

Precourse 4

Installation instruction

Get system informations

To get system info from filesystem, I create a helper function to execute a command then push stream output to string buffer. This string will be used to display system info.

```
39  std::string exec(const char *cmd)
40  {
41      std::array<char, 512> buffer;
42      std::string result;
43      std::unique_ptr<FILE, decltype(&pclose)> pipe(popen(cmd, "r"), pclose);
44      if (!pipe)
45      {
46          throw std::runtime_error("popen() failed!");
47      }
48      while (fgets(buffer.data(), buffer.size(), pipe.get()) != nullptr)
49      {
50          result += buffer.data();
51      }
52      return result;
53  }
```

Then apply the `exec()` to get result

```
55  void printf_system_info()
56  {
57      std::string result_cpuMHz = exec("cat /proc/cpuinfo | grep \"cpu MHz\" | head -n1");
58      std::string result_L1d = exec("lscpu | grep \"L1d cache:\"");
59      std::string result_L2 = exec("lscpu | grep \"L2 cache:\"");
60      std::string result_L3 = exec("lscpu | grep \"L3 cache:\"");
61      std::string result_mem = exec("cat /proc/meminfo | grep MemTotal");
62      std::string result_disk = exec("df -h .");
63      std::string result_num_cpu = exec("lscpu | grep \"CPU(s):\"");
64      std::cout << std::endl;
65      std::cout << result_num_cpu;
66      std::cout << result_cpuMHz << std::endl;
67      std::cout << result_L1d;
68      std::cout << result_L2;
69      std::cout << result_L3;
70      std::cout << result_mem;
71      std::cout << "Disk usage info:" << std::endl;
72      std::cout << result_disk << std::endl;
73  }
```

The output terminal after executing `printf_system_info()` is

```

CPU(s):                2
NUMA node0 CPU(s):    0,1
cpu MHz                : 998.026

L1d cache:            128 KiB (2 instances)
L2 cache:             1 MiB (2 instances)
L3 cache:             32 MiB (2 instances)
MemTotal:             2007552 kB
Disk usage info:
Filesystem            Size  Used Avail Use% Mounted on
/dev/mapper/ubuntu--vg-ubuntu--lv 9.8G  7.2G  2.2G  78% /

```

I run this code in VM with 2 core cpu and 2GiB RAM. This VM is ubuntu x86_64 architecture virtualization, run on Mac M1 chip, so the benchmark result can be unstable.

Benchmark the cache

Follow the cache size in the above output terminal, total cache is ~ 33Mb. So for this benchmark, the buffer size is set increase from 2kb to 32mb exponentially, the stride is set from 1 to 31 with range 2 between 2 stride. To measure the bandwidth, I access the memory multiple times and use the `std::chrono` to measure execution time. The more runs step, the more stable the result is. For this benchmark, I repeat access whole buffer 1000 times.

The algorithm to access buffer with `stride` is described as follow:

```

94  void test_access_memory(int num_elements, int stride, int runs)
95  {
96      long result;
97      for (int j = 0; j < runs; j++)
98      {
99          result = 0;
100         for (int i = 0; i < num_elements; i += stride)
101             [
102                 result += data[i];          thuan, 2 days ago • add bandwidth measurement of cache
103             ]
104         volatile long temp; // volatile tell compiler not to optimize the loop
105         temp = result;
106     }
107 }

```

I define a `volatile` variable and assign result for it, this trick can tell the compiler not to optimize the loop. The function to measure the bandwidth is described as follow:

```

158 long double run_bandwidth(int size, int stride)
159 {
160     long num_elements = (long)size / sizeof(long);
161     auto start = std::chrono::high_resolution_clock::now();
162     test_access_memory(num_elements, stride, MAXRUNS * (1 << (int)(log2(MAXBYTES) - log2(size) + 1)));
163     auto elapsed = std::chrono::high_resolution_clock::now() - start;
164     long long microseconds = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
165     long double result = (long double)(size / stride);
166     result *= MAXRUNS * (1 << (int)(log2(MAXBYTES) - log2(size) + 1));
167     result /= ((long double)microseconds);
168     return result;
169 }
170

```

We have the equation to calculate the bandwidth:

Bandwidth (MB/s) \approx total bytes read (bytes) / total execution time (micro seconds)

total bytes read = (size of buffer) * (number of runs) / stride

The logic for benchmark the latency is a bit different:

```

227 long double run_latency_ns(int size, int stride)
228 {
229     long num_elements = (long)
230     auto start = std::chrono::high_resolution_clock::now();
231     test_access_memory(num_elements, stride, MAXRUNS * (1 << (int)(log2(MAXBYTES) - log2(size) + 1)));
232     auto elapsed = std::chrono::high_resolution_clock::now() - start;
233     long long microseconds = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
234
235     long double result = ((long double)microseconds * 1000) / (MAXRUNS * num_elements * (1 << (int)(log2(MAXBYTES) - log2(size) + 1)) / (long)stride);
236     return result;
237 }
238

```

We calculate the latency (nano second) by divide total execution time with total number of element accessed. **MAXRUNS** is set to 100 so number of run is set to **MAXRUNS * (1 << (int)(log2(MAXBYTES) - log2(size) + 1))** means the smaller buffer size, the more run steps, and the run step will increase exponentially. The output terminal for benchmark cache is showed as follow:

Throughput/Bandwidth with different buffer size and stride (MB/sec)																
	s1	s3	s5	s7	s9	s11	s13	s15	s17	s19	s21	s23	s25	s27	s29	s31
32m	4977	2079	2030	1786	1513	1358	1262	1123	921	752	702	787	707	855	969	898
16m	6334	2859	2566	2405	2163	2022	1823	1654	1509	1235	1216	1164	1148	1192	1277	1206
8m	6474	3146	3019	2815	2608	2498	2469	2282	2343	2369	1774	2346	2291	2665	2900	2483
4m	6813	3268	3045	2956	2817	2792	2737	2658	2796	2757	2521	2759	2694	2797	2884	2682
2m	6823	3468	3365	3236	3080	2989	2941	2847	3001	2999	2654	2974	2883	3010	3120	2883
1m	6850	3481	3363	3224	2933	2668	2626	2589	2679	2629	2433	2630	2600	2676	2730	2570
512k	6817	3142	3075	2947	2873	2774	2751	2684	2803	2769	2508	2752	2682	2778	2921	2757
256k	6849	3498	3363	3188	3097	3022	2961	2900	3125	2918	2659	3144	3247	3416	3590	3464
128k	6822	3541	3520	3401	3295	3429	3336	3634	3633	3640	3616	3654	3551	3620	3641	3598
64k	6955	3656	3637	3646	3636	3648	3629	3622	3607	3611	3613	3607	3594	3585	3605	3552
32k	6935	3650	3639	3624	3592	3612	3588	3583	3594	3562	3556	3530	3545	3508	3491	3494
16k	6935	3645	3585	3600	3572	3539	3544	3464	3485	3479	3447	3416	3421	3382	3376	3326
8k	6845	3598	3565	3521	3475	3418	3406	3343	3306	3295	3523	3415	3493	3491	3378	3365
4k	6676	3539	3439	3363	3319	3479	3435	3380	3343	3375	3298	3243	3231	3225	3167	3107
2k	6623	3405	3226	3436	3355	3275	3243	3089	3030	3034	2934	2869	2817	2826	2833	2703

Latency with different buffer size and stride (nano seconds)																
	s1	s3	s5	s7	s9	s11	s13	s15	s17	s19	s21	s23	s25	s27	s29	s31
32m	1.6328	4.2550	5.2782	5.9175	5.0495	5.7120	6.2756	6.9670	8.5176	10.3596	11.3118	10.0748	10.5561	8.4651	8.0694	8.7862
16m	1.2425	2.7295	3.0672	3.2870	3.6749	3.9331	4.2665	4.6713	5.2348	6.5429	6.4587	6.8348	6.8971	6.4949	5.8581	6.4509
8m	1.2075	2.4977	2.6415	2.7804	2.9417	3.1293	3.1383	3.3450	3.4128	3.3184	4.5039	3.4614	3.5424	2.9746	2.7245	3.2506
4m	1.1698	2.3416	2.3926	2.4894	2.7789	2.8804	2.9099	2.9826	2.8758	2.8999	3.1606	2.8985	2.9404	3.3648	2.7379	2.9281
2m	1.1681	2.3160	2.3794	2.4858	2.5970	2.6806	2.7046	2.8035	2.6476	2.6860	3.0060	2.6979	2.7701	2.6473	2.6381	2.7963
1m	1.1657	2.3233	2.7813	2.8248	2.7657	2.8456	2.8444	2.8291	2.8321	2.9096	3.1676	2.8399	2.9314	2.9423	2.7503	3.0212
512k	1.1683	2.4983	2.5905	2.5803	2.7516	2.7136	2.7773	2.8657	2.8111	2.8370	3.0928	2.8517	2.9292	2.8407	2.6773	2.9373
256k	1.1690	2.2819	2.3775	2.4882	2.5849	2.6476	2.7026	2.7576	2.6559	2.6075	3.0239	2.7260	2.4832	2.3498	2.2383	2.3326
128k	1.1742	2.2978	2.2915	2.3347	2.4068	2.3483	2.4310	2.2025	2.2022	2.2028	2.2153	2.1981	2.2035	2.2397	2.2083	2.2102
64k	1.1504	2.1957	2.2235	2.2113	2.2193	2.2065	2.2213	2.2109	2.2376	2.2210	2.2427	2.2271	2.2696	2.2402	2.2372	2.2613
32k	1.1569	2.2161	2.2152	2.2150	2.2336	2.2310	2.2484	2.2514	2.2400	2.2587	2.2641	2.2828	2.2828	2.2844	2.2926	2.3064
16k	1.1539	2.2064	2.2229	2.2371	2.2534	2.2698	2.2782	2.3142	2.3175	2.3179	2.3511	2.3657	2.3500	2.3801	2.3857	2.4151
8k	1.1745	2.2506	2.2645	2.3055	2.3187	2.3525	2.3854	2.4165	2.4454	2.4621	2.3076	2.3396	2.3057	2.3253	2.4557	2.4018
4k	1.2027	2.2670	2.3209	2.3929	2.4862	2.3077	2.3223	2.3975	2.4090	2.3602	2.4332	2.4600	2.4573	2.4423	2.5219	2.5381
2k	1.2093	2.3482	2.4645	2.3272	2.3813	2.4491	2.4655	2.5786	2.6351	2.6283	2.7593	2.8308	2.8145	2.8249	2.8807	2.9886

Benchmark the DRAM

To benchmark DRAM, I use the same logic with benchmark cache, but increase the buffer size from 64Mb to 512 Mb, So that the array have to store in main memory. To test DRAM benchmark, I use multi thread to run both 2 cpu to access memory at the same time, and because I only read the array to get elements without changing them, So I don't need to lock array when a thread access it.

```

170 long double run_multithread_bandwidth(int size, int stride)
171 {
172     long num_elements = (long)size / sizeof(long);
173     auto start = std::chrono::high_resolution_clock::now();
174     std::thread th1(test_access_memory, num_elements, stride, (int)MAXRUNS / 2);
175     std::thread th2(test_access_memory, num_elements, stride, (int)MAXRUNS / 2);
176     th1.join();
177     th2.join();
178     auto elapsed = std::chrono::high_resolution_clock::now() - start;
179     long long microseconds = std::chrono::duration_cast<std::chrono::microseconds>(elapsed).count();
180     long double result = (long double)(size / stride);
181     result *= MAXRUNS;
182     result /= ((long double)microseconds);
183     return result;
184 }

```

I create 2 thread to use 2 cpu, and wait until both 2 threads finish their jobs to measure execution time.

The output result in terminal is

Throughput/Bandwidth of DRAM with different buffer size and stride (MB/sec)																
	s1	s3	s5	s7	s9	s11	s13	s15	s17	s19	s21	s23	s25	s27	s29	s31
512m	5620	2535	2270	1956	1764	1618	1489	1377	802	553	726	540	530	202	216	181
256m	6017	2426	2075	1721	1513	1355	1207	1102	768	324	610	260	513	318	235	243
128m	5903	2329	1893	1449	1252	1093	959	875	734	548	585	537	335	231	213	235
64m	5580	2221	1688	1326	1104	939	850	757	722	284	251	236	236	234	224	227

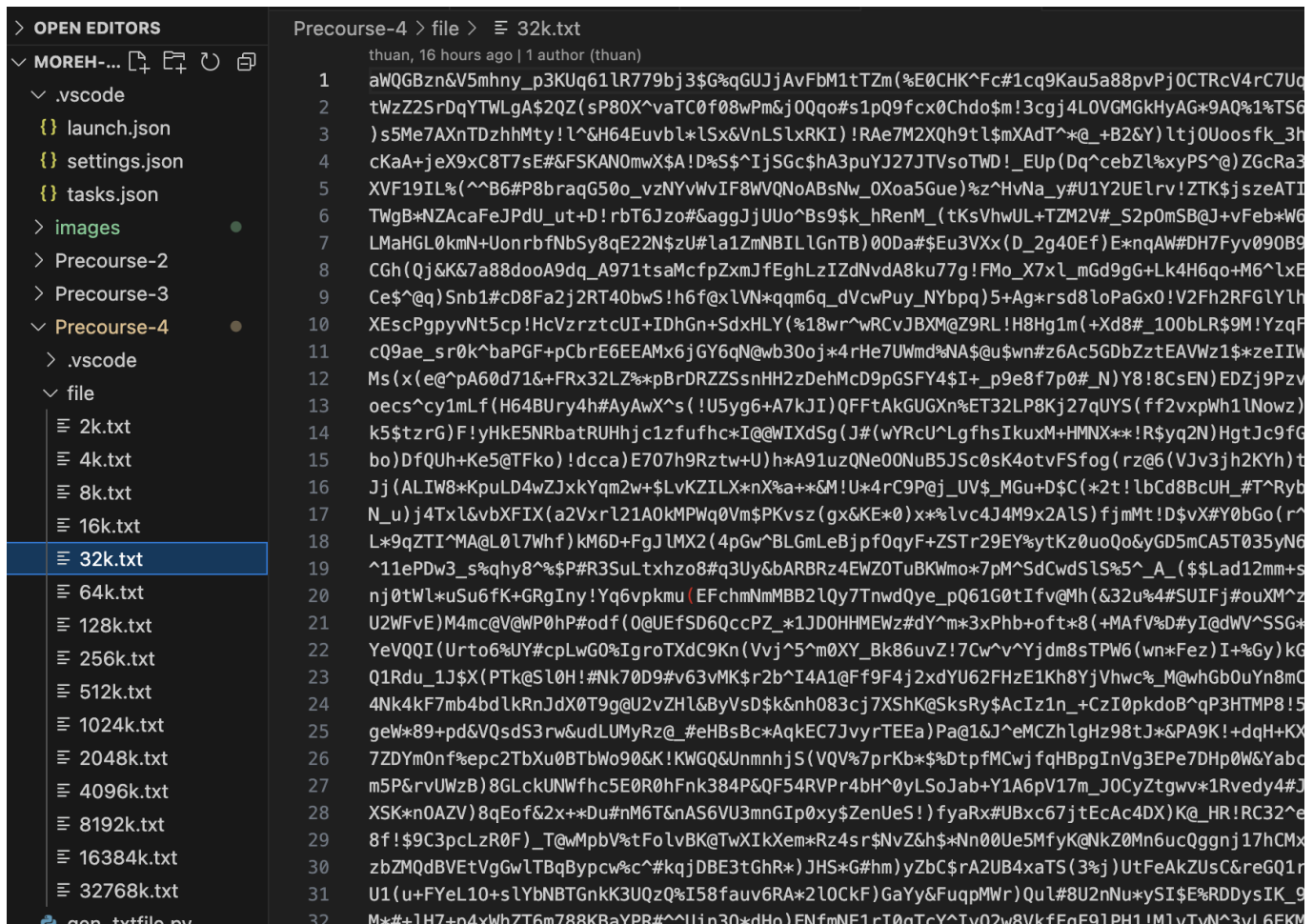
Latency of DRAM with different buffer size and stride (nano seconds)																
	s1	s3	s5	s7	s9	s11	s13	s15	s17	s19	s21	s23	s25	s27	s29	s31
512m	1.323	3.144	3.530	4.088	4.496	4.858	5.323	5.768	10.065	14.548	11.209	24.394	15.928	36.732	52.049	47.565
256m	1.320	3.260	3.806	4.609	5.350	5.974	6.578	7.306	10.628	32.585	12.348	17.526	28.362	33.009	55.519	38.338
128m	1.390	3.467	4.375	5.366	6.447	7.439	8.163	9.106	10.767	17.741	21.602	28.054	37.522	15.851	41.446	38.919
64m	1.438	3.691	4.741	6.109	7.244	8.406	9.442	10.285	11.160	14.291	14.479	14.592	30.791	35.467	35.146	35.086

Disk benchmark

In this section I will benchmark by read file with different size from 2kb to 32 Mb, run the test multiple times and then get the average result. The file is text format contains random latin characters.

```
[thuan@ubuntu:~/moreh-precources/Precourse-4$ ls -lh --sort=size file/
total 64M
-rw-rw-r-- 1 thuan thuan 32M Dec 29 07:50 32768k.txt
-rw-rw-r-- 1 thuan thuan 16M Dec 29 07:45 16384k.txt
-rw-rw-r-- 1 thuan thuan 8.0M Dec 29 07:44 8192k.txt
-rw-rw-r-- 1 thuan thuan 4.0M Dec 29 07:44 4096k.txt
-rw-rw-r-- 1 thuan thuan 2.0M Dec 29 07:43 2048k.txt
-rw-rw-r-- 1 thuan thuan 1.0M Dec 29 07:43 1024k.txt
-rw-rw-r-- 1 thuan thuan 512K Dec 29 07:43 512k.txt
-rw-rw-r-- 1 thuan thuan 256K Dec 29 07:43 256k.txt
-rw-rw-r-- 1 thuan thuan 128K Dec 29 07:43 128k.txt
-rw-rw-r-- 1 thuan thuan 64K Dec 29 07:43 64k.txt
-rw-rw-r-- 1 thuan thuan 32K Dec 29 07:43 32k.txt
-rw-rw-r-- 1 thuan thuan 16K Dec 29 07:43 16k.txt
-rw-rw-r-- 1 thuan thuan 8.0K Dec 29 07:43 8k.txt
-rw-rw-r-- 1 thuan thuan 4.0K Dec 29 07:43 4k.txt
-rw-rw-r-- 1 thuan thuan 2.0K Dec 29 07:43 2k.txt
```

This is an example content of 1 text file



```
1  aWQGBzn&V5mhny_p3KUq61lR779bj3$G%qGUJjAvFbM1tTZm(%E0CHK^Fc#1cq9Kau5a88pvPj0CTrcV4rC7Uq
2  tWzZ2SrDqYTWLgA$2QZ(sP80X^vaTC0f08wPm&j0Qqo#s1pQ9fcx0Chdo$m!3cgj4LOVGMGkHyAG*9AQ%1%TS6
3  )s5Me7AXnTDzhMty!l^&H64Euvbl*LSx&VnLSlXRKI)!RAe7M2XQh9t!l$mXAdT^*+_B2&Y)ltj0Uoosfk_3H
4  cKaA+jeX9xC8T7sE&FSKAN0mwX$A!D%$^IjSGc$hA3puYJ27JTVsoTWD!_Eup(Dq^cebZl%xySP^@)ZGcRa3
5  XVf19IL%(^B6#P8braqG500_vzNYvWvIF8WVQNoABsNw_0Xoa5Gue)%z^HvNa_y#U1Y2UElrv!ZTK$jszeATI
6  TWgB*NZAcaFeJpDU_ut+D!rbT6Jzo#&aggJjUuo^Bs9$k_hRenM_(tKsVhwUL+TZM2V#_S2p0mSB@J+vFeb*W6
7  LMaHGL0kmN+UonrbfNbSy8qE22N$zU#1a1ZmNBILlGnTB)00Da#$Eu3VXx(D_2g40Ef)E*nqAW#DH7Fyv090B9
8  CGh(Qj&K&7a88dooA9dq_A971tsaMcfpZxmJfEghLzIZdNvdA8ku77g!FMo_X7xl_mGd9gG+Lk4H6qo+M6^lxE
9  Ce$^@q)Snb1#cd8Fa2j2RT40bwS!h6f@xLVN*xqm6q_dVcwPuy_NYbpq)5+Ag*rsd8loPaGx0!V2Fh2RF6LYlh
10 XEscPgpyvNt5cp!HcVzrztcUI+IdhGn+SdxHLY(%18wr^wRCvJBXM@Z9RL!H8Hg1m(+Xd8#_100bLR$9M!YzqF
11 cQ9ae_s r0k^baPGF+pCbrE6EEAMx6jGY6qN@wb30oj*4rHe7UWmd%NA$@u$wn#z6Ac5GDbZztEAVWz1$*zeIIW
12 Ms(x(e^pA60d71&+FRx32LZ*#pBrDRZSSnHH2zDehMcD9pGSFY4$I+_p9e8f7p0#_N)Y8!8CsEn)EDZj9Pzv
13 oecs^cy1mLf(H64Bury4h#AyAwX^s(!U5yg6+A7kJI)QFFtAkGUGXn#ET32LP8Kj27qUYS(ff2vxpWh1lNowz)
14 k5$ztzrG)F!yHkE5NRbatRUHhjczfufhc*I@wIXdSg(J#(wYRcU^LgfhSIkuxM+HMNX**!R$yq2N)HgtJc9fG
15 bo)DfQUh+Ke5@TFko)!dcca)E707h9Rztw+U)h*A91uzQNe00NuB5JSc0sK4otvFSfog(rz@6(VJv3jh2KYh)t
16 Jj(ALIW8*KpuLD4wZJxkYqm2w+$LvKZILX*nX%a+*M!U*4rC9P@j_UV$MGU+d$C(*2t!lbCd8BcUH_#T^Ryb
17 N_u)j4Tx1&vbfXf(a2Vxrl21A0kMPWq0Vm$PKvsz(gx&KE*0)x*$lvc4J4M9x2A1S)fjmMt!D$vX#Y0bGo(r^
18 L*9qZTI^MA@L017Whf)kM6D+FgJlMX2(4pGw^BLGmLeBjpf0qyF+ZSTr29EY$ytKz0uoQo&yGD5mCA5T035yN6
19 ^11ePDw3_s%qhy8^%P#R3SuLtxhzo8#q3Uy&bARBRz4EWZ0TuBKWmo*7pM^SdCwdSL5%5^_A_($$Lad12mm+s
20 nj0tWl*usufK+GRgIny!Yq6vpmu(EFchmNmMBB2lQy7TnwdQye_pQ61G0tIfv@Mh(&32u%4#SUIFj#ouXM^z
21 U2WFvE)M4mc@V@P0hP#odf(0@UEfSD6QccPZ_*1JD0HMEWz#dY^m*3xPhb+oft*8(+MAfV#d#yI@dWV^SSG*
22 YeVQQI(Urto6%UY#cpLWg0%IgroTXdC9Kn(Vvj^5^m0XY_Bk86uvZ!7Cw^v^Yjdm8sTPW6(wn*Fez)I+%Gy)kG
23 Q1Rdu_1J$X(PTk@SL0H!#Nk70D9#v63vMK$r2b^I4A1@Ff9F4j2xdYU62FHZE1Kh8YjVhwC*_M@whGb0uYn8mC
24 4Nk4kF7mb4bdLkRnJdX0T9g@U2vZHL&ByVsD$k&nh083cj7XShK@SksRy$AcIz1n_+CzI0pkdoB^qP3HTMP8!5
25 geW*89+pd&VQsdS3rw&udLUMyRz@_#eHBsBc*AqkEC7JvyrTEEA)Pa@1&J^eMCZhlgH98tJ*&PA9K!+dqH+KX
26 7ZDYmOnf%epc2TbXu0BTbWo90&K!KWGQ&UnmnhjS(VQV%7prKb*$%DtpfMCwjfqHBpgInvg3EPe7DHP0W&Yabc
27 m5P&rvUwzB)8GLckUNWfhc5E0R0hFnk384P&QF54RVP4bH^0yLS0Jab+Y1A6pV17m_J0CyZtgwv*1Rvedy4#J
28 XSK*n0AZV)8qEof&2x+*Du#nM6T&nAS6VU3mnGIp0xy$ZenUeS!)fyaRx#UBxc67jtEcAc4DX)K@_HR!RC32^e
29 8f!$9C3pclzR0F)_T@wMpbv%tFoLvBK@TwXIkXem*Rz4sr$NvZ&h$*Nn0Ue5MfyK@NkZ0Mn6ucQgggj17hCMx
30 zbZMQdBVEtVgGwLTbQBypcw%c^#kqjDBE3tGhR*)JHS*G#hm)yZbC$rA2UB4xaTS(3%j)UtFeAkZUSC&reGQ1r
31 U1(u+FYeL10+sLYbNBTGnkK3UQz0%I58fauv6RA*2l0ckF)GaYy&FuqpMwR)Qul#8U2nNu*ySI$E%RDDysIK_9
32 M*#+lH7+p4xWhZT6m788KBaYPR#^Uin30*dHo)ENfmNE1rI0oTcY^TvQ2w8VkfEqF9lPH1lMlvTvW^3vl6EKW
```

The bandwidth is calculated by $\text{total file size read divide/total execution time}$, this is the result of output terminal

Throughput and latency read of Disk with different file size		
	bwidth(MB/s)	latency(ms)
32m	474	71.297
16m	1795	9.354
8m	2080	4.080
4m	2069	2.047
2m	2134	0.975
1m	2053	0.500
512k	1898	0.270
256k	1709	0.148
128k	1349	0.089
64k	957	0.063
32k	671	0.045
16k	410	0.036
8k	218	0.037
4k	100	0.035
2k	56	0.033