Electric Power Systems Research 116 (2014) 87–93

Contents lists available at ScienceDirect

Electric Power Systems Research

journal homepage: www.elsevier.com/locate/epsr

GPU-based power flow analysis with Chebyshev preconditioner and conjugate gradient method

Xue Li, Fangxing Li*

Department of EECS, The University of Tennessee, Knoxville, TN 37996, USA

A R T I C L E   I N F O

Article history:
Received 10 April 2014
Accepted 12 May 2014

Keywords:
Power flow
Parallel processing
CUDA
Graphic processing unit (GPU)
Chebyshev preconditioner
Conjugate gradient method

A B S T R A C T

Traditionally, linear equations in power system applications are solved by direct methods based on LU decomposition. With the development of advanced power system controls, the industrial and research community is more interested in simulating larger, interconnected power grids. Iterative methods such as the conjugate gradient method have been applied to power system applications in the literature for its parallelism potential with larger systems. Preconditioner, used for preconditioning the linear system for a better convergence rate in iterative computations, is an indispensable part of iterative solving process. This work implemented a polynomial preconditioner Chebyshev preconditioner with graphic processing unit (GPU), and integrated a GPU-based conjugate gradient solver. Results show that GPU-based Chebyshev preconditioner can reach around 46× speedup for the largest test system, and conjugate gradient can gain more than 4× speedup. This demonstrates great potentials for GPU application in power system simulation.

1. Introduction

Power flow is the fundamental component in power system analysis and simulation. It is usually modeled as a nonlinear system. The Newton–Raphson method converts this nonlinear system to a group of linear equations with the introduction of Jacobian matrix as Eq. (1) shows.

$$\begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} = \begin{bmatrix} \dfrac{\partial \Delta P}{\partial \delta} & \dfrac{\partial \Delta P}{\partial V} \\ \dfrac{\partial \Delta Q}{\partial \delta} & \dfrac{\partial \Delta Q}{\partial V} \end{bmatrix} \begin{bmatrix} \Delta \delta \\ \Delta V \end{bmatrix} \tag{1}$$

Each iteration in Newton–Raphson method requires solving a set of sparse linear equations. We measured the linear equation solving time and total run time for large systems in MATPOWER. The results show that about 40–50% of the total time is spent on solving linear equations. Therefore improving the efficiency of solving linear system is of great importance for accelerating power flow analysis.

LU factorization, the most commonly used direct method, is widely deployed in solving power flows. However, LU factorization is intrinsically a serial algorithm and difficult to parallelize due to the tight data dependency during factorization. LU factorization will be effective for smaller systems while its performance is limited for larger systems. Leon and Semlyen have proven that the iterative solver can provide about 25% performance improvement over LU direct solver for the systems larger than 3183-bus system [1].

On one hand, iterative methods have been adapted to power system computation in various aspects. Pai et al. [2] have implemented the Generalized Minimal Residual (GMRES) method on a Cray machine for dynamic power system simulation. Pai and Dag [3] further applied several iterative solvers including conjugate gradient and GMRES to dynamic power flow simulation and state estimation. On the other hand, the graphic processing unit (GPU) has been widely adopted in high performance computing recently as a parallel hardware architecture. The GPU was originally designed for graphic displaying and processing. It has massive parallel computing units on board to perform graphic computations. Computational unified device architecture (CUDA) [4] provides a C-like programing interface for users to utilize these computation resources. GPU as a co-processor helps a commodity server deliver more computational throughput.

There has been research about GPU implementation of iterative method in different realms. Helfenstein and Koko implemented a SSOR preconditioner and conjugate gradient solver on GPU to solve generalized Poisson equations [5]. Zhang and Zhang presented a sliced block ELLPACK format to implement a least-square

* Corresponding author. Tel.: +1 865 974 8401.
E-mail address: fli6@utk.edu (F. Li).

http://dx.doi.org/10.1016/j.epsr.2014.05.005
0378-7796/© 2014 Published by Elsevier B.V.

Nomenclature

A	linear system A
b	right hand side of linear equations
x	solution of linear equations $Ax=b$
α	smallest eigenvalue of matrix A
β	largest eigenvalue of matrix A
P	precondition matrix
Z	transformation matrix to shift A's spectrum to $[\alpha, \beta]$
c_k	decay rate
T_k	Chebyshev polynomials
r	degree of Chebyshev polynomials
ratio	artificial condition number
G	approximation of matrix A's inverse

polynomial preconditioned conjugate gradient method for finite element problem [6]. Researches based on high performance computing especially GPU related methods begin to emerge in power system applications too. Garcia implemented a preconditioned biconjugate gradient method with GPU and CUDA [7]. Li et al. [8] discussed the limitation of using direct solver to solve large systems, and introduced GPU-based conjugate gradient normal residual with Jacobi preconditioner. Multifrontal method with CUDA library solved AC power flow was adapted as well [9]. Gopal et al. [10] implemented a DC power flow based contingency analysis with GPU. Kamiabad also did a prototype implementation of a Chebyshev polynomial preconditioner and conjugate gradient method with CUBLAS [11]. However, the speedup for their Chebyshev preconditioner is limited to up to 8×.

In this work, a polynomial preconditioner Chebyshev preconditioner with graphic processing unit (GPU) will be implemented and integrated with a GPU-based conjugate gradient solver for linearized DC power flows. Results show that our GPU-based Chebyshev preconditioner can reach around 46× speedup and the conjugate gradient can gain more than 4× speedup compared with corresponding CPU implementation. This demonstrates great potentials for GPU applications in power systems.

The rest of this paper is organized as follows. Section 2 takes a closer look at iterative solutions and the polynomial preconditioner Chebyshev preconditioner. Section 3 introduces GPU and CUDA technology in detail. Section 4 presents the algorithm that this work uses and the corresponding GPU-based implementation. Computing experiments are shown in Section 5. A further discussion is extended in Section 6. Section 7 closes the whole paper.

2. Iterative solver and preconditioner

2.1. Conjugate gradient

The linear system $Ax=b$ can be solved either directly by using LU decomposition, or indirectly by finding out the minimum value for a quadratic form as Eq. (2) shows:

$$f(x)=\frac{1}{2}x'Ax-b'x+c \tag{2}$$

In Eq. (2), A is a symmetric positive definite matrix and b is a vector. The derivative of Eq. (2) is $f'(x)=Ax-b$. Therefore the x which provides minimum value of Eq. (2) satisfies $Ax-b=0$. Thus, it is the solution of $Ax=b$ as well.

An intuitive way to find out the minimum value of Eq. (2) is to use the steepest descent method. The method will choose the direction that has the greatest change in a small range as the update direction. Steepest descent is straightforward and easy to implement. However, since it picks this direction leading to a local minimum instead of a global minimum in each iteration, there is no guarantee on the convergence rate.

Conjugate gradient, instead, guarantees that the method will converge within n (the size of the system) steps. It is an orthogonal method. Each residual and each newly generated direction vector is A-orthogonal to all the previous selected direction vectors. The A-orthogonality guarantees that the update of current direction is only related to the last step information. A detailed algorithm is showed in Section 4.

2.2. Chebyshev preconditioner

To improve the convergence rate of iterative solver, preconditioner is commonly deployed. A left preconditioner is a matrix that can be left-multiplied to matrix A, and also to vector b correspondingly to reduce the condition number of A. The condition number of symmetric positive definite matrix is defined as the ratio of the largest eigenvalue and the smallest eigenvalue. The larger the condition number is, the more iterations the solver requires. Ideally, the preconditioning matrix P would be the precise inverse of matrix A. However, the cost of computing the inverse of A is usually very high. The goal of a preconditioner is two folds: close approximation of A inverse; and easiness to obtain.

Chebyshev preconditioner is a polynomial based preconditioner. The inverse of matrix A is shown by Eq. (3) in Chebyshev polynomial pattern. Assume α is the smallest eigenvalue of A, and β is the largest eigenvalue, then matrix A has the spectrum of $[\alpha,\beta]$. Z transforms A's spectrum from $[\alpha,\beta]$ to $[-1,1]$, defined as Eq. (4). T_k is Chebyshev polynomial, defined as Eq. (5). These polynomials are orthogonal. c_k and constant q are defined as Eqs. (6) and (7), respectively. c_k is the decay rate for the entries of A^{-1} decaying away from main diagonal of the matrix A. This decay rate can be estimated using constant number q, which is a function of the condition number of A. Detailed discussions of these parameters can be found in Dag's work [12].

$$A^{-1}=\frac{c_0}{2}I+\sum_{k=1}^{k=\infty}c_kT_k(Z) \tag{3}$$

$$Z=\frac{2}{\beta-\alpha}\left[A-\frac{\beta+\alpha}{2}I\right] \tag{4}$$

$$\begin{cases} T_0=I \\ T_1=z \\ T_k=2zT_{k-1}(z)-T_{k-2}(z) \end{cases} \tag{5}$$

$$c_k=\frac{1}{\sqrt{\alpha\beta}}(-q)^k \tag{6}$$

$$q=\frac{1-\sqrt{(\alpha/\beta)}}{1+\sqrt{(\alpha/\beta)}} \tag{7}$$

The definitions of the parameters above show that the calculation of the preconditioner requires mostly matrix and vector multiplications, which fits characteristics of parallel computation platforms.

3. GPU and SPARSE MATRIX STORAGE

3.1. GPU and CUDA

GPU for general purpose computations has been widely deployed nowadays. GPU was originally designed for graphic processing, which requires intensive floating point computations.

Before the release of CUDA, there were only a few general purpose computations that could run on GPU. However they have to be done through graphic application programming interface (API). The higher learning cost for using graphic API limits the development of general purpose computation on GPU. CUDA introduces a C-like programming interface for users. The C-like programming interface significantly reduces the learning cost of conducting general computations like matrix operations on GPU. With such software developments, more computing units have been added to GPU chip to accommodate the needs for large scale general purpose computations. The evolution of software and hardware on GPU together has popularized the GPU for general purpose computation. Different GPU architecture has different hardware and software designs. The discussions in this work will use Fermi architecture as an example.

GPU and CPU play different roles in computations. The Fermi GPU architecture from NVIDIA has 448–512 CUDA cores, while the mainstream CPU has 12–16 cores on chip due to the power and cooling limitations. GPU inherits the parallel computing advantage it has as a graphic processor, and CPU is designed to be more versatile and flexible. These differences together make the modern hybrid system architecture: parallelized computation work is offloaded to GPU, and CPU processes the rest computations, usually the code parts with heavily data dependence or intensive logical operations. After 2010 when CUDA was more widely accepted, the fastest supercomputers around the world equipped with Intel CPU all chose NVIDIA GPU as co-processor to boost computational throughput [13]. This trend of adopting GPU in the computation system shows great acceptance of this hybrid architecture in academia and industry.

CUDA cores on GPU are organized as Streaming Multiprocessors (SM). Each SM has 32 CUDA cores and 4 special function units for sin, cos, square root, etc. operations [14]. Each CUDA core has one floating point processing unit, and one integer processing unit. Threads on GPU are grouped together as a warp. The Fermi architecture has 32 threads as a warp. A warp is the minimum scheduling unit on GPU. All the threads in one warp will perform exactly the same work, which is named as Single Instruction Multithread (SIMT) technology. There are two warps executing concurrently on each SM, and up to 48 warps can be kept active to do fast context switching to compensate for the latency brought by memory related operations.

The SIMT brings massive parallelism in a GPU system. However, the other side of the story is that, since the hardware executes the exact same instruction for all the 32 threads, a conditional statement may happen, and the whole warp may have to run multiple times to finish all the branches. Such situation may harm the overall performance severely. Therefore, parallelization of programs with a large number of conditional statements may not be a good choice.

These give us the design consideration of a promising parallelized implementation or algorithm: it should have a large portion of parallelized code; less logical statements; and plenty of data to fully drive the GPU's computation ability and hide the memory latency.

3.2. *CUBLAS and CUSPARSE*

Basic Linear Algebra Subroutine (BLAS) [15] is a commonly used linear algebra library. CUDA Basic Linear Algebra Subroutines (CUBLAS) [16] is a CUDA implementation of BLAS. CUBLAS can provide single, double floating precisions, and complex numbers based dense matrix computations. CUBLAS makes calling algebra functions based on GPU implementation as easy as calling a BLAS function from CPU. CUBLAS hides implementation details of threads, blocks and grids inside each computation kernel.

Other than the support of dense matrix operations, NVIDIA also introduces CUDA Sparse Matrix Library (CUSPARSE) for sparse matrix operations. Sparse matrix functions are different from dense matrix operations: the storage of matrices and sparsity of two operands have to be considered. The matrix computations involved in power system application are usually very sparse. With support from CUSPARSE, users do not have to worry about special operations for sparse matrix. CUSPARSE has provided a set of functions like matrix format conversion, sparse matrix and dense vector operations, sparse matrix and sparse matrix operations. Same as CUBLAS, CUSPARSE has encapsulated the implementation details, so that users can call CUSPARSE functions directly without the effort of optimizing details like threads, block, and grid allocation. The standard interface such as CUBLAS and CUSPARSE further reduces the learning cost and development cost.

3.3. *Sparse matrix storage*

Computations based on sparse matrix usually utilize the sparse matrix storage format. Different storage formats will yield different memory access pattern and hence influence the performance.

Coordinate format (COO) is a commonly used storage format. Each non-zero element in sparse matrix will be represented by three entries: the row number, the column number, and the non-zero element value. Each entry itself forms an array with the number of non-zero elements as the length. Compressed sparse row format (CSR) compresses the row indices array compared with COO. Blocked compressed sparse row (BSR) [17] is another storage format. It stores non-zero blocks of elements with their row and column indices. Assume the block dimension is *blockDim*, the original matrix will be split into $(m/blockDim)+1$ by $(n/blockDim)+1$ subblocks. The indices of these sub-blocks will be stored in row-majored order. The advantage of BSR is that it provides a chance for reusing the vector data while performing matrix-vector multiplication. One vector data can be reused for *blockDim* times for the multiplication between the corresponding sub-blocks and the vector. The disadvantage of BSR is that it introduces more fill-ins. Not every element inside a non-zero block is actually non-zero, and then zero elements inside this block now are considered as non-zero elements and participate in the computations.

CUSPARSE has a better support for CSR based operations since it is more widely used. Matrix-vector multiplication has been supported in both CSR and BSR. However, sparse matrix-dense matrix multiplication, sparse matrix-sparse matrix multiplication are supported in CSR only for now.

4. Implementation

4.1. *Chebyshev preconditioner algorithm*

Chebyshev preconditioner algorithm is presented in Fig. 1. β is the largest eigenvalue of matrix A. α is the smallest eigenvalue of matrix A. *ratio* is used to estimate the value of α. Z transforms A's spectrum to $[-1,1]$. The decay rate c_k is related to α and β. r is the degree of Chebyshev preconditioner. Dag and Semlyen [12] discussed how to choose *ratio* and r in detail. Matrix G is the approximation of A's inverse. The output of Chebyshev preconditioner algorithm is matrix G. Bolded lines are implemented by either CUBLAS or CUSPARSE on GPU, since they are all matrix related computations.

Left multiplying G to A will generate the preconditioned matrix A with smaller condition numbers so that the system can converge faster in the iterative solver step.

4.2. *Conjugate gradient algorithm*

Fig. 2 shows the algorithm of the conjugate gradient method. x_0 is the initial value of the solution of $Ax=b$. If there is no

```
Initialization:
    $\beta = \max(eig(A))$; $\alpha = \beta/ratio$;
    $\boldsymbol{Z} = \boldsymbol{\frac{2}{\beta-\alpha}A - \frac{\beta+\alpha}{\beta-\alpha}I}$;
    $q = (1-\sqrt{(\alpha/\beta)})/(1+\sqrt{(\alpha/\beta)})$;
    $c_0 = (-q)^0/\sqrt{\alpha\beta}$; $c_1 = (-q)^1/\sqrt{\alpha\beta}$;
    $T_{p0} = I$; $T_{p1} = Z$;
    $\boldsymbol{G = \frac{c_0 I}{2} + c_1 T_{p1}}$
for i = 2:r
        $\boldsymbol{T = 2ZT_{p1} - T_{p0}}$;
        $c = (-q)^i/\sqrt{\alpha\beta}$;
        $\boldsymbol{G = G + cT}$;
        $T_{p0} = T_{p1}$; $T_{p1} = T$;
endfor
```

Fig. 1. Algorithm of Chebyshev preconditioner.

pre-knowledge of x_0, it can be set to all 0's. If there is, a cultivated x_0 can help conjugate gradient method converge in less iterations. r is the residual, which measures the error between b and Ax_k. If r is less than user-defined error tolerance, or the iteration has exceeded the allowed maximum iterations, the algorithm will stop.

Same as above, bolded lines are implemented either by CUBLAS or CUSPARSE on GPU. It can be seen that a majority of the computations can be ported on GPU for computation.

4.3. Hardware platform

The experiments are carried out on a server equipped with the NVIDIA GPU Tesla M2070, which is a Fermi architecture product. It has 14 stream multiprocessors and each processor has 32 CUDA cores, which makes the total CUDA cores on the chip be 448. The CUDA driver version is 5.0. The server has an 8-core Intel Xeon E5607 2.27 GHz CPU and 24 GB memory. Operation system is Ubuntu 11.10 with Linux Kernel version 3.0.0.

4.4. Software implementation

The test cases are power system examples from Matrix Market [18] and MATPOWER [19], and a sample case from UCTE [20]. Test matrices from Matrix Market are 494-bus, 662-bus, 685-bus, 1138-bus. Test matrices from MATPOWER are case2383wp and case2736sp. The sample case from UCTE in summer 2002 has 1253 buses. The maximum allowed error is 1×10^{-3}. The upper limit for iteration is 1000 iterations. Double precision floating point format is applied for both GPU and Matlab implementation.

```
Initialization
    $r = b - A * x_0; x = x_0;$
    $p = r$;
    $\boldsymbol{r_1 = r' * r}$; $k = 1$;
while( $r_1 > tolerance$ and $iteration < max\ iter$)
    if (k>1)
        $\beta = \frac{r_1}{r_0}$;
        $\boldsymbol{p = r + \beta * p}$;
    endif
    $\boldsymbol{Ap = A * p}$;
    $\boldsymbol{temp = p' * Ap}$;
    $\alpha = \frac{r_1}{temp}$;
    $\boldsymbol{x = x + \alpha * p}$;
    $\boldsymbol{r = r - \alpha * p}$;
    $r_0 = r_1$;
    $\boldsymbol{r_1 = r' * r}$;
endwhile
```

Fig. 2. Algorithm of the conjugate gradient method.

The bolded lines in Figs 1 and 2 are implemented either with CUSPARSE or CUBLAS. If there is sparse matrix in the computation, corresponding functions from CUSPARSE will be called. If the linear computation involves only two dense vectors, CUBLAS functions will be called for speedup purpose. We implement Chebyshev preconditioner based on the algorithm in Fig. 1, and integrate the conjugate gradient implementation adapted from NVIDIA CUDA computing SDK 5.0 Samples. Artificial condition number ratio used in the Chebyshev preconditioning is 5 for all the experiments based on Dag's discussion [12].

5. Computational experiment

This section will present computational experiment results beginning with selecting the degree for the Chebyshev Preconditioner, and then the performance comparison between Matlab implementation and our GPU implementation. Finally further performance improvement is discussed. Since Matlab's default floating point processing precision is double precision, our GPU implementations are all based on double precision floating point numbers for fair comparison purpose.

Note, throughout the computational experiments, the linearized DC power flow results are always verified with commercial software so the accuracy is ensured. The performance comparison is solely on the computational performance.

5.1. Degree of Chebyshev preconditioner

Chebyshev preconditioner can effectively reduce condition number. Condition number is generally considered as an indicator of matrix attribute. The smaller the condition number is, the less iterations the matrix needs to converge. A deeper degree of Chebyshev preconditioning can further reduce the condition number. The IEEE 30-bus and IEEE 118-bus systems from the standard IEEE test cases, the 494-bus and 662-bus systems from MatrixMarket, and the 1138-bus system from MatrixMarket, are selected as computation examples in small scale, medium scale, and large scale, respectively. Table 1 shows the condition number of these systems with different degrees of Chebyshev preconditioner. Degree 0 is the condition number of the original system without any preconditioning. Clearly, the condition number drops when the larger degree is set to Chebyshev preconditioner as indicated in Table 1.

Fig. 3 shows the conjugate gradient iteration comparison using Chebyshev preconditioner of different degrees. The first bar of each system, marked as *Degree = 0* in the figure, is the iteration number that is needed for the original system to be solved by the conjugate gradient method without the plugin of Chebyshev preconditioner. The rest of the bars in each group are the iteration number needed for solving the system by the conjugate gradient method with Chebyshev preconditioner, and the degree is set to different numbers for comparison. The original systems without any precondition require many more iterations for most cases. Fig. 3

Table 1
Condition number comparison of various systems.

Degree	IEEE 30-bus	IEEE 118-bus	494-bus	662-bus	1138-bus
0	961.534	4.85E+03	3.89E+06	8.27E+05	1.23E+07
2	168.637	941.108	6.49E+05	1.82E+05	2.07E+06
3	114.248	708.869	4.75E+05	1.39E+05	1.43E+06
5	78.962	483.311	3.25E+05	1.01E+05	9.69E+05
8	53.999	334.441	2.21E+05	7.02E+04	7.22E+05
10	44.773	277.241	1.87E+05	5.84E+04	6.11E+05

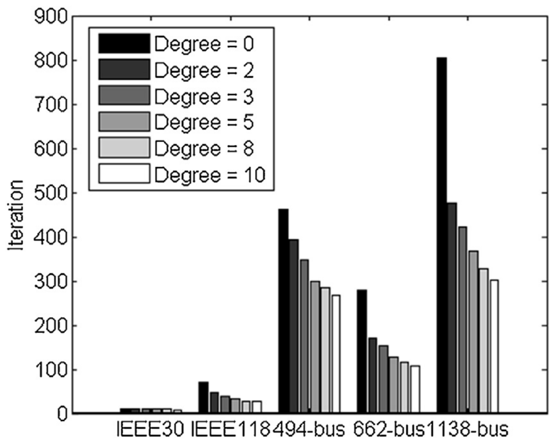

Fig. 3. Iteration comparison of Chebyshev preconditioner with various degrees.

shows that deeper degrees can always lead to a significant iteration number reduction in iterative solving.

Fig. 4 compares the performance of Chebyshev preconditioner implementation with GPU and with Matlab on CPU with different degrees. The IEEE 30-bus and the IEEE 118-bus systems are the smallest test cases. The GPU implementation is less efficient than the Matlab computations for the IEEE 30-bus system, no matter what degree of Chebyshev preconditioner is. The reason is that data involved in computations with small scale cannot fully drive the computational ability of the GPU card. The speedup gained in small systems can hardly offset the data copy overhead inherited in GPU computing. For the IEEE 118-bus system, as the degree increases, more data and computations will emerge and advantage of GPU's massive parallel computation ability begin to benefit the computation efficiency. For the medium scale systems, 494-bus and 662-bus, and the large scale system 1138-bus, speedup can be achieved for all degrees. The maximum speedup is 12.54, reached by the 1138-bus system when the degree is 2 for these five example systems.

Table 1 shows that a larger degree can lead to a better preconditioning in terms of condition number. However, it comes with the price that the number of non-zero elements will increase. Fig. 5 shows the exponential increase of nonzero elements when the degree is deeper. An increase of non-zero elements will affect not only the efficiency of Chebyshev preconditioner, but also the conjugate gradient method solving process. From the five example cases, choosing degree as 2 offers best performance improvement. Therefore 2 is selected as the Chebyshev preconditioner degree for

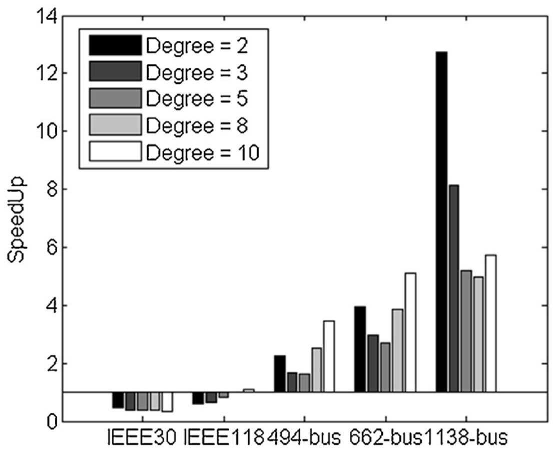

Fig. 4. Speedup comparison of Chebyshev preconditioner on GPU over Matlab with various degrees.

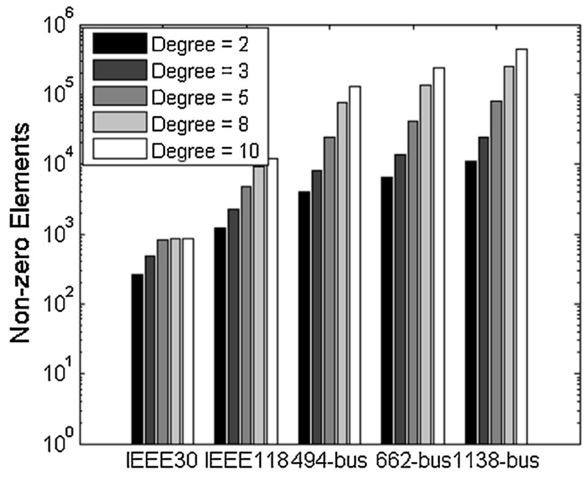

Fig. 5. Non-zero elements increase with deeper degree.

the rest experiments based on the consideration of the trade-off between the condition number reduction and the non-zero element increase.

5.2. Chebyshev preconditioner and conjugate gradient method

This section presents the performance result of the conjugate gradient method with Chebyshev preconditioner. The test matrices are from IEEE standard bus systems (IEEE30, IEEE57, IEEE 118, and IEEE300), MatrixMarket (494-bus, 662-bus, 685-bus, and 1138-bus), UCTE (1253 buses), and MATPOWER sample cases (Case2383wp and Case2736sp). The stop criterion for conjugate gradient method is 1×10^{-3}.

Table 2 shows the runtime of Matlab implementation on CPU and our GPU implementation of Chebyshev preconditioner. Speedup of GPU implementation over Matlab implementation is shown in the third column. The runtime of Chebyshev preconditioner on Matlab increases significantly while the size of test matrices grows. However, the runtime of Chebyshev preconditioner on GPU is in a stable range due to GPU's capability of handling large scale data.

GPU implementation of Chebyshev preconditioner begins to gain performance speedup when the system is larger than the standard IEEE 300-bus system. When the system scale is larger than 1000 by 1000, the performance improvement is significant. GPU implementation can gain about 46× speedup and almost 200 ms absolute runtime saving for the largest system case2736sp. Table 2 shows consistent results as Fig. 4. Chebyshev preconditioner on GPU can hardly improve computation performance for smaller systems, but it is able to gain runtime saving when the test systems are larger. The reason is that there is enough data

Table 2
Chebyshev preconditioner performance comparison between Matlab and GPU implementation.

System	Size	CPU(ms)	GPU(ms)	Speedup
IEEE30	29 by 29	0.737	1.537	0.48
IEEE57	56 by 56	0.530	1.538	0.34
IEEE188	117 by 177	0.914	1.579	0.58
IEEE300	299 by 299	3.472	3.254	1.07
494-bus	494 by 494	8.273	3.338	2.48
662-bus	662 by 662	15.328	3.407	4.50
685-bus	685 by 685	16.652	3.350	4.97
1138-bus	1138 by 1138	42.885	3.419	12.54
UCTE	1253 by 1253	52.565	3.389	15.51
Case2383wp	2382 by 2382	152.813	4.117	37.12
Case2736sp	2735 by 2735	199.502	4.263	46.80

Table 3
Conjugate gradient performance comparison between Matlab and GPU implementation.

System	Size	CPU(ms)	GPU(ms)	Speedup
IEEE30	29 by 29	1.259	1.781	0.71
IEEE57	56 by 56	2.100	2.966	0.71
IEEE188	117 by 177	6.058	7.844	0.77
IEEE300	299 by 299	12.522	13.075	0.96
494-bus	494 by 494	52.705	47.757	1.10
662-bus	662 by 662	36.449	28.106	1.30
685-bus	685 by 685	35.601	24.925	1.43
1138-bus	1138 by 1138	128.215	75.054	1.71
UCTE	1253 by 1253	8.554	4.600	1.86
Case2383wp	2382 by 2382	202.052	71.539	2.82
Case2736sp	2735 by 2735	124.189	25.727	4.83

Table 4
Conjugate gradient with Chebyshev preconditioner performance comparison between Matlab and GPU implementation.

System	Size	CPU(ms)	GPU(ms)	Speedup
IEEE30	29 by 29	1.996	3.318	0.60
IEEE57	56 by 56	2.630	4.504	0.58
IEEE188	117 by 177	6.972	9.423	0.74
IEEE300	299 by 299	15.994	16.329	0.98
494-bus	494 by 494	60.978	51.095	1.19
662-bus	662 by 662	51.777	31.513	1.64
685-bus	685 by 685	52.253	28.275	1.85
1138-bus	1138 by 1138	171.100	78.473	2.18
UCTE	1253 by 1253	61.119	7.989	7.65
Case2383wp	2382 by 2382	354.865	75.656	4.69
Case2736sp	2735 by 2735	323.690	29.990	10.79

to better utilize the computation capability of the GPU and offset computation overhead like data copy.

Table 3 shows the runtime of Matlab implementation and GPU implementation of the conjugate gradient method. Corresponding speedup in the third column shows GPU's advantages in a large system. The runtime of conjugate gradient is related to the data of each system. System size, matrix condition number and sparsity, have their influences on the performance of conjugate gradient method. The runtime is no longer monotonically increasing when the system size is larger. The conditioner number of 662-bus is less than 494-bus as Table 1 indicates. In Table 3, it is shown that the runtime of 662-bus is shorter than 494-bus. The maximum GPU implementation speedup of conjugate gradient is 4.83× for case2736sp in our experiments. The absolute runtime saving is around 100 ms for one solving in the same case.

Table 4 shows the total runtime, including Chebyshev preconditioner and conjugate gradient method, and the corresponding speedup of GPU implementation over Matlab implementation. GPU implementation begins to improve performance at 494-bus,

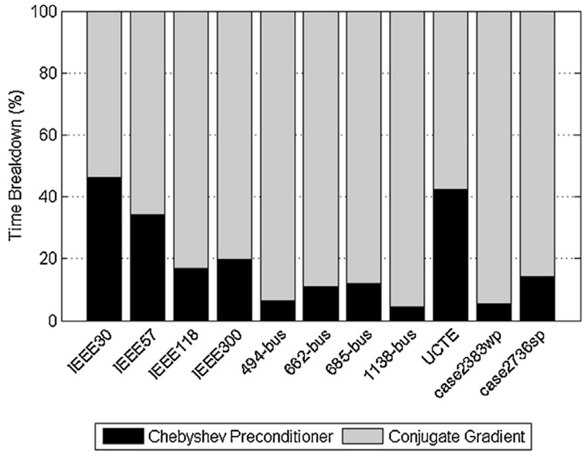

Fig. 6. Runtime breakdown of conjugate gradient method with Chebyshev preconditioner.

because of the performance improvement in the Chebyshev preconditioner part. The total time speedup is better than the conjugate gradient only results. The fastest speedup reaches 10.79×. The absolute runtime saving is about 300 ms for case2736sp for one solution.

5.3. Improvement on Chebyshev preconditioner and conjugate gradient method

The runtime breakdown of Chebyshev preconditioner and conjugate gradient is shown in Fig. 6. Chebyshev preconditioner can help to reduce the iterations needed in the conjugate gradient method. However, for most cases, it consumes less than 20% of the total runtime; for some other cases it only reaches 50% of the total runtime. The Chebyshev preconditioner computation time never occupied more than 50% of the total execution time. Therefore, in order to further improve the overall performance, the performance of conjugate gradient computation needs to be enhanced.

Computations involved in conjugate gradient contain a lot of matrix vector multiplications. Blocked compressed sparse row (BSR) provides data reuse for matrix vector multiplication. Our improved conjugate gradient implementation uses BSR based GPU matrix-vector multiplication. The result is shown in Table 5. The block size we choose based on empirical experience is 3. The BSR-based conjugate gradient implementation shows greater improvement when system is relatively small. It makes GPU implementation run faster than the CPU version when the system is only around a 300 by 300 scale.

Table 5
Conjugate gradient performance comparison between Matlab, CSR and BSR based GPU implementation.

System	Size	CPU(ms)	GPU-CSR (ms)	GPU-BSR(ms)	CG Speedup		Total Speedup	
					GPU-CSR vs. CPU	GPU-BSR vs. CPU	GPU-CSR vs. CPU	GPU-BSR vs. CPU
IEEE30	29 by 29	1.259	1.781	1.567	0.71	0.80	0.60	0.64
IEEE57	56 by 56	2.100	2.966	2.536	0.71	0.83	0.58	0.64
IEEE188	117 by 177	6.058	7.844	6.681	0.77	0.91	0.74	0.84
IEEE300	299 by 299	12.522	13.075	11.429	0.96	1.10	0.98	1.09
494-bus	494 by 494	52.705	47.757	41.116	1.10	1.28	1.19	1.37
662-bus	662 by 662	36.449	28.106	26.026	1.30	1.40	1.64	1.76
685-bus	685 by 685	35.601	24.925	21.331	1.43	1.67	1.85	2.12
1138-bus	1138 by 1138	128.215	75.054	68.201	1.71	1.88	2.18	2.39
UCTE	1253 by 1253	8.554	4.600	4.411	1.86	1.94	7.65	7.82
Case2383wp	2382 by 2382	202.052	71.539	73.438	2.82	2.75	4.69	4.57
Case2736sp	2735 by 2735	124.189	25.727	26.774	4.83	4.64	10.79	10.45

6. Discussion

Our work discusses the GPU-based implementation of an iterative solver: conjugate gradient solver, and a polynomial preconditioner: the Chebyshev preconditioner. Because of its potential in parallelism and scalability, iterative linear solvers have been adapted to power system applications [3,12,21]. Preconditioner plays an important role in the iterative solver. Previously, preconditioner like ILU was widely used to precondition the matrix. However, they suffer a tight data dependency issue and hence are difficult to parallelize. The Chebyshev preconditioner, a polynomial preconditioner, is a parallelizable method. The conjugate gradient method is one of the iterative solvers. It has been introduced to power system applications for its potential parallelism [7].

The limitation of the conjugate gradient method is that it requires a symmetric positive definite linear system. This fits the model of linearized DC power flow as used in this paper. For linear systems that are not symmetric, a transition can be used to accommodate the computational needs: left multiply the matrix's transpose to both of left hand side and right hand side to eliminate the undesired matrix characteristics while guaranteeing that no extra work is required for solving the system. Solving $Ax = b$ can be alternatively turned into solving $A^TAx = A^Tb$.

Chebyshev preconditioner can provide a major condition number reduction with deeper preconditioner degree. However, a deeper degree will lead to significant increase of non-zero elements. Such increase of non-zero elements will cause severe performance degradation in the conjugate gradient step. The degree for Chebyshev preconditioner should not be chosen without the consideration of iterative solver step. The proper degree should be chose based on a trade-off between reducing condition number and inhibiting the growth of non-zero elements.

7. Conclusion

Power system applications such as power system optimization, control and analysis require intensive computational ability [22]. Solving sparse linear systems is a critical computation element involved in these applications. Our work presents a GPU-based Chebyshev preconditioner, and integrates the iterative conjugate gradient solver for a whole iterative solving chain. Our implementation uses native functions from CUSPARSE and CUBLAS libraries which are already optimized. Implementations based CUSPARSE and CUBLAS libraries require minimum modifications when there are updates for either GPU, the hardware platform, or CUDA, the software platform. It will be NVIDIA who considers the compatibility of their previous functions and libraries, not the end programmers. In addition, the functions will be further optimized based on newer software environments.

Our work targets at solving the fundamental computation of power system and sparse linear systems. Table 5 shows that the maximum overall speedup can reach 10.79 with the case2736sp system; Table 2 shows that the maximum Chebyshev preconditioner speedup can reach 46.80 with the same system. This work will be not only for solving DC power flow in power system, but also for any sparse linear systems that are symmetric positive definite.

Our work considers Chebyshev preconditioner and conjugate gradient method together to choose the proper degree for Chebyshev preconditioner. Fig. 5 shows the runtime breakdown of the iterative solver and the preconditioner. The iterative solver actually consumes more runtime. Thus, we can conclude that to improve the overall linear equations solving capability, besides improving the performance of Chebyshev preconditioner, one must take the performance improvement of iterative solver into consideration as well. Further improvement on the iterative solver implementation is critical for the overall performance improvement of iterative solutions of linear systems.

Acknowledgement

This work was supported in part by US NSF grant ECCS-1128381. This work also made use of the Shared Facilities and the Industry Partnership Program supported by CURENT, an Engineering Research Center (ERC) Program of NSF and DOE under NSF grant EEC-1041877.

References

[1] F. de Leon, A. Semlyen, Iterative solvers in the Newton power flow problem: preconditioners, inexact solutions and partial Jacobian updates, IEE Proc. Gener. Transm. Distrib. 149 (Jul) (2002) 479–484.
[2] M.A. Pai, P.W. Sauer, A.Y. Kulkarni, Conjugate gradient approach to parallel processing in dynamic simulation of power systems, in: American Control Conference, 1992, pp. 1644–1647.
[3] M.A. Pai, H. Dag, Iterative solver techniques in large scale power system computation, in: Proc. of the 36th IEEE Conference on Decision and Control, vol. 4, 1997, pp. 3861–3866.
[4] NVIDIA, Parallel Programming and Computing Platform | CUDA | NVIDIA, http://www.nvidia.com/object/cuda_home_new.html, 2013.
[5] R. Helfenstein, J. Koko, Parallel preconditioned conjugate gradient algorithm on GPU, J. Comput. Appl. Math. 236 (2012) 3584–3590.
[6] J. Zhang, L. Zhang, Efficient CUDA polynomial preconditioned conjugate gradient solver for finite element computation of elasticity problems, Math. Prob. Eng. 2013 (2013).
[7] N. Garcia, Parallel power flow solutions using a biconjugate gradient algorithm and a Newton method: a GPU-based approach, in: IEEE PES General Meeting Minneapolis, MN, 2010, pp. 1–4.
[8] Z. Li, V.D. Donde, J.C. Tournier, F. Yang, On limitations of traditional multi-core and potential of many-core processing architectures for sparse linear solvers used in large-scale power system applications, in: IEEE PES General Meeting, Detroit, 2011, pp. 1–8.
[9] X. Li, F. Li, J.M. Clark, Exploration of multifrontal method with GPU in power flow computation, in: IEEE PES General Meeting, Vancouver, Canada, 2013, pp. 1–6.
[10] A. Gopal, D. Niebur, S. Venkatasubramanian, DC power flow based contingency analysis using graphics processing units, in: IEEE Power Tech, Lausanne, 2007, pp. 731–736.
[11] A.A. Kamiabad, Implementing a Preconditioned Iterative Linear Solver Using Massively Parallel Graphics Processing Units (M.Sc. Thesis), University of Toronto, 2011.
[12] H. Dag, A. Semlyen, A new preconditioned conjugate gradient power flow, in: IEEE Trans. Power Syst., vol. 18, 2003, pp. 1248–1255.
[13] TOP500.org, Top #1 Sytems, 2013, http://www.top500.org/featured/top-systems/
[14] NVIDIA, NVIDIA's Next Generation CUDA Compute Architecture: Fermi, 2009, http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf
[15] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, et al., LAPACK Users' guide vol. 9, in: Society for Industrial Mathematics, 1987.
[16] NVIDIA, CUBLAS | NVIDIA Developer Zone, 2012, https://developer.nvidia.com/cublas
[17] NVIDIA, CUSPARSE LIBRARY, 2012, http://docs.nvidia.com/cuda/pdf/CUDA_CUSPARSE_Users_Guide.pdf
[18] R.F. Boisvert, R. Pozo, K. Remington, R. Barrett, J.J. Dongarra, Matrix market: a web resource for test matrix collections, in: Quality of Numerical Software, Assessment and Enhancement, 1997, pp. 125–137.
[19] R.D. Zimmerman, C.E. Murillo-Sánchez, R.J. Thomas, MATPOWER: steady-state operations, planning, and analysis tools for power systems research and education, IEEE Trans. Power Syst. 26 (Feb) (2011) 12–19.
[20] Q. Zhou, J.W. Bialek, Approximate model of European interconnected system as a benchmark system to study effects of cross-border trades, IEEE Trans. Power Syst. 20 (May) (2005) 782–788.
[21] R. Idema, D.J.P. Lahaye, C. Vuik, L. van der Sluis, Scalable Newton–Krylov solver for very large power flow problems, IEEE Trans. Power Syst. 27 (Feb) (2012) 390–396.
[22] R.C. Green, L. Wang, M. Alam, High performance computing for electric power systems: Applications and trends, in: IEEE PES General Meeting, 2011, pp. 1–8.

Xue Li received her BS degree from Northwestern Polytechnical University China in 2007 and MS degree in 2011 from University of Tennessee. She is presently pursuing her Ph.D. degree in the Department of EECS, The University of Tennessee, Knoxville, TN, 37922, USA. Her research interest is computational methods for power system analysis.

Fangxing Li, also known as Fran Li, received his Ph.D. degree from Virginia Tech in 2001. Presently, he is an Associate Professor at The University of Tennessee, Knoxville, TN, USA. Dr. Li is a registered Professional Engineer (P.E.) in North Carolina, and a Fellow of IET (formerly IEE).