Performance

A20379025 Hsueh Yi Chen
A20385942 Yitong Huang

CPU Benchmark:

- The program finds GFLOPS and GIOPS per second
- The start time and end time is calculated
- The total time is difference between start time and end time
- The performance is measured using the formula

OPS = number of instructions / Total time

System Configuration:

1. **for without AVX program and 600 sample program**
   Chameleon Cloud(openstack KVM)
   Processor: Intel Xeon E312xx (Sandy Bridge) @ 3.09 GHz
   1 processor, 32 cores
   OS:CentOS7
   RAM: 4GB
   Disk: 40GB
2. **for without AVX program**
   Chameleon Cloud(baremetal)
   Processor: Intel(R) Xeon(R) CPU E5-2670 v3 @ 2.30GHz

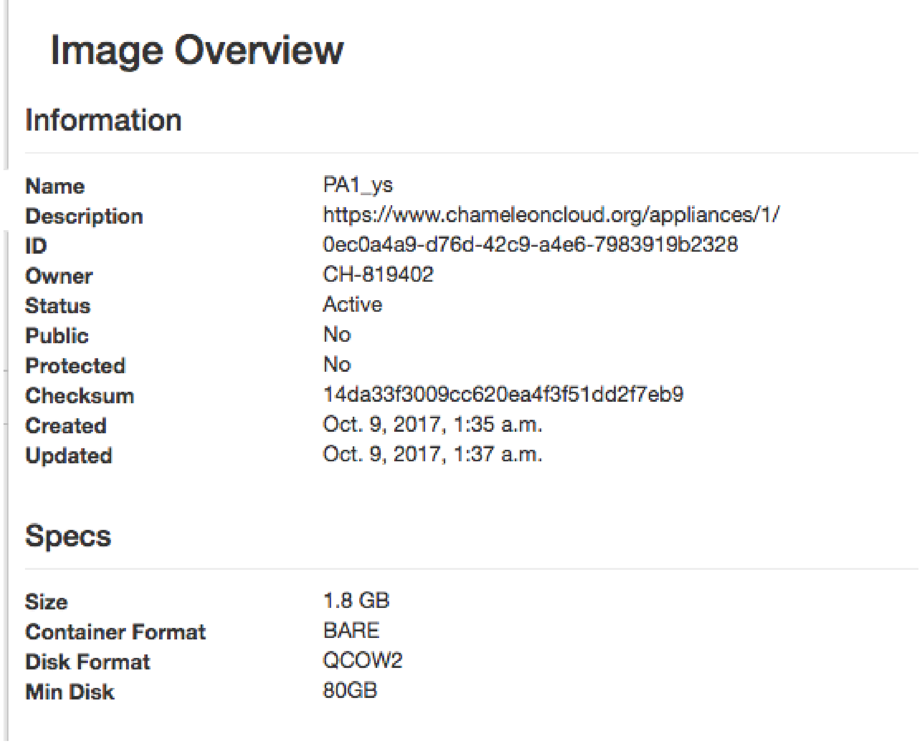


Image Overview

Information

Name	PA1_ys
Description	https://www.chameleoncloud.org/appliances/1/
ID	0ec0a4a9-d76d-42c9-a4e6-7983919b2328
Owner	CH-819402
Status	Active
Public	No
Protected	No
Checksum	14da33f3009cc620ea4f3f51dd2f7eb9
Created	Oct. 9, 2017, 1:35 a.m.
Updated	Oct. 9, 2017, 1:37 a.m.

Specs

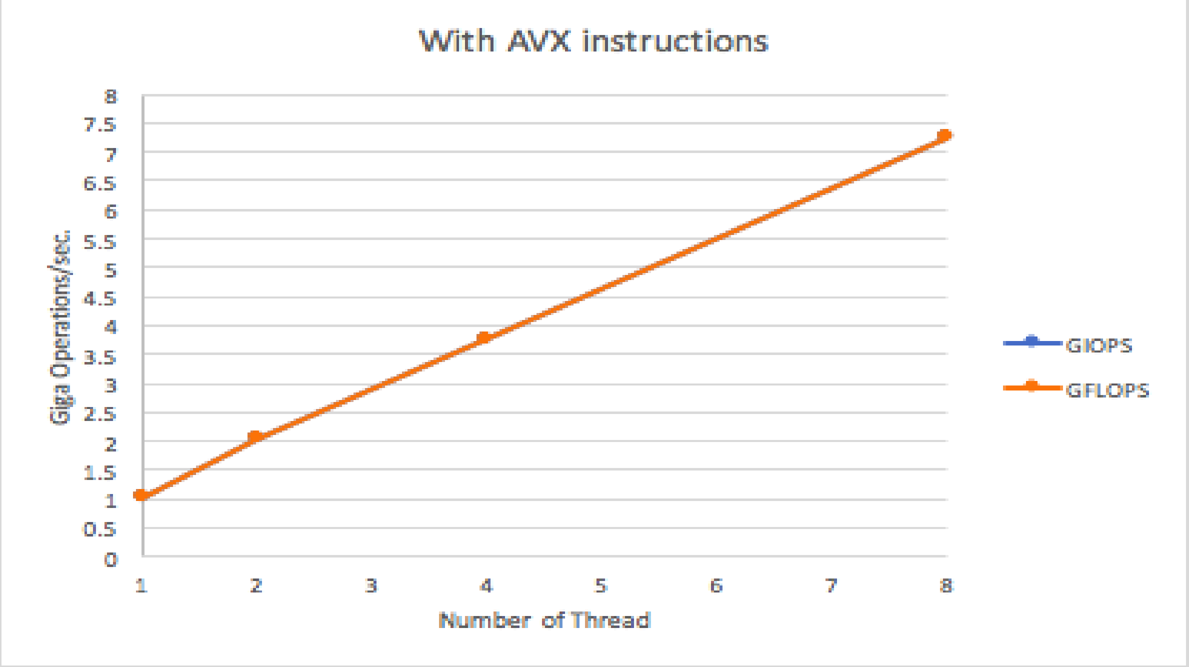
Size	1.8 GB
Container Format	BARE
Disk Format	QCOW2
Min Disk	80GB

Experiment 1~8:

Two programs are run 1 time to calculate GIOPS and GFLOPS with AVX instructions and without AVX instructions, respectively.The result is recorded in the table 1 and table 2 shown below:

Operations/sec	GIOPS	GFLOPS
1 thread	1.012469	1.012507
2 threads	2.023625	2.023738
4 threads	3.763475	3.765623
8 threads	7.250366	7.2515

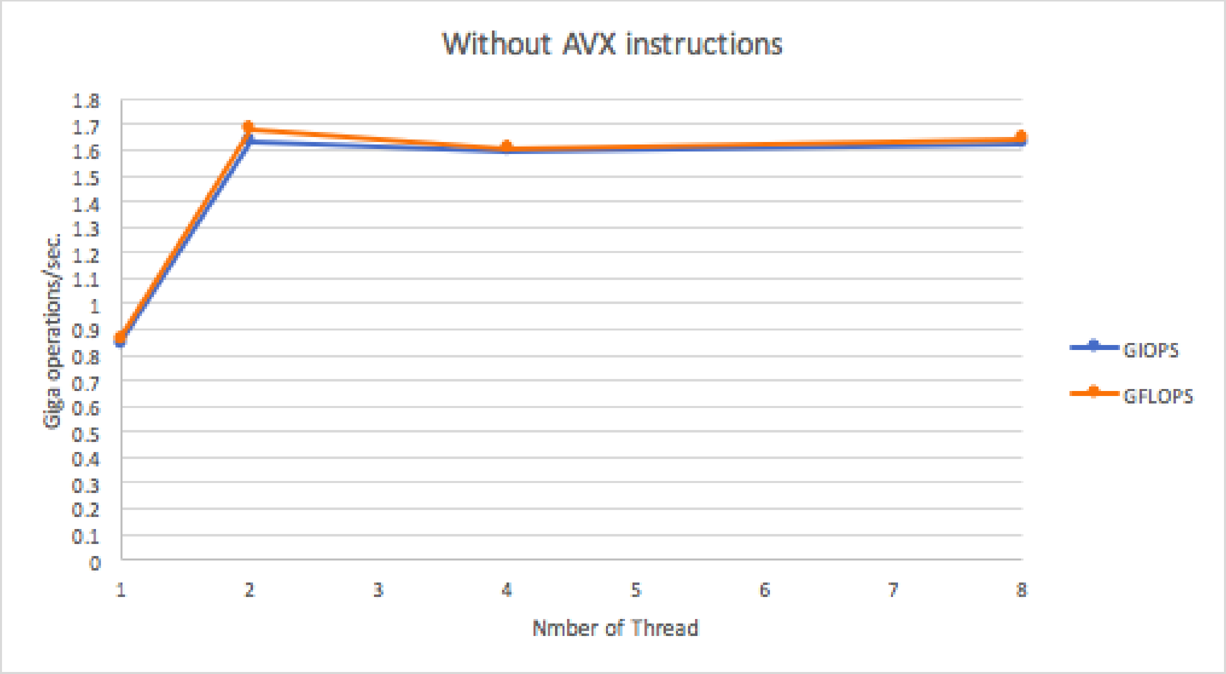
Table 1. Performance with AVX instructions



Operations/sec	GIOPS	GFLOPS
1 thread	0.846123	0.863293
2 threads	1.634691	1.680904
4 threads	1.598174	1.606577

8 threads	1.624932	1.643697

Table 2. Performance without AVX instructions



Observation:

- After using AVX instructions, the efficiency of GFLOPS and GIOPS is 2 or 3 times better than the result without AVX instructions.
- Based on the result without AVX instructions, GIOPS and GFLOPS are always growing up when having 2 threads.
- In case of the result with AVX instructions, there is a constant linear growth as the number of threads increases. On the other hand, the result without AVX instructions continues to rise until hitting a plateau after getting more than 2 threads.

Experiment 9,10:

A program was run for 600 seconds and recorded 600 samples for IOPS and FLOPS. Each Dataset is plotted in the line chart 1 and chart 2. The charts are shown below:

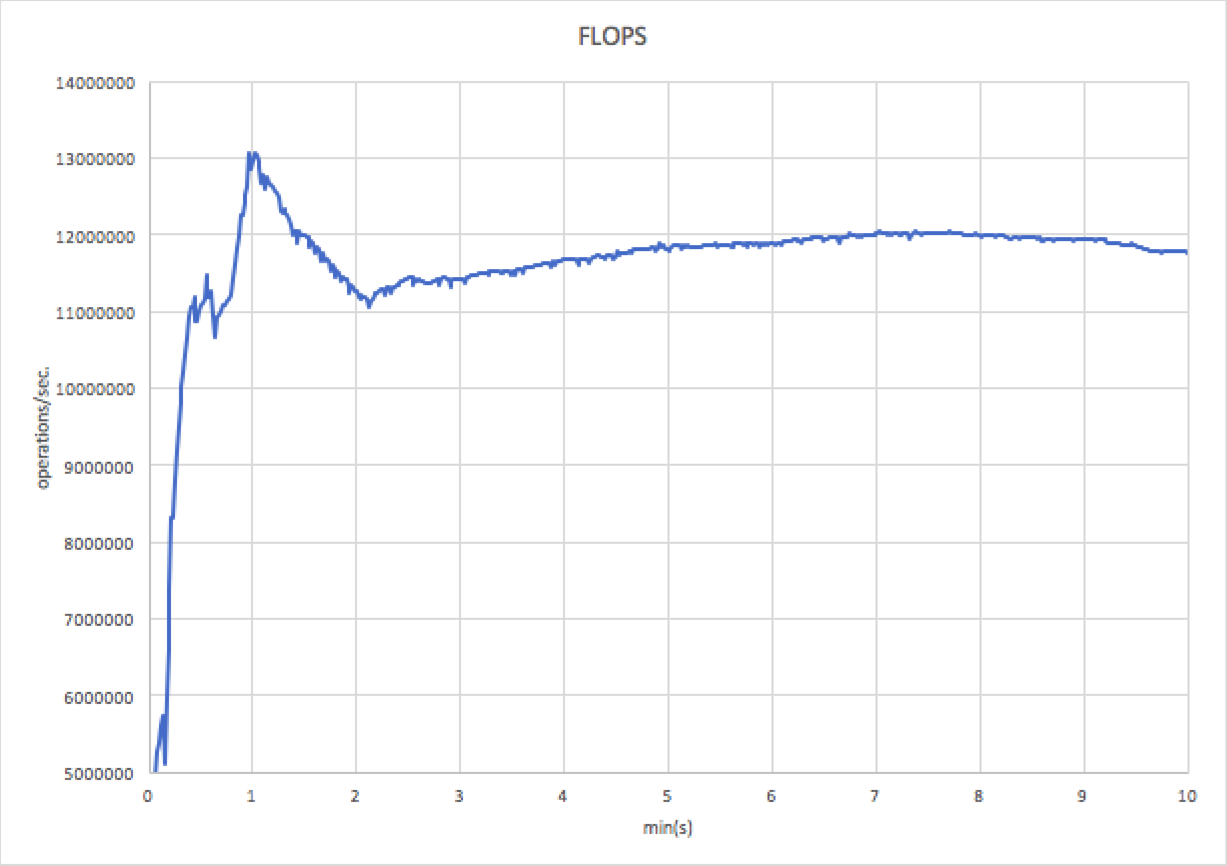


Chart 1.600 samples for FLOPS

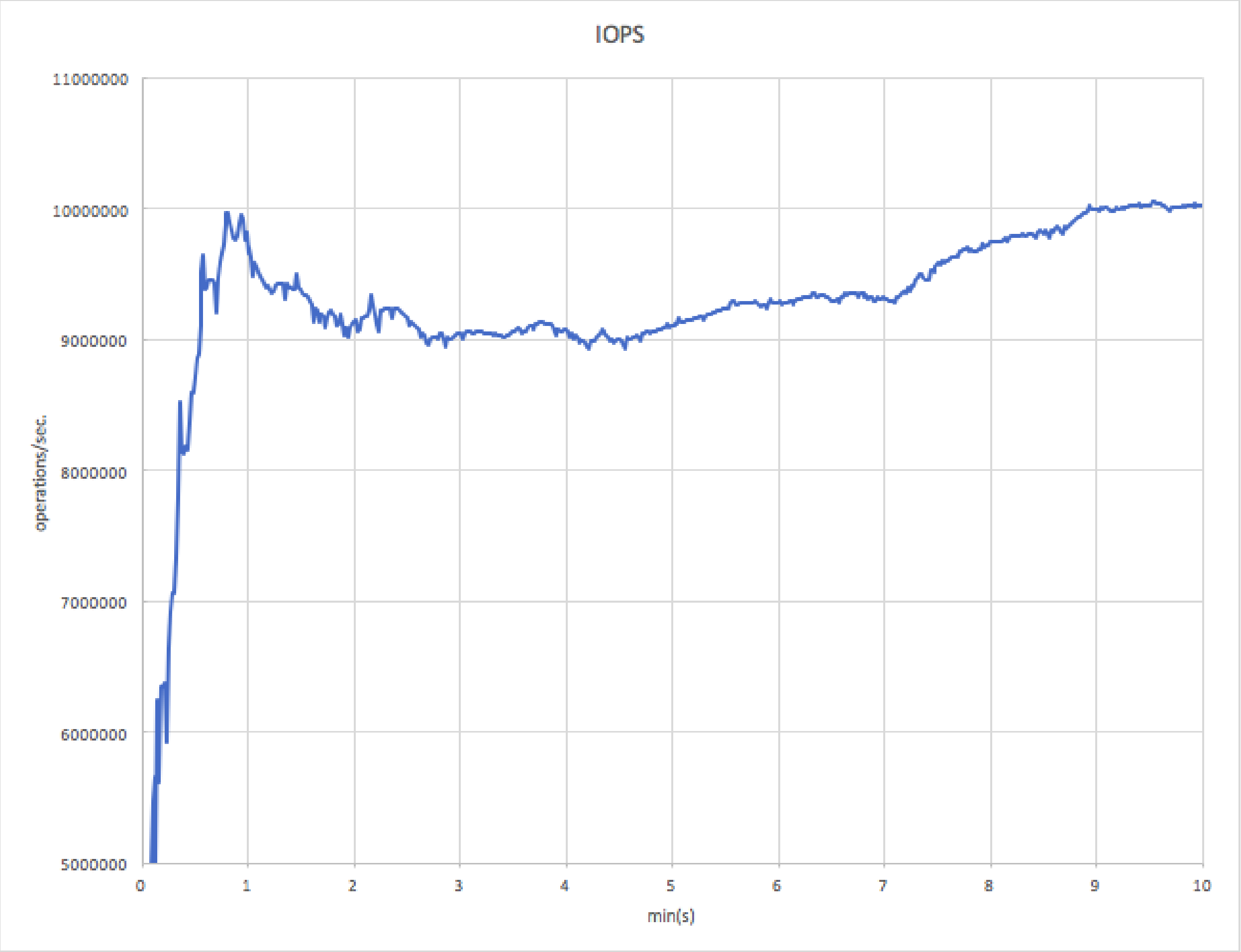


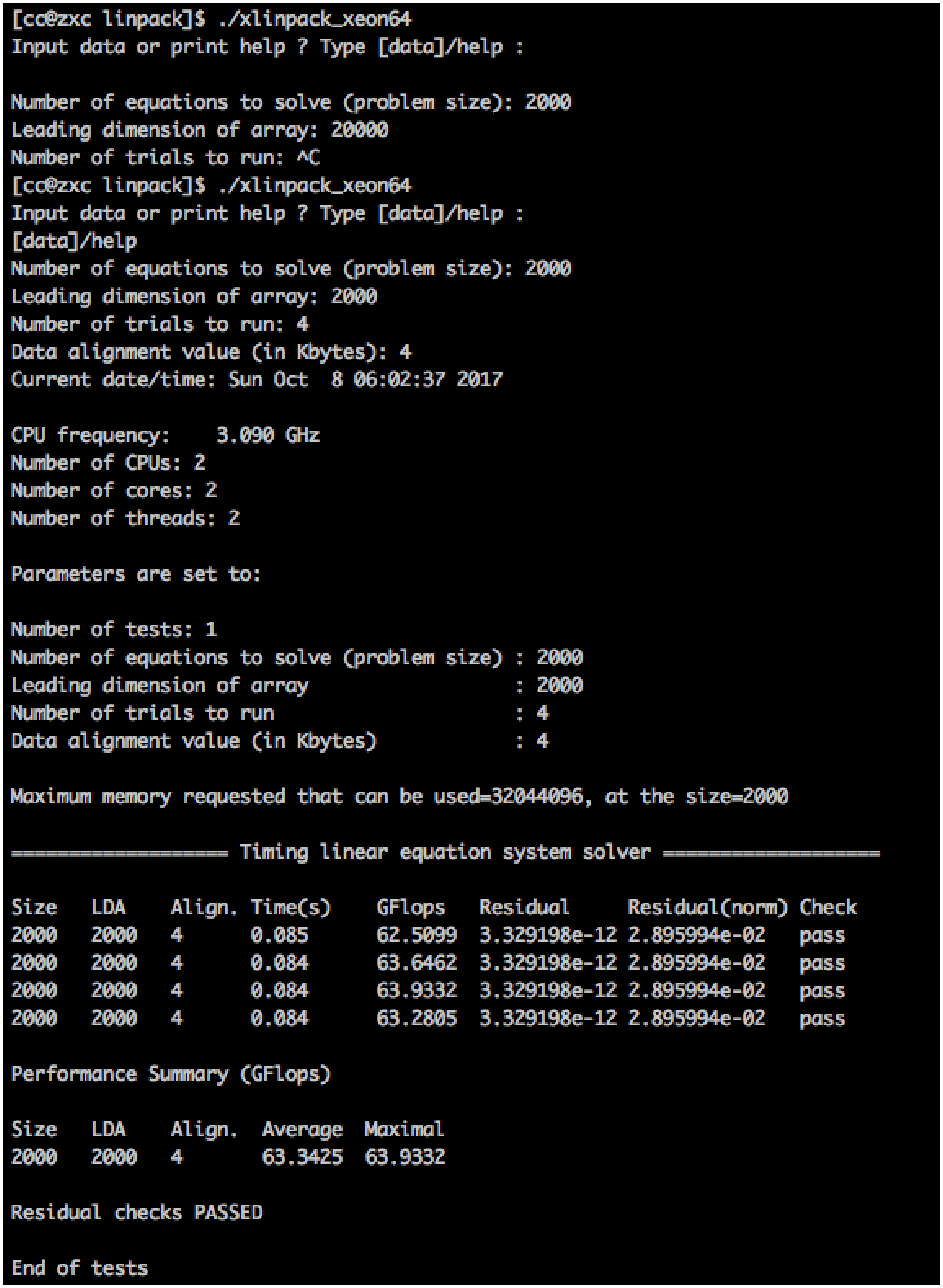
Chart 2.600 samples for IOPS

Observation:

- In the case of the result for IOPS and FLOPS, each of them has an apparent shooting up in the first minute. But in the second minute, each of them declines a little simultaneously. Then both of them upturn gradually in final eight minutes.
- The difference between IOPS and FLOPS is each sample is 10 million times larger than each sample in IOPS.

Experiment 11:

Linpack was run and the following results was shown below:



```
[cc@zxc linpack]$ ./xlinpack_xeon64
Input data or print help ? Type [data]/help :

Number of equations to solve (problem size): 2000
Leading dimension of array: 20000
Number of trials to run: ^C
[cc@zxc linpack]$ ./xlinpack_xeon64
Input data or print help ? Type [data]/help :
[data]/help
Number of equations to solve (problem size): 2000
Leading dimension of array: 2000
Number of trials to run: 4
Data alignment value (in Kbytes): 4
Current date/time: Sun Oct  8 06:02:37 2017

CPU frequency:    3.090 GHz
Number of CPUs: 2
Number of cores: 2
Number of threads: 2

Parameters are set to:

Number of tests: 1
Number of equations to solve (problem size) : 2000
Leading dimension of array                  : 2000
Number of trials to run                     : 4
Data alignment value (in Kbytes)            : 4

Maximum memory requested that can be used=32044096, at the size=2000

=================== Timing linear equation system solver ===================

Size   LDA    Align. Time(s)    GFlops   Residual     Residual(norm) Check
2000   2000   4      0.085      62.5099  3.329198e-12 2.895994e-02   pass
2000   2000   4      0.084      63.6462  3.329198e-12 2.895994e-02   pass
2000   2000   4      0.084      63.9332  3.329198e-12 2.895994e-02   pass
2000   2000   4      0.084      63.2805  3.329198e-12 2.895994e-02   pass

Performance Summary (GFlops)

Size   LDA    Align.  Average  Maximal
2000   2000   4       63.3425  63.9332

Residual checks PASSED

End of tests
```

Theoretical Peak Performance:
The CentOs Linux instance has a processor: Intel Xeon E312xx (Sandy Bridge) @3.09GHz
The closest to it is Intel® Xeon® Processor E3-1225 and its performance is :

Processor number	Frequency Type	Clock	GFLOP
E3-1225	Base	3.10GHz	99.2
	Max Turbo	3.40GHz	109

Conclusion:

- Compared to theoretical Peak performance, the algorithm used in my own program is not as efficient as theoretical Peak performance. It wastes much time for lots of processing like assigning variables or passing pointer.
- In the case of Linpack performance, values of GFLOPS are much larger than values from my own program. Although I uses AVX instructions to improve the efficiency, but the difference of GFLOPS between Linpack and my own program is still large.

Memory Benchmark:

- The program measures the latency and throughput of memory speed by allocating 1.28GB memory
- There are three operations, sequential read+write, sequential write and random write
- Latency is measured in microseconds/8B
- Throughput is measured in MB/s

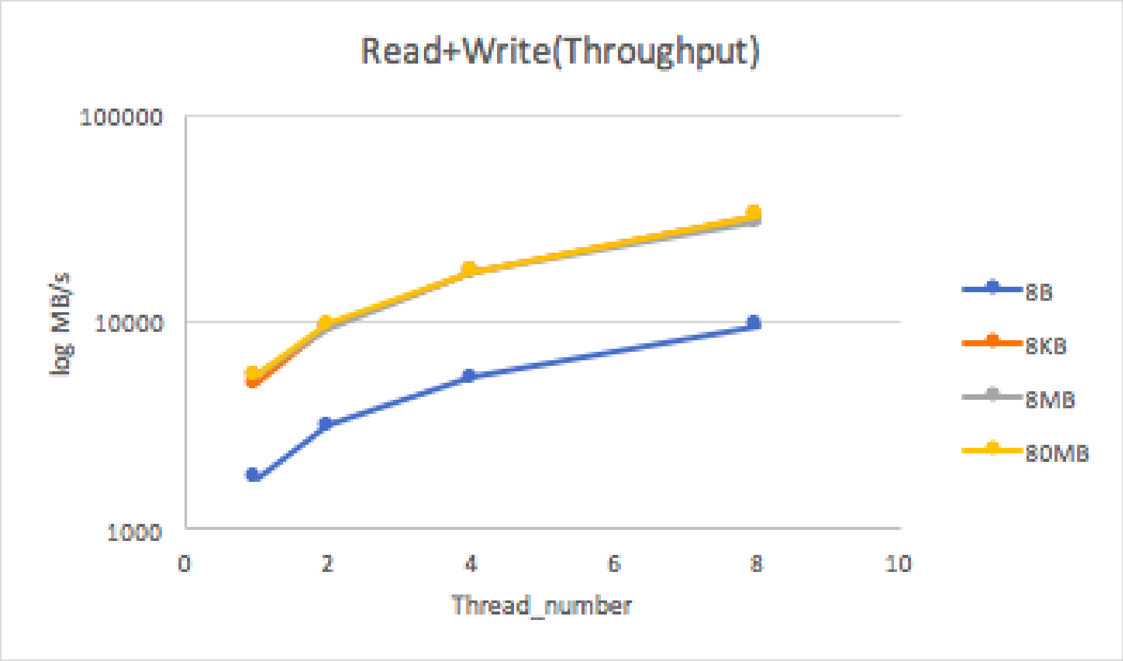
System Configuration:

Chameleon Cloud(openstack KVM)
Processor: Intel Xeon E312xx (Sandy Bridge) @ 3.09 GHz
OS:CentOS7
RAM: 16GB
Disk: 160GB
(instance size:xlarge, since it will get out of memory when running in medium size)

Experiment 1~48:

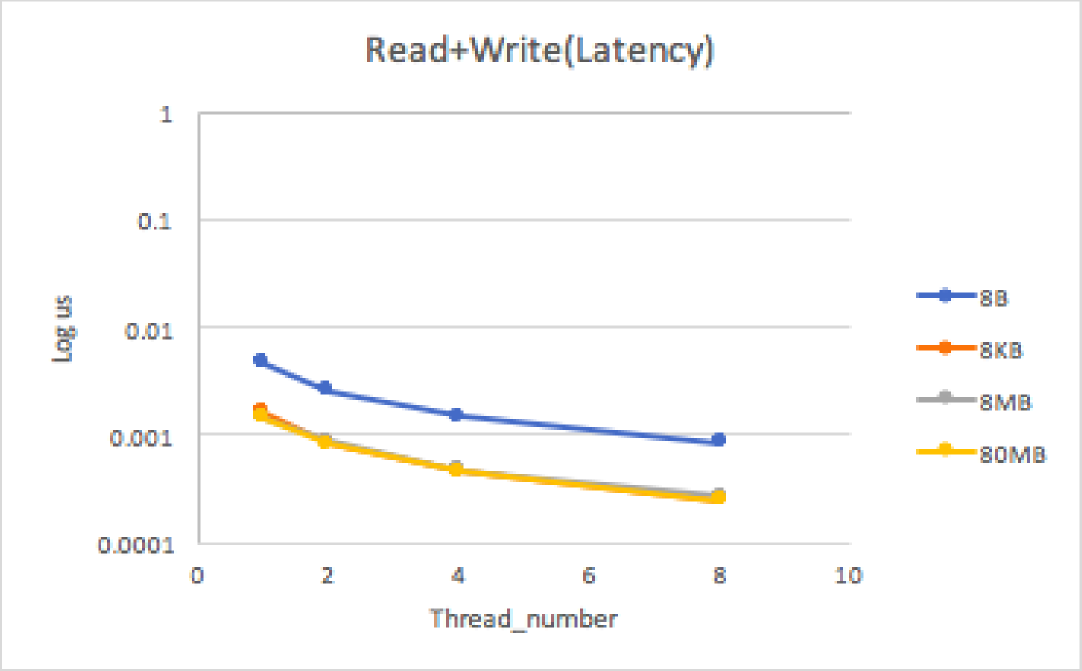
- Sequential read+write:
  Throughput:

Block size	1 thread	2 threads	4 threads	8 threads
8B	1753.866728	3137.485599	5376.840938	9487.806686
8KB	4992.530706	9448.098201	17447.48715	32504.63445
8MB	5529.969844	9299.89247	16986.26501	29970.96563
80MB	5549.77454	9638.917128	17392.72223	32482.36309



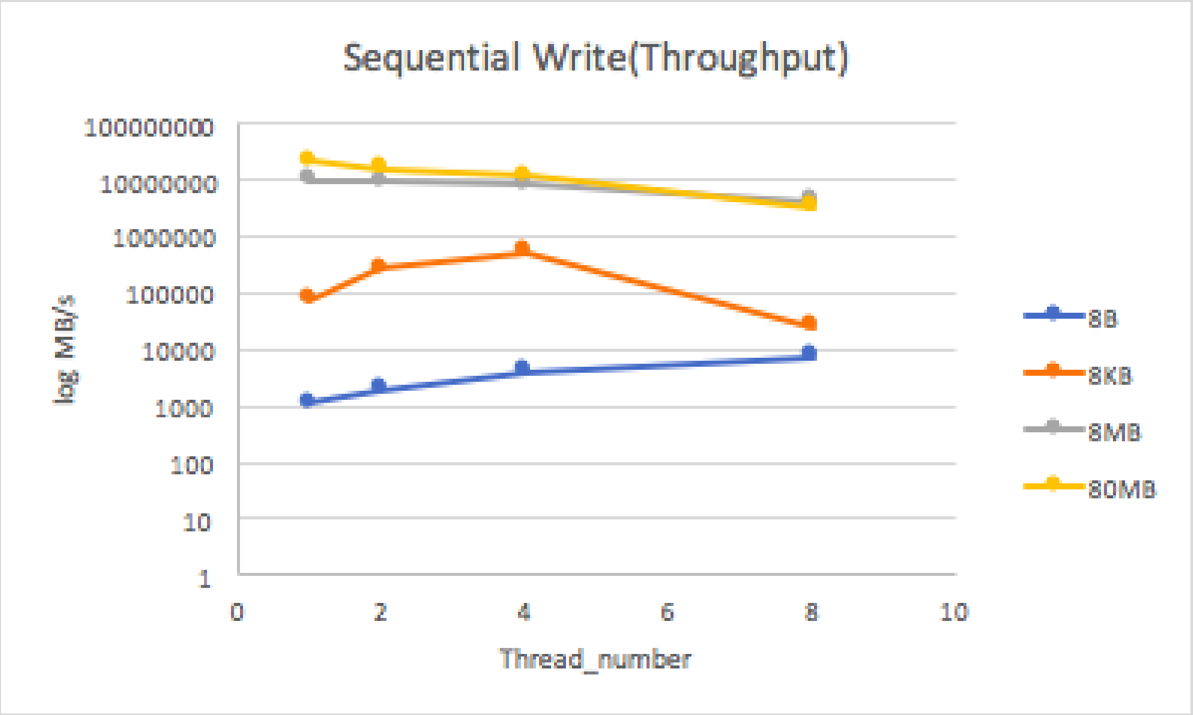
Latency:

Block size	1 thread	2 threads	4 threads	8 threads
8B	0.00456135	0.002549812	0.001487862	0.000843187
8KB	0.001602394	0.000846731	0.000458519	0.000246119
8MB	0.001446662	0.000860225	0.000470969	0.000266925
80MB	0.0014415	0.000829969	0.000459963	0.000246287



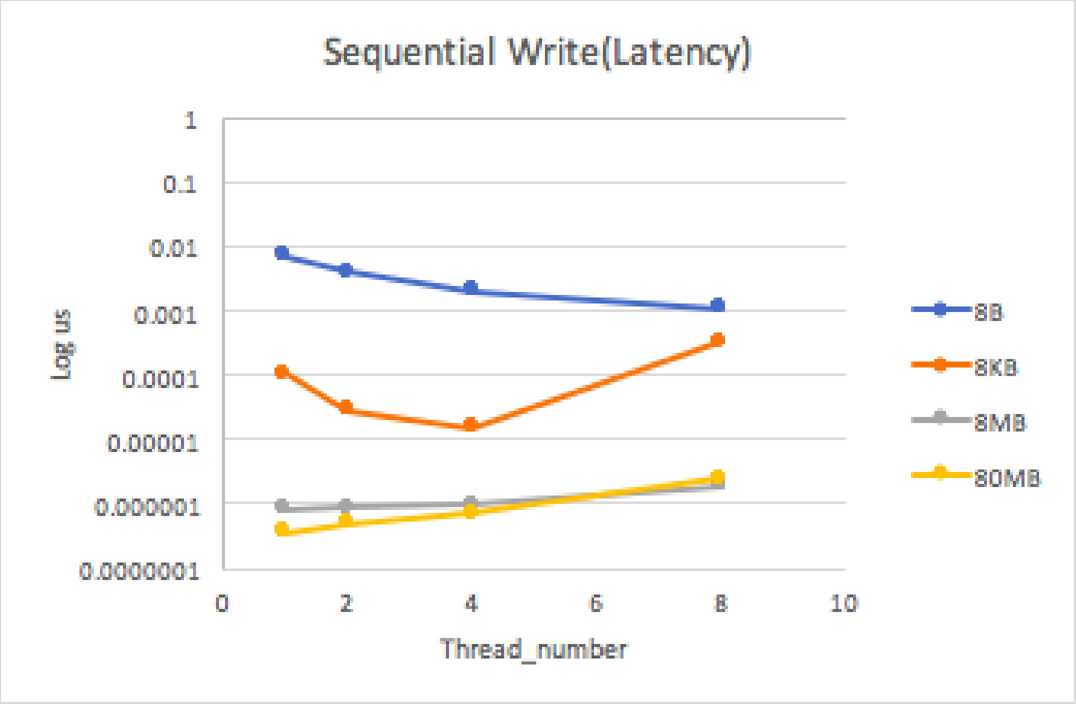
- Sequential Read:
  Throughput:

Block size	1 thread	2 threads	4 threads	8 threads
8B	1140.350408	1969.912662	3947.620017	7419.257611
8KB	77122.37151	276996.3211	521810.0285	24931.34142
8MB	9624060.15	9343065.693	8258064.516	4155844.156
80MB	21694915.25	16202531.65	11531531.53	3224181.36



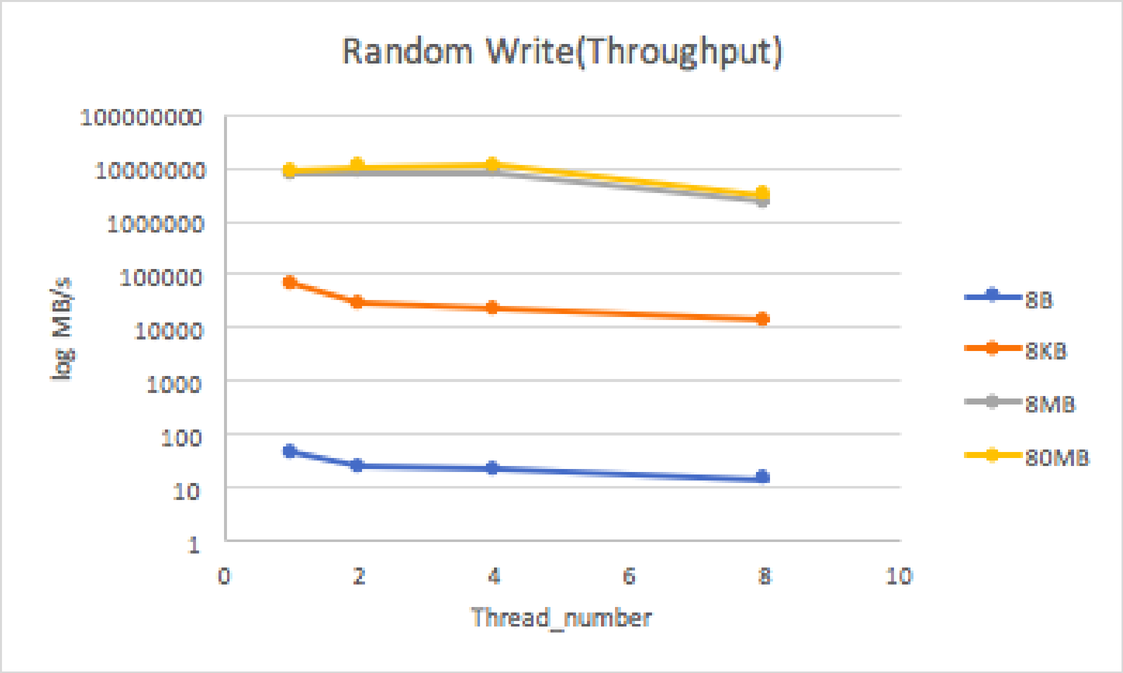
Latency:

Block size	1 thread	2 threads	4 threads	8 threads
8B	0.007015388	0.004061094	0.002026537	0.001078275
8KB	0.000103731	0.000028881	0.000015331	0.000320881
8MB	0.000000831	0.000000856	0.000000969	0.000001925
80MB	0.000000369	0.000000494	0.000000694	0.000002481



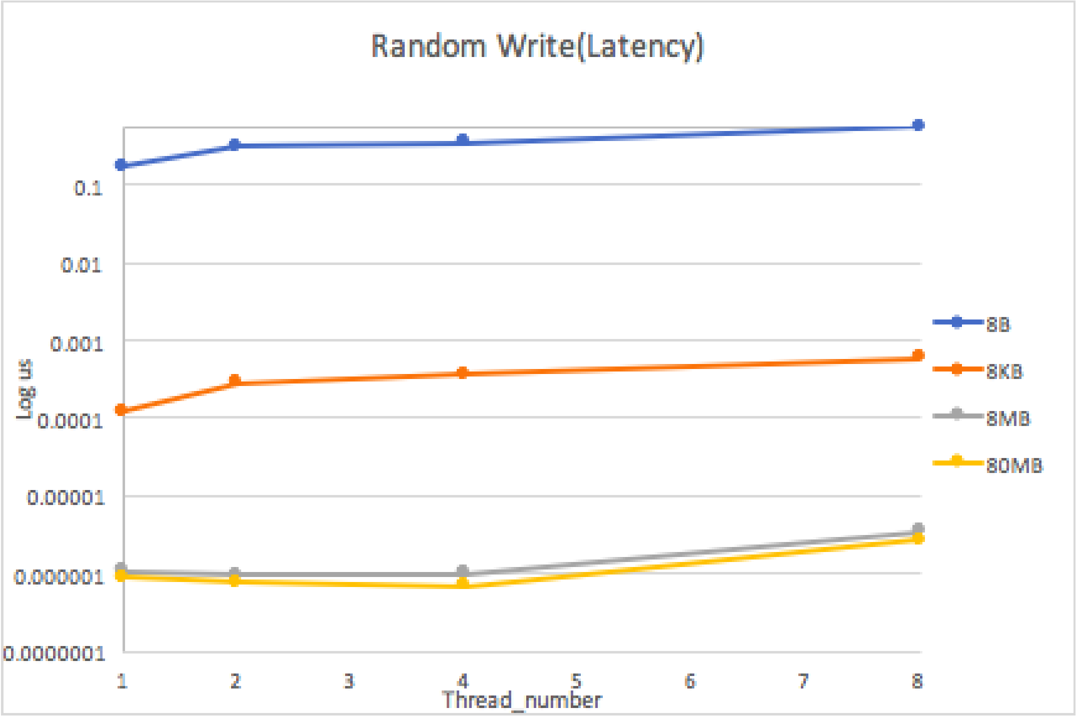
- Random Write:
  Throughput:

Block size	1 thread	2 threads	4 threads	8 threads
8B	46.189383	25.173689	22.304231	14.148185
8KB	66273.16972	27964.69457	21829.96504	13555.15784
8MB	7619047.619	8476821	8152866.242	2335766.423
80MB	9014084.507	10322580.65	11428571.43	3033175



Latency:

Block size	1 thread	2 threads	4 threads	8 threads
8B	0.173199975	0.317792119	0.358676344	0.565443563
8KB	0.000120713	0.000286075	0.000366469	0.000590181
8MB	0.00000105	0.000000944	0.000000981	0.000003425
80MB	0.000000887	0.000000775	0.0000007	0.000002638

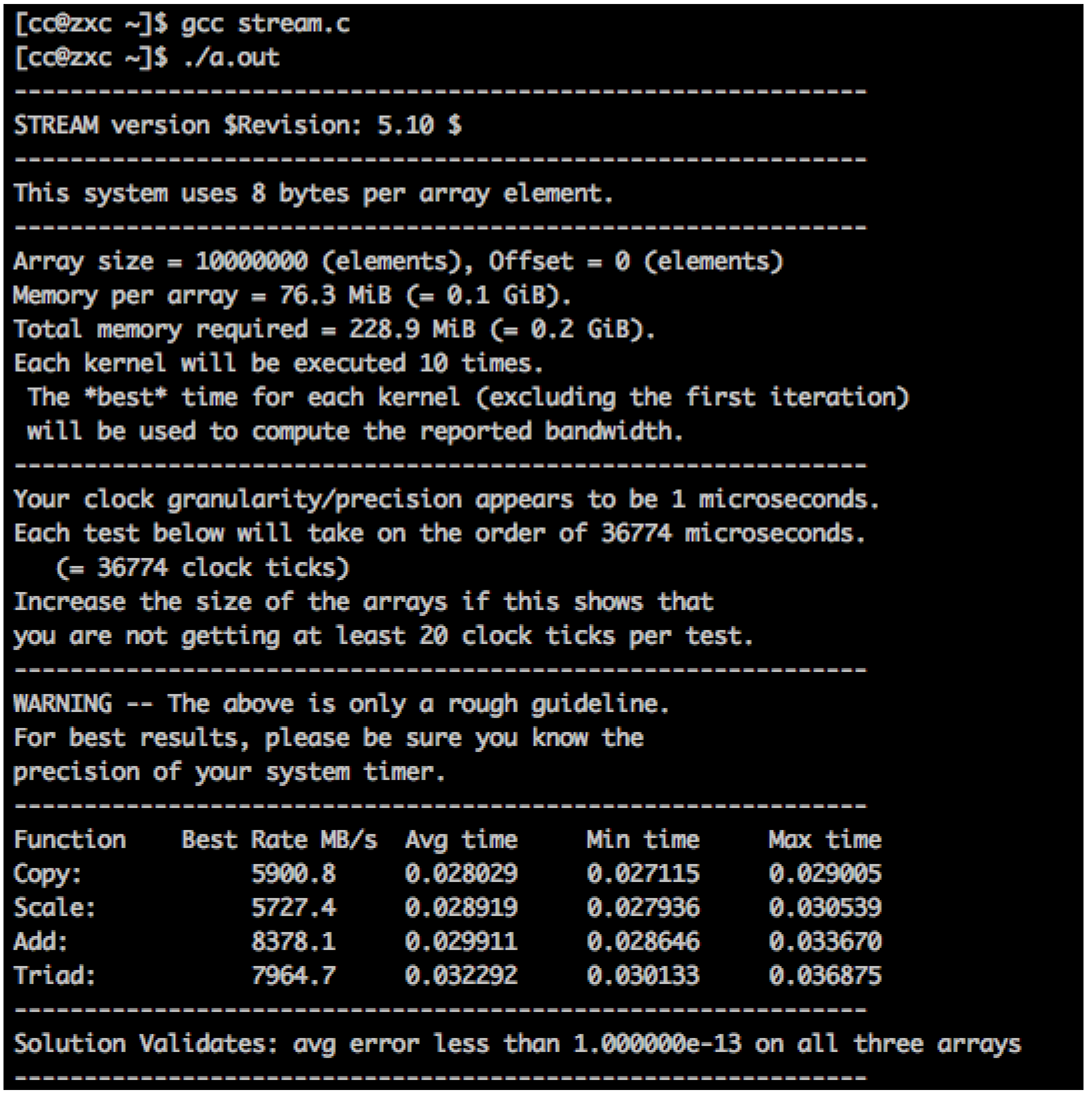


Observation:

- As the number of thread grows up, the speed of transfer also increases until hitting a plateau when having 4 threads. Some operations like random write and sequential write even get worse efficiency when having 8 threads.
- Sequential write is always faster than random write
- Increasing the block size can’t always improve the speed of transfer. In most of experiments, 8MB and 80MB almost get the same performance.
- Small block size like 8B always get the highest latency.
- Based on the result of each operations, 4 threads of concurrency can get the best performance.

Experiment 49:

Stream was run to estimate the memory benchmarking



```
[cc@zxc ~]$ gcc stream.c
[cc@zxc ~]$ ./a.out
-------------------------------------------------------------
STREAM version $Revision: 5.10 $
-------------------------------------------------------------
This system uses 8 bytes per array element.
-------------------------------------------------------------
Array size = 10000000 (elements), Offset = 0 (elements)
Memory per array = 76.3 MiB (= 0.1 GiB).
Total memory required = 228.9 MiB (= 0.2 GiB).
Each kernel will be executed 10 times.
 The *best* time for each kernel (excluding the first iteration)
 will be used to compute the reported bandwidth.
-------------------------------------------------------------
Your clock granularity/precision appears to be 1 microseconds.
Each test below will take on the order of 36774 microseconds.
   (= 36774 clock ticks)
Increase the size of the arrays if this shows that
you are not getting at least 20 clock ticks per test.
-------------------------------------------------------------
WARNING -- The above is only a rough guideline.
For best results, please be sure you know the
precision of your system timer.
-------------------------------------------------------------
Function    Best Rate MB/s  Avg time     Min time     Max time
Copy:            5900.8     0.028029     0.027115     0.029005
Scale:           5727.4     0.028919     0.027936     0.030539
Add:             8378.1     0.029911     0.028646     0.033670
Triad:           7964.7     0.032292     0.030133     0.036875
-------------------------------------------------------------
Solution Validates: avg error less than 1.000000e-13 on all three arrays
-------------------------------------------------------------
```

Theoretical Peak Performance:

The CentOs Linux instance has a processor: Intel Xeon E312xx (Sandy Bridge) 3.09 GHz RAM:16GB

The closet to its is Intel Xeon E312xx (Sandy Bridge) @ 1.80 GHz RAM:32GB and its performance is :

Operation type	Speed
Sequential Read	3.52GB/s
Sequential	2.94GB/s

Write	
Stdlib Copy	1.96GB/s

Conclusion:

- The formulated values of each operation mentioned above are much larger than result in Stream benchmark and theoretical performance.
- Compared to theoretical memory bandwidth and Stream benchmark, the algorithm used in my own program missed some part of memory when allocating memory. Therefore, the actual values would be less than formulated values.

Disk Benchmark

- The program finds throughput in MB per second and latency in ms
- The 10 GB binary file is allocated first
- The start time and end time is calculated only for actual read+rewrite, sequential read and random read
- The total time is difference between start and end time
- The actual operation data size is adjusted during experiment to make sure the run time is at least 10 seconds but no so long

- The throughput performance is measured using formula

Throughput = total operation data size / total time

- The latency performance is measured using formula

Latency = total time / total operation data size x 8 (since requirement asked to use 8B block size to measure latency)

System Configuration

Chameleon Cloud(openstack KVM)

Processor: Intel Xeon E312xx (Sandy Bridge) @ 3.09 GHz
1 processor, 32 cores
OS:CentOS7
RAM: 4GB
Disk: 40GB

Experiment 1~16

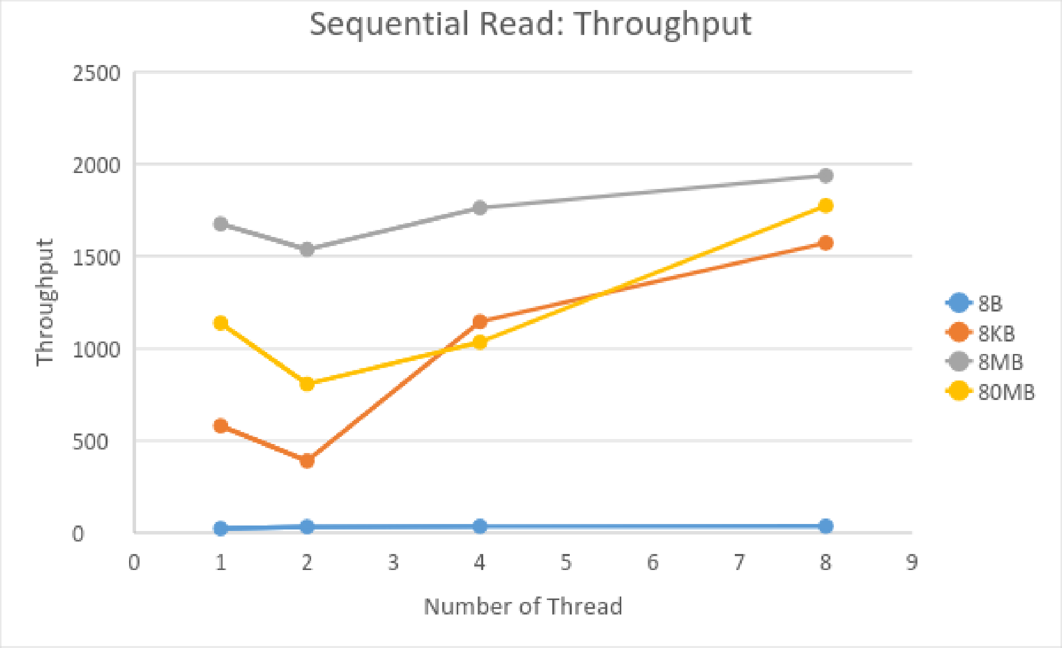
- Read Sequential program is run for testing sequential read performance within the 10 GB binary data file. The sequential read data size in below table means the data size operated by each experiment. This data size is adjusted during testing to make sure the run time is 10s. But the sequential read data size was fixed for each block size to make sure strong scaling.

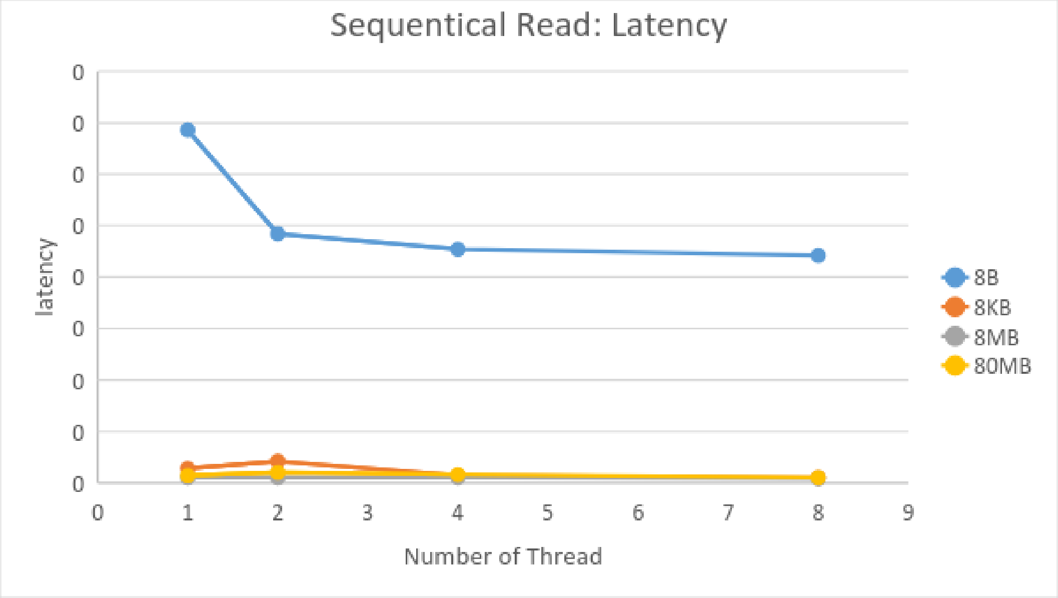
Result shows in below table and graphs.

block_size	thread number	throughtput (MBps)	latency(ms)	run time(s)	Sequential Read	file datasize

					datasize(B)	
8B	1	23.347399	0.000343	42.831324	1.00E+09	10GB
	2	33.107913	0.000242	30.20426	1.00E+09	10GB
	4	35.227066	0.000227	28.387263	1.00E+09	10GB
	8	36.245878	0.000221	27.589344	1.00E+09	10GB
8KB	1	580.293798	0.000014	51.69795	3.00E+10	10GB
	2	390.117129	0.000021	76.899981	3.00E+10	10GB
	4	1145.140031	0.000007	26.197669	3.00E+10	10GB
	8	1571.941037	0.000005	19.084685	3.00E+10	10GB
8MB	1	1675.124039	0.000005	29.848536	5.00E+10	10GB
	2	1536.660188	0.000005	32.538098	5.00E+10	10GB
	4	1762.40535	0.000005	28.370318	5.00E+10	10GB
	8	1936.964278	0.000004	25.813589	5.00E+10	10GB
80MB	1	1137.94976	0.000007	43.938671	5.00E+10	10GB
	2	806.951948	0.00001	61.961558	5.00E+10	10GB
	4	1034.113558	0.000008	48.350589	5.00E+10	10GB
	8	1774.294539	0.000005	28.180214	5.00E+10	10GB

Table Performance of sequential read





Experiment 17~32

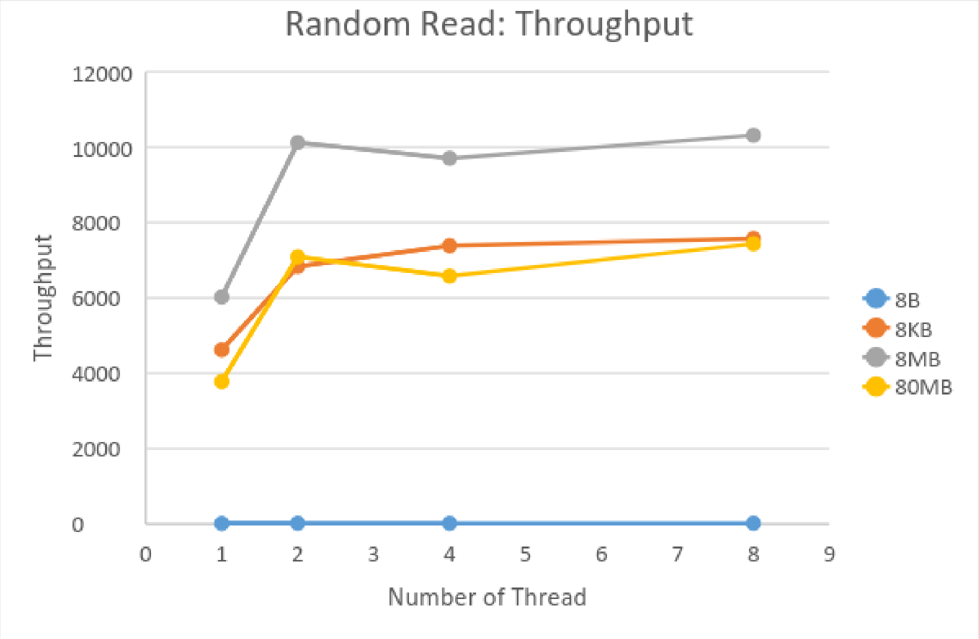
- Read Random program is run for testing Random read performance within the 10 GB binary data file.
- The randomization was implemented by rand() function. We seek a random block sized pointer before every reading. And then read the randomly seeking block.
- Other notifications are the same with experiment 1~16.

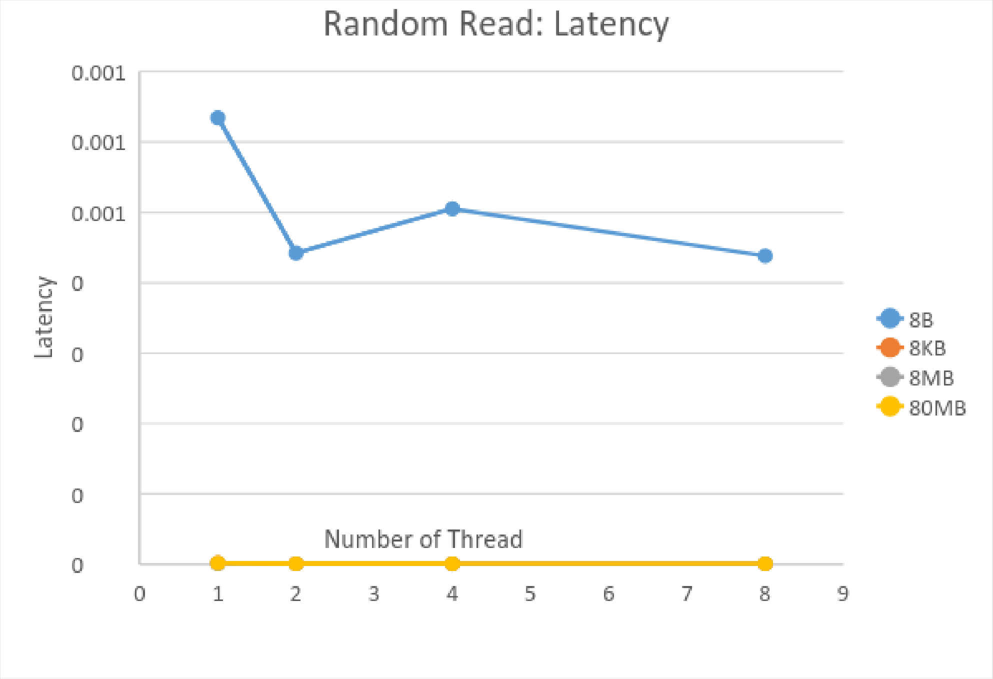
Result shows in below table and graphs.

block_size	thread number	throughtput (MBps)	latency(ms)	run time(s)	Random Read datasize(B)	file datasize
8B	1	12.625589	0.000634	79.204226	1.00E+09	10GB
	2	18.104972	0.000442	55.233447	1.00E+09	10GB
	4	15.836879	0.000505	63.143756	1.00E+09	10GB
	8	18.266729	0.000438	54.744339	1.00E+09	10GB
8KB	1	4622.064161	0.000002	17.308284	8.00E+10	10GB
	2	6836.899511	0.000001	11.70121	8.00E+10	10GB
	4	7386.729248	0.000001	10.830233	8.00E+10	10GB
	8	7576.294088	0.000001	10.559252	8.00E+10	10GB
8MB	1	6027.785294	0.000001	33.179682	2.00E+11	10GB
	2	10124.20053	0.000001	19.754646	2.00E+11	10GB
	4	9708.827844	0.000001	20.599809	2.00E+11	10GB
	8	10318.74739	0.000001	18.715544	2.00E+11	10GB
80MB	1	3779.746878	0.000002	26.456798	1.00E+11	10GB
	2	7090.083106	0.000001	14.104207	1.00E+11	10GB
	4	6581.964264	0.000001	15.193033	1.00E+11	10GB

	8	7437.105123	0.000001	13.446092	1.00E+11	10GB

Table Performance of random read





Experiment 33~48

- Read+write program is run for testing read and rewrite performance within the 10 GB binary data file.

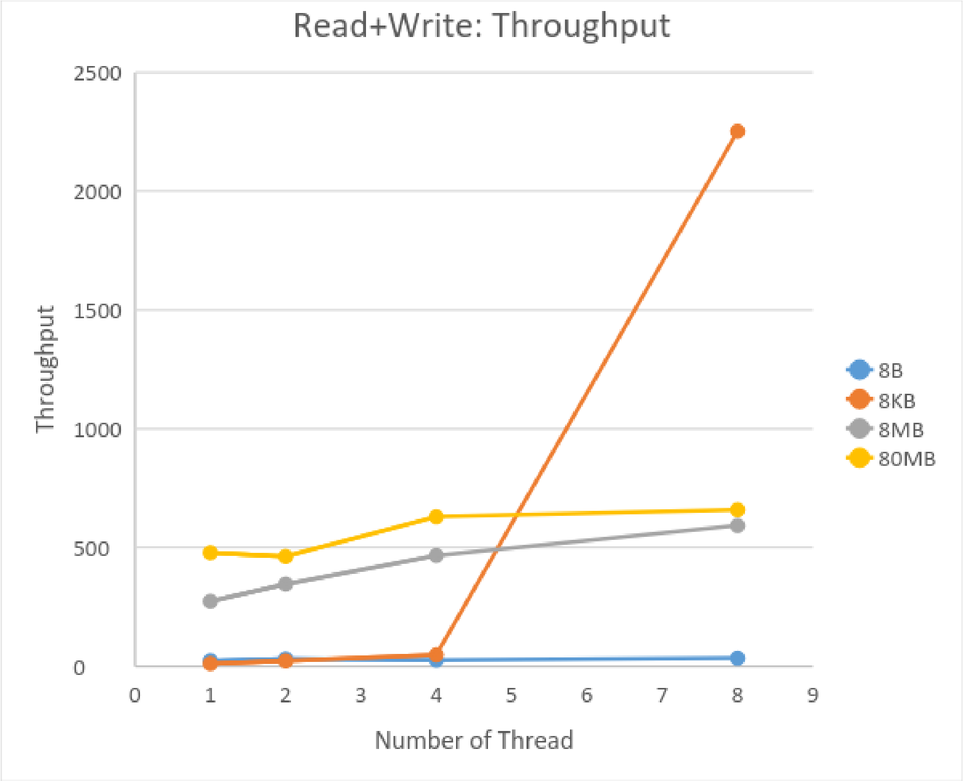
- The program was implemented as following, first sequentially read the whole file; then sequentially re-write the file for a fixed data size.
- Other notifications are the same with experiment 1~16.

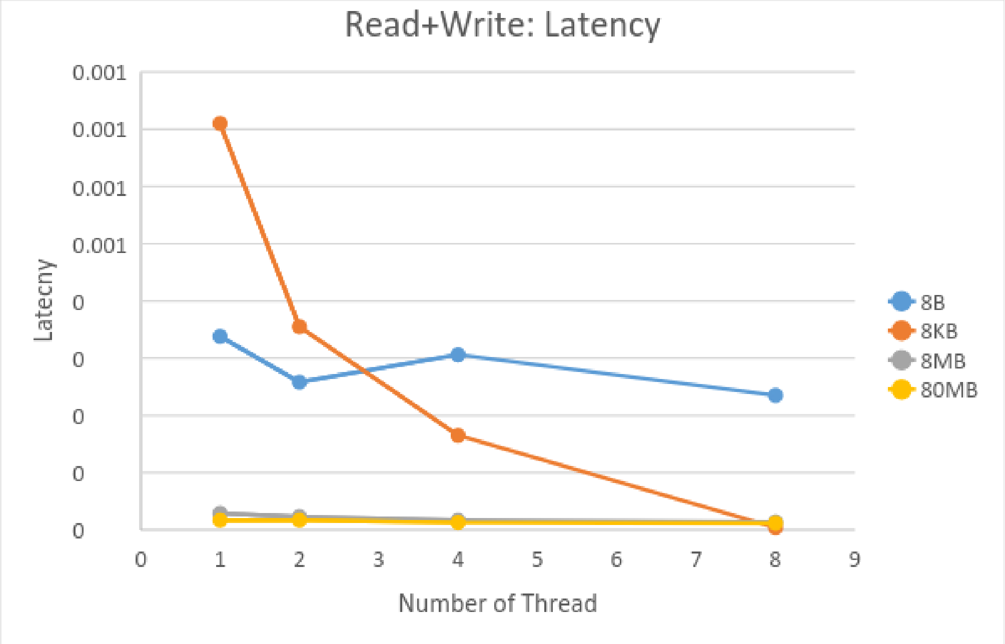
Result shows in below table and graphs.

block_size	thread number	throughtput (MBps)	latency(ms)	run time(s)	Read Datasize(B)	Write Datasize(B)	file Datasize
8B	1	23.691809	0.000338	44.319115	1.00E+09	5.00E+08	10GB
	2	30.948178	0.000258	33.927683	1.00E+09	5.00E+08	10GB
	4	26.165928	0.000306	40.128521	1.00E+09	5.00E+08	10GB
	8	34.01607	0.000235	30.867764	1.00E+09	5.00E+08	10GB
8KB	1	11.26622	0.00071	710.087304	1.00E+09	5.00E+09	10GB
	2	22.529044	0.000355	399.484324	1.00E+09	5.00E+09	10GB
	4	48.502913	0.000165	123.70391	1.00E+09	5.00E+09	10GB
	8	2251.984162	0.000004	2.664317	1.00E+09	5.00E+09	10GB
8MB	1	273.694321	0.000029	40.190823	1.00E+09	1.00E+10	10GB
	2	346.312504	0.000023	31.763219	1.00E+09	1.00E+10	10GB
	4	466.457173	0.000017	23.582015	1.00E+09	1.00E+10	10GB
	8	592.145975	0.000014	18.5765	1.00E+09	1.00E+10	10GB
80MB	1	477.937483	0.000017	23.015563	1.00E+09	1.00E+10	10GB
	2	462.167589	0.000017	23.80089	1.00E+09	1.00E+10	10GB
	4	629.337257	0.000013	17.478705	1.00E+09	1.00E+10	10GB
	8	658.779646	0.000012	16.697541	1.00E+09	1.00E+10	10GB

Table . Performance of read+write

Here 5.00E+8=500000000





Observation and conclusion 1:

- In general, the throughput increases as number of threads increase; the latency decreases as the number of threads decreases. It agrees with the strong scaling performance.

- In general, the throughput increases as the block size increases. My explanation is: as each block size gets smaller, you pay more and more of a penalty for the disk seeks.
- Read operations (both sequential read and random read) speed are obviously faster than read+write operation.
- For small block size (8B), sequential read is faster than random read; while for larger block size, random read is faster than sequential read. My explanation is that rand() function has the advantage to make disk takes less time for seeking, especially as block size gets large.
- The optimal concurrency to get the best performance is thread number 8, because we are using strong scaling, the more number of threads will increase the performance.
- The hard ware we are testing is

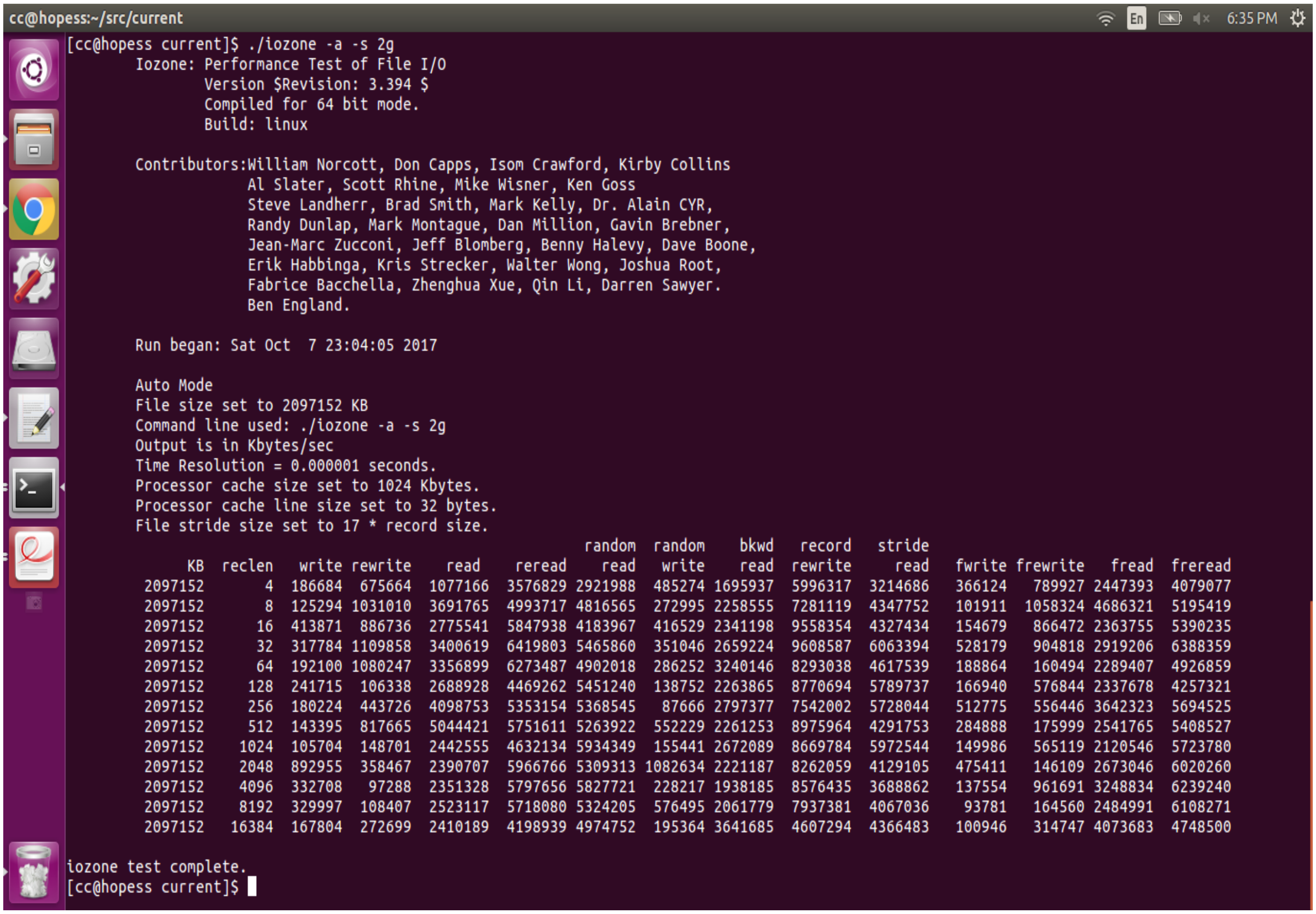
"device": "sda",
"driver": "mptsas",
"interface": "SCSI",
"vendor": "SEAGATE"

- Based on my performance evaluation, it is HDD.

Experiment 49

- Run IOZONE benchmark
- Since our node RAM size is 4G. As per IOZONE instruction, we should Change the -s to reflect half of your total physical RAM, i.e. 2G.

The screenshot is shown as below:



```
[cc@hopess current]$ ./iozone -a -s 2g
        Iozone: Performance Test of File I/O
                Version $Revision: 3.394 $
                Compiled for 64 bit mode.
                Build: linux

        Contributors:William Norcott, Don Capps, Isom Crawford, Kirby Collins
                     Al Slater, Scott Rhine, Mike Wisner, Ken Goss
                     Steve Landherr, Brad Smith, Mark Kelly, Dr. Alain CYR,
                     Randy Dunlap, Mark Montague, Dan Million, Gavin Brebner,
                     Jean-Marc Zucconi, Jeff Blomberg, Benny Halevy, Dave Boone,
                     Erik Habbinga, Kris Strecker, Walter Wong, Joshua Root,
                     Fabrice Bacchella, Zhenghua Xue, Qin Li, Darren Sawyer.
                     Ben England.

        Run began: Sat Oct  7 23:04:05 2017

        Auto Mode
        File size set to 2097152 KB
        Command line used: ./iozone -a -s 2g
        Output is in Kbytes/sec
        Time Resolution = 0.000001 seconds.
        Processor cache size set to 1024 Kbytes.
        Processor cache line size set to 32 bytes.
        File stride size set to 17 * record size.
                                                            random  random    bkwd   record   stride
              KB  reclen   write rewrite     read   reread    read   write    read  rewrite     read   fwrite frewrite   fread  freread
         2097152       4  186684  675664  1077166  3576829 2921988  485274 1695937  5996317  3214686   366124   789927 2447393  4079077
         2097152       8  125294 1031010  3691765  4993717 4816565  272995 2258555  7281119  4347752   101911  1058324 4686321  5195419
         2097152      16  413871  886736  2775541  5847938 4183967  416529 2341198  9558354  4327434   154679   866472 2363755  5390235
         2097152      32  317784 1109858  3400619  6419803 5465860  351046 2659224  9608587  6063394   528179   904818 2919206  6388359
         2097152      64  192100 1080247  3356899  6273487 4902018  286252 3240146  8293038  4617539   188864   160494 2289407  4926859
         2097152     128  241715  106338  2688928  4469262 5451240  138752 2263865  8770694  5789737   166940   576844 2337678  4257321
         2097152     256  180224  443726  4098753  5353154 5368545   87666 2797377  7542002  5728044   512775   556446 3642323  5694525
         2097152     512  143395  817665  5044421  5751611 5263922  552229 2261253  8975964  4291753   284888   175999 2541765  5408527
         2097152    1024  105704  148701  2442555  4632134 5934349  155441 2672089  8669784  5972544   149986   565119 2120546  5723780
         2097152    2048  892955  358467  2390707  5966766 5309313 1082634 2221187  8262059  4129105   475411   146109 2673046  6020260
         2097152    4096  332708   97288  2351328  5797656 5827721  228217 1938185  8576435  3688862   137554   961691 3248834  6239240
         2097152    8192  329997  108407  2523117  5718080 5324205  576495 2061779  7937381  4067036    93781   164560 2484991  6108271
         2097152   16384  167804  272699  2410189  4198939 4974752  195364 3641685  4607294  4366483   100946   314747 4073683  4748500

iozone test complete.
[cc@hopess current]$
```

Observation and conclusion 2:

The IOZONE benchmark has a better performance than mine. We can improve our implementation, such as look for better disk access APIs, or increase thread number and block size.

Network Benchmark

- The program finds throughput in Mb per second and latency in ms
- We use socket programming and client-server protocol type to do the implementation.
- The total time difference between start and end time is calculated round trip ping-pong message transferring between client and server
- The actual operation data size is adjusted during experiment to make sure the run time is not that long
- The throughput performance is measured using formula

Throughput = total transferring data size / total time

- The latency performance is measured using formula

Latency = total time / total times of round trip (i.e. latency is the RTT per professor's instruction)

- We only tested loopback performance due to limited resource on Chameleon

System Configuration

Chameleon Cloud(openstack KVM)

Processor: Intel Xeon E312xx (Sandy Bridge) @ 3.09 GHz
1 processor, 32 cores
OS:CentOS7
RAM: 4GB
Disk: 40GB

Experiment 1~16

- TCP program is run for testing data transferring speed under TCP protocol between client and server. Data size is fixed for each thread to ensure strong scaling. We didn’t set data size too large, otherwise the running time would be long. Also it will not affect the result since throughput and latency are unit performance measurement. Num_loops in below table is the number of round trips.

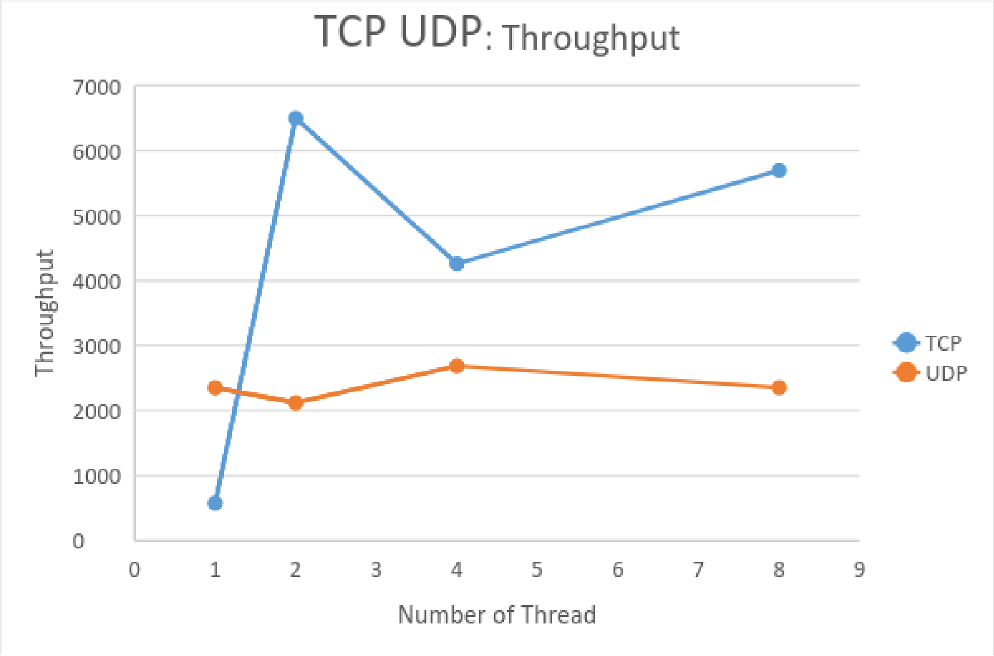
Result shows in below table and graphs.

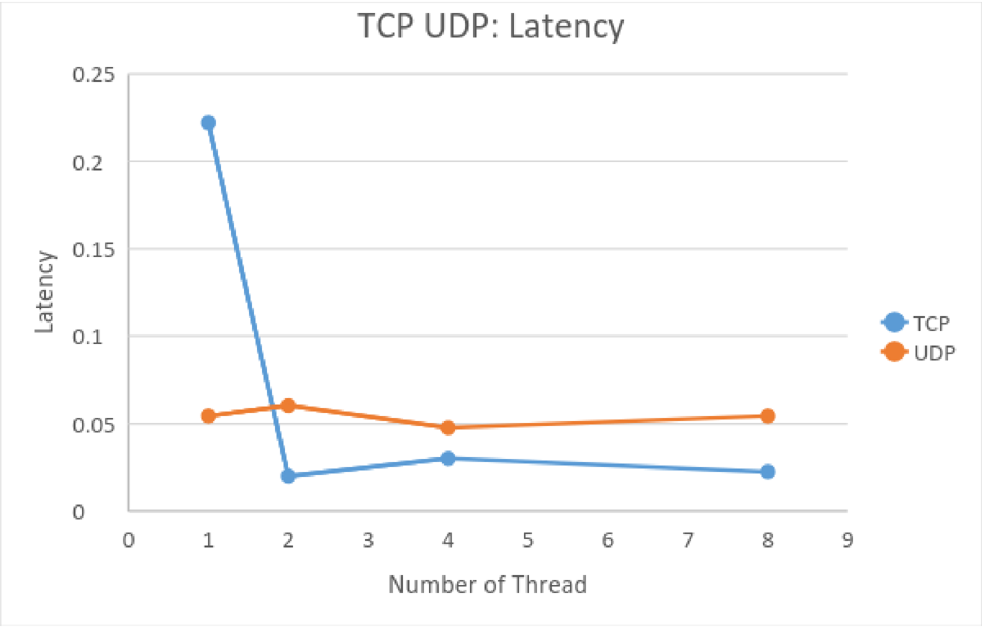
thread	throughput (Mb/sec)	latency(ms/RT)	Num_loops	datasize
1	576.492249	0.222032	200	25.6MB
2	6497.961426	0.01998	200	25.6MB
4	4259.041504	0.030054	200	25.6MB
8	5694.759277	0.02247	200	25.6MB

Table TCP loopback performance

thread	throughput (Mb/sec)	latency(ms/RTT)	Num_loops	datasize
1	2353.336182	0.054391	200	25.6MB
2	2122.10791	0.060317	200	25.6MB
4	2686.520996	0.047645	200	25.6MB
8	2354.763184	0.054358	200	25.6MB

Table UDP loopback performance



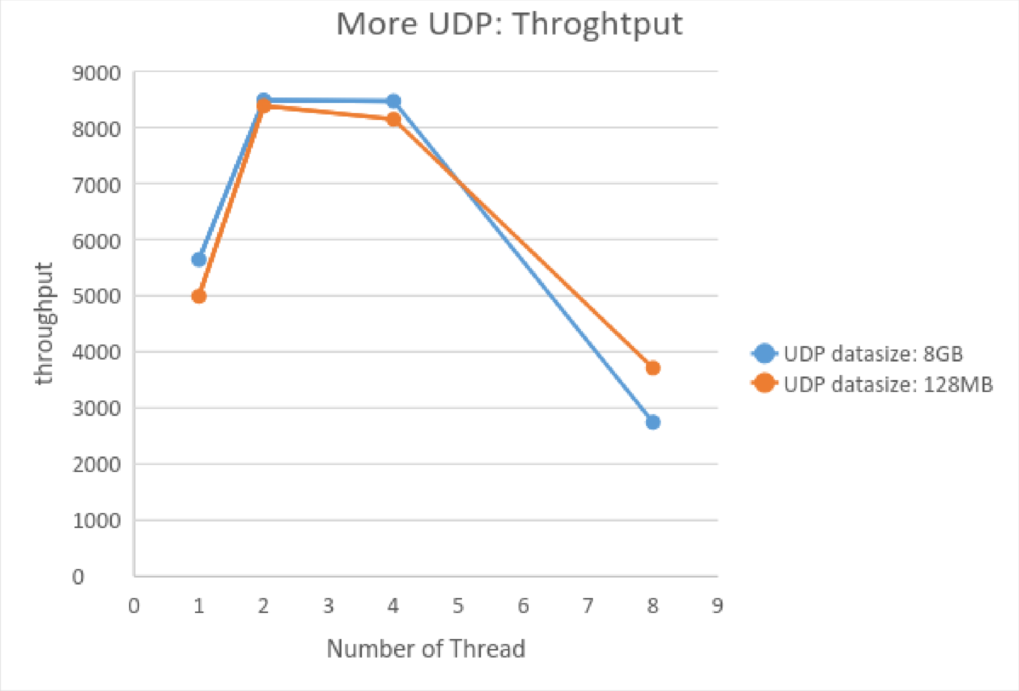


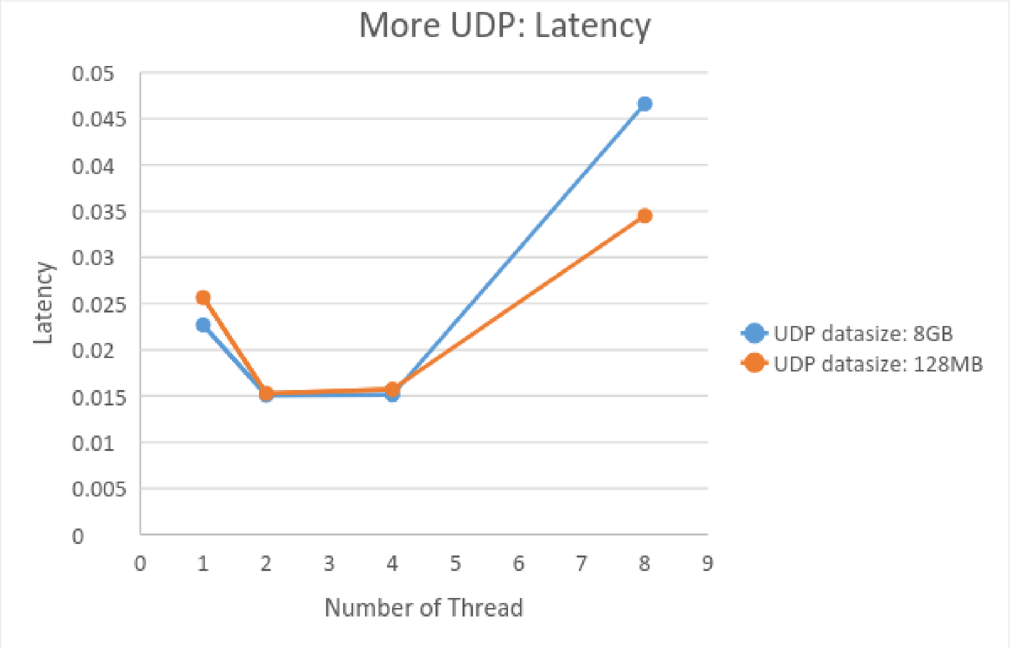
- To see if the transferring data size has effect on the performance, we increase the transferring data size for only UDP to save cloud resource. Since UDP is faster than TCP.

More results are shown below:

thread	throughput (Mb/sec)	latency(ms/RTT)	Num_loops	datasize
1	5644.057617	0.022679	62500	8GB
2	8486.951172	0.015082	62500	8GB
4	8468.522461	0.015115	62500	8GB
8	2745.610352	0.04662	62500	8GB
1	4991.004395	0.025646	1000	128MB
2	8383.00293	0.015269	1000	128MB
4	8145.578125	0.015714	1000	128MB
8	3712.01001	0.034483	1000	128MB

Table more UDP performance with larger data size





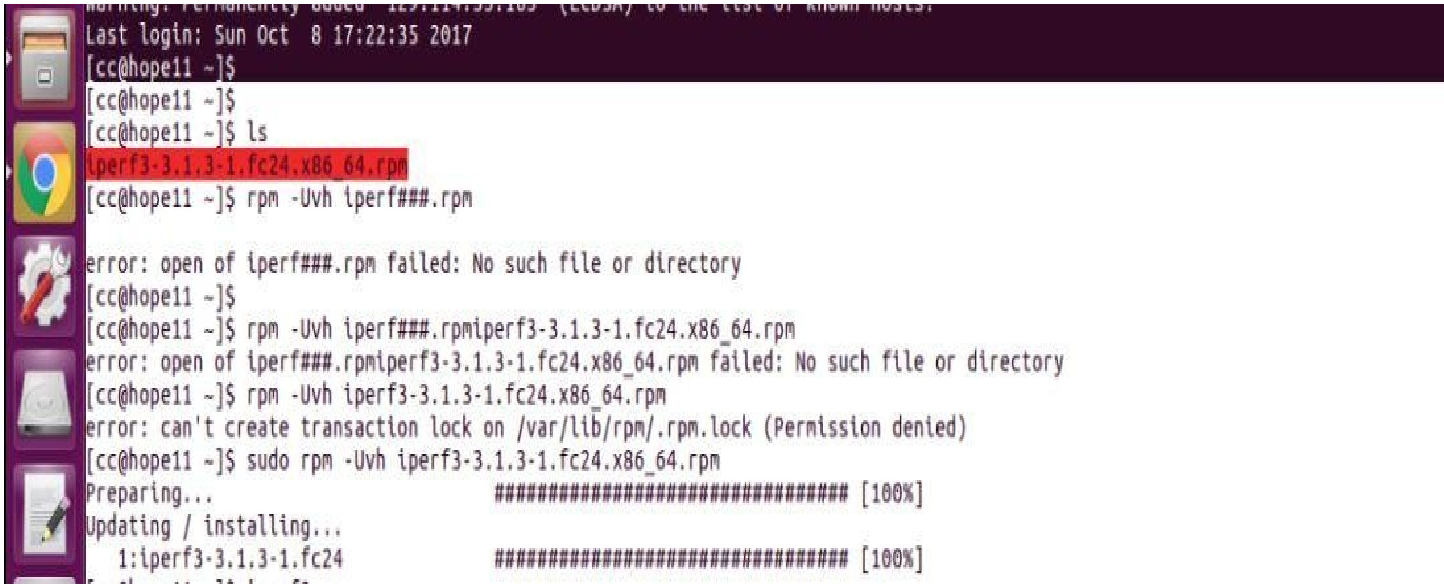
Observation and conclusion 1:

- In general, with small transfer data size, the speed of TCP and UDP does not have big difference; with larger data size, UDP is much faster than TCP (since it takes long to get a result if we using TCP to transfer 8GB data, so we only test UDP for lager data size). My explanation is because UDP's nonexistent acknowledge packet (ACK) that permits a continuous packet stream, instead of TCP that acknowledges a set of packets, calculated by using the TCP window size and round-trip time (RTT).
- In general, both TCP and UDP speed increases as threads increases. Since we use strong scaling.
  But for larger scale data transfer with UDP, larger thread number does not help transfer faster. My explaination is: That's caused by a lock contention on the UDP receive buffer side. Since both threads are using the same socket descriptor, they spend a disproportionate amount of time fighting for a lock around the UDP receive buffer.
- There is not much difference of transferring speed as data size becomes larger, since throughput and latency are both unit performance measurement.

Experiment 17

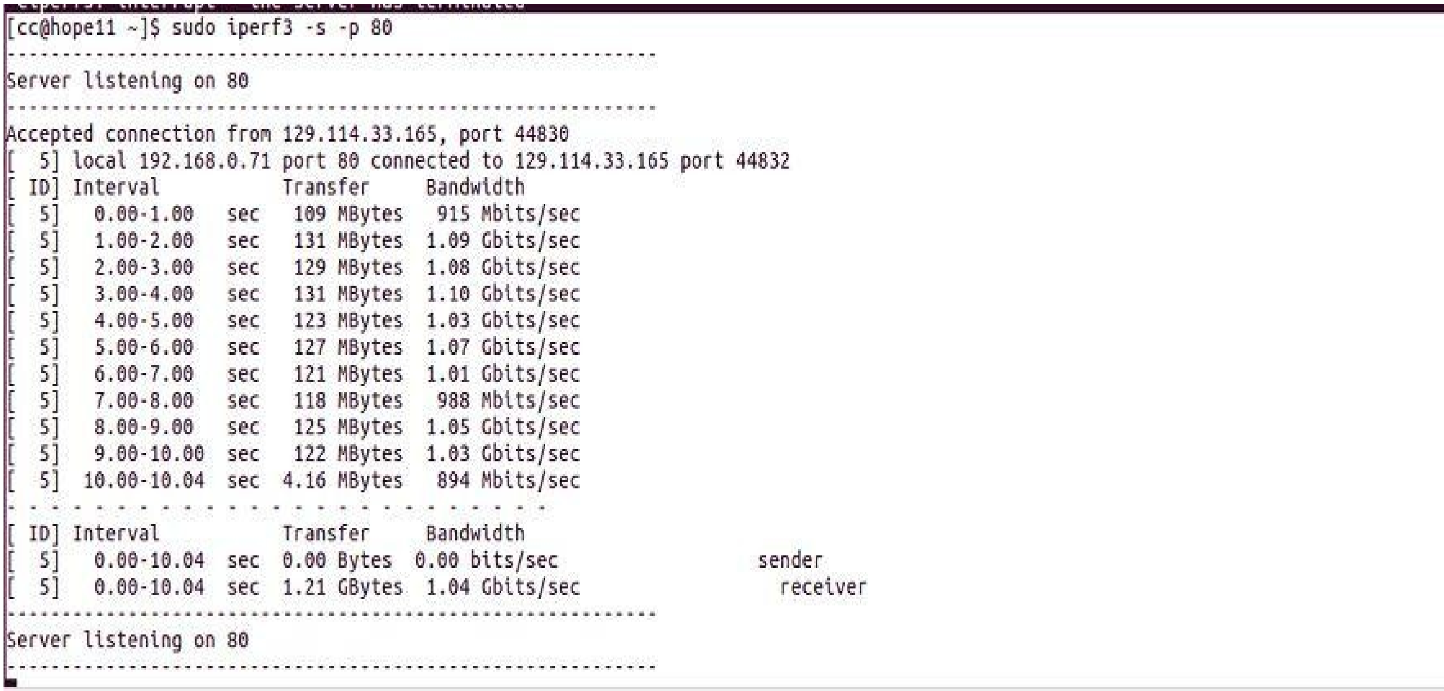
- Run iperf benchmark for loopback interface

The screen short is shown as below:

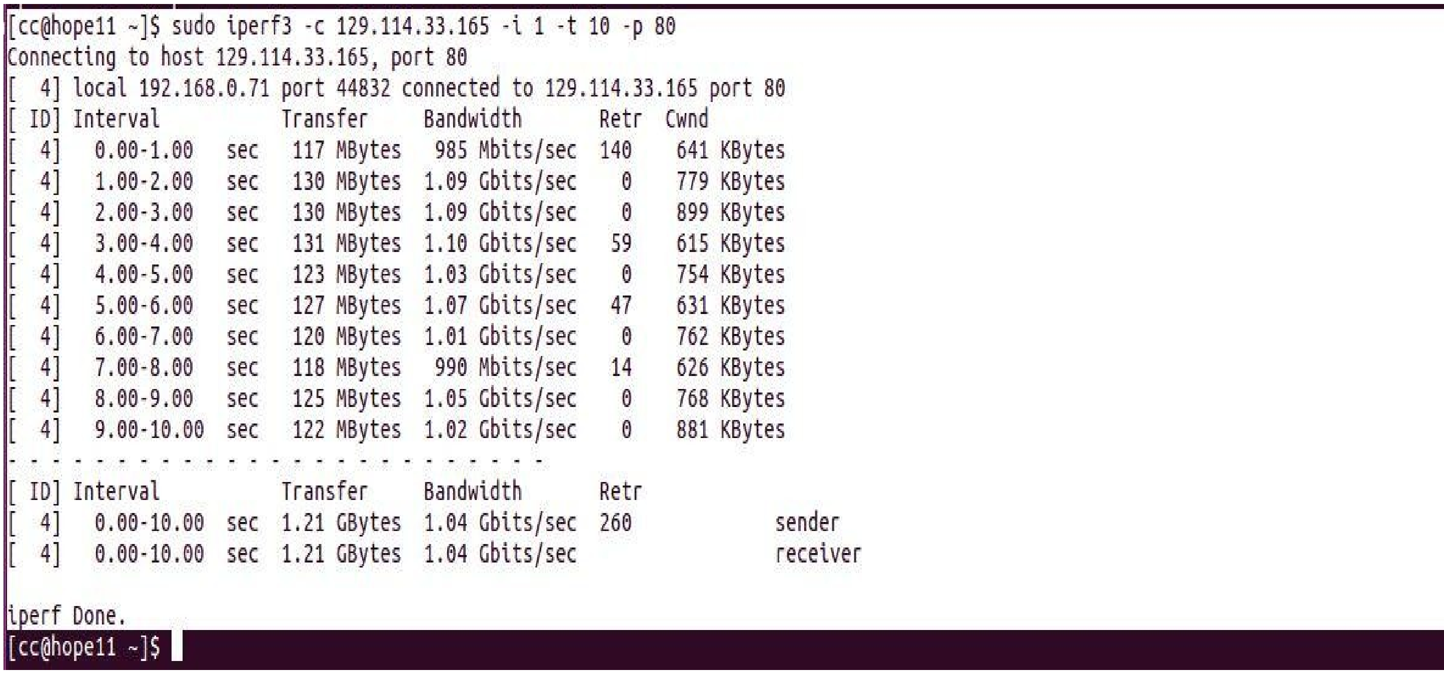


```
Warning: Permanently added '129.114.33.165' (ECDSA) to the list of known hosts.
Last login: Sun Oct  8 17:22:35 2017
[cc@hope11 ~]$
[cc@hope11 ~]$
[cc@hope11 ~]$ ls
iperf3-3.1.3-1.fc24.x86_64.rpm
[cc@hope11 ~]$ rpm -Uvh iperf###.rpm

error: open of iperf###.rpm failed: No such file or directory
[cc@hope11 ~]$
[cc@hope11 ~]$ rpm -Uvh iperf###.rpmiperf3-3.1.3-1.fc24.x86_64.rpm
error: open of iperf###.rpmiperf3-3.1.3-1.fc24.x86_64.rpm failed: No such file or directory
[cc@hope11 ~]$ rpm -Uvh iperf3-3.1.3-1.fc24.x86_64.rpm
error: can't create transaction lock on /var/lib/rpm/.rpm.lock (Permission denied)
[cc@hope11 ~]$ sudo rpm -Uvh iperf3-3.1.3-1.fc24.x86_64.rpm
Preparing...                          ################################# [100%]
Updating / installing...
   1:iperf3-3.1.3-1.fc24              ################################# [100%]
```



```
^Ciperf3: interrupt - the server has terminated
[cc@hope11 ~]$ sudo iperf3 -s -p 80
-----------------------------------------------------------
Server listening on 80
-----------------------------------------------------------
Accepted connection from 129.114.33.165, port 44830
[  5] local 192.168.0.71 port 80 connected to 129.114.33.165 port 44832
[ ID] Interval           Transfer     Bandwidth
[  5]   0.00-1.00   sec   109 MBytes   915 Mbits/sec
[  5]   1.00-2.00   sec   131 MBytes  1.09 Gbits/sec
[  5]   2.00-3.00   sec   129 MBytes  1.08 Gbits/sec
[  5]   3.00-4.00   sec   131 MBytes  1.10 Gbits/sec
[  5]   4.00-5.00   sec   123 MBytes  1.03 Gbits/sec
[  5]   5.00-6.00   sec   127 MBytes  1.07 Gbits/sec
[  5]   6.00-7.00   sec   121 MBytes  1.01 Gbits/sec
[  5]   7.00-8.00   sec   118 MBytes   988 Mbits/sec
[  5]   8.00-9.00   sec   125 MBytes  1.05 Gbits/sec
[  5]   9.00-10.00  sec   122 MBytes  1.03 Gbits/sec
[  5]  10.00-10.04  sec  4.16 MBytes   894 Mbits/sec
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth
[  5]   0.00-10.04  sec  0.00 Bytes  0.00 bits/sec                  sender
[  5]   0.00-10.04  sec  1.21 GBytes  1.04 Gbits/sec                  receiver
-----------------------------------------------------------
Server listening on 80
-----------------------------------------------------------
```



```
[cc@hope11 ~]$ sudo iperf3 -c 129.114.33.165 -i 1 -t 10 -p 80
Connecting to host 129.114.33.165, port 80
[  4] local 192.168.0.71 port 44832 connected to 129.114.33.165 port 80
[ ID] Interval           Transfer     Bandwidth       Retr  Cwnd
[  4]   0.00-1.00   sec   117 MBytes   985 Mbits/sec  140    641 KBytes
[  4]   1.00-2.00   sec   130 MBytes  1.09 Gbits/sec    0    779 KBytes
[  4]   2.00-3.00   sec   130 MBytes  1.09 Gbits/sec    0    899 KBytes
[  4]   3.00-4.00   sec   131 MBytes  1.10 Gbits/sec   59    615 KBytes
[  4]   4.00-5.00   sec   123 MBytes  1.03 Gbits/sec    0    754 KBytes
[  4]   5.00-6.00   sec   127 MBytes  1.07 Gbits/sec   47    631 KBytes
[  4]   6.00-7.00   sec   120 MBytes  1.01 Gbits/sec    0    762 KBytes
[  4]   7.00-8.00   sec   118 MBytes   990 Mbits/sec   14    626 KBytes
[  4]   8.00-9.00   sec   125 MBytes  1.05 Gbits/sec    0    768 KBytes
[  4]   9.00-10.00  sec   122 MBytes  1.02 Gbits/sec    0    881 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0.00-10.00  sec  1.21 GBytes  1.04 Gbits/sec  260             sender
[  4]   0.00-10.00  sec  1.21 GBytes  1.04 Gbits/sec                  receiver

iperf Done.
[cc@hope11 ~]$
```

Observation and Conclusion 2:

- As we can see, the iperf transfer speed is faster than my implementation. My explanation here is, for latency, we use ping utility measurement to measure the RTT. That means we measure latency for data transferred per round trip; while the iperf measures data transferred per second.
- The theoretical memory performance we got from part 2 is that Stdlib copy is 1.96GB per second, comparing with my highest testing throughput which is 8486.52 Mb/s, we get about 8486.52Mb/1.96GB=54.28% performance of the theoretical memory performance. It’s reasonable.
- The theoretical loopback is faster than 40Gbps, so the theoretical loopback latency is at least 1/(40Gb/64KB/4) = 0.0000256 seconds = 0.0256ms; Comparing this theoretical latency with my testing latency, they are pretty close. Some of mine are a little faster due to multi threads. Some of mine are a little slower due to loopback network traffic.