Randomization Algorithm to Compute Low-Rank Approximation

Ru HAN
The Chinese University of Hong Kong
hanru0905@gmail.com
August 4, 2017

1. Background

A low-rank representation of a matrix provides a powerful tool for analyzing the data represented by the matrix. It has been successfully used to filter out noises and extract the underlying features of the data in many data analysis tools, including Latent Semantic Indexing (LSI), genetic clustering, subspace tracking, and image processing. In this project, I implement "randomized" algorithm to compute the low-rank representation in the LAPACK/MAGMA/UMA/CUBLAS software framework. Compared to the traditional algorithms, the randomized algorithm has been shown to be more efficient to compute the low-rank representation of the modern large data sets. This is especially true on the modern computer architecture including the hybrid CPU/GPU computers, where the data access has become significantly more expensive compared to the computation.

2. Introduction

Let's say there is a matrix A whose size is M by N, then there exist orthogonal matrices $U = [u_1 u_2, ..., u_M] \in R_M \times R_M$ and $V = [v_1 v_2, ..., v_N] \in R_N \times R_N$ such that $U^T A V = \Sigma = diag(\sigma_1, ..., \sigma_v)$, where Σ is a diagonal matrix whose size is M by N; also, $v = min\{M, N\}$ and $\sigma 1 \geq \sigma 2 \geq ... \geq \sigma v \geq 0$. The σ_i is the i-th singular value of A, the column vectors u_i, vi are the i-th left and the i-th right singular vectors of A, respectively. 【2】.This is the general SVD decomposition.

In low rank SVD approximation, a rank parameter k is given, this k is much smaller than both m and n. the low- rank matrix approximation problem is to find a good approximation to A of rank k, in the sense that by projecting A onto its top k left or right singular vectors. 【2】

In this project, I implemented randomized algorithm to compute the low-rank representation in the LAPACK, MAGMA, UMA and CUBLAS-XT software framework.

3. Algorithm and Math Model

3.1. Algorithm Explanation

The following shows the simple Matlab code for randomized singular value decomposition approximation algorithm which is called Power iteration:

```
function [u, s, v] = svd_rand (A, k, ell, max_iters)
n = size(A,2);
q = randn (n, k+l);
[q, r] = qr (q,0);
    for iter=1:(max_iters-1)
        p = A*q;
        q = A'*p;
        [q, r] = qr(q,0);
    end
    p = A*q;
    [p, b] = qr(p,0);
end
[x, s, y] = svd(b);
u = p*x (:,1: k);
s =s (1: k,1: k);
v = q*y(:,1: k);
```

This algorithm is not hard to understand, for the given M-by-N matrix A, we produce a random matrix Q whose size is N by (K+L); usually we choose L to be equal to K, the use of L is to make the results more accurate. To be clear, I here list the sizes of all matrixes occurred. Dimension of P: M by K+L; B is (K+L)-by-(K+L); X is KL-by-KL; Y^T is K+L-by-K+L; SI is K+L-by-1, by the way, we only need first k values of SI; in this way, we get vector

S, whose size is K-by-1; eventually, we will use S_k=diag(S); Also, the dimension of U_k is M-by-K and the dimension of V_k is N-by-K.

After the whole process, there are three matrix U_K, S_K and V_K that we want. Here, the error is a 2-norm $||A-U_KS_KV_K^T||_2$, which is also equal to $(k+1)_{th}$ largest singular value of A. By the way, it is also a nice way to check whether the code is right or not.

We denote A^TA by A_1, it is clear that A_1 is symmetric, then A_1 can be decomposed to be $U\sum U^T$, where U is orthogonal matrix. Then, $AU=U\sum$, $AU_j=U_j\sigma_j$. For any random matrix P, P can be denoted as $\sum c_j \sigma jUj$ for some c_j; as we can see, $AP=\sum c_j\sigma_jAU_j=\sum c_j \sigma_j^2U_j$.

Additionally, computational cost is a major concern of the project. Let's figure in what cases the randomized algorithm can outperform the general singular value decomposition.

If we use LAPACK to compute SVD, it needs a m*n*n floating point operations (flops); as for randomization algorithm, it needs {2*[2*m*(k+l) *n] *max_iterations} floating point operations. In order to make randomized algorithm performs a fewer flops than LAPACK, we have to decide iteration times based on M, N, K, L.

By simple calculation, we need to let m*n*n>{2*[2*m*(k+l) *n] *max_iterations}; which means n>4*(k+l) *max_iterations. In most cases, the value of n is much larger than k so randomization algorithm should work. Since we usually take L to be equal to K; we can decide iteration times based on M, N, K.

For example, if we choose M=3000; N=500; K=8; then LAPACK's SVD decomposition needs 3000*500*500 flops; randomization algorithm needs {2*[2*3000*(8+8) *500] *max_iterations} flops; so, max_iterations should be less than or equal to 7.

We can see if the N/K is larger, we can use more iteration times; if the N/K is smaller, we can only use less iteration times. It is not hard to see that the randomization is suitable for large database.

3.2. Optimization of the algorithm

3.2.1. Cholesky QR Decomposition

From the efficiency (Glops/s ： giga-flops per second: 10^9 flops per second) I have recorded for each step of the algorithm, I found out the QR decomposition using LAPACK or MAGMA is very slow. Therefore, I use Cholesky QR Decomposition instead.

It works assuming A's condition number is small (e.g., less than 10^8 in double precision). In many data analysis applications, the matrix has low-rank plus noises, and these noises seem to make Cholesky QR Decomposition stable.

The following is the algorithm of Cholesky QR Decomposition:

(1) G=ATA

(2) G=RTR

(3) q=AR-1

The following figure shows the algorithm of Cholesky QR Decomposition:

3.2.2. Use of symmetry of A^T*A

In computing something like R = A^T*A, instead of calling DGEMM or ZGEMM to form the entire matrix, one may consider taking advantage of the symmetry and compute only the lower or upper part of the matrix. This can be performed using ZHERK (symmetric rank-k

update), which takes less time compared to ZGEMM since it need only half work of ZGEMM.

And in Cholesky factorization, then Cholesky subroutine may need access only the lower or upper part of the matrix.

4. Project Scheme

The following shows the outline of the project:

(1) Use LAPACK/BLAS to implement the randomized algorithm on CPU

(2) Use MAGMA to implement basic randomized algorithm on GPU

(3) Implement out-of-memory randomized algorithm when the matrix does not fit on the GPU, there are mainly two methods: using single queue, manual pipelining and using UMA and CUBLAS-XT on GPU.

(4) set up tester to compare performances

4.1. Implementation of the randomized algorithm on CPU using LAPACK

For the randomization on CPU, we use double complex version. There are totally four version: double complex version (LAPACK function starting with 'z'), single complex version (LAPACK function starting with 'c'), double real version (LAPACK function starting with 'd') and single real version (LAPACK function starting with 's').

4.2. Implementation of the randomized algorithm on GPU using MAGMA

Let's see the major time-consuming parts of the whole process which are P=A*Q and $Q=A^T*P$. The dimension of A is M-by-N, the dimension of Q is N-by-(K+L) and the dimension of P is M-by-(K+L). As for the flops, they are all 2*N*M*K flops. Therefore, the time spent on P=A*Q and $Q=A^T*P$ should be very close.

When allocating CPU memory that will be used to transfer data to the GPU, there are two types of memory to choose from: pinned and non-pinned memory. Pinned

memory is memory allocated using the cudaMallocHost function, which prevents the memory from being swapped out and provides improved transfer speed. [5]

To reduce the time spent on set and get matrix, I use CPU pinned memory instead of just using CPU memory.

4.3. Implementation of the out-of-memory randomized algorithm on GPU

Since GPU has much smaller memory size than the CPU, it is very normal that the matrix does not fit on the GPU. That is when we should extend GPU-implementation.

For each GPU device, it has about11439.9 MiB memory, which is equal to 1.19963e+10 bytes, and I use double precision. Sqrt(12e9/8) =3.8730e+04. After some simple calculation, we can see that the device can only fit at most forty thousand by forty thousand matrix. Therefore, it is very common that many matrixes can not fit in the GPU. Here is where we need out-of-memory algorithm implementation.

4.3.1. Manual pipelining

When matrix A does not fit in the memory, we implement the out-of-memory algorithm. And we mainly need to care about two places of the code, where the matrix A is involved.

One possible solution is to handle the “out of core” data movement directly but allocating say a dAtmp on GPU and explicitly transfer each part of A into dAtmp and perform dAtmp * dQ or transpose (dAtmp) * dQ.

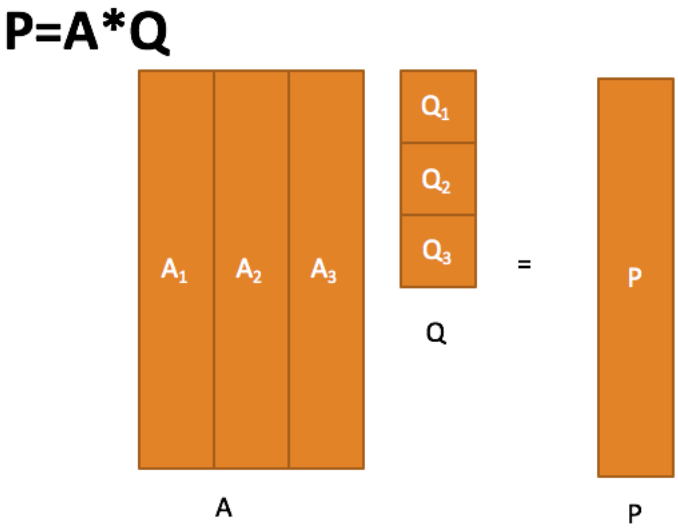
Firstly, when we can P=A*Q;
we divide A and Q into several parts. Let’s call them A_1, A_2, A_3, Q_1, Q_2, Q_3, then P equals to $A_1*Q_1+A_2*Q_2+A_3*Q_3$.
The algorithm is as follows:
P=0;
for i=1,2,3….

set (A_i to dA);

$P=P+A_iQ_i$;

end

We can also see the process through the figure:



Similarly, we can get the result of $Q=A^T*P$.

Algorithm:

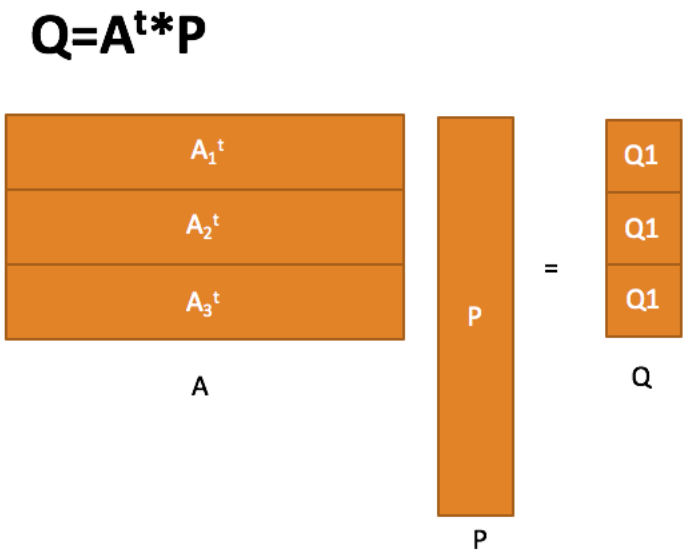
for i=1,2,3….

set (A_i to dA);

$Q_i=A_i^T*P$;

end

As for how many i we need, in another word, how many parts A and Q are needed to divide, it depends on the size of the A.



Here, we denote the number of rows of A_i to be NB. The size of NB is a major key of the efficiency of the algorithm.

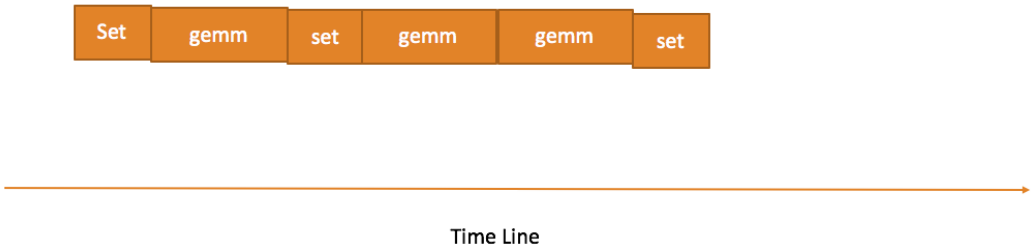
The setting of "NB" is for performance and there is some flexibility. And this value of NB can be determined by calling cudaMemGetInfo () after all the matrixes on GPU like dU, dV, ..., device arrays have been allocated. One may then assume say 80% of remaining device memory is available for dA. Then we can solve for NB using the following formula:

*NB = (0.8 * (freeMem/sizeof (magmaDoubleComplex)))/ (N * num_queues);*

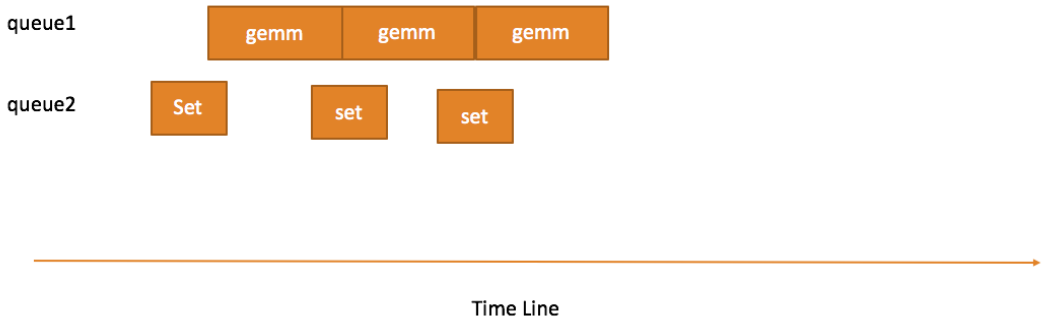
NB = MAX (1, MIN (MIN (N, KL), NB));

This method takes advantage of available GPU device memory.

When we use single queue to implement the out-of-memory case, I need to copy matrix from CPU to GPU; and for each parts of A, before doing GEMM I have to do the copy action, which is called set-matrix. Also, GEMM is a function for matrix multiplication. So, it will take a lot more time comparing to CPU and for cases that matrix fits. We can see the process as follows:

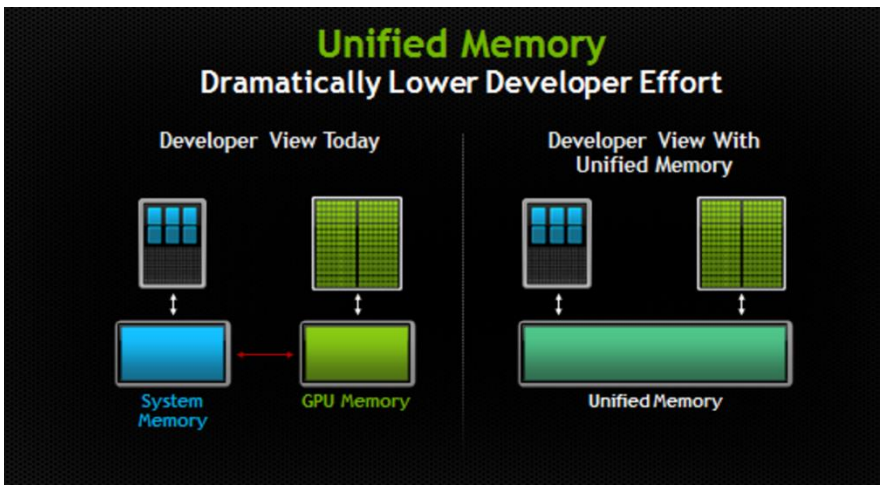


A more sophisticated version may allocate dAtmp [0], dAtmp [1] and try to overlap data movement while performing computations. In another word, I Use more than one queue, so that I can do the set action and GEMM at the same time; then it will save a lot of time.

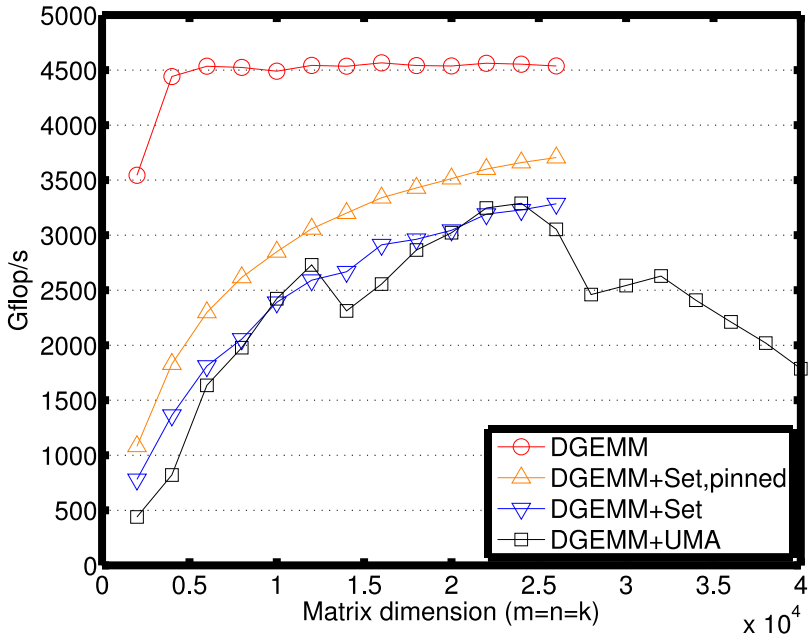


4.3.2. using UMA and CUBLAS-XT

UMA is a programming model, Unified Memory Access. Unified Memory creates a pool of managed memory that is shared between the CPU and GPU. Managed memory is accessible to both the CPU and GPU using a single pointer. The key is that the system automatically migrates data allocated in Unified Memory between host and device. [1]



The following figure shows the performance of GEMM (matrix-matrix multiplication) between DGEMM, DGEMM with set and pinned, DGEMM with set, DGEMM with UMA. From this, we can see when the memory fits on GPU, the performance of MAGMA is much better, but when the memory does not fit, we need UMA.



The NVIDIA CUBLAS library is a fast GPU-accelerated implementation of the standard basic linear algebra subroutines (BLAS).

The CUBLAS-XT can accept arrays on CPU and break up the matrix on CPU into blocks and perform data transfer and computations on GPU. When I use CUBLAS-XT, the data/matrices that I compute are on the CPU memory, so you don't have to copy them to the GPU. The data arrays on CPU can be larger than available GPU device memory. CUBLAS-XT takes data pointer to CPU memory. The CUBLAS-XT library will allocate device storage, perform data transfers, perform computations and transfer data back. This enables all the functions like GEMM to work on matrices larger than amount of GPU device memory.
It can also take advantage of multiple GPUs on the same node. The library is copying things as necessary and uses as many GPUs are specified to accelerate the computation.

It is a convenient interface to get something to work, but it may be doing more than minimal data transfer. If the arrays are sufficiently large that it is dominated by computation and not data movement, then this ability to use multiple GPUs may be very useful.

Optimization of CUBLAS-XT: In the configuration for simple CUBLASXT, there is a call cublasXTSetBlockDim to set the block dimension. One would like this BlockDim to be sufficiently large so that time for computation is least comparable to time for data movement. However, a very large BlockDim may use too much GPU device memory or limit amount of parallel computation. So, I set BlockDim to be 256 which is relatively big for better performance.

5. Performance Results

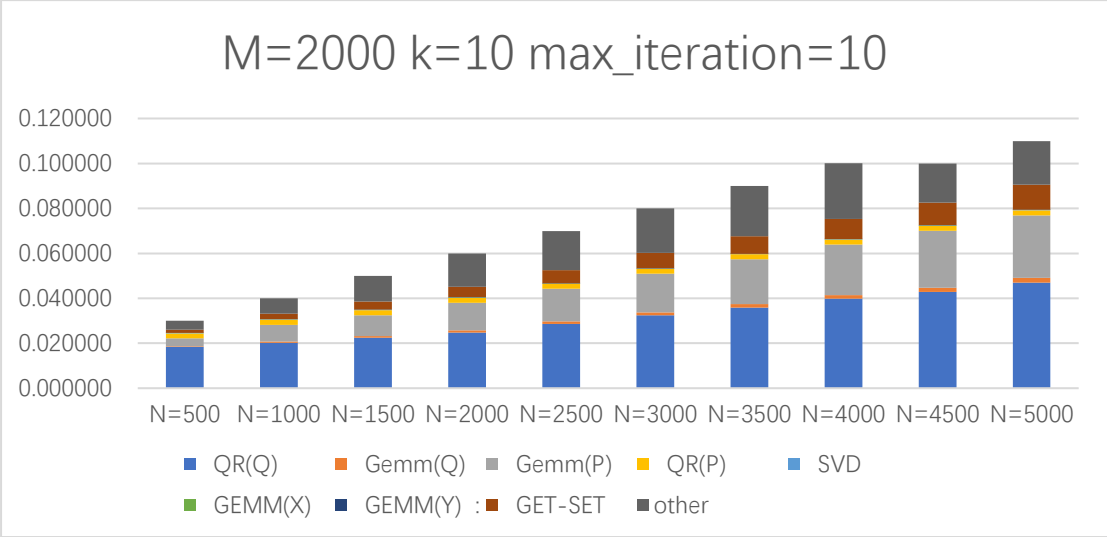
I set up a tester called 'testing_zgesvd_rand.cpp' to compute time and error. In this tester, I used four functions I created in the source file called 'zgesvd_rand.cpp', 'zgesvd_rand_cpu.cpp', 'zgesvd_rand_m.cpp' and 'zgesvd_rand_uma.cpp'. For each test, we can compare the time and error of LAPACK/CPU/GPU/NGR/UMA/CUBLAS.

First set of comparisons is between LAPACK SVD, CPU and in-core GPU.

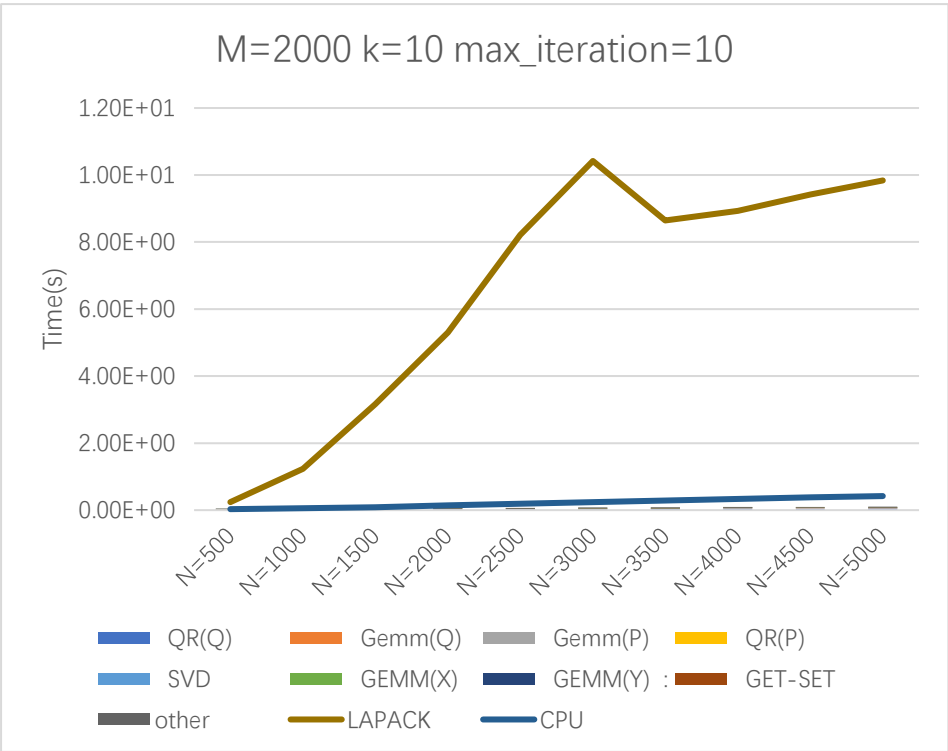
In the GPU case, I timed 8 different steps of SVD algorithm and they are QR(Q) for [Q,R] = QR(Q,0); GEMM(Q) for Q = A^T*P; GEMM(P) for P = A*Q; QR(P) for [P,B] = QR (P,0); SVD for [X,S,Y] = SVD(B); GEMM(X) for U = P*X(:,1:k); GEMM(Y) for V = Q*Y(:,1:k); GET-SET for all the set-matrix and get-matrix performances. The following are 4 sets of comparisons.

Comparison 1: M=2000, k=10, max_iteration=10; change N

GPU	QR(Q)	Gemm(Q)	Gemm(P)	QR(P)	SVD	GEMM(X)	GEMM(Y) :	GET-SET	other
N=500	0.018300	0.000316	0.003510	0.002270	0.000132	0.000038	0.000025	0.001410	0.004000
N=1000	0.020200	0.000565	0.007440	0.002290	0.000147	0.000038	0.000025	0.002500	0.006800
N=1500	0.022500	0.000705	0.009240	0.002310	0.000146	0.000038	0.000028	0.003580	0.011400
N=2000	0.024800	0.000928	0.012300	0.002250	0.000153	0.000038	0.000028	0.004690	0.014800
N=2500	0.028600	0.001100	0.014600	0.002220	0.000149	0.000039	0.000030	0.005730	0.017500
N=3000	0.032400	0.001290	0.017200	0.002220	0.000149	0.000039	0.000034	0.006820	0.019900
N=3500	0.035900	0.001470	0.020000	0.002230	0.000146	0.000038	0.000037	0.007860	0.022300
N=4000	0.039800	0.001680	0.022400	0.002240	0.000145	0.000038	0.000037	0.008930	0.024800
N=4500	0.042900	0.001860	0.025300	0.002260	0.000152	0.000037	0.000039	0.010000	0.017400
N=5000	0.047000	0.002060	0.027800	0.002340	0.000177	0.000038	0.000041	0.011100	0.019400

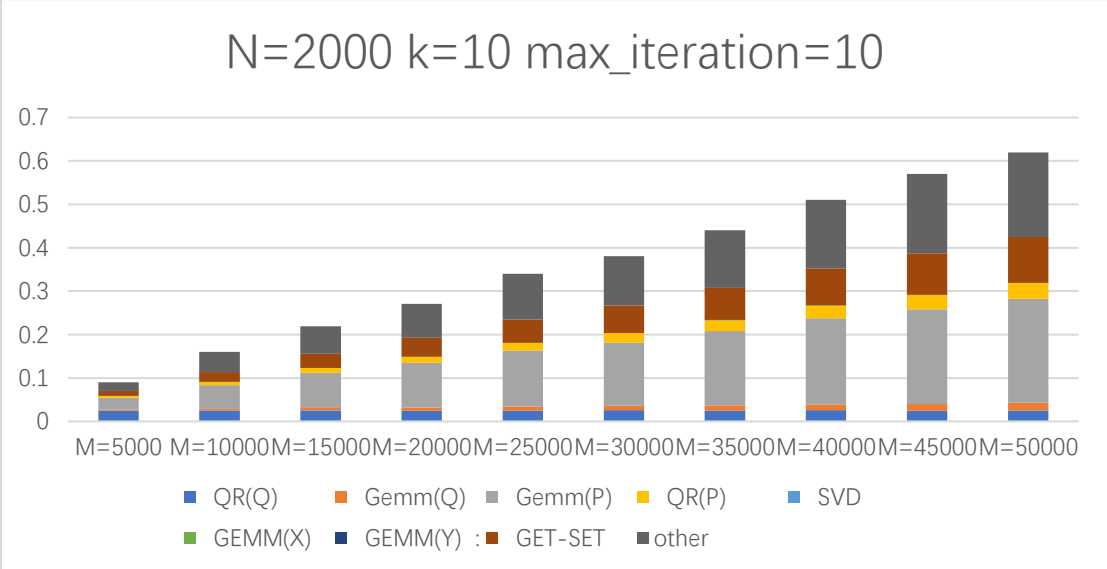


	LAPACK	CPU	GPU
N=500	0.24	0.03	0.030001
N=1000	1.23	0.06	0.040005
N=1500	3.16	0.09	0.049947
N=2000	5.3	0.14	0.059987
N=2500	8.21	0.19	0.069968
N=3000	10.42	0.24	0.080052
N=3500	8.64	0.29	0.089981
N=4000	8.93	0.33	0.100070
N=4500	9.42	0.38	0.099948
N=5000	9.84	0.42	0.109956

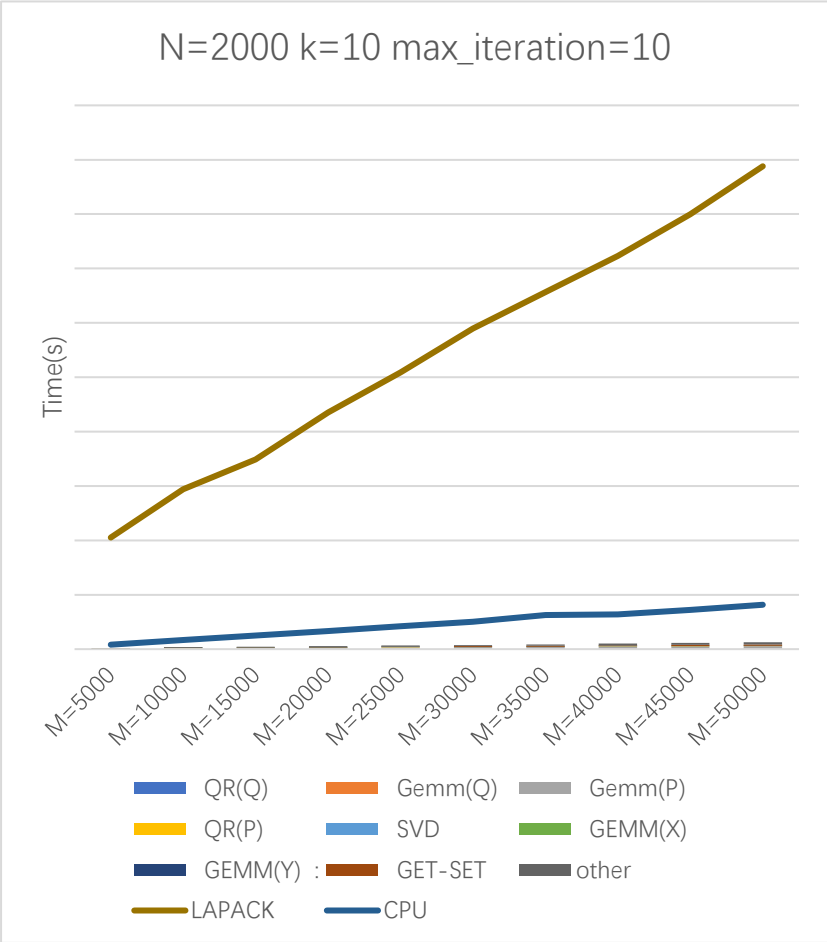


Comparison 2: N=2000, k=10, max_iteration=10; change M

GPU	QR(Q)	Gemm(Q)	Gemm(P)	QR(P)	SVD	GEMM(X)	GEMM(Y) :	GET-SET	other
M=5000	0.0245	0.00206	0.0279	0.00456	1.24e-04	0.000051	0.0000279	0.011	0.0198
M=10000	0.025	0.00402	0.0542	0.00825	0.000149	0.000127	0.0000331	0.0216	0.047
M=15000	0.0251	0.00608	0.0807	0.0114	0.000123	0.00015	0.0000329	0.0319	0.064
M=20000	0.0246	0.00725	0.103	0.0149	0.000134	0.000155	0.0000391	0.0423	0.078
M=25000	0.0247	0.00917	0.129	0.0186	0.000125	0.000179	0.000031	0.0529	0.105
M=30000	0.0262	0.0101	0.145	0.0221	0.000149	0.000183	0.000031	0.0631	0.114
M=35000	0.0248	0.012	0.171	0.0254	0.000166	0.000206	0.00003	0.0737	0.133
M=40000	0.0256	0.0137	0.198	0.0303	0.000161	0.000221	0.00003	0.0844	0.158
M=45000	0.0245	0.0155	0.218	0.0339	0.000174	0.00025	0.00003	0.0946	0.183
M=50000	0.0252	0.0174	0.24	0.0367	0.00016	0.000247	0.00003	0.105	0.195

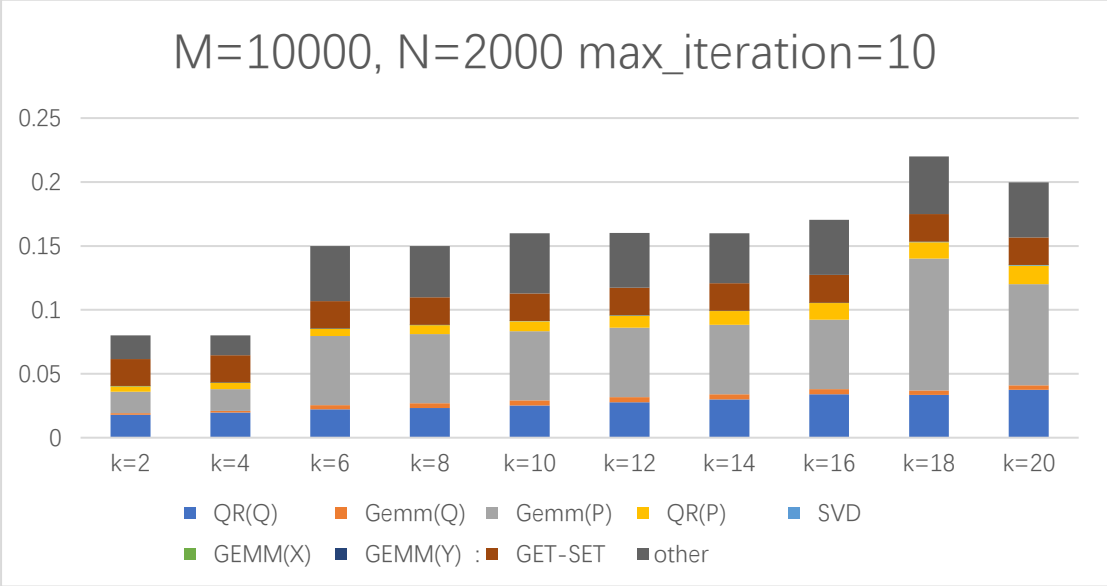


	LAPACK	CPU	GPU
M=5000	10.26	0.41	0.0898989
M=10000	14.71	0.85	0.1603791
M=15000	17.43	1.24	0.2194859
M=20000	21.78	1.66	0.2703781
M=25000	25.45	2.1	0.339705
M=30000	29.47	2.52	0.380863
M=35000	32.82	3.15	0.440302
M=40000	36.15	3.2	0.510412
M=45000	39.98	3.62	0.569954
M=50000	44.4	4.08	0.619737

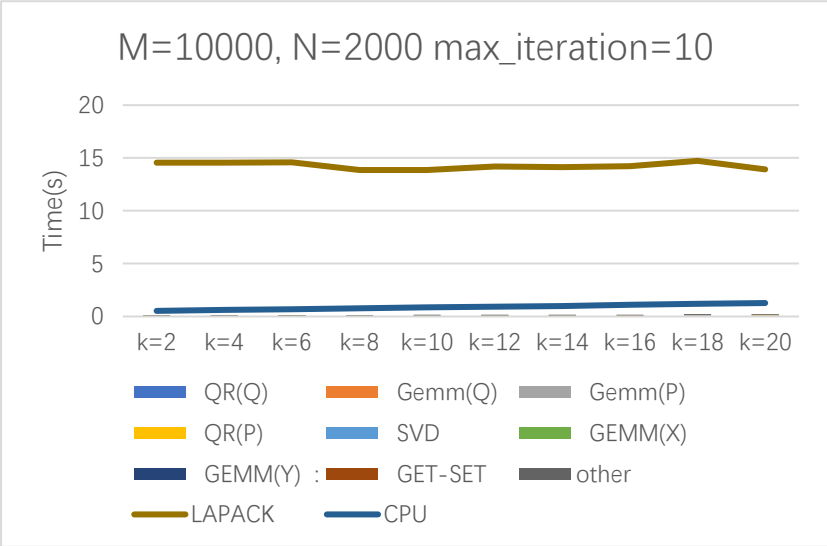


Comparison 3: M=10000, N=2000, max_iteration=10; change k

	QR(Q)	Gemm(Q)	Gemm(P)	QR(P)	SVD	GEMM(X)	GEMM(Y) :	GET-SET	other
k=2	0.0179	0.00128	0.0169	0.0042	0.0000372	0.000042	0.0000188	0.0212	0.0184
k=4	0.0197	0.00131	0.0169	0.00513	0.000073	0.000046	0.0000219	0.0213	0.0155
k=6	0.0221	0.00348	0.054	0.00573	0.000067	0.000047	0.0000219	0.0214	0.043
k=8	0.0231	0.00398	0.054	0.00699	0.000093	0.000112	0.000031	0.0215	0.04
k=10	0.0251	0.00401	0.0541	0.00792	0.00012	0.000133	0.0000329	0.0215	0.047
k=12	0.0278	0.00401	0.0542	0.00942	0.000163	0.000109	0.000047	0.0215	0.043
k=14	0.0299	0.00402	0.0543	0.0108	0.000214	0.000121	0.0000501	0.0215	0.039
k=16	0.034	0.00403	0.0544	0.0128	0.000275	0.00012	0.0000498	0.0217	0.043
k=18	0.0334	0.00368	0.103	0.0127	0.000465	0.000123	0.0000532	0.0216	0.045
k=20	0.0375	0.0036	0.0789	0.0144	0.000582	0.000119	0.0000498	0.0216	0.043

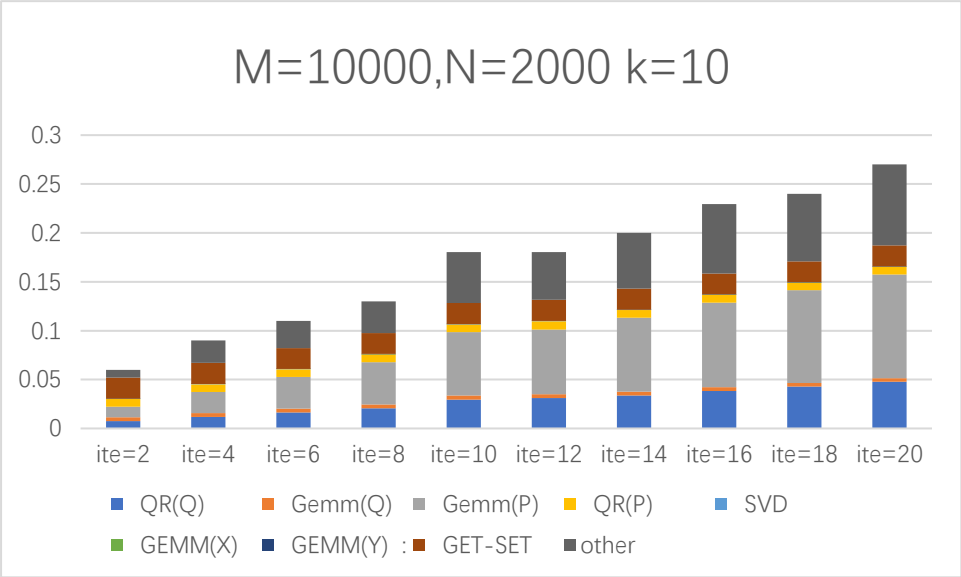


	LAPACK	CPU	GPU
k=2	14.55	0.53	0.079978
k=4	14.55	0.61	0.0799809
k=6	14.59	0.69	0.1498459
k=8	13.86	0.78	0.149806
k=10	13.86	0.85	0.1599159
k=12	14.19	0.92	0.160249
k=14	14.14	1	0.1599051
k=16	14.21	1.11	0.1703748
k=18	14.73	1.19	0.2200212
k=20	13.91	1.27	0.1997508

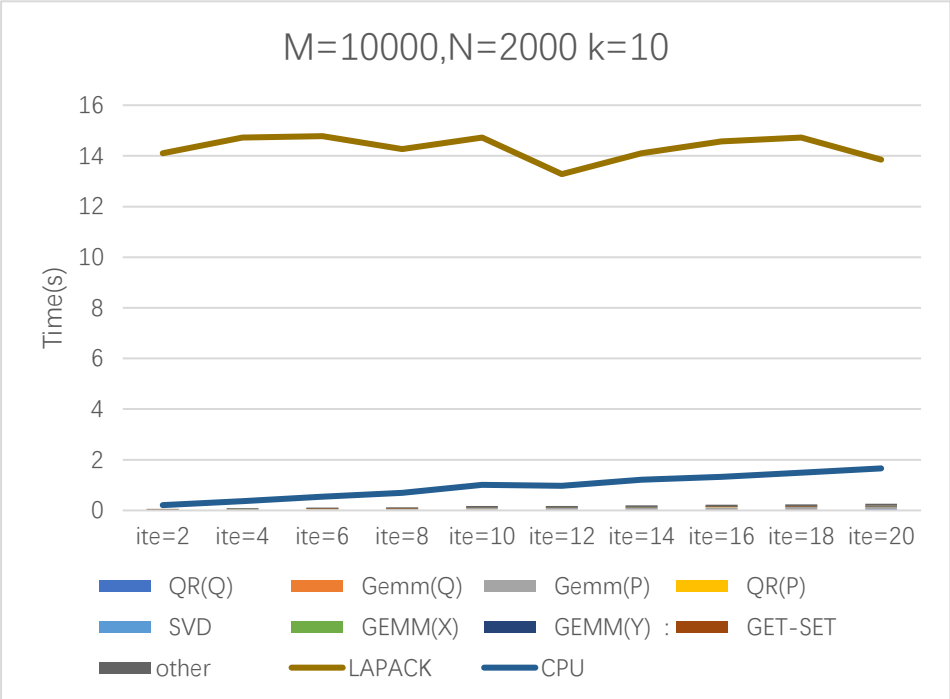


Comparison 4: M=10000, N=2000 k=10; change max_iteration

	QR(Q)	Gemm(Q)	Gemm(P)	QR(P)	SVD	GEMM(X)	GEMM(Y) :	GET-SET	other
ite=2	0.00737	0.00401	0.0109	0.008	0.000129	0.000126	0.0000331	0.0215	0.008
ite=4	0.0118	0.00402	0.0216	0.00792	0.000124	0.000127	0.0000319	0.0215	0.0228
ite=6	0.0163	0.00402	0.0324	0.0079	0.000123	0.000134	0.0000372	0.0214	0.0276
ite=8	0.0206	0.004	0.0433	0.00794	0.000121	0.000125	0.0000319	0.0215	0.0323
ite=10	0.0296	0.00402	0.0649	0.00796	0.000122	0.000128	0.0000331	0.0215	0.052
ite=12	0.0311	0.00409	0.0661	0.00851	0.000148	0.000133	0.0000319	0.0214	0.049
ite=14	0.0337	0.00401	0.0757	0.00788	0.000131	0.000127	0.0000329	0.0215	0.057
ite=16	0.0382	0.00403	0.0865	0.00819	0.000124	0.00012	0.000031	0.0214	0.071
ite=18	0.0429	0.0036	0.0948	0.00786	0.000129	0.000128	0.0000319	0.0215	0.069
ite=20	0.0478	0.00359	0.106	0.00795	0.000131	0.000125	0.000031	0.0215	0.083



	LAPACK	CPU	GPU
ite=2	14.1	0.21	0.0600681
ite=4	14.72	0.37	0.0899229
ite=6	14.78	0.54	0.1099142
ite=8	14.27	0.69	0.1299179
ite=10	14.73	1.01	0.1802631
ite=12	13.28	0.97	0.1805129
ite=14	14.11	1.21	0.2000809
ite=16	14.57	1.33	0.229595
ite=18	14.72	1.49	0.2399489
ite=20	13.85	1.66	0.270127



6. Motivation and Application of Randomized Approximation

Singular Value Decomposition has a lot of applications in large-scale data analysis. A general family of applications for this algorithm is Principal Component Analysis applications, where a database (of documents, images etc.) that exists in a high dimensional space is projected to a lower dimensional space using SVD. It has been successfully used to filter out noises and extract the underlying features of the data in many data analysis tools. While the most obvious benefit of randomization is that it can lead to faster algorithms. [2]

For example, we can use the randomized SVD onto Latent Semantic Indexing(LSI), genetic clustering, subspace tracking and image processing.

In astronomy, for example, very small angular regions of the sky imaged at a range of electromagnetic frequency bands can be represented as a matrix—in that case, an object is a region and the features are the elements of the frequency bands.

Similarly, in genetics, DNA Single Nucleotide Polymorphism or DNA microarray expression data can be represented in such a framework, with A_{ij} representing the expression level of the i-th gene or SNP in the j-th experimental condition or individual. [2]

What's more, Latent semantic indexing. Latent semantic indexing(LSI) is an indexing and retrieval method that uses a mathematical technique called singular value decomposition (SVD) to identify patterns in the relationships between the terms and concepts contained in an unstructured collection of text. A matrix containing word counts per paragraph (rows represent unique words and columns represent each paragraph) is constructed from a large piece of text and a mathematical technique called (SVD) is used to reduce the number of rows while preserving the similarity structure among columns. [4]

7. Conclusion

In this project, I mainly have done the following things:

(1) Use LAPACK/BLAS to implement the randomized algorithm on CPU

(2) Use MAGMA to implement basic randomized algorithm on GPU

(3) Implement out-of-memory randomized algorithm when the matrix does not fit on the GPU, there are mainly two methods: using single queue, manual pipelining and using UMA and CUBLAS-XT on GPU.

(4) set up tester to compare performances, then optimize the algorithm.

8. Future Work

If the node has multiple GPU and power iteration multiplication by A is very expensive, then one might consider using multiple GPU. Ideally this can be done simply by using CUBLAS-XT.

When the matrix is too big, we should try to get familiar with different randomized sampling and updating methods for the out-of-core Matrix.

Also, there are many applications for us to explore like Latent Semantic Indexing (LSI), genetic clustering, subspace tracking, and image processing.

9. Acknowledgements

This project is sponsored by Oak Ridge National Laboratory, Joint Institute for Computational Sciences, University of Tennessee, Knoxville and The Chinese University of Hong Kong.

Most sincere gratitude to my mentors: Dr. Ed D‘Azevedo’s and Dr. Ichitaro Yamazaki.

Reference

[1] Harris, M. and →, V. (2017). Unified Memory in CUDA 6. [online] Parallel Forall. Available at: https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/ [Accessed 21 Jun. 2017].

[2] Mahoney, M. (2011). Randomized algorithms for matrices and data. Hanover, Mass.: Now Publishers.

[3] Drinea, E., Drineas, P. and Huggins2, P. (2017). A Randomized Singular Value Decomposition Algorithm for Image Processing Applications. [ebook] 1 Computer Science Department, Harvard University Cambridge, MA 02138, USA 2 Computer Science Department, Yale University New Haven, CT 06520, USA. Available at: http://ai2-s2-pdfs.s3.amazonaws.com/e881/439705f383468b276415b9d01d0059c1d3e5.pdf [Accessed 26 Jun. 2017].

[4] En.wikipedia.org. (2017). Latent semantic analysis. [online] Available at: https://en.wikipedia.org/wiki/Latent_semantic_analysis [Accessed 29 Jun. 2017].

[5] Cs.virginia.edu. (2017). Pinned vs. non-pinned memory. [online] Available at: https://www.cs.virginia.edu/~mwb7w/cuda_support/pinned_tradeoff.html [Accessed 11 Jul. 2017].