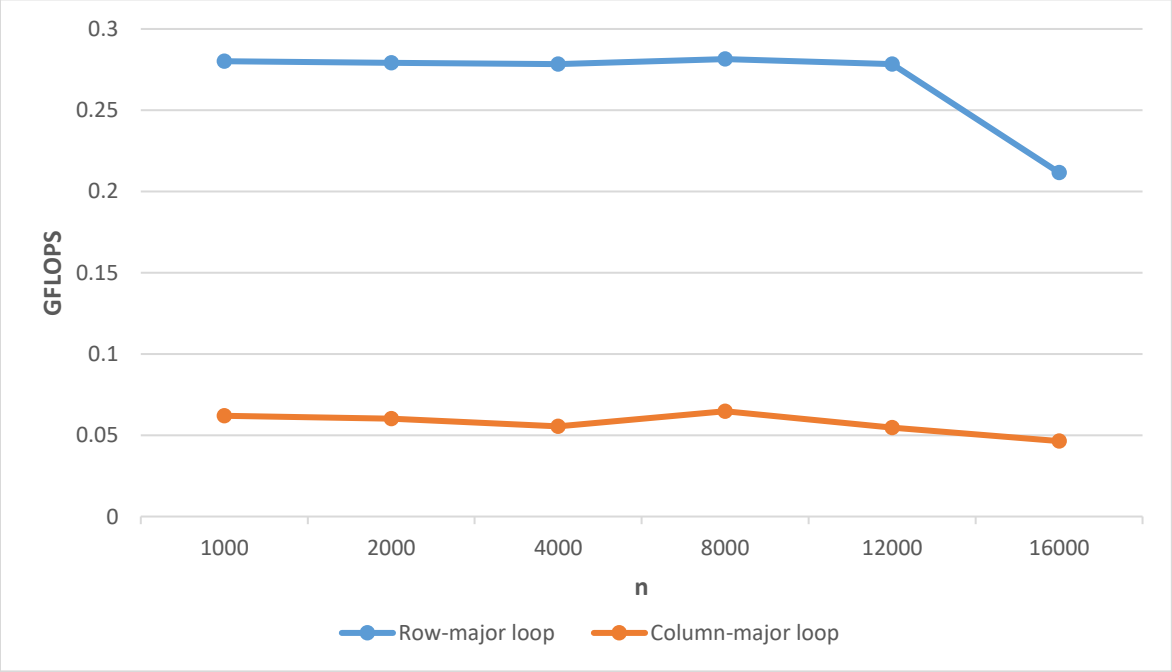
(a) run the program for a range of values of n. Does the performance depends on n for row-major and column-major implementation of matrix-vector multiplication loops? How the two implementations compare to each other and to the 2.6 GFLOPS limit?

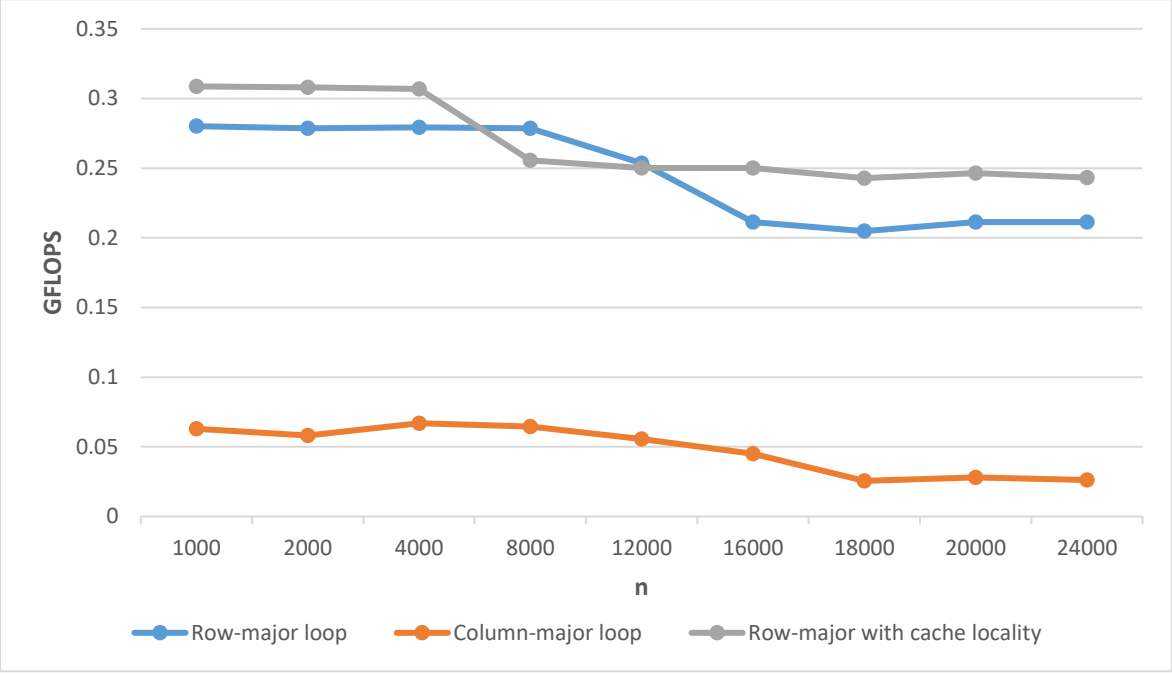
n	Row-major loop	Column-major loop
1000	0.280149	0.062056
2000	0.279252	0.060175
4000	0.278285	0.055475
8000	0.281461	0.064805
12000	0.278342	0.054748
16000	0.211640	0.046414



The performance does depend on n for both cases and in both cases as the number of operations increases the performance deteriorates. The row major loop outperforms the column major loop as expected from C and C++ programing language but neither one is even close to the limit of 2.6 GFLOPS.

(b) apply Tip 1 and see whether it improves the performance of the matvec program for the range of values of n. Please follow the theme of measuring the time, printing result, and checking whether the b vector matches the reference. To ensure consistence of your program, it is a good idea to copy a chunk of the code (e.g. the row-major implementation), paste it below, and modify accordingly.

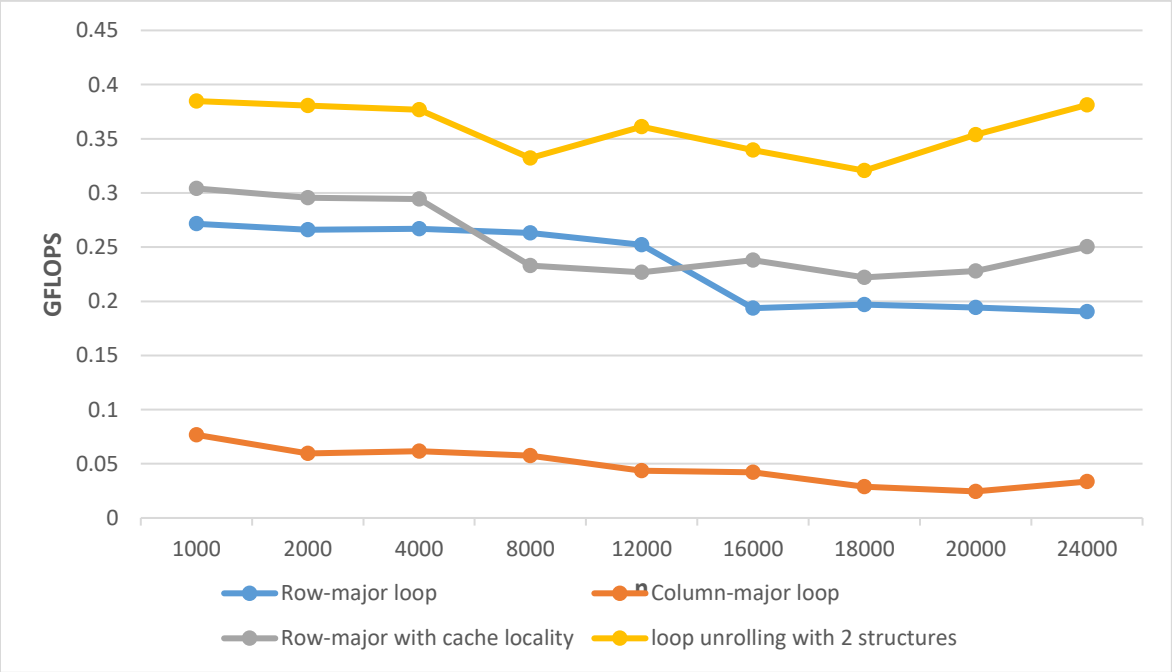
n	Row-major loop	Column-major loop	Row-major with cache locality
1000	0.280163	0.062818	0.308694
2000	0.278591	0.058176	0.307998
4000	0.279271	0.066925	0.306788
8000	0.278614	0.064504	0.255735
12000	0.253573	0.055678	0.250248
16000	0.211409	0.044921	0.250145
18000	0.204791	0.025559	0.242855
20000	0.211388	0.027922	0.246475
24000	0.211224	0.02613	0.243179



Row major with cache locality improves the GFLOPS at early stages but as the number of computations increased the GFLOPS decreased. However, overall the cache locality improves the speed of computation.

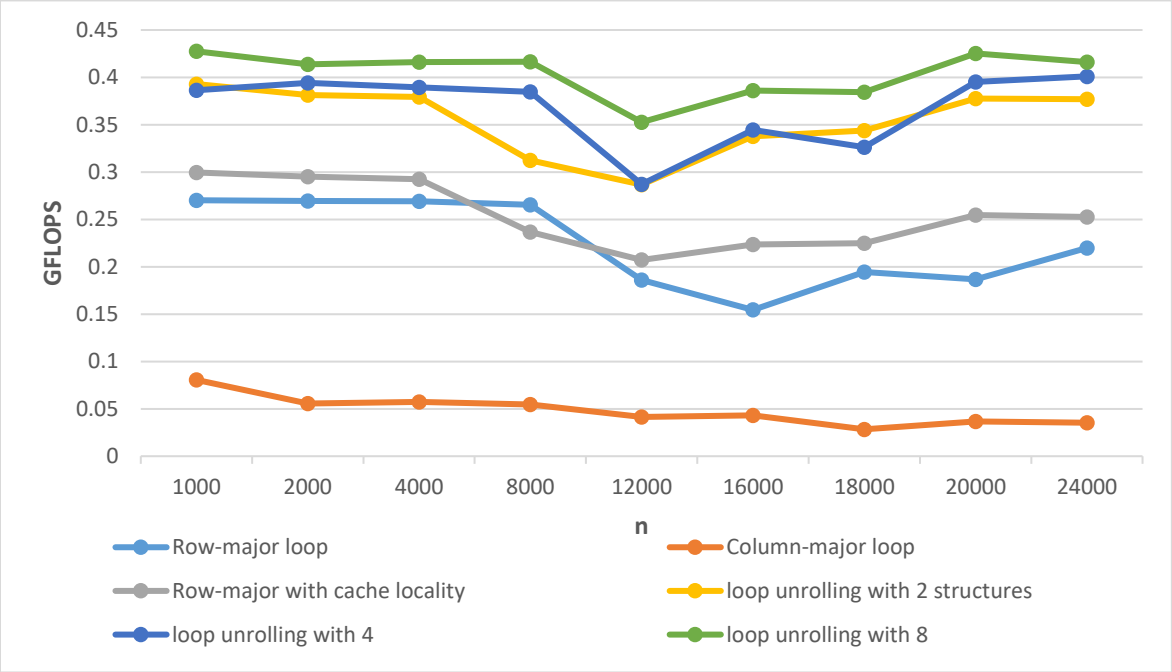
(c) apply the loop unrolling technique to the row-major algorithm (see Tip 2), and check whether it provides any improvement in terms of performance. What happens when you unroll with more than two independent instructions? Is there a limit to the number of independent instructions, after which there is no performance gain?

n	Row-major loop	Column-major loop	Row-major with cache locality	loop unrolling with 2 structures
1000	0.271493	0.076722	0.304119	0.384772
2000	0.265871	0.059647	0.295524	0.380470
4000	0.266874	0.061487	0.294395	0.376766
8000	0.262971	0.057615	0.232962	0.332129
12000	0.252187	0.043711	0.226725	0.361224
16000	0.193546	0.042137	0.237833	0.339652
18000	0.196981	0.028991	0.222054	0.320547
20000	0.194118	0.024507	0.227905	0.353730
24000	0.190525	0.033638	0.250247	0.381248



Loop unrolling with 2 structures does effectively increases GFLOPS.

n	Row-major loop	Column-major loop	Row-major with cache locality	loop unrolling with 2 structures	loop unrolling with 4 structures	loop unrolling with 8 structures
1000	0.270218	0.080588	0.299565	0.392933	0.386453	0.427487
2000	0.269478	0.055803	0.295058	0.381264	0.39399	0.413784
4000	0.269071	0.057207	0.292626	0.379255	0.38957	0.416257
8000	0.265452	0.054534	0.236665	0.312452	0.38457	0.416598
12000	0.185945	0.041565	0.207182	0.286726	0.287135	0.352649
16000	0.15446	0.04312	0.223423	0.337646	0.344525	0.386032
18000	0.194429	0.028417	0.225013	0.343844	0.326307	0.384417
20000	0.186862	0.036671	0.254568	0.377641	0.395208	0.425404
24000	0.219968	0.035332	0.252656	0.376903	0.401025	0.416048

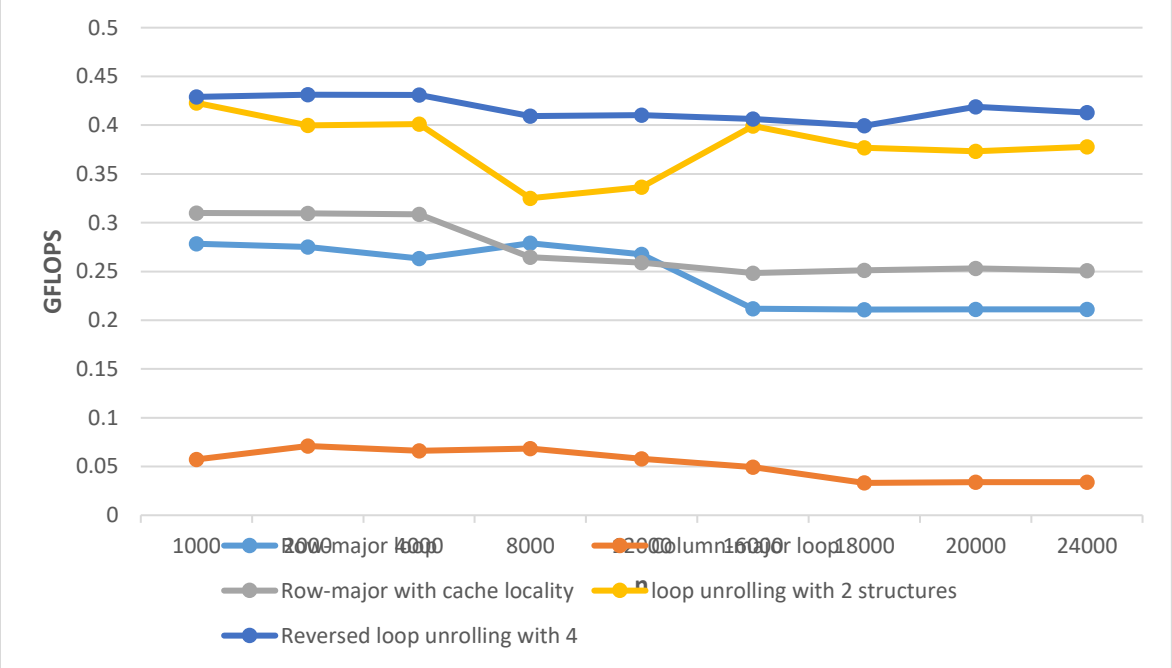


(d) apply the cache blocking technique described in Tip 3, and again see whether there is performance boost. Experiment with different block sizes. You can assume that R1 processor has 32KB of L1 cache, the cache line size is 64B, and to compute a single dot product you need a row of matrix A, and vector x. The result

vector b is written to memory only after the dot product is computed, so it does not need to be in cache.

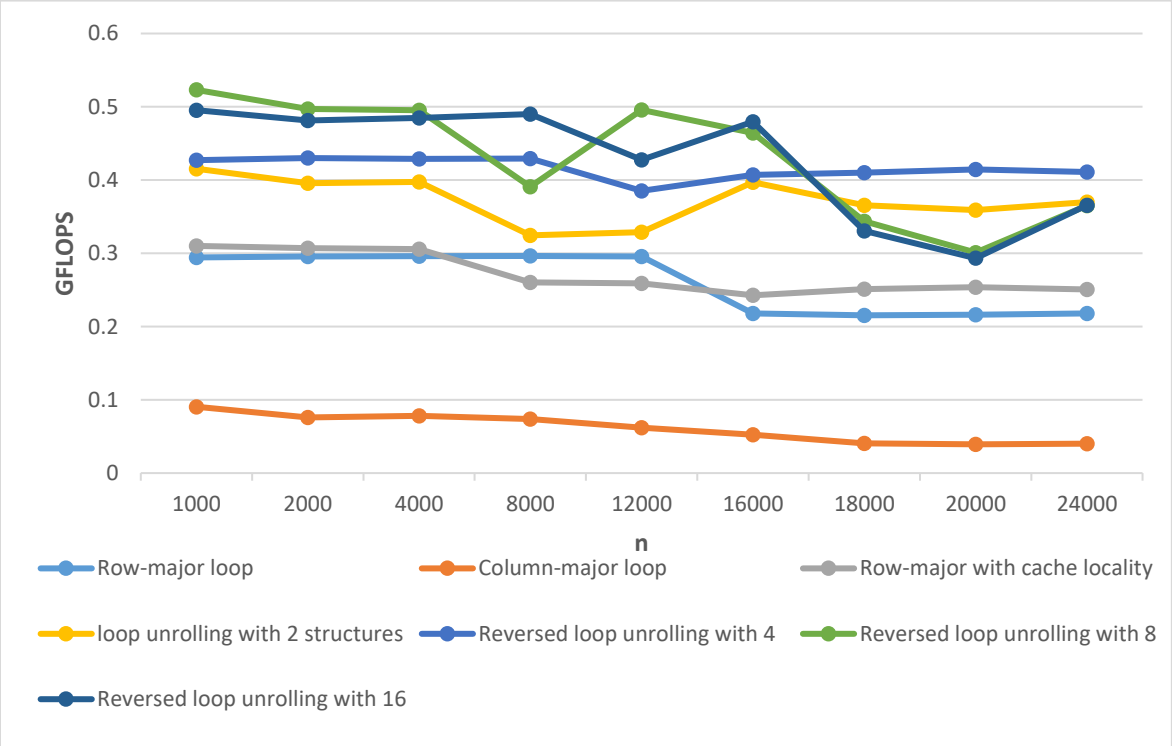
Doing it in reverse order:

n	Row-major loop	Column-major loop	Row-major with cache locality	loop unrolling with 2 structures	Reversed loop unrolling with 2 structures
1000	0.278347	0.057248	0.309966	0.422871	0.429
2000	0.275199	0.071019	0.309438	0.399742	0.431238
4000	0.263354	0.06596	0.30864	0.401297	0.431064
8000	0.278924	0.068536	0.26454	0.325118	0.409204
12000	0.267595	0.057924	0.259071	0.336606	0.410177
16000	0.211677	0.049218	0.248381	0.399125	0.406357
18000	0.21094	0.033163	0.251091	0.377016	0.399364
20000	0.211062	0.034044	0.253013	0.373271	0.418863
24000	0.211262	0.033791	0.25081	0.377764	0.412841



The reversed loop unrolling with 2 structures improves the GFLOPS but is still close to the normal loop unrolling with 2 structures

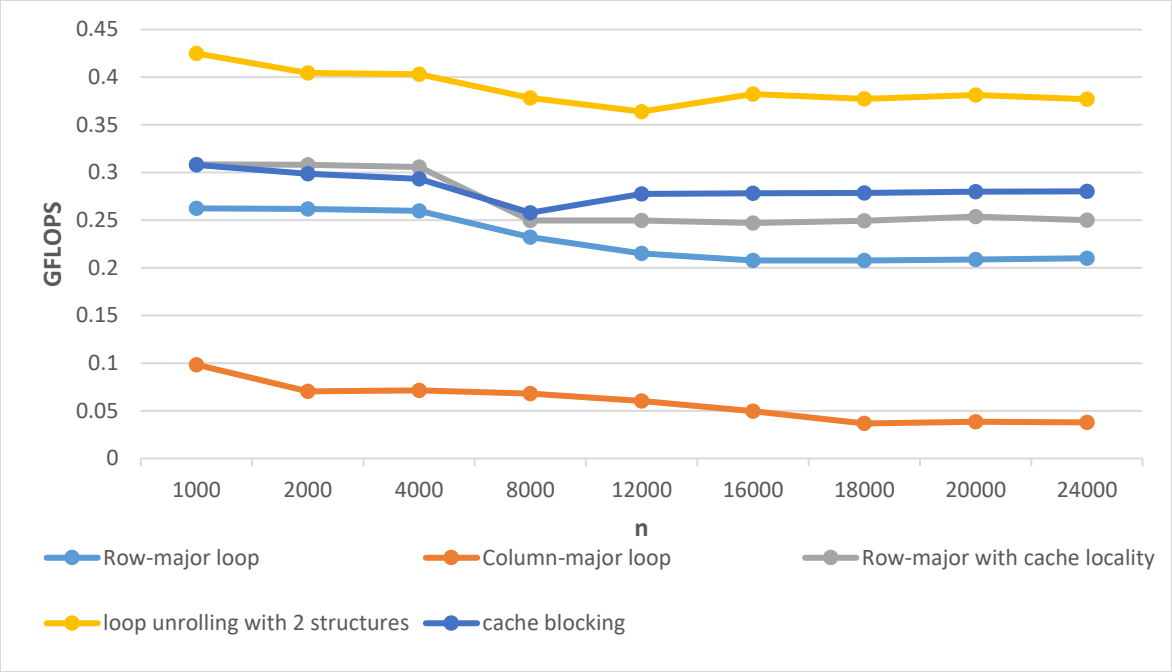
n	Row-major loop	Column-major loop	Row-major with cache locality	loop unrolling with 2 structures	Reversed loop unrolling with 2 structures	Reversed loop unrolling with 8 structures	Reversed loop unrolling with 16 structures
1000	0.294325	0.090543	0.3101	0.415426	0.427342	0.523056	0.495376
2000	0.295848	0.07604	0.307077	0.395664	0.430026	0.496907	0.481449
4000	0.296	0.078115	0.305757	0.397494	0.428736	0.495348	0.484739
8000	0.296359	0.073856	0.260282	0.324555	0.429375	0.390716	0.490216
12000	0.295675	0.061945	0.259053	0.328939	0.385254	0.495652	0.427478
16000	0.217877	0.052355	0.242684	0.396969	0.407047	0.464162	0.479516
18000	0.215369	0.040562	0.251246	0.365553	0.41033	0.343533	0.330512
20000	0.216249	0.039278	0.253802	0.359084	0.414488	0.301039	0.293277
24000	0.218102	0.040261	0.250792	0.369916	0.410877	0.365062	0.365644



Increasing the number of structures for smaller number of computations increases GFLOPS drastically but this value fades as the n increases to a point where it becomes equal or less than the number of GFLOPS in 2 structural case.

(d) apply the cache blocking technique described in Tip 3, and again see whether there is performance boost. Experiment with different block sizes. You can assume that R1 processor has 32KB of L1 cache, the cache line size is 64B, and to compute a single dot product you need a row of matrix A, and vector x. The result vector b is written to memory only after the dot product is computed, so it does not need to be in cache.

n	Row-major loop	Column-major loop	Row-major with cache locality	loop unrolling with 2 structures	cache blocking
1000	0.262412	0.098417	0.308594	0.424976	0.307979
2000	0.261765	0.070507	0.308101	0.404407	0.298554
4000	0.259933	0.071485	0.305636	0.403215	0.29349
8000	0.232333	0.067911	0.249681	0.378401	0.257831
12000	0.21518	0.060436	0.249574	0.363913	0.277619
16000	0.207806	0.049537	0.247063	0.382411	0.278298
18000	0.207694	0.036715	0.249417	0.377226	0.27868
20000	0.208893	0.038545	0.25386	0.381365	0.279805
24000	0.210241	0.037828	0.250147	0.376994	0.280188



cache blocking technique does increases GFLOPS but not as much as loop unrolling.

(e) Can you mix the techniques tested above to see whether you can get even better performance boost?

Working on it.

(f) How does compiler optimizations (-O1, -O2, -O3) affect the performance results when you use them in conjunction with your code optimizations?

Using –O1-3 significantly increases GFLOPS as depicted in below figure

n	Row-major loop	Column-major loop	Row-major with cache locality	loop unrolling with 2 structures	cache blocking
1000	0.905848	0.479661	1.081208	1.383129	1.084484
2000	0.866567	0.238421	1.046145	1.215651	0.97982
4000	0.87666	0.2378	1.043486	1.226896	0.964409

8000	0.879322	0.225908	1.042014	1.233511	0.960372
12000	0.879292	0.179575	0.55944	0.97556	0.771187
16000	0.880905	0.11869	0.593831	0.890853	0.806694
18000	0.870776	0.121582	0.605193	0.862814	0.841689
20000	0.689084	0.07142	0.558272	1.221469	0.838181
24000	0.463346	0.122896	0.58238	0.996655	0.843138