Randomization Algorithm to Compute Low-Rank Approximation

Student: Ru HAN (The Chinese University of Hong Kong)
Mentors: Dr. Ed D‘Azevedo’s, Dr. Ichitaro Yamazaki

Outline

- Background
- Algorithm and Math Model
- Project Scheme
- Performance Results
- Motivation and Application of Randomized Approximation
- Future Work

Background-General SVD

$A = U\sum V^T$

$U = [u_1 u_2, \ldots, u_M] \in R_M \times R_M$

$V = [v_1 v_2, \ldots, v_N] \in R_N \times R_N$

$\Sigma = \mathrm{diag}(\sigma 1, \ldots, \sigma \nu) = U^T A V$, $\Sigma \in R_M \times R_N$, $\nu = \min\{M, N\}$, $\sigma 1 \geq \sigma 2 \geq \ldots \geq \sigma \nu \geq 0$.

Background-General SVD

Example: $A=U\Sigma V^T$

A

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 2 \\ 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{U} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{\Sigma} = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & 3 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{V}^* = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ \sqrt{0.2} & 0 & 0 & 0 & \sqrt{0.8} \\ 0 & 0 & 0 & 1 & 0 \\ -\sqrt{0.8} & 0 & 0 & 0 & \sqrt{0.2} \end{bmatrix}$$

$$\mathbf{U}\mathbf{U}^\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \mathbf{I}_4$$

$$\mathbf{V}\mathbf{V}^\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} = \mathbf{I}_5$$

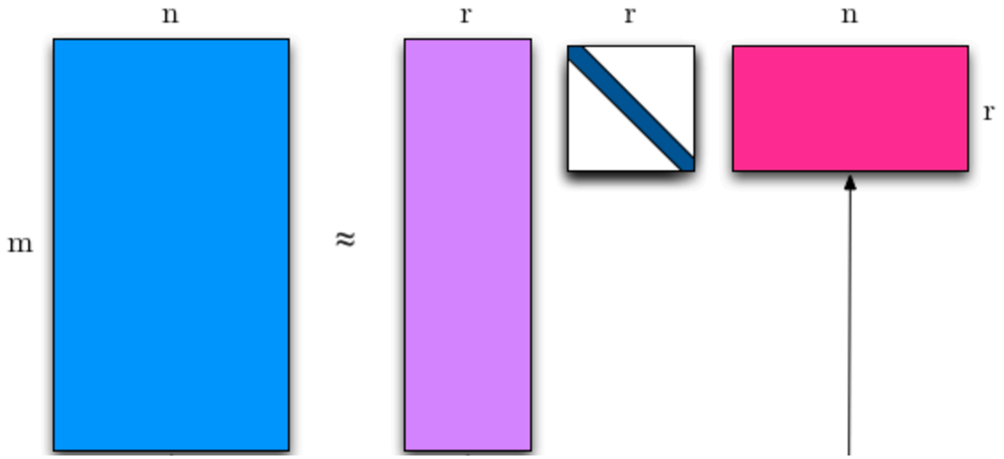
Background

- Low-Rank SVD Approximation

$A = U_k \Sigma_k V_k^t$

Σ_k: largest k singular values of A

large matrix in image processing



- LAPACK/MAGMA/CUBLAS-XT software framework

Algorithm--Power iteration

Matlab Code “svd_rand./” SVD approximation

```
function [u,s,v] = svd_rand(A, k, l, max_iters)
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_k = p*x(:,1:k);

s = s(1:k,1:k);

v_k= q*y(:,1:k);

Algorithm and Math Model

- Input: mxn matrix A, int k,

1. Draw a random nx(k+l) matrix Ω.
2. Compute QR of $(AA^T)^q A \Omega$
3. and SVD:
4. Truncate SVD $Q^T A = \hat{U}\hat{\Sigma}\hat{V}^T$

$$\hat{U}_k \hat{\Sigma}_k \hat{V}_k^T$$

- Output:

$$B = (Q\hat{U}_k)\hat{\Sigma}_k \hat{V}_k^T$$

QR needs done carefully for numerical accuracy.

Algorithm is old one when q = 0; but q = 1 far more accurate.

Should converge faster when singular values do not decay very fast.

Thm [Limited Warranty] (Halko/Martinsson/Tropp, 2011)

$$\|A - B\|_2 = O(\sigma_{k+1}) > \sigma_{k+1}$$

with failure probability $5p^{-p}$

Computational Cost

LAPACK SVD: M*N*N floating point operations (FLOPS)

randomization algorithm: {2*[2*M*(K+L)*N]*max_iterations} FLOPS

M*N*N>{2*[2*M*(K+L)*N]*max_iterations}

N>4*(K+L)*max_iterations

P=A*Q/ Q=A^T*P: 2*N*M*K FLOPS

Matrix	Size
A	M-by-N
Q	N-by-(K+L)
P	M-by-(K+L)
B	(K+L)-by-(K+L)
X	(K+L)-by-(K+L)
Y^T	(K+L)-by-(K+L)
SI	(K+L)-by-1
S	K-by-1
u_k	M-by-K
v_k	N-by-K

QR Decomposition

Consider the decomposition of

$$A = \begin{pmatrix} 12 & -51 & 4 \\ 6 & 167 & -68 \\ -4 & 24 & -41 \end{pmatrix}.$$

Recall that an orthonormal matrix Q has the property

$$Q^T Q = I.$$

Then, we can calculate Q by means of Gram–Schmidt as follows:

$$U = (\mathbf{u}_1 \quad \mathbf{u}_2 \quad \mathbf{u}_3) = \begin{pmatrix} 12 & -69 & -58/5 \\ 6 & 158 & 6/5 \\ -4 & 30 & -33 \end{pmatrix};$$

$$Q = \left(\frac{\mathbf{u}_1}{\|\mathbf{u}_1\|} \quad \frac{\mathbf{u}_2}{\|\mathbf{u}_2\|} \quad \frac{\mathbf{u}_3}{\|\mathbf{u}_3\|} \right) = \begin{pmatrix} 6/7 & -69/175 & -58/175 \\ 3/7 & 158/175 & 6/175 \\ -2/7 & 6/35 & -33/35 \end{pmatrix}.$$

[q,r] = qr(q,0);
q=q*r

In linear algebra, a **QR decomposition** (also called a **QR factorization**) of a matrix is a **decomposition** of a matrix A into a product A = **QR** of an orthogonal matrix Q and an upper triangular matrix R.

Thus, we have

$$Q^T A = Q^T Q R = R;$$

$$R = Q^T A = \begin{pmatrix} 14 & 21 & -14 \\ 0 & 175 & -70 \\ 0 & 0 & 35 \end{pmatrix}.$$

Optimization of the algorithm Cholesky QR

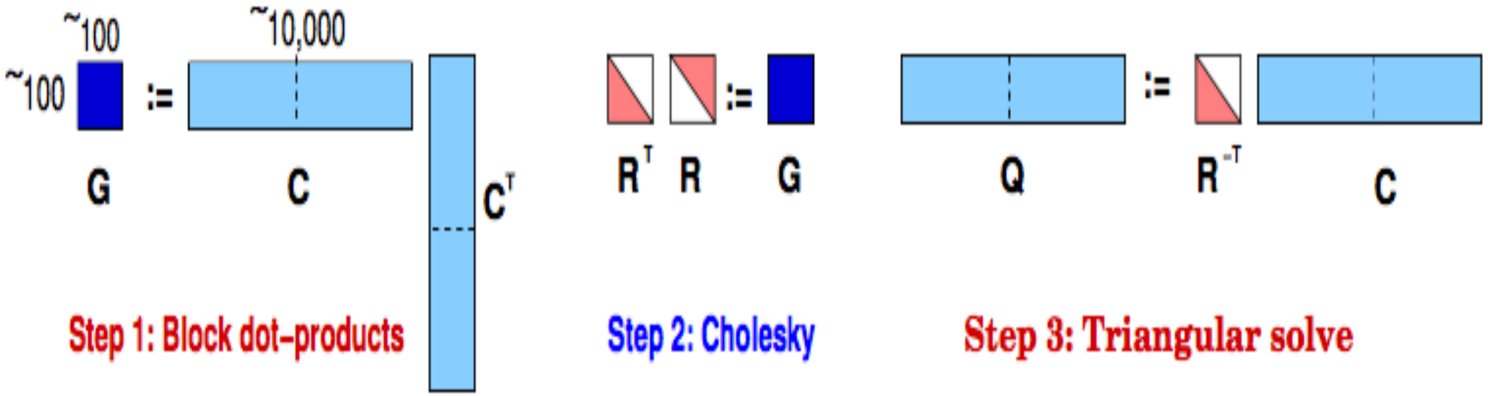
efficiency (Gflops/s): giga-flops per second: 10^9 flops per second

algorithm of Cholesky QR Decomposition:

(1) $G=C^TC$

(2) $G=R^TR$

(3) $Q=CR^{-1}$



Optimization of the algorithm Cholesky QR

(1)G=C^TC

(2)G=R^TR

(3) Q=CR⁻¹

- suppose C is an m $\times$ n matrix with linearly independent columns
- the matrix G = C^TC is positive definite

every positive definite matrix G $\in$ Rn$\times$n can be factored as G= R^TR where R is upper triangular with positive diagonal elements

- complexity of computing R is $(1/3)n^3$ flops
- R is called the Cholesky factor of G

$$\begin{bmatrix} 25 & 15 & -5 \\ 15 & 18 & 0 \\ -5 & 0 & 11 \end{bmatrix} = \begin{bmatrix} R_{11} & 0 & 0 \\ R_{12} & R_{22} & 0 \\ R_{13} & R_{23} & R_{33} \end{bmatrix} \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ 0 & R_{22} & R_{23} \\ 0 & 0 & R_{33} \end{bmatrix}$$

$$= \begin{bmatrix} 5 & 0 & 0 \\ 3 & 3 & 0 \\ -1 & 1 & 3 \end{bmatrix} \begin{bmatrix} 5 & 3 & -1 \\ 0 & 3 & 1 \\ 0 & 0 & 3 \end{bmatrix}$$

Optimization of the algorithm Cholesky QR

EXAMPLE

$$B = \begin{bmatrix} 3 & -6 \\ 4 & -8 \\ 0 & 1 \end{bmatrix}, \qquad A = B^T B = \begin{bmatrix} 25 & -50 \\ -50 & 101 \end{bmatrix}$$

1. Cholesky factorization:

$$A = \begin{bmatrix} 5 & 0 \\ -10 & 1 \end{bmatrix} \begin{bmatrix} 5 & -10 \\ 0 & 1 \end{bmatrix}$$

2. QR factorization

$$B = \begin{bmatrix} 3 & -6 \\ 4 & -8 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 3/5 & 0 \\ 4/5 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 5 & -10 \\ 0 & 1 \end{bmatrix}$$

Optimization of the algorithm

```
[han123@comet-33-02 testing]$ ./testing_dgesvd_rand --range 10000,2000,20 -l --niter 1 -c
% MAGMA 2.2.0 svn compiled for CUDA capability >= 3.0, 32-bit magma_int_t, 64-bit pointer.
% CUDA runtime 7000, driver 8000. OpenMP threads 1. MKL 11.3.3, MKL threads 1.
% device 0: Tesla P100-PCIE-16GB, 405.0 MHz clock, 16276.2 MiB memory, capability 6.0
% Wed Jul 26 07:35:06 2017
% Usage: ./testing_dgesvd_rand [options] [-h|--help]

 Error is ||A - Uk*Sk*Vk^T||_2, L=K, performs 1 iterations

%   M     N     K      LAPACK time (s)    Randomized time (s)      LAPACK error         Randomized error
%                                         CPU, GPU, NGR, UMA                            CPU, GPU, NGR, UMA
%===================================================================================================================

Intel MKL ERROR: Parameter 5 was incorrect on entry to DGEQRF.
  QR(Q)     : 5.31e-04 second,   24.27Gflop/s
  Gemm(Q)   : 8.73e-04 second, 3665.15Gflop/s
  Gemm(P)   : 1.27e-03 second, 1256.96Gflop/s
  QR(P)     : 3.13e-04 second,  103.37Gflop/s
  SVD       : 5.89e-04
  GEMM(X)   : 9.80e-05 second,  163.28Gflop/s
  GEMM(Y)   : 4.60e-05 second,   69.54Gflop/s
  GET-SET   : 1.88e-02
  laset     : 0.00e+00
  lacpy     : 0.00e+00
  ungqr     : 0.00e+00
    Total   : 2.26e-02
simpleCUBLASXT test running..
10000  2000    20     13.28            0.19, 0.03, 0.15, 4.01      4.09e+01       4.15e+01,4.15e+01,4.15e+
01,4.15e+01      (S[20]=4.09e+01)
[han123@comet-33-02 testing]$ make testing_dgesvd_rand
```

Project Scheme

1. Implementing the randomized algorithm using LAPACK on CPU

2. Implementing the randomized algorithm using MAGMA on GPU

3. Implementing the out-of-memory randomized algorithm on GPU

-manual pipelining.

-UMA

-CUBLAS-XT

4. set up tester to compare performances

Out-of-Memory GPU Implementation

Device: Tesla K80, 823.5 MHz clock, 11439.9 MiB memory, capability 3.7

1 MiB = 2^{20} bytes = 1024 kibibytes = 1048576bytes

11439.9Mib$*$1048576=1.1996e+10 bytes

Sqrt(12e9/8)=3.8730e+04

Out-of-Memory GPU Implementation manual pipelining

P=A*Q

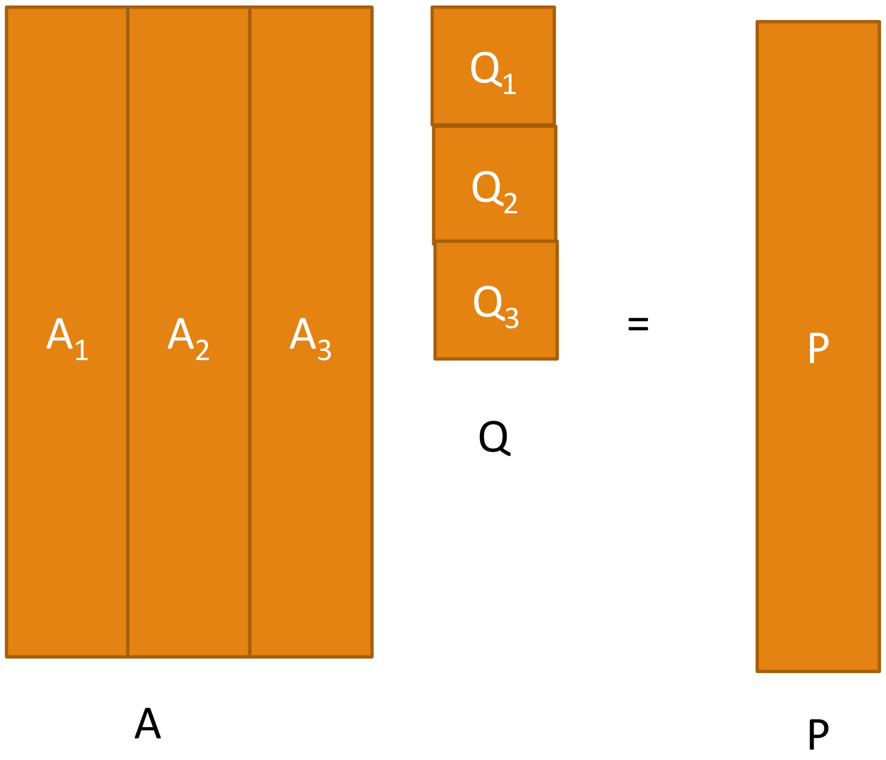
P=0;

For k=1,2,3…..

set (A_k to dA);

$P=P+A_kQ_k$;

end



Out-of-Memory GPU Implementation manual pipelining

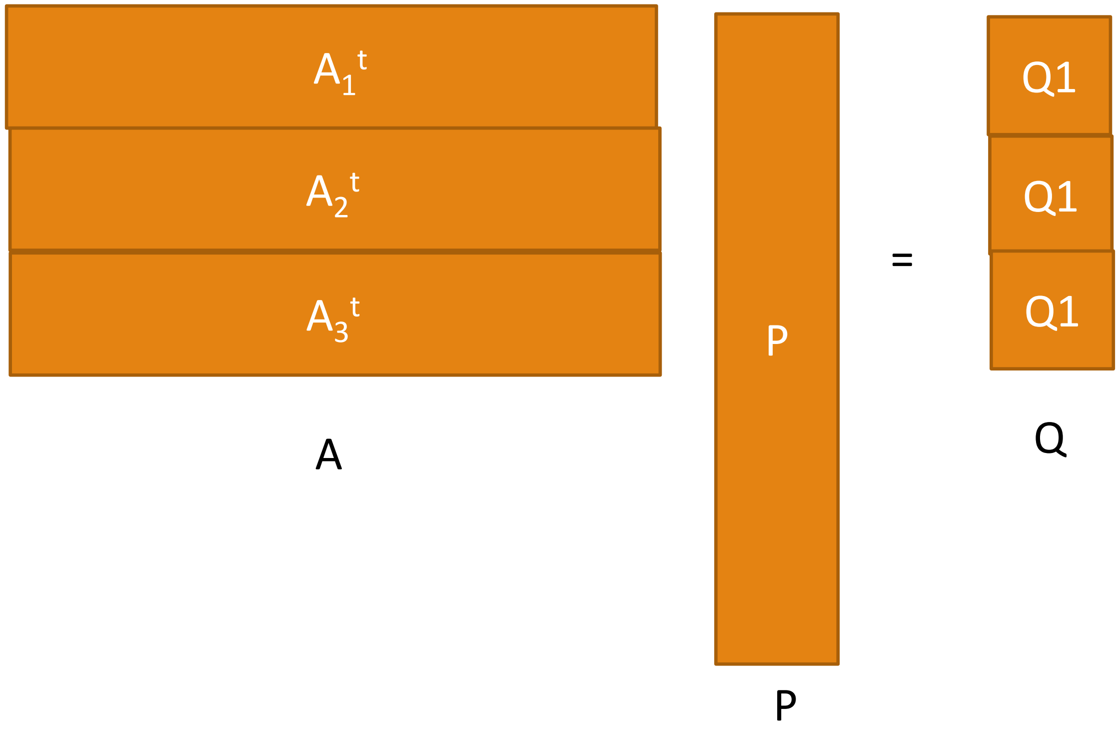
$Q=A^t*P$

For k=1,2,3…..

set (A_k to dA);

$Q_k=A_k^tP$;

end



Out-of-Memory GPU Implementation manual pipelining

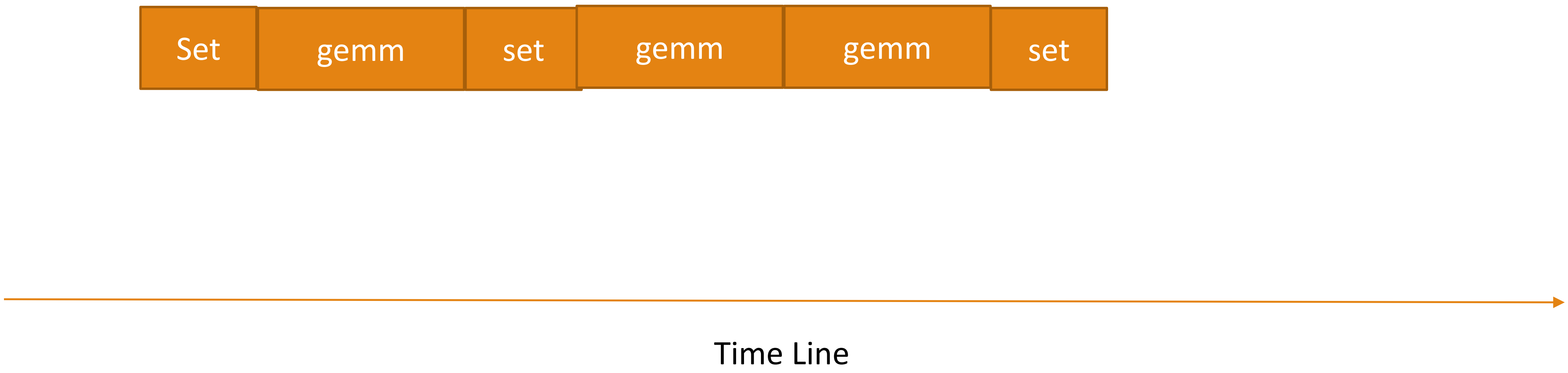
NB: the number of rows of A_i

calling *cudaMemGetInfo()*

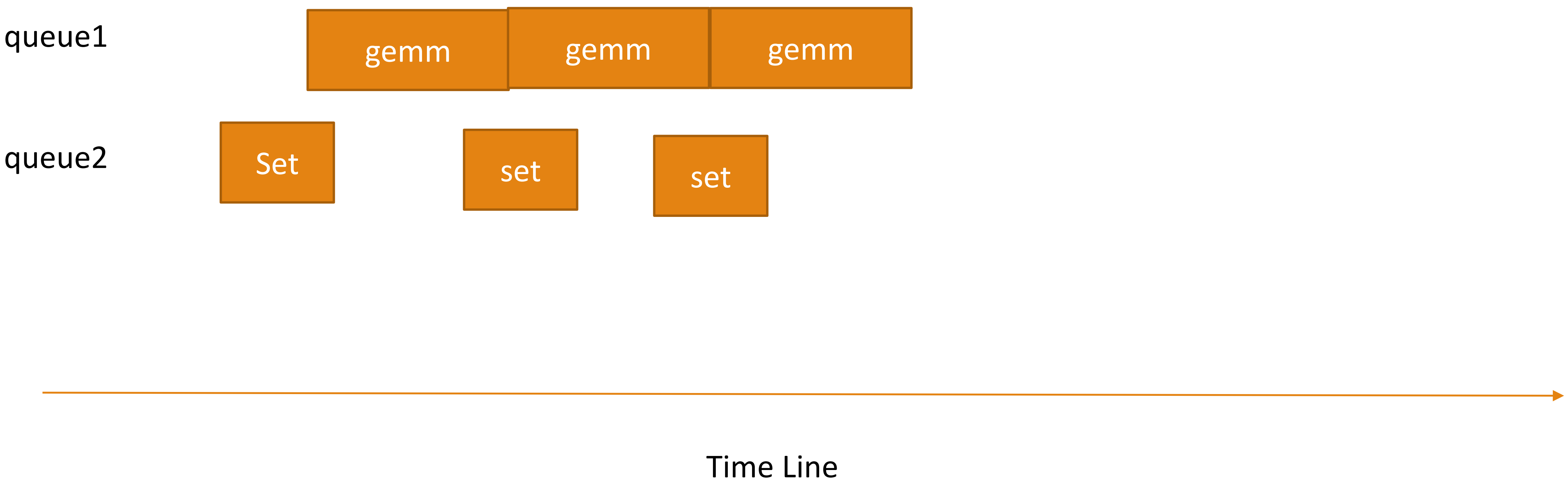
*NB = (0.8 * (freeMem/sizeof (magmaDoubleComplex)))/ (N * num_queues);*

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

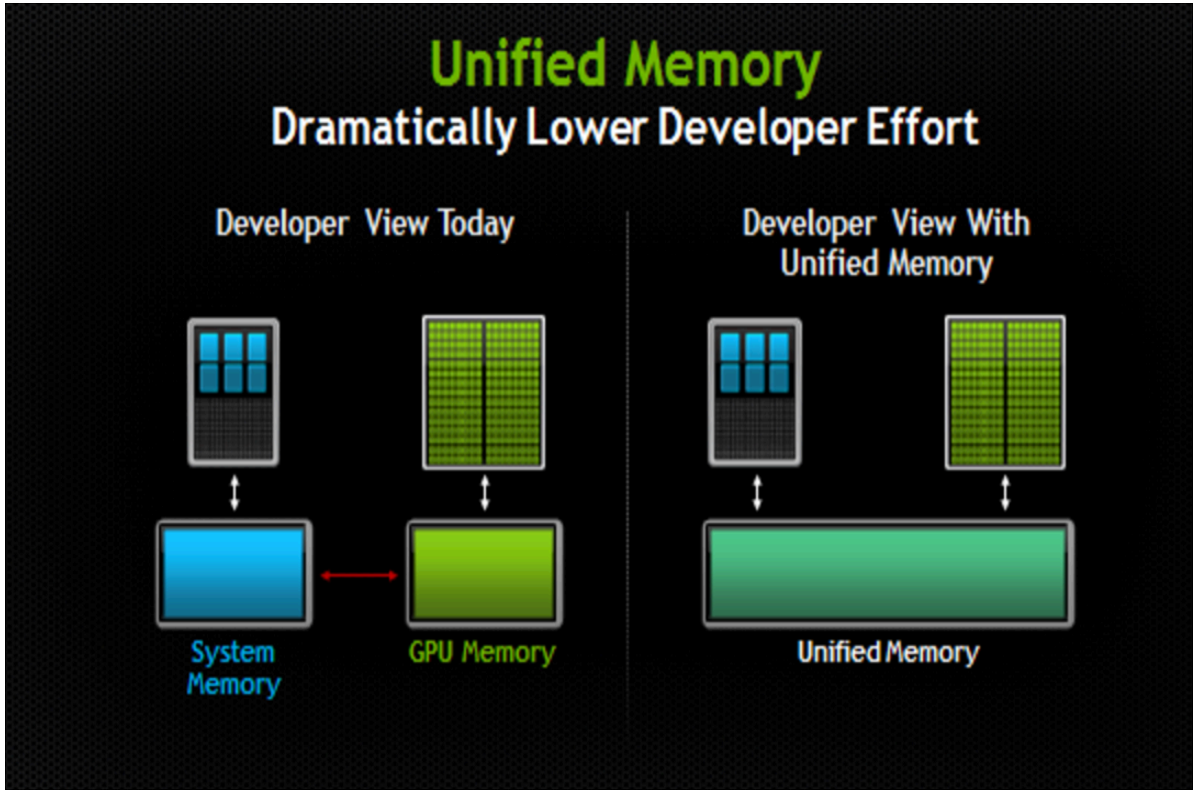
Out-of-Memory GPU Implementation manual pipelining



Out-of-Memory GPU Implementation manual pipelining



Out-of-Memory GPU Implementation UMA

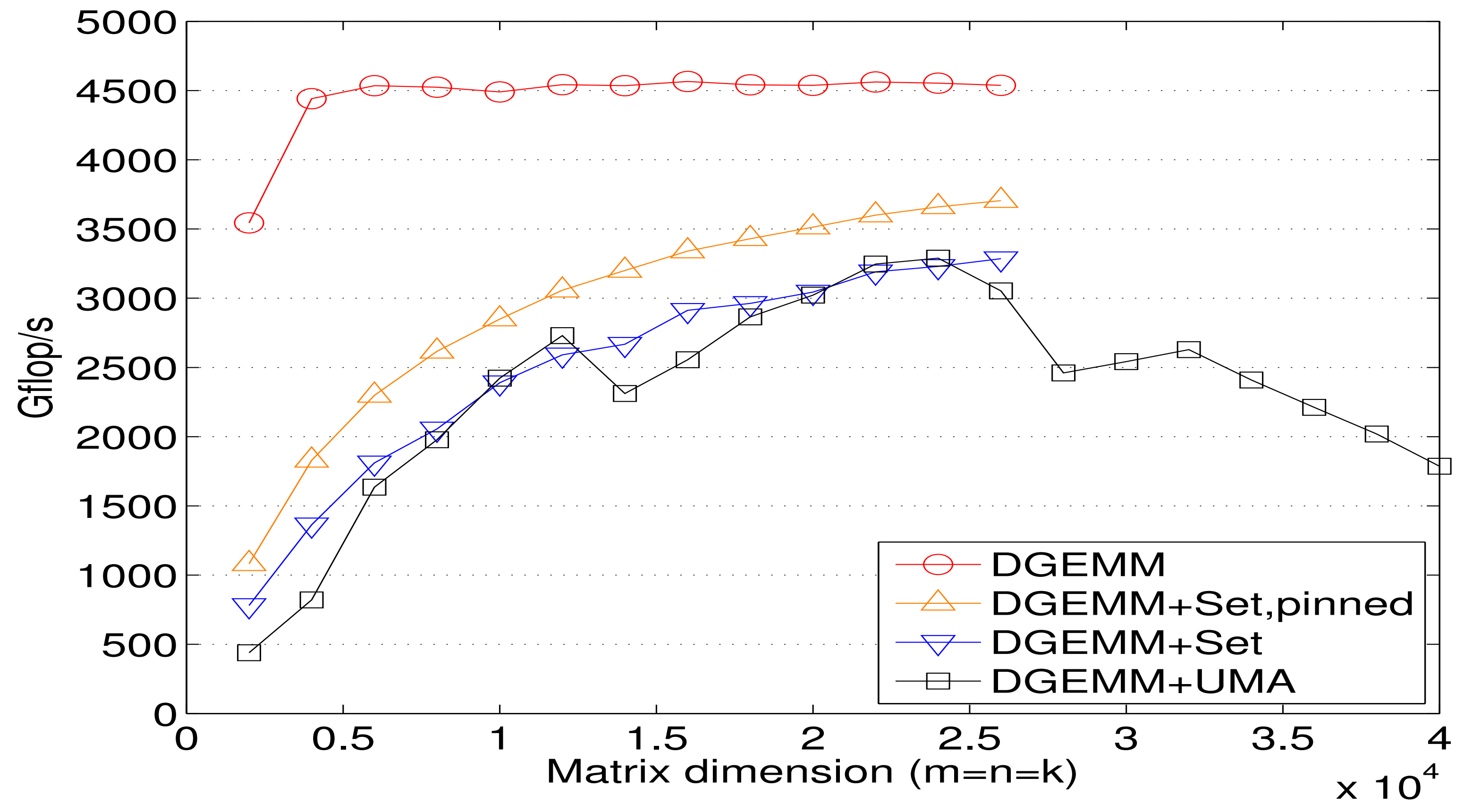


UMA

Unified Memory Access.

Unified Memory creates a pool of managed memory that is shared between the CPU and GPU.

Out-of-Memory GPU Implementation UMA



Out-of-Memory GPU Implementation CUBLAS-XT

- NVIDIA cuBLAS library : a fast GPU-accelerated implementation of the standard basic linear algebra subroutines (BLAS).

- accept arrays on CPU and break up the matrix on CPU into blocks and perform data transfer and computations on GPU.

- multiple GPUs on the same node

Performance Results

‘zgesvd_rand_cpu.cpp ’: CPU

‘zgesvd_rand.cpp’:in-core on GPU

‘zgesvd_rand_m.cpp’ : out-of-core using manual pipelining

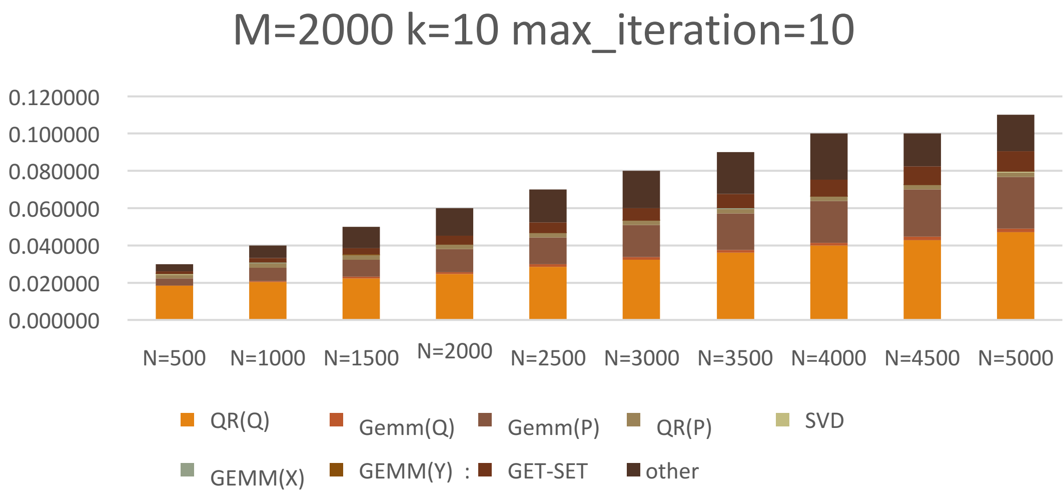
‘zgesvd_rand_uma.cpp’: out-of-core using UMA&CUBLAS.

Performance Results

Name	Steps
QR(Q)	[q,r] = qr(q,0);
Gemm(Q)	q = A'*p
Gemm(P)	p = A*q;
QR(P)	[p,b] = qr(p,0)
SVD	[x,s,y] = svd(b)
Gemm(X)	u = p*x(:,1:k)
Gemm(Y)	v = q*y(:,1:k)
GET-SET	setmatrix and getmatrix

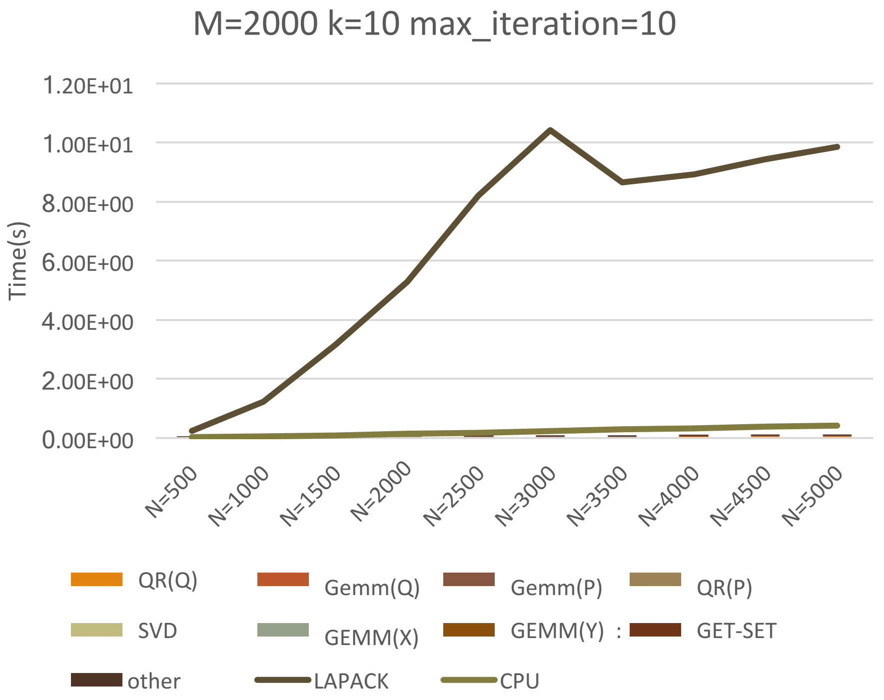
Comparison 1: M=2000, k=10, max_iteration=10 change N

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



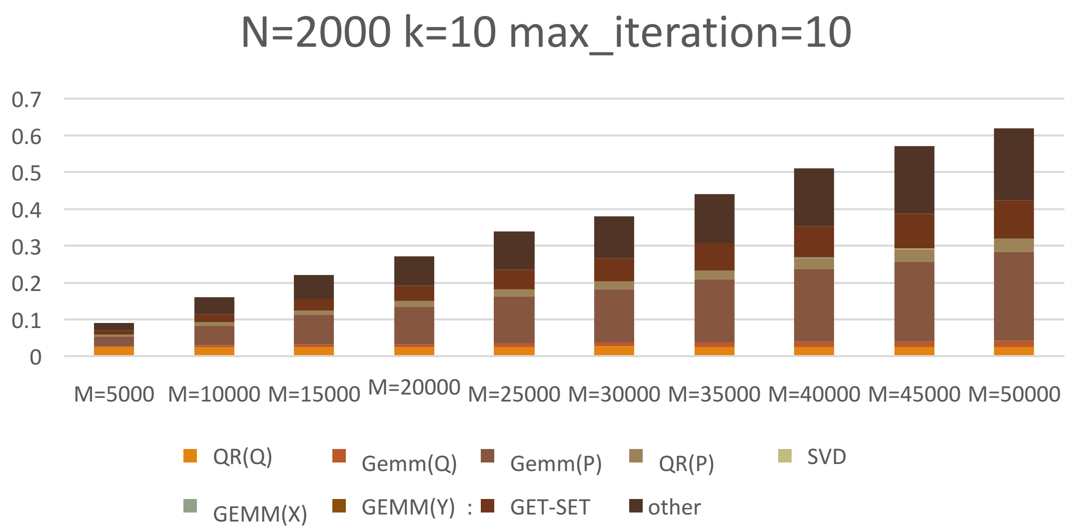
Comparison 1: M=2000, k=10, max_iteration=10 change N

	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



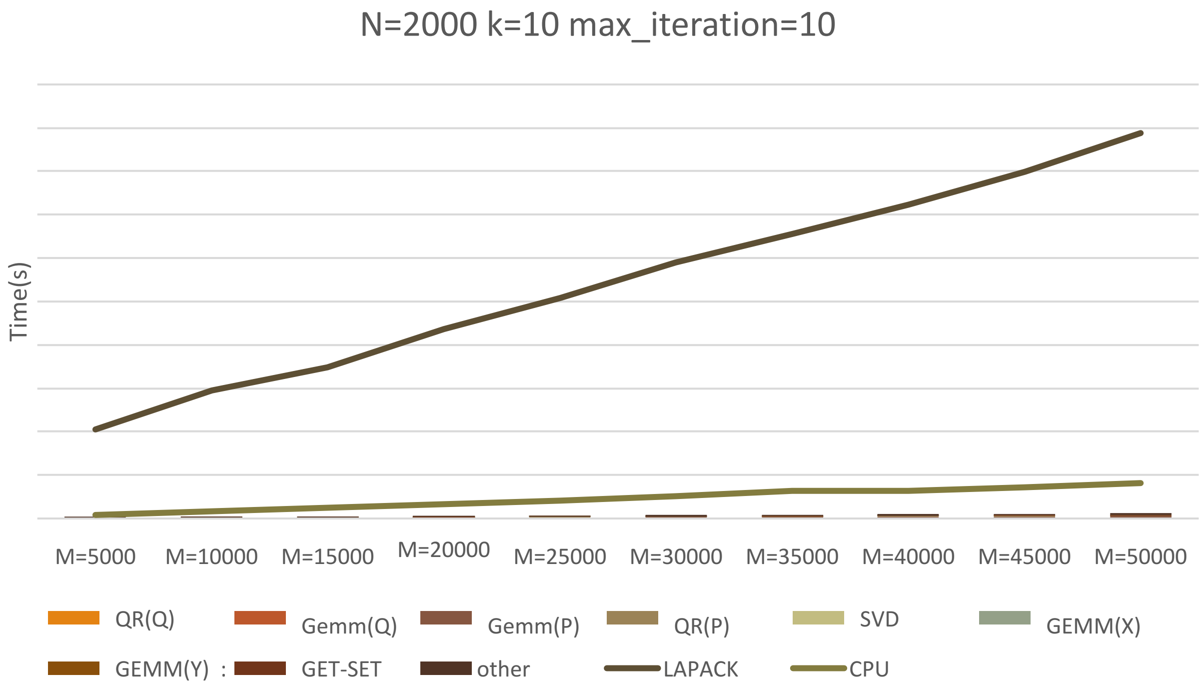
Comparison 2: N=2000, k=10, max_iteration=10 change M

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



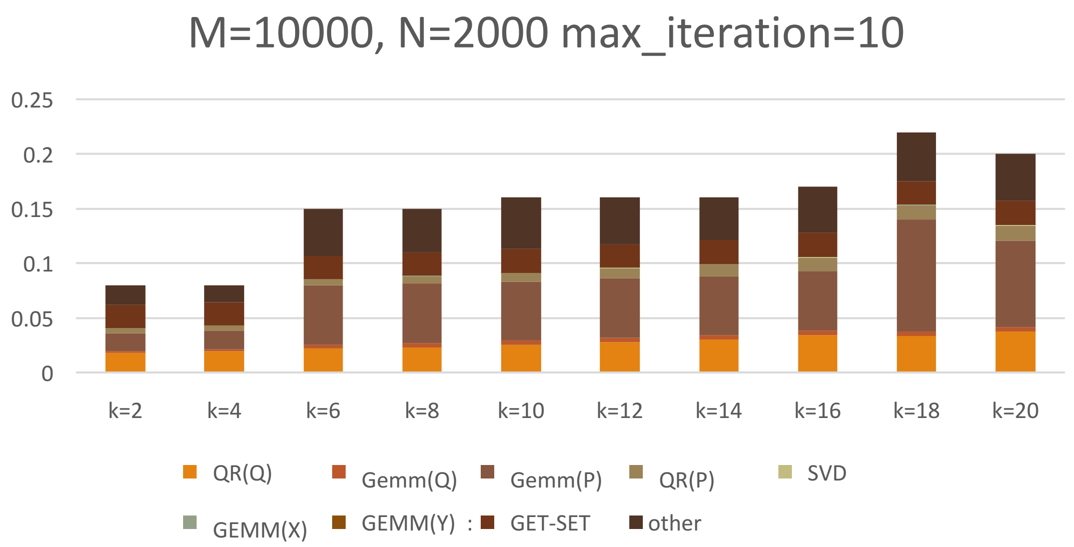
Comparison 2: N=2000, k=10, max_iteration=10 change M

	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



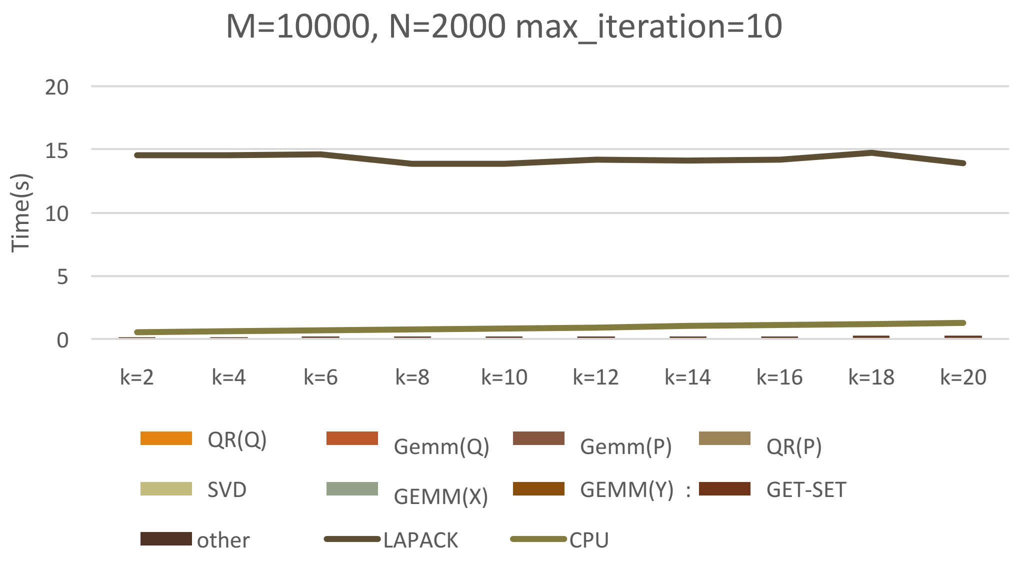
Comparison 3: M=10000, N=2000, max_iteration=10 change k

	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



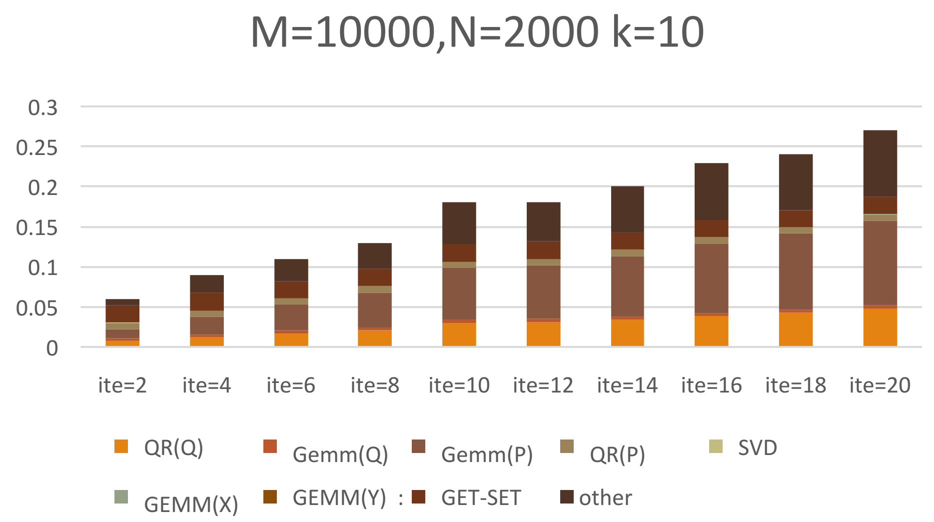
Comparison 3: M=10000, N=2000, max_iteration=10 change k

	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



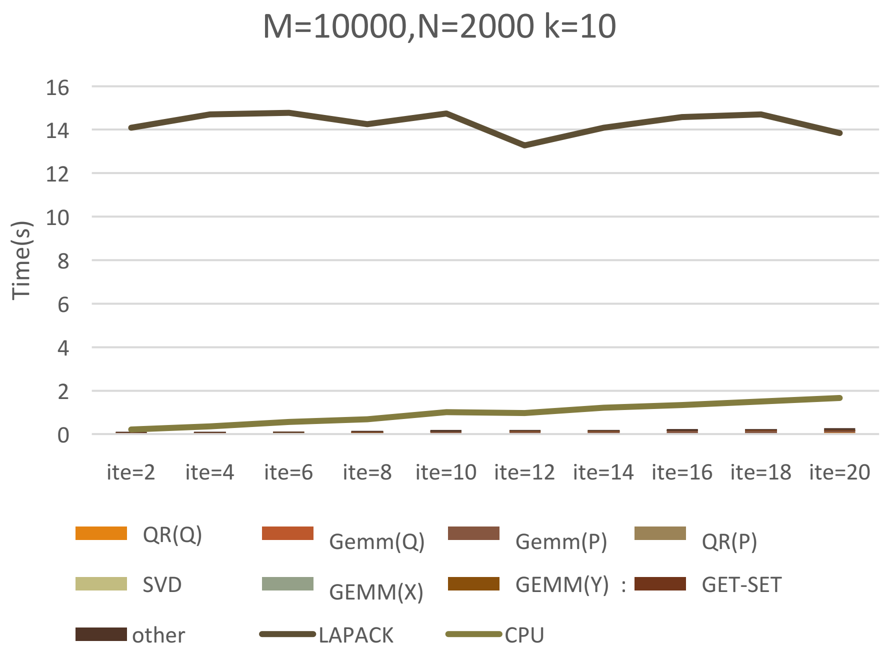
Comparison 4: M=10000,N=2000 k=10 change max_iteration

	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



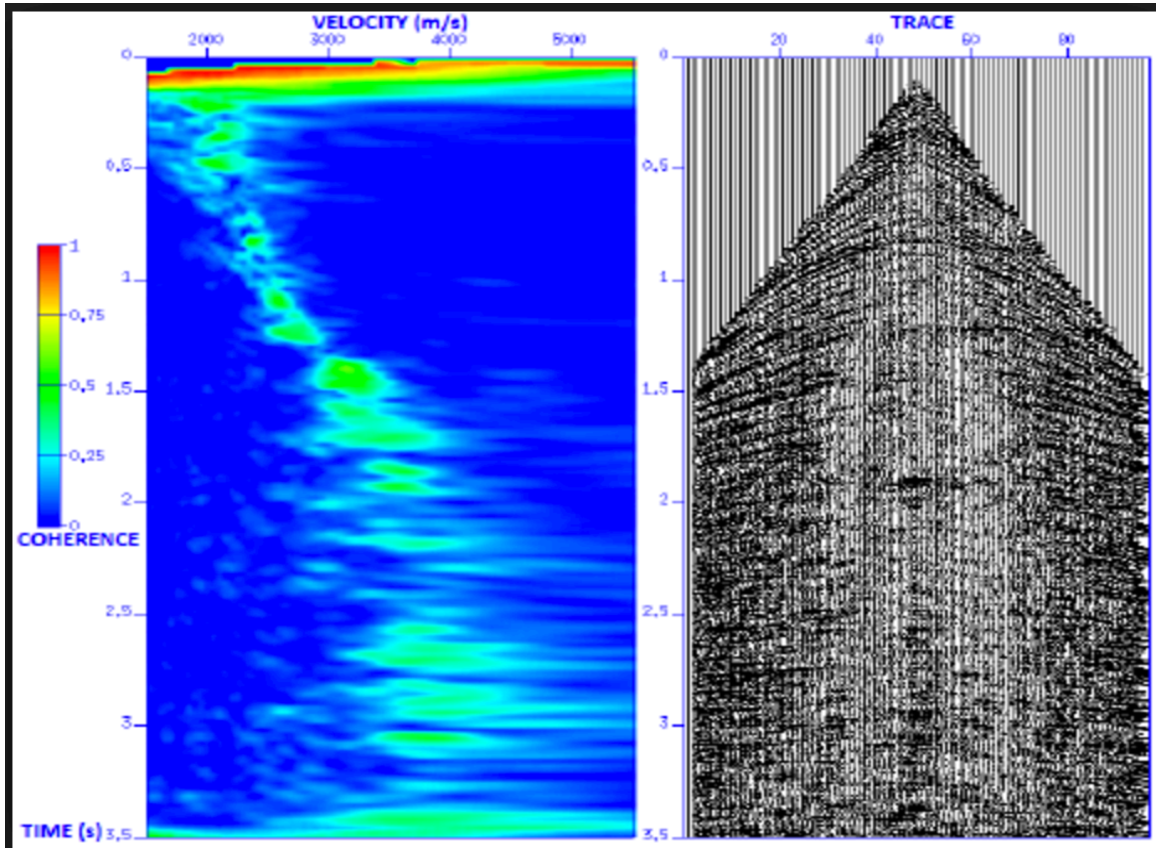
Comparison 4: M=10000,N=2000, k=10 change max_iteration

	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



Motivation and Application

- Latent Semantic Indexing (LSI)
- Genetic clustering
- subspace tracking
- image processing



Future Work

multiple GPU: CUBLAS-XT

randomized sampling and updating methods

Acknowledgements

This project is sponsored by

Oak Ridge National Laboratory

Joint Institute for Computational Sciences

University of Tennessee

The Chinese University of Hong Kong

Most sincere gratitude to my mentors:

Dr. Ed D‘Azevedo’s,   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].