An Implementation of Partial Random Butterfly Transformation (PRBP) in Solving Dense Linear System on Graphical Processing Units

Reference: Accelerating Linear System Solutions Using Randomization Techniques
Baboulin, Dongarra, Herrmann and Tomov, 2013

Methods:

This project implements a linear system solver using partial random butterfly transformation (PRBP). The classic solver for this problem (Gaussian Elimination or LU decomposition step) requires pivoting. In parallel programming environment, this creates high communication burdens between processors. This new implements is aim to using randomization techniques to shuffle the matrix in order to stabilize the Gaussian Elimination without pivoting.

This random butterfly transformation works by generating random number filled, recursive butterfly matrices U and V. As the butterfly matrix is sparse, to be memory efficient, we only store them in an n-length array. The authors point out that the depth of recursion can be set to 2 to accommodate most of the cases. So in order to generate two such matrices, we need four n-length arrays. I use cuRand, the CUDA-library optimized for generating random numbers on devices, to do this step.

Then for a linear system as Ax = b, we calculate the randomized matrix $A_r = U^tAV$. Then we solve the system $A_ry = U^tb$. The final solution is x = Vy.

I have used library optimized for sparse matrix cuSparse, a CUDA-optimized library for operating sparse matrices, to do the recursion and some calculation of the following matrices. But, unfortunately this library does not support matrix multiplication by having sparse matrix on the right side and dense matrix on the left hand. I have to switch back to dense matrix multiplication by using cuBLAS Level 3 functions. The LU decomposition and triangular linear system solving is also done in cuBLAS.

To access the numerical correctness of this method, I generate the solution myself and compute out the b as the input. I compare the solution I got with the ground truth x and report L1-norm. Notice the cuBLAS has a ready-to-use function to solve the least square problem in linear system. It is not what we want to do here and I don’t know if pivoting cannot be disabled in it.

So the over algorithm works in three stages:

STAGE 1: a. Generate random numbers in the device memory (cuRand)

b. Scale and transform the random numbers (generic, block size =1024)

c. Recursive butterfly matrix multiplication (cuSparse)

STAGE 2: a. Use cuSparse and cuBLAS to calculate A_r and $U^t b$

STAGE 3: a. Use LU decomposition (w/o pivoting) and triangle solver to solve

$$A_r y = U^t b$$

I have tried to use the matrix inversion here, but somehow it does not work. And after all, the goal is not to inverse the matrix. The cost is more expensive.

b. Get solution by x = Vy

Environment:

As long as the CUDA toolkit 6.0 is loaded, the code should work properly.

Command “make” generates four executable files:

prbp: the floating version

prbpd: the double precision version

prbp_norand: the couble precision version w/o random butterfly step

prbp_dr: the floating version + use double precision random number

The command is executed as :

./prbp <size of matrix> <seed number for butterfly matrix generation>

Performance:

FLOAT	1024	2048	4096	8192	1024	2048	4096	8192
Run1	770.143188	2727.86963	15678.4023	121139.055		2.03E-02	1.03E-01	
Run2	768.270508	2700.50415	15897.4111	121052.57	1.75E-02	2.70E-02	2.03E+00	3.46E-01
Run3	766.419312	2712.06812	15706.6973	121150.695	1.05E-02	3.35E-02	3.32E-01	3.46E-01
Run4		2715.28149	15673.9639	121130.586	1.15E-02	1.74E-02	6.03E-01	3.46E-01
Avg	768.277669	2713.93085	15739.1187	121118.227	0.01318715	0.02456982	0.76738233	0.34570807
SD	1.86194833	11.2525531	106.519594	44.5403484	0.0038079	0.00718402	0.86728918	0.00025282

DOUBLE	1024	2048	4096	8192	1024	2048	4096	8192
Run1		2797.50317	18287.5781	145886.406		3.20E-10	7.85E-10	1.48E-09
Run2	793.070251	2769.74487	18308.9023	146062.141	5.15E-11	1.57E-09	4.09E-10	7.90E-10
Run3	791.605469	2799.92041	18275.582	146060.359	6.85E-11	2.23E-09	4.09E-10	7.90E-10
Run4	789.936768	2783.9126	18248.7363	145950.734	5.33E-11	1.76E-09	4.09E-10	7.90E-10
Avg	791.537496	2787.77026	18280.1997	145989.91	5.7778E-11	1.4686E-09	5.031E-10	9.614E-10
SD	1.56784699	13.9302057	25.0968097	86.4641226	9.2869E-12	8.1454E-10	1.8808E-10	3.4328E-10

So first it is noticeable that the floating version is quite inaccurate, while surprisingly, the double precision version is not much slower. In general, when n is in the range of several thousand, we lost about 6 magnitude precisions, which is about n square.

Without using the randomized butterfly scheme generates wrong result:

NORAND	1024	2048	4096	8192
Time	268.226196	2009.30701	16821.9395	139108.859
Error	2.54E+00	2.24E+00	4.65E+00	1.65E+00

And the time difference is the cost we pay for the random butterfly operations. The overhead is significant when the size of the matrix is small. In the largest matrix case, we are barely paying 5% more time to do the randomized butterfly scheme.

To make sure our randomization scheme does not affect the precision I tried to use a double precision random number in single floating number version (prdb_dr). The error does not change.

The time complexity of linear system solving is $8/3n^3 + 12n^2 + 4n/3$. Based on this, the GFLOPS is barely ten. I did not run any profiling on this implementation.

To-Dos

This is a very simple implementation of the algorithm that is described in the paper. Without MAGMA and LAPACK installed, I did not do what the authors described as “hybrid system”. Almost the entire work is done on GPU. I believe the authors were using CPU to do the more refined panel factorization while GPU is responsible for updating the trailing matrix. This is the method that LAPACK is using.

Also in the authors’ latest work, they introduced an advanced queue system: QUARK to further optimize the work coordinating between GPU and CPU.