Report of Linear Solver Implementation on GPU

XIANG LI

Abstract

As the development of technology and the linear equation solver is used in many aspects such as smart grid, aviation and chemical engineering, therefore, it is indispensable to develop an efficient and fast-speed algorithm based on the GPU implementation. GPU is short for Graphic Process Unit, it is not only very efficient manipulating computer graphics but also because of its highly parallel structure makes it more effective than general-purpose CPU for algorithms where processing of large blocks of data is done in parallel.

In this report, we mainly discuss three methods for solving the linear equation Ax=b where A is a matrix and b and x are both vectors. The three methods discussed in this report are Conjugate Gradient Algorithm, Preconditioned Conjugate Gradient Algorithm, and LU Decomposition. As for GPU, One of the best of GPU is that it can reduce the calculation time especially for the large size of matrix. In the following tests, you can see it is true. In this report, all the test data are from real power system and they are all sparse positive definite symmetric matrixes. The device I used is ASUS GEFORCE GTX TITAN which is the newest product in the market and it can do the calculation using dynamic parallelization. And all the running time is calculated by CUDA Event Library.

Contents

1 Conjugate Gradient Algorithm

Conjugate Gradient Algorithm is based on the descent method, that is, it approximates the ultimate solution descending from the initial solution. So, whats the most significant is how to determine the descent direction. In terms of the steepest descent method, the direction is the negative gradient. But the weakness is the slow convergence, always in need of hundreds of iterations. This problem is solved by the Conjugate Gradient that we choose the conjugate vector of the matrix as the descent direction. With this method, we can get the accurate solution within N iterations for the N-order matrix. Seemingly, the algorithm as stated requires storage of all previous searching directions and residue vectors, as well as many matrix-vector multiplications, and thus can be computationally expensive. However, a closer analysis of the algorithm shows that r_{k-1} is conjugate to p_i for all $i < k$, and therefore only r_k, p_k, and x_k are needed to construct r_{k+1}, p_{k+1}, and x_{k+1}. Furthermore, only one matrix-vector multiplication is needed in each iteration.

The algorithm is detailed below for solving $Ax = b$ where A is a real, symmetric, positive-definite matrix. The input vector x_0 can be an approximate initial

solution which are all zeros.

$r_0 := b - Ax_0$

$p_0 := r_0$

$k := 0$

repeat

$\alpha_k := \frac{r_k^T r_k}{p_k^T A p_k}$

$x_{k+1} := x_k + \alpha_k p_k$

$x_{k+1} := r_k - \alpha_k A p_k$

if r_{k+1} is sufficiently small then exit loop

$\beta_k := \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$

$p_{k+1} := r_{k+1} + \beta_k p_k$

$k := k + 1$

end repeat

The result is $x_k + 1$

The implementation of conjugate gradient algorithm for GPU is just like this. But when it comes to matrix calculation, the C++ program must call functions from CUBLAS which is a CUDA Library for linear solver calculation. Moreover, CUDA has CUSPARSE Library which is designed especially for sparse matrix to reduce the complication when solving problems about it, Because sparse matrix is compressed in a certain format(CSR) to make it much more easier to manipulate in CUDA.

1.1 Testing Result

After lots of tests using different matrixes, I found that the condition number of the matrix A effects the iteration of conjugate gradient algorithm a lot. So the final test data I used are divided into three groups by their different condition number which are low, medium and high condition number. Here are the results of all the test data.

Table 1: Result of Low Condition Number

Matrix Size	Iteration	Running Time(ms)
99*99	11	3.37
198*198	11	3.86
498*498	11	3.88
999*999	12	4.15
5001*5001	12	4.26

Table 2: Result of Medium Condition Number

Matrix Size	Iteration	Running Time(ms)
102*102	51	12.83
504*504	63	15.31
1875*1875	92	21.55
5001*5001	95	22.53

Table 3: Result of High Condition Number

Matrix Size	Iteration	Running Time(ms)
102*102	54	13.42
504*504	90	21.06
1875*1875	109	25.08
5001*5001	110	25.63

As we can see from the charts, for the same condition number of different size of matrixes, the iteration and running time are getting bigger as the size of matrix goes up except for the low condition number. My explanation for this is that for low condition number, the sparse matrix is almost the same for different size of matrix. All the test data are composed of three by three blocks and some of them are the same. This explanation is based on the fact that if the test data is like diagonal blocked matrix for conjugate gradient algorithm, what really matters is each block which the calculation really based on. Here comes the result that the more kind of blocks are in the matrix, the more iteration will be.

Here are some diagrams for the result in the chart. So you can see the tendency of the result more clearly.

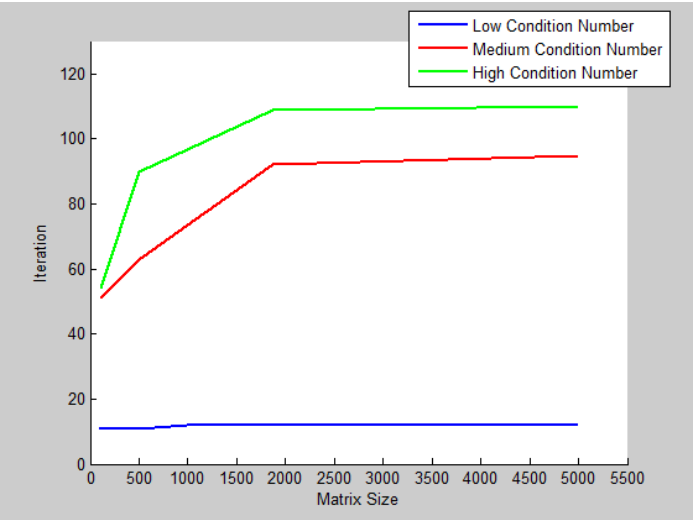


Figure 1: Iteration Result Comparison between Different Condition Number

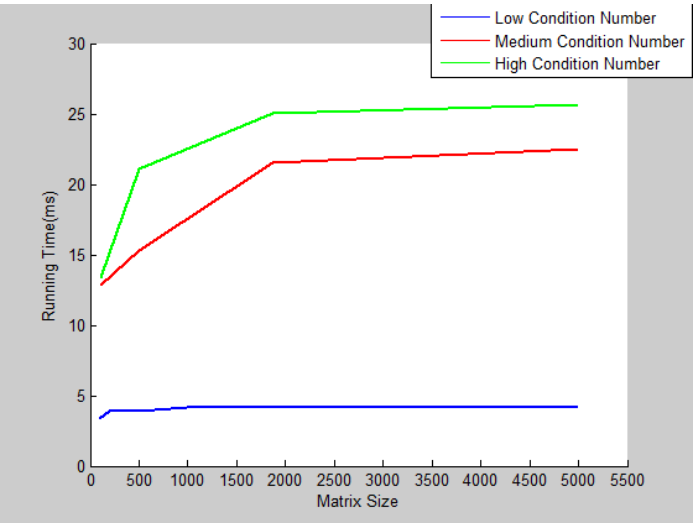


Figure 2: Running Time Result Comparison between Different Condition Number

As the results shown above, the condition number definitively decide the iteration and running time for different matrix. And as the size of matrix goes up, the iteration and running time increase either.

1.2 Resource Distribution

In this part we will discuss about the resource distribution in CUDA. I use a tool in CUDA named Nsight to see the resource distribution of GPU. Here are what I got.

Figure 3: CUDA Resource Distribution Progress

As you can see from the graph above. I use Nsight to see the resource distribution which is distributed by CUDA. It is not the real distribution in GPU. The physical resource distribution could change if you use different type of GPU. And if you want to know the GPU real distribution such as threads and grids used in GPU, the only way is to understand the assembly language of GPU which is extremely hard. So here we are using the abstract resource distribution CUDA provided us.

The distribution is based on the kernel function which might be called in the code.

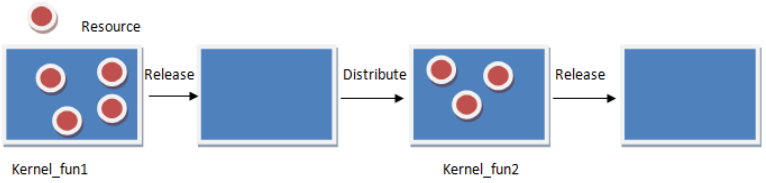


Figure 4: Example of Resource Distribution based on Functions

As you can see from the graph above. If the kernel function1 runs out, then the resources will be released and give it to the next kernel function. Then the resource distribution for the code is based on the how many function calls it used. The distribution for one single function is different if we use the different size and shape of the matrix. Since we can not get how many thread or grids using in one program, what we can get is the cores using in each program. Here are the number cores used for different matrixes in CUDA.

Table 4: Resource Distribute of Low Condition Number

Matrix Size	Cores (Total 2688)	Occupancy
99*99	102	3.79%
201*201	102	3.79%
501*501	102	3.79%
999*999	111	4.13%
5001*5001	111	4.13%

Table 5: Resource Distribute of Medium Condition Number

Matrix Size	Cores (Total 2688)	Occupancy
102*102	462	17.19%
504*504	570	21.21%
1875*1875	831	30.94%
5001*5001	858	31.92%

Table 6: Resource Distribute of High Condition Number

Matrix Size	Cores (Total 2688)	Occupancy
102*102	489	18.19%
504*504	813	30.25%
1875*1875	984	36.61%
5001*5001	993	36.94%

As you can see from the tables above. The GPU cores a whole program used are varied from different type of matrixes. And if you compare it with the iteration of CG you can easily find out that the cores it used is positively related to the iteration times.

	Name	Launches	Device %
1	axpy_kernel_val<float,int=0>	174	0.00
2	dot_kernel<float,int=128,int=0,int=0>	117	0.00
3	reduce_1Block_kernel<float,int=128,i...	117	0.00
4	csrMv_kernel<float,int=128,int=1,int=...	59	0.00
5	scal_kernel_val<float,float,int=0>	57	0.00
6	copy_kernel<float,int=0>	1	0.00

Figure 5: Kernel Functions Launch Time

Function Name	Grid Dimensions	Block Dimensions	Registers per Thread	Static Shared Memory per Block (bytes)	Dynamic Shared Memory per Block (bytes)
csrMv_kernel<float,int=128,int=1,int=2>	(16, 1, 1)	(4, 32, 1)	18	528	256
axpy_kernel_val<float,int=0>	(2, 1, 1)	(256, 1, 1)	14	0	0
dot_kernel<float,int=128,int=0,int=0>	(4, 1, 1)	(128, 1, 1)	11	512	0
reduce_1Block_kernel<float,int=128,int=7>	(1, 1, 1)	(128, 1, 1)	11	768	0
copy_kernel<float,int=0>	(2, 1, 1)	(256, 1, 1)	8	0	0
scal_kernel_val<float,float,int=0>	(2, 1, 1)	(256, 1, 1)	8	0	0

Figure 6: Resource Distribution for different kernel functions

As for the threads, blocks and memory , the distribution is based on the functions. Here is a result for the matrix which condition number is medium and size is 504*504. It used 525 CUDA cores in total and called six kernel functions. For example, the function name which is csrMv kernel<float,int=128,int=1,int=2>, it used 16 blocks and 4*32 threads. In each thread it used 18 registers. And the static shared memory per block is 528, dynamic shared memory per block is 256.

Figure 7: Resource Distribution for different kernel functions

Here is a result for function named csrMv kernel<float,int=128,int=1,int=2>. In my opinion, the achieved in the graph means the blocks or the warps it really used and theoretical in the graph means the space this function allocated but may not be used. And finally The active warps(32 threads) in each clock is 4.28.

1.3 Conjugate Gradient Algorithm for Dense Matrix

I also have the c++ program of dense conjugate gradient algorithm. But I did not do the whole test for dense conjugate gradient algorithm for two reasons. First of all, conjugate gradient algorithm is working pretty well with sparse matrix. So maybe CG is designed for sparse matrix. Secondly, I do not have proper test data to test it. If the matrix A is randomized, the condition number of it is uncertain which might lead to inaccurate result of the test.

To finish dense conjugate gradient algorithm, what I have done is converting the CUSPARSE functions to CUBLAS fucntions. Because in CUDA, the format for dense matrix and sparse matrix is different. The function to do multification of matrix is different either.

1.4 Conclusion

As a result, the conjugate gradient algorithm is working really well for sparse matrix. To be more specific, there are three conclusions.

1) The iteration time is indeed related to the condition number but it also have great thing to do with the block. All the abnormal part of the result is because the matrix maybe copied from one block to another. This kind of copy reduce the iteration and running time for CG. This kind of matrix do exist in the real power system.

2) The second result is that as the size of matrix goes up, the more it grows up, the less iteration time will grow up. That proves that the CG is much better in the large size of sparse matrix .

3) The last conclusion we can draw from the resource distribution is the cores using in every data is related to the iteration time of every test data.

2 Preconditioned Conjugate Gradient Algorithm

Preconditioned Conjugate gradient algorithm is also a method to solve linear equation Ax=b. The convergence of the iterative methods depends highly on the spectrum of the coefficient matrix and can be significantly improved using preconditioning. The preconditioning modifies the spectrum of the coefficient matrix of the linear system in order to reduce the number of iterative steps required for convergence. It often involves finding a preconditioning matrix M. It is assumed that M is the result of decomposition which is also symmetric positive definite. There are several ways to do the decomposition, such as incomplete Cholesky factorization (what I use), LU factorization and so on. As for Cholesky factorization, when M is available in the form of an incomplete Cholesky factorization, the relationship between A and M is shown as followed.

$$M = LL^T$$

After doing the preconditioning, the matrix M will be conveyed to the loop of conjugate gradient program to do the convergence. The pseudocode for Preconditioned

CG program is shown as followed.

Letting initial guess be x_0, compute $r = f - Ax_0$

For $i = 1, 2$...until convergence do

Solve $Mz = r$

$\rho_i = r^T z$

If $i == 1$,then

$p = z$

else

$\beta = \frac{\rho_i}{\rho_i - 1}$

$p = z + \beta p$

end if

Compute $q = Ap$

$\alpha = \frac{\rho_i}{p^T q}$

$x = x + \alpha p$

$r = r - \alpha q$

end For

The implementation of preconditioned conjugate gradient method in GPU uses the CUDA library. And for the resource allocation I use NSight in CUDA to find out the cores used in each program for each different matrix which is all the same as conjugate gradient algorithm. But preconditioned conjugate gradient algorithm does precondition the matrix, so the iterative part of the algorithm is a little different.

2.1 Testing Result

Table 7: Test Result of Low Condition Number

Matrix Size	Iteration	Running Time	Cores (Total 2688)	Occupancy
99*99	2	10.87	498	18.53%
201*201	2	12.01	604	22.47%
501*501	2	12.15	604	22.47%
999*999	2	12.32	604	22.47%
5001*5001	3	16.69	627	23.32%

Table 8: Test Result of Medium Condition Number

Matrix Size	Iteration	Running Time	Cores (Total 2688)	Occupancy
102*102	11	14.05	678	25.22%
504*504	11	17.42	834	31.03%
1875*1875	11	20.21	857	31.88%
5001*5001	11	22.91	857	31.88%

Table 9: Test Result of High Condition Number

Matrix Size	Iteration	Running Time	Cores (Total 2688)	Occupancy
102*102	11	15.73	678	25.22%
504*504	11	17.52	834	31.03%
1875*1875	11	20.30	857	31.88%
5001*5001	11	22.93	857	31.88%

We can see from the above charts that the result is quite surprising because the iteration of different size of matrixes sometimes is exactly the same. I found this the first time I test the preconditioned conjugate gradient method using the old data, then I thought it might because the blocks are the same in each matrix. So I find another data to test which the blocks are not exactly the same. But surprisingly the results have no change.

It really surprised me in the first place when I assuming the result of PCG maybe like the result of CG, then to explain this question, I found some information and ask for other people to get the explanation of this result. Now I think maybe

it is the preconditioner that has done some work about the matrix and then the decomposed matrix is not so different from each other. So the iteration is almost the same .

Moreover, even the iteration is the same for different matrixes, the running time and the cores used in GPU vary from each other. So I suppose that the preconditioner does reduce the iteration, but as for running time and cores used, the working for GPU, does not reduce at all.

Carefully looking into these charts, one can find that the cores used in GPU between matrixes of medium condition number and matrixes of high condition number is exactly the same, that is because the kernel functions called is the same, even the launch time for them are the same , which result in the same number cores in use.

2.2 Comparison with Conjugate Gradient Algorithm

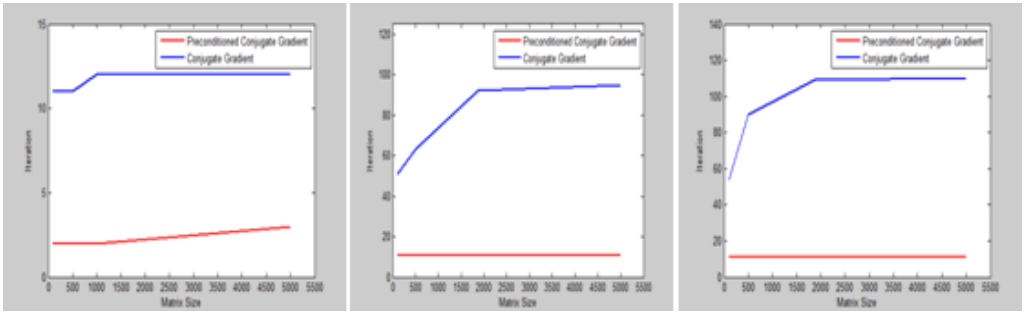


Figure 8: Iteration & Matrix Relation of Three Different Condition Number

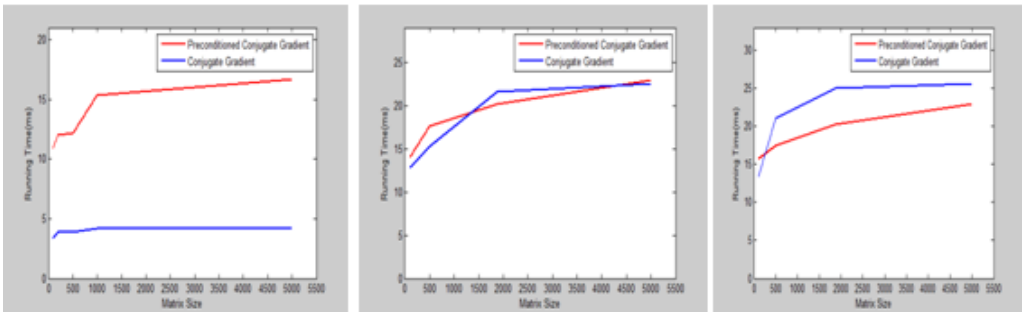


Figure 9: Running Time & Matrix Relation of Three Different Condition Number

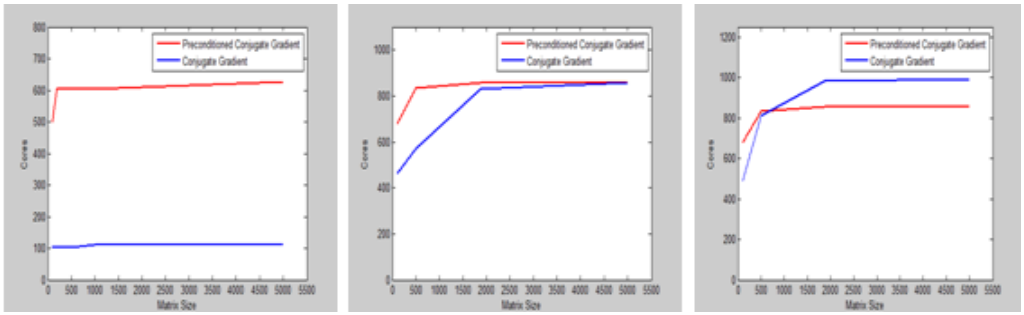


Figure 10: Cores & Matrix Relation of Three Different Condition Number

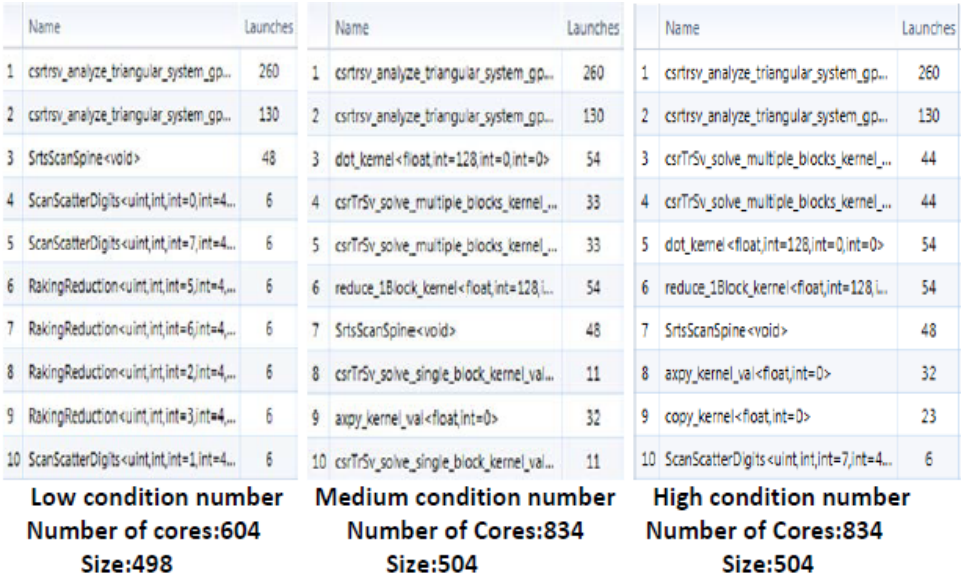
To better understand the advantage of preconditioned conjugate gradient method .I did some comparison between PCG and CG. The graphs were shown above. The first three graphs are about Iteration and Matrix size. The first one is the result of low condition number. The second one is the result of medium condition number and the last one is the result of high condition number. Besides, the blue lines represent conjugate gradient method, red ones represent preconditioned conjugate gradient method. So does the other two groups of graphs.

From the graphs one can draw a conclusion that PCG really decrease the iteration. But as for the time and cores used in GPU, PCG is no better than CG, some time even worse than it, just as I said in last part.

Additionally, if the testing matrix whose condition number is high, just like Figure 10 shows, preconditioned conjugate gradient method is better than conjugate gradient method in all three aspects including iteration, running time and cores used which means PCG works better when the matrix's condition number is high.

2.3 Resource Analysis

In the first part, one can see the number of cores used in GPU is different between matrixes of low condition number and matrixes of medium condition number or high condition number. So to probe this, I decide to do some resource analysis about this to see what effect the number of cores in use. This is the kernel functions called for different matrixes.



	Name	Launches
1	csrtrsv_analyze_triangular_system_gp...	260
2	csrtrsv_analyze_triangular_system_gp...	130
3	SrtsScanSpine<void>	48
4	ScanScatterDigits<uint,int,int=0,int=4...	6
5	ScanScatterDigits<uint,int,int=7,int=4...	6
6	RakingReduction<uint,int,int=5,int=4,...	6
7	RakingReduction<uint,int,int=6,int=4,...	6
8	RakingReduction<uint,int,int=2,int=4,...	6
9	RakingReduction<uint,int,int=3,int=4,...	6
10	ScanScatterDigits<uint,int,int=1,int=4...	6

Low condition number
Number of cores:604
Size:498

	Name	Launches
1	csrtrsv_analyze_triangular_system_gp...	260
2	csrtrsv_analyze_triangular_system_gp...	130
3	dot_kernel<float,int=128,int=0,int=0>	54
4	csrTrSv_solve_multiple_blocks_kernel_...	33
5	csrTrSv_solve_multiple_blocks_kernel_...	33
6	reduce_1Block_kernel<float,int=128,i...	54
7	SrtsScanSpine<void>	48
8	csrTrSv_solve_single_block_kernel_val...	11
9	axpy_kernel_val<float,int=0>	32
10	csrTrSv_solve_single_block_kernel_val...	11

Medium condition number
Number of Cores:834
Size:504

	Name	Launches
1	csrtrsv_analyze_triangular_system_gp...	260
2	csrtrsv_analyze_triangular_system_gp...	130
3	csrTrSv_solve_multiple_blocks_kernel_...	44
4	csrTrSv_solve_multiple_blocks_kernel_...	44
5	dot_kernel<float,int=128,int=0,int=0>	54
6	reduce_1Block_kernel<float,int=128,i...	54
7	SrtsScanSpine<void>	48
8	axpy_kernel_val<float,int=0>	32
9	copy_kernel<float,int=0>	23
10	ScanScatterDigits<uint,int,int=7,int=4...	6

High condition number
Number of Cores:834
Size:504

Figure 11: Kernel Functions Called for Three Different Condition Number

From the figure above, we can find that the functions of matrixes of medium and high condition number are almost the same, so does the launches. But for the matrix of low condition number, the functions and launches are far from the other two. I think thats why the number of cores in use is different from the other two. So we can conclude that the number of cores in use is determined by the kernel functions called and the number they launches.

2.4 Conclusion

I implemented a parallel algorithm of Preconditioned Conjugate Gradient on GPU and then did the comparison between CG and PCG. There are three conclusions that we can draw from the experimental test results.

1) Under normal circumstances, PCG reduce the number of iterative steps, but it has no positive effect for the running time and cores used in GPU comparing with CG.

2) If the matrix A whose condition number is high, PCG performs better than CG in both three aspects. Maybe that is because the condition number of matrix A does not have impact on PCG, the results of medium condition number and high condition number are just the same which include iteration and cores used.

3) Last but not least, the number of cores used in GPU depends on the kernel functions called and the time they launches in the program.

3 LU Decomposition

LU Decomposition is one of the method to solve linear equations Ax=b especially when A is asymmetric. LU Decomposition with partial pivoting which is used for factoring general asymmetric matrices. Without pivoting, LU decomposition is quite similar to Cholesky in the last section when A is symmetric, positive and definitive. In practice, partial pivoting provides accurate solution to this problem. Because in GPU, LU Decomposition with partial pivoting is difficult to accomplish. And I do not have enough time to finish it, I only implement the LU Decomposition for dense matrix and specific blocked matrix which are the test date for CG and PCG. As for general sparse matrix, I finally got how to implement thorough reading lots of papers and slides. Moreover, the format for dense matrix and sparse matrix in CUDA is different which makes LU Decomposition for sparse matrix much more harder. The interchange between sparse matrix rows is difficult to accomplish in CSR format. I will introduce what I have done about LU Decomposition.

3.1 Dense LU Decomposition Implementation on GPU

LU Decomposition Implementation for dense matrix is just like in matlab. I found a program on the Internet to solve the LU Decomposition result(L+U) of matrix A, after figure out what it did, I just have to do the triangular solution for equation.

The Matlab Code for LU Decomposition is showed as followed.

function $[A, p] = \text{myplu}(A)$

$[n, n] = \text{size}(A);$

$p = 1 : n;$

for $i = 1 : n - 1$

$[a, k] = \max(\text{abs}(A(i : n, i)));$

if $a == 0$

error('Error:not valic input')

end

$k = k + i - 1;$

if$k \ = i$

$t = A(i, :); A(i, :) = A(k, :); A(k, :) = t;$

$tmp = p(i); p(i) = p(k); p(k) = tmp;$

end

$A(i + 1 : n, i) = A(i + 1 : n, i)/A(i, i);$

$A(i + 1 : n, i + 1 : n) = A(i + 1 : n, i + 1 : n) - A(i + 1 : n, i) * A(i, i + 1 : n);$

end

3.2 Sparse LU Decomposition Implementation on GPU

For sparse matrix, it must be compressed in CSR format, but luckily, there is a function in CUSPARSE to finish the LU Decomposition. But this function is not perfect, but has lots of flaws and can not use as a method to solve general LU Decomposition directly.

3.2.1 Sparse LU Decomposition Implementation on GPU for blocked matrix

But for specific kind of matrix, we can still use this function which is what I did for the test data in CG and PCG. First of all, I will introduce the problem of the function. The function only returns the original format of A instead of the real format because the function think the matrix A format is the same as the format of L+U which lead to the inaccurate result of L+U.

To solve this problem, I have to predict the format of L+U, then make some changes about the matrix A and save some zeros in matrix A. Then the result is accurate. But this method only work for specific matrix which I can predict what the L+U look like. But for general matrix I have to find a method to predict what the matrix will be like before using this function.

3.2.2 General LU Decomposition

For general LU Decomposition, there is a way to predict the output format of L+U which may use graph and I will introduce in the following part. As a matter of fact, in sparse matrix, some onzero values in matrix A is useless, and we can just take it for zero instead which make the decomposition much more easier. The method to prune the matrix named elimination tree. It is used in SUPERLU and is the depth-first search of $G^+(A)$ which I will introduce later.

Symmetric LU Decomposition If the matrix A is symmetric and position definitive, the way to predict the format of L+U is easier than asymmetric matrix. There are several steps to solve the linear equation using LU Decomposition.

STEP 1 convert matrix A to graph G(A)

First of all, there may be some new nozeros in the result of LU Decomposition. For this kind of unkown nozeros, we have to predict the matrix format before calculating the really value. Then we have to use the graph to represent the matrix. For the graph, the column and row number is the vertex in the graph. If there is a nozero value in i column, j row. Then in the graph, there will be a directed edge between vertex i and j. Of course, for symmetric matrix, the diagonal value must not be zero, so the graph will not have loop in each vertex. Let G = G(A) be the graph of a symmetric, positive definite matrix, with vertices 1, 2, , n.

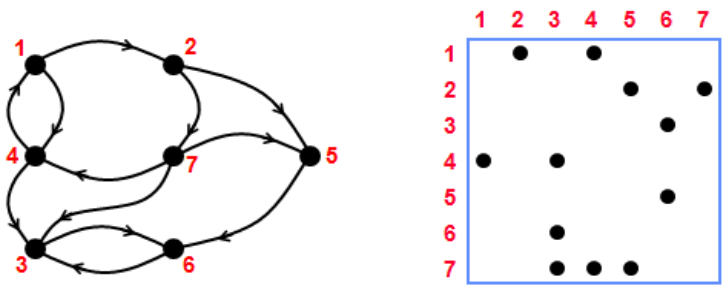


Figure 12: Convert from A to G(A)

STEP 2 Add edges to G(A) to make it G^+(A) which is the result of LU Decomposition

Let G^+ = G^+(A) be the filled graph. Then (v, w) is an edge of G^+ if and only if G contains a path from v to w of the form (v, x_1, x_2, , x_k, w) with x_i<min(v, w) for each i. Like following graph showed

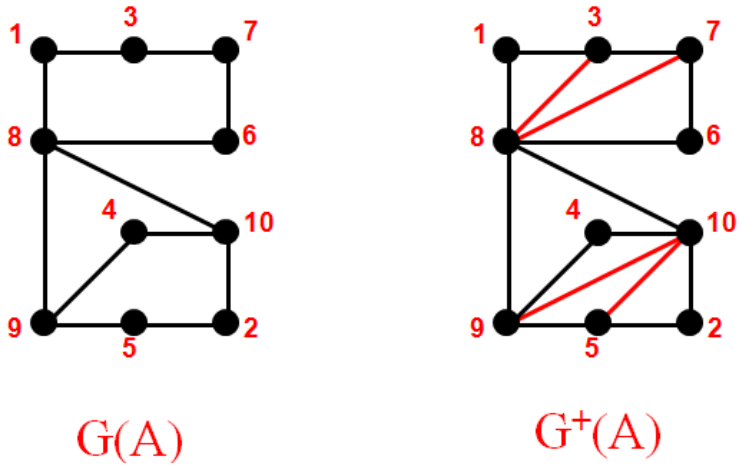


Figure 13: Predict the output format

STEP 3 Convert G^+(A) to matrix A.

After we predicted the format of the result, we can just convert the graph to matrix. And them compress the matrix to CSR format using the function in CUSPARSE.

STEP 4 Triangular Solves

for $r = 1 : n$

$x(i) = b(i)$

for$j = 1 : (i - 1)$

$x(i) = x(i) - L(i, j) * x(j);$

end;

$x(i) = x(i)/L(i, i);$

end;

Asymmetric LU Decomposition For asymmetric LU Decomposition, the algorithm will be much more complicated because since the matrix is not symmetric, we can not just consider part of the matrix. We need to consider more about the matrix and the graph from the matrix will have loops. So the progress will much more complicated.

If there are the result of L+U. There should be more nozeros too, the following graph showed the graph. And the way to add fill edge is just like symmetric matrix. Add fill edge a $->$ b if there is a path from a to b through lower-numbered vertices.

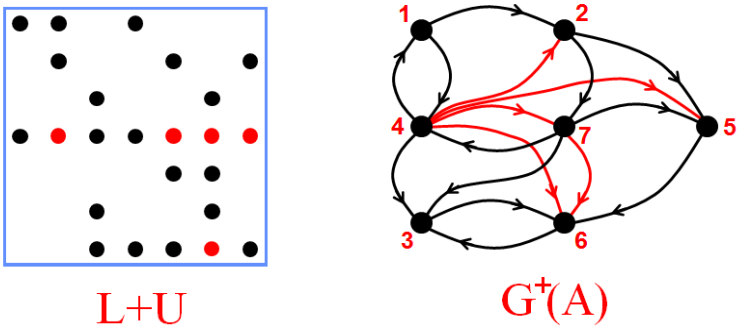


Figure 14: Graphs and Sparse Matrices: LU Decomposition

As we can see from the graph. The result of LU Decomposition can also be predicted through graph. But the conversion from $G^+(A)$ to compressed CSR format will take lots of work to do.

3.3 Conclusion

LU Decomposition is very useful for most matrix because it is general and can solve any kind of linear equations if there is an answer. But it is this kind of generality make it a little difficult to implement. LU Decomposition with partial pivoting can solve nearly every linear system no matter matrix A is symmetric or not. If I have more time, I will spare no effort to finish it.

4 Conclusion

After doing all these work, there are several conclusions we can draw

1) Conjugate Gradient Algorithm works well with sparse matrix especially for large size of matrix.

2) Preconditioned Conjugate Gradient reduce the iteration a lot. When the matrix A has high condition nubmer, PCG performs much better than CG in iteration, running time and cores used in GPU.

3) LU Decomposition is useful because it can be widely used even matrix A is not symmetric , positive definitive. It can be implemented on GPU using C++ and graphs.