12-2015

Using GPU to Accelerate Linear Computations in Power System Applications

Xue Li
University of Tennessee - Knoxville, xli44@vols.utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Xue Li entitled "Using GPU to Accelerate Linear Computations in Power System Applications." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Engineering.

Fangxing Li, Major Professor

We have read this dissertation and recommend its acceptance:

Hairong Qi, Yilu Liu, Xueping Li, Mallikarjun Shankar

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

Using GPU to Accelerate Linear Computations in Power System Applications

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Xue Li

December 2015

Acknowledgements

I would like to thank my advisor Dr. Fangxing Li for his guidance, patience, time, encouragement and support for this dissertation and other research works during my Ph.D. study at the University of Tennessee, Knoxville. This work cannot be finished without his continuous efforts on bridging the gap between my Computer Engineering background and the related power system knowledge. I would also like to thank my committee members Dr. Yilu Liu, Dr. Hairong Qi, Dr. Mallikarjun Shankar and Dr. Xueping Li for their time and suggestions on this work.

My special thanks go to Dr. Yanli Wei, Dr. Zhiqiang Jin, Dr. Yao Xu, Dr. Qinran Hu, Dr. Hao Huang, Dr. Sarina Adhikari, Haoyu Yuan, Kumaraguru Prabakar, Can Huang, Xin Fang, Hantao Cui, Linquan Bai and Riyasat Azim for their enlightening discussions on both my research and course work. These years working with them are the treasure of my life. I would also like to thank Joshua M. Clark, Daniel Cash and Andrew Hnilica for their contributions while working with me as undergraduate researchers. I am also grateful for all the faculty, students and staff in the Center for Ultra-wide-area Resilient Electric Energy Transmission networks (CURENT) for creating an open and diverse culture.

At last, I owe my deepest gratitude to my parents and family, especially to my grandfather. It's their support that makes it possible for me to pursue the happiness that I desire.

Abstract

With the development of advanced power system controls, the industrial and research community is becoming more interested in simulating larger interconnected power grids. It is always critical to incorporate advanced computing technologies to accelerate these power system computations. Power flow, one of the most fundamental computations in power system analysis, converts the solution of non-linear systems to that of a set of linear systems via the Newton method or one of its variants. An efficient solution to these linear equations is the key to improving the performance of power flow computation, and hence to accelerating other power system applications based on power flow computation, such as optimal power flow, contingency analysis, etc.

This dissertation focuses on the exploration of iterative linear solvers and applicable preconditioners, with graphic processing unit (GPU) implementations to achieve performance improvement on the linear computations in power flow computations. An iterative conjugate gradient solver with Chebyshev preconditioner is studied first, and then the preconditioner is extended to a two-step preconditioner. At last, the conjugate gradient solver and the two-step preconditioner are integrated with MATPOWER to solve the practical fast decoupled load flow (FDPF), and an inexact linear solution method is proposed to further save the runtime of FDPF. Performance improvement is reported by applying these methods and GPU-implementation. The final complete GPU-based FDPF with inexact linear solving can achieve nearly 3x performance improvement over the MATPOWER implementation for a test system with 11,624 buses. A supporting study including a quick estimation of the largest eigenvalue of the linear system which is required by the Chebyshev preconditioner is presented as well. This dissertation demonstrates the potential of using GPU with scalable methods in power flow computation.

Table of Contents

Chapter 1

Introduction

1.1 General introduction

The North American transmission grid is one of the largest engineering systems Crow (2002). There are many applications in modern power system analysis that are computationally intensive. Some examples of this are, simulation, optimization, contingency analysis, etc. With the further development of the smart grid, the power system model will become even more complex, and hence, the corresponding computational implementations are facing more challenges. At the same time, the penetration of renewable resources, application of distributed generators, and demand for energy storage make the existing models even more complex and also necessitate real time analysis and response. Powerful and efficient software and hardware are necessary to accommodate such computational needs.

Power flow, as one of the most fundamental computations in power system analysis and simulation, is usually modeled as a nonlinear system and solved iteratively through linearization. For instance, the Newton-Raphson method converts the nonlinear system to a set of linear equations with the introduction of Jacobian matrix. Solving these nonlinear systems takes a significant portion of time in the overall power flow solution. Therefore, it is of great importance to improve the efficiency of the linear system solution in order to enhance the power flow analysis computation efficiency.

Traditionally, the linear equation system is solved by a direct method with sparse matrix techniques, such as LU decomposition (Tinney and Walker (1967), Tinney and Hart (1967)). The process is carried out in $\mathcal{O}(n^3)$ basic floating point operations Crow (2002). Direct solvers usually have a high memory requirement due to the reordering step and the inevitable fill-ins during factorization. Because of the potential scalability issue of direct solvers and the development of parallel computing platform, iterative solvers can be considered as an alternative to traditional LU-based direct methods.

The classic iterative methods to solve linear systems include the Jacobi method (Golub and van Van Loan (1996)), the Gauss-Seidel method (Golub and van Van Loan (1996)), the relaxation method (Axelsson (1972), Chazan and Miranker (1969)) as well as others. All of these have a similar form as $Mx^{k+1} = Nx^k + b$, where $A = M - N$ is called a *splitting* of matrix A. Since this method involves a solving process every iteration, it usually requires that the matrix M should be much more easier to solve than A, such as diagonal matrix or triangular matrix. To make the method converge, the spectrum radius of $M^{-1}N$ should be smaller than 1.

However, except in the case that matrix A is well structured, it will not be easy to find a proper M, that is, firstly, simple enough to solve, and secondly, has a good convergence rate. Therefore another category of iterative method called Krylov space method has emerged. Conjugate gradient method works on symmetric positive definite matrix (Dennis Jr and Turner (1987), Stewart (1973)). Minimum residual method (MINRES) can be applied to systems which are symmetric but not necessarily positive definite (Paige and Saunders (1975)). Generalized minimal residual method (GMRES) (Saad and Schultz (1986)), Arnoldi method (Arnoldi (1951) Saad (1981)) and Biconjugate Stabilized method (BiCG-STAB) Van der Vorst (1992) extend the iterative solvers to asymmetric systems. All of these methods can be termed as the projection methods on different Krylov subspaces.

Although iterative solver doesn't require reordering as the direct solver does, a preconditioner is usually necessary to improve the convergence rate of the system. A preconditioner is also a matrix which transforms the original linear system into another one with clustered eigenvalue spectrum, or smaller condition number. This process is called preconditioning. Incomplete LU (ILU) factorization de Leon and Semlyen (2002), approximated inverse preconditioner Benzi and Tuma (1998) and polynomial-based preconditioner such as Chebyshev preconditioner Dag and Semlyen (2003) are several examples of preconditioners.

On the other hand, the development of a parallel computation hardware platform also makes revolutionary changes. Recently, graphic processing unit (GPU) has been enabled to carry general purpose computations, although it was originally designed for a graphic processing purpose. GPU has massive parallel computational units on board, and therefore it is now being used as a co-processor to accelerate specific types of computations. As a peripheral equipment, GPU, which communicates with CPU by PCI-E, has its own features. From the perspective of CPU, GPU is its peripheral equipment, therefore it won't be efficient if CPU communicates with GPU as frequently as with its own memory like what happens when computation is carried out on a single desktop or server. However, the communication between CPU and GPU does not like that between several independent servers either. There is no shared network which may get congested and delay the work. The communication delay between GPU and CPU will be much shorter than that of several servers. Such difference in communication delay calls for different algorithms with different granularity of parallelism. Therefore, the parallel algorithm on GPU requires specific considerations to efficiently perform.

With the development of the software support of GPU, the example cases in computational finance, computational fluid dynamics, computational structural mechanics, electronic design automation, numerical analysis, computational chemistry and biology have already been successfully accelerated NVIDIA (2014).

There are also works attempting to use GPU in power system applications. Gopal et al. (2007) employed GPU to simulate DC power flow. Newton-Raphson and Gauss-Jacobi were mapped to GPU to solve the linearized power flow. Garcia (2010) used a GPU version of biconjugate gradient to solve Newton power flow and achieved approximately 2 times speedup. Guo et al. (2012) implemented the Gauss-Seidel power flow, Newton-Raphson power flow and PQ decoupled power flow with CUDA on GPU. Jalili-Marandi and Dinavahi (2010) Jalili-Marandi et al. (2012) Jalili-Marandi and Dinavahi (2009) explored the possibility of solving transient stability simulation of large-scale system on a single GPU. Jalili-Marandi et al. (2012) investigated the potential for transient stability simulation by the use of multiple GPUs. A Gauss-Jacobi instantaneous relaxation (GJ-IR) method was proposed and a GPU-based sparse linear LU solver was employed. A 10 times speedup of linear solving was reported for a synthetic system with size of 22,000. Liu et al. (2013) used a GPU-based GMRES iterative solver with incomplete LU as its preconditioner for large scale transient analysis. The results shown that GPU-GMRES can yield about 3 to 10 times speedup over the corresponding implementations on CPU. Yu et al. (2014) used the Jacobian-free Newton-Raphson method for the transient dynamic simulations. Karimipour and Dinavahi (2013) implemented a weighted least squared (WLS) state estimation for large-scale power system. Rakai and Rosehart (2014) extended the discussion to optimal power flow (OPF). Predictor-corrector (PC) interior-point method was employed to solve the OPF. The most computationally intensive part of it was a matrix factorization, which was mapped to GPU. A speedup over 4 was reported for single-precision floating point computation for system with 3120 buses. Li et al. (2014) discussed using GPU to accelerate the optimization problem in a commercial power system simulation and analysis software. The authors profiled the most computationally intensive parts of their software tool and used GPU to accelerate them. The results showed the great potential of acceleration that GPU can bring, especially for realistic power systems. However, the authors also pointed out that to use a fully GPU-based implementation may incur a total re-architecture of the whole existing software, which may be too expensive for the industries. Ablakovic et al. (2012) used OpenCL and GPU to perform a real time three-phase distribution power flow.

These aforementioned works proved that the integration of GPU-based implementation aiming at accelerating the power system computation have been widely accepted and deployed in different fields of research and practice.

1.2 Dissertation outline

This work will focus on applying parallel methods with GPU-based implementation to accelerate the linear computations in power flow analysis, and integrate such linear solving method into power flow computation.

Chapter 2 reviews the literature related to parallel computations in power system applications, iterative solver, commonly used preconditioner, and the integration of GPU in linear system solving.

Chapter 3 introduces the computational needs in power system applications, and then introduces the background of linear system solving. A brief introduction of GPU is followed.

Chapter 4 uses GPU-accelerated conjugate gradient method and Chebyshev preconditioner to solve power flow. The Chebyshev preconditioner is a polynomial preconditioner, which can reduce the condition number significantly and can be parallel. The results are conducted with several practical bus systems. The maximum speedup for Chebyshev preconditioner and conjugate gradient solver can reach up to 46 and 4 times for the largest test system, respectively.

Chapter 5 proposes a method to estimate the largest eigenvalue for the use of Chebyshev preconditioner. The calculation of eigenvalues are usually too time-consuming to be deployed practically. Therefore, an estimation of it using the features of power grid is proposed and discussed in this Chapter. The precision and performance improvement of using an estimated value are presented too.

Chapter 6 utilizes two GPU-based preconditioners to precondition the linear computations. The two-step preconditioner *Jach* integrates a Jacobi-like and a Chebyshev preconditioner. The results show that the two-step preconditioner can always perform the best compared with using each preconditioner alone. The implementation on GPU brings up to 8.9x performance improvement for the largest test system over corresponding CPU implementation.

Chapter 7 studies a complete fast decoupled power flow (FDPF) by the integration of the GPU-based conjugate gradient solver and the two-step preconditioner $JaCh$ with MATPOWER, a Matlab-based open source software package for solving power flow and optimal power flow. The results show that GPU-based FDPF performs better when the system size is approaching 9000-bus scale. With an inexact linear solution strategy, the performance improvement is around 2 times compared with MATPOWER for the several test system around 10000-bus scale. At the same time, the computation precision is well maintained too.

Chapter 8 concludes this work and provides suggestions for future work in applying GPU-based parallel computational methods for power system applications.

1.3 Contributions

The contributions of this work are listed as followed.

- This work discusses the numerical attributes, such as symmetric/asymmetric and positive definiteness, of the linearized systems from commonly applied power system computations.

- GPU-based Chebyshev preconditioner is developed with the integration of an iterative conjugate gradient solver to solve power flow. With the goal of improving the computation efficiency of linear solution, this work considers the Chebyshev preconditioner and conjugate gradient together to choose the proper degree for Chebyshev preconditioner so that the overall computation is accelerated.

- The speedup with the GPU implementation can reach 46x for Chebyshev preconditioner and 4x for the conjugate gradient solver among the test matrices from practical power systems.

- A two-step preconditioner is implemented with GPU and provides up to 8.9x speedup for the whole solving process compared with its corresponding CPU implementation.

- In order to construct Chebyshev preconditioner, the largest eigenvalue is required. An estimation of the largest eigenvalue is proposed and verified. The estimation accuracy is precise enough to well keep the preconditioning effects of the preconditioner, and the overhead is negligible.

- A complete software architecture for fast decoupled power flow on GPU is proposed and implemented for the first time. The GPU-FDPF works as precisely as the original CPU-based FDPF with MATPOWER, the Matlab-based open source tool. The GPU-FDPF begins to achieve performance improvement for systems around 10,000 buses.

- The integration of GPU-FDPF with inexact inner linear solution is proposed. GPU-FDPF with inexact linear solution well maintains the FDPF precision and further improves the performance for systems larger than 9,000 buses to around 2 times over MATPOWER implementation on CPU.

Chapter 2

Literature Review

This chapter reviews related literature on high performance computation in power system, iterative methods, and preconditioner. A review of GPU's integration in linear system solving is followed.

2.1 Parallel computations in power system applications

The computations involved in power system analysis, simulation, control and optimization are getting more and more intense in modern electrical power systems. This is because the renewable resources, distributed energy generation and storage, etc. are penetrated rapidly into an already complicated system. At the same time, more information about real-time system states and the prediction of future system activities are always critical to the power industry. Therefore, the interests of improving computation efficiency never fade in the research of power systems.

2.1.1 Power flow

Power flow is the most fundamental computation in power system applications. There are many works focusing on improving its computation efficiency.

Amano et al. (1996) used an epsilon decomposition method to eliminate the weak coupling elements so as to decompose the algebraic equations in power flow and then applied block-parallel Newton method to solve power flow concurrently. However, they used a constant Jacobian matrix, which tended to show a slower convergence rate.

Chen and Chen (2000) proposed a novel factorization tree with the consideration of maximum number of fill-in and degree of every node so as to balance the workload on each core of a multiprocessor architecture. Their work was demonstrated with IEEE-57 and IEEE-118 systems. Wang et al. (2007) presented a partitioning scheme so that the Jacobian matrix could be presented in doubly-bordered block diagonal (DBBD) LU factorization form, and implemented the algorithm on hardware, an SOPC (system-on-a-programmable-chip) containing a FPGA (field-programmable-gate-array). Their experiments with IEEE-57, IEEE-118, IEEE-300 bus systems and one 1648-bus, one 7917-bus system showed up to 7x speedup compared with a single processor. Koester et al. (1993) presented a reordering scheme to generate block-diagonal-bordered form matrices and, at the same time, minimize the fill-in and number of coupling equations. With such a decoupling scheme, many diagonal block matrices could be factorized simultaneously so as to save the runtime, and the existing techniques based on dense matrix could be applied.

Applying the direct method in power systems is more about reordering the Jacobian matrix so as to decouple the original matrix and reduce potential fill-ins. Another trend is to apply iterative solvers to get the solution of load flow equations. Conjugate gradient (CG) method can be used for symmetric positive definite linear systems. Wallach (1968) reformulated the load flow problem as an optimization problem and then deployed the steepest descent and conjugate gradient methods to solve it. It brought up advantages of CG methods such as no matrix inversion required and no additional storage space needed, and it guaranteed to converge within n (size of the matrix) steps. Galiana et al. (1994) applied a conjugate gradient solver with incomplete Cholesky as preconditioner to solve fast decoupled power flow and DC load flow. Their method had been tested on randomly generated 5000-bus to 10000-bus systems. A computation efficiency comparison between iterative method and direct method was given for DC load flow and fast decoupled power flow when the size of the largest block was varied. It showed that the CPU time for preconditioned CG method varies as $\mathcal{O}(n^{1.5})$, whereas the direct method varies as $\mathcal{O}(n^2)$. The advantage of iterative method over direct method becomes significant when system size is larger than 3000.

For asymmetric cases such as AC power flow or power system dynamics, conjugate gradient variants can be applied. de Leon and Semlyen (2002) compared different conjugate gradient-based methods which were applicable for asymmetric systems, such as bi-conjugate stabilized (BiCG), conjugate gradient square (CGS), general minimized residual (GMRES), bi-conjugate (BiCG), quasi-minimal residual (QMR) to solve the AC power flow of 118-, 354-, 1062-, 3186- and 6372-bus systems. Their work also tested preconditioners such as scaling, ILU(m), FD preconditioner (Flueck and Hsiao-Dong (1998)). Partial Jacobian updates and inexact solutions had been integrated as well. Their work demonstrated that FLOPS saving compared with direct LU solution as great as 35% could be reached for larger systems that were 3000 buses and more. Garcia (2010) implemented a preconditioned BiCG with the Newton method on GPU to solve power flow. The experiment on IEEE-118 showed 2.47x speedup. Li et al. (2011) applied a GPU-based CG with Jacobi preconditioner and conjugate gradient normal residual (CGNR) with Jacobi preconditioner for power system state estimation and power flow computation. The experiment with synthetic large scale power system examples (22K by 22K buses) showed a significant speedup of about 49x.

The convergence rate of an iterative solver is usually tightly related to an effective preconditioner. Dag and Alvarado (1997) proposed a preconditioner called the XD method. Firstly, it reordered the whole matrix with the purpose of reducing fill-ins and then it did a complete LDU factorization instead of an incomplete one. Finally it discarded some elements with different levels. While standard $ILU(0)$,

$ILU(1)$, $ILU(2)$ were based on incomplete LDU decomposition with different fill-ins, XD methods such as $ILU_{(0)}$, $ILU_{(1)}$, and $ILU_{(2)}$ etc did the complete LDU decomposition first and then discarded elements according to the non-zero pattern of corresponding levels. The advantage of the XD method was that it kept all the XD methods with different levels always positive definite, which avoided the problem of conjugate gradient method diverging due to the preconditioner. The disadvantage was the high computation cost which included a reordering process and a complete LDU decomposition. The experiments on gain matrices of state estimation from six different systems with sizes 14, 57, 68, 118, 300 and 414 buses proved that XD preconditioned gain matrix worked better than ordinary $ILU(0)$, $ILU(1)$ and $ILU(2)$ methods.

2.1.2 Power system dynamics

Power system dynamics are usually formulated as a set of algebraic equations, which will be transformed to a set of linear equations via methods such as simultaneous implicit approaches. The numerical characteristics of such linear system is asymmetric.

Decker et al. (1996) presented using Bi-conjugate gradient (Bi-CG) method and a stabilized Bi-CG called BiCG-Stab method to solve power system dynamics. It implemented the proposed method on a cluster that had 8 processing nodes connected with a hypercubic topology and demonstrated the CG-based variants' robustness and accuracy when applied to power system dynamics computations. Pai et al. (1995) tested an traditional ILU(m) preconditioner for GMRES method to solve power system dynamics, and they also proposed a dishonest preconditioner which used the ILU(m) preconditioner from last Newton iteration for the current Newton iteration so as to avoid repeated factorization of the preconditioner of each Jacobian matrix. This was based on the observation that the Jacobian matrix would not significantly change every Newton iteration. Khaitan and McCalley (2010) applied

GMRES to solve long-term time domain simulations with ILU and multifrontal as preconditioner. The results tested with 6-, 32-, 194- and 385-generator test systems showed that, firstly, GMRES with multifrontal preconditioner constantly had the best performance among GMRES with ILU and direct multifrontal solver, and secondly, that direct multifrontal solver could provide close performance to GMRES with multifrontal preconditioner for small scale systems. However when system scale was large, for example a 385-generator system, the direct solver was much slower than the proposed GMRES with multifrontal preconditioner. The amount of time saving and speed-up of GMRES with multifrontal preconditioner increased significantly with the increase of system size.

2.1.3 Contingency

Contingency analysis in a power system security assessment will detect and evaluate the limited violations of the system operating. It will usually carry out a set of specifically designed contingency cases. Theoretically, the complete system status should be evaluated for each contingency case. The rank of the effects of those violation will be given. With such advisory results, the operational planner can make corresponding control decisions to eliminate the influence from the violations. The evaluation of each contingency case is normally carried out as a load flow. As expected, the contingency analysis will involve many load flow computations which make the contingency screening process extremely computationally expensive. Therefore the improvement of such computational efficiency will be of great importance to power system security assessment.

As discussed, the most computationally intensive section in contingency analysis is solving load flow. Therefore, the methods discussed here have their common points discussed in DC or FDPF in section 2.1.1. The difference would be that the contingency analysis will have more repeated similar load flow computations. As a result, the data reuse among different contingency cases could be explored. Alves et al. (1999) compared sparse Gaussian elimination and preconditioned conjugate gradient method for contingency analysis of IEEE-30, 57, 118, 300 bus systems and two Brazilian 810- and 1663-bus systems. They compared the impact from different reordering schemes to the final iterations that CG needed to converge, the difference was actually insignificant. They also pointed out that a complete Cholesky preconditioner could produce good preconditioning effects. However, if a complete Cholesky decomposition was available, a forward/backward substitution would make a better overall runtime result. Mori et al. (1995) utilized Tchebyshev iteration (TI) methods as the iterative solver and incomplete Cholesky (IC) as the preconditioner to solve the fast decoupled power flow of contingency screening. The proposed ICTI method for fast decoupled power flow could gain 22.7x speedup for a 2107-node system.

2.1.4 State estimation

Practical power systems are very complex. Analytical functions can hardly be available for every variable. As a result, the system status can be achieved by measurements. However, firstly, some measurements are too difficult or too expensive to get due to factors such as high temperature or moving parts. Secondly, there are inevitable errors during measurement. Therefore, redundant measurements are usually deployed to estimate the unavailable variables and to resolve the conflicting measured results. State estimation will eliminate the random measurement errors, correct the measure faults, and estimate those unavailable measurements. This process minimizes the error between all the measurements and the system states. Weighted least square (WLS) is the most common optimization method to do this.

With the increase of system size and complexity, faster state estimation is needed. Nieplocha et al. (2006) compared a parallel LU solver and a conjugate gradient-based parallel solver with Jacobi preconditioner for WLS state estimation on a 1177-bus system with 1770 lines and 6144 measurements. The results showed that

CG-based solver was about 4.75 times faster than the state-of-the-art parallel LU solver. Chen et al. (2013) proposed two methods to solve WLS state estimation: gain matrix factorization and QR factorization. A preconditioner called ParaSail from a software library Hyper was integrated with conjugate gradient method to solve the gain matrix factorization. Another software package PETSc were used to solve the QR factorization. Both methods were implemented parallel. A speedup around 5x for both cases was reported with the practical system with 7500 buss and 9300 branches from Bonneville Power Administration (BPA). Karimipour and Dinavahi (2013) implemented parallel WLS state estimation on a GPU simulator and reached 38 times speedup compared with CPU for a 4992-bus system.

2.2 Iterative linear solvers

Iterative solver is an alternative to those well known direct solvers. With the development and availability of parallel computation in terms of both hardware and software, iterative solver gradually gets more attention for its scalability and inherited parallelism. This section will introduce some iterative solvers in engineering computation.

One of the most successful and popular methods to solve positive definite symmetric (SPD) linear systems is the conjugate gradient method. Shewchuk (1994) introduced the essential concepts for conjugate gradient methods and its convergence analysis. It stated and proved that the conjugate gradient method would converge and give an exact solution after n iterations, and that CG would be quicker if there were repeated eigenvalues. Practically, clustered eigenvalues could give better convergence properties.

Jennings (1977) discussed the influence of eigenvalue spectrum to the convergence rate of CG method. The rule of thumb was that clustered eigenvalue would yield better performance and using the close-to-minimum and close-to-maximum eigenvalue in CG instead of the actual minimum and maximum eigenvalue would lead to a better convergence. This work also presented an estimated upper bound for iterations required for CG methods to converge based on the spectrum of eigenvalues, which was much tighter than the commonly accepted upper bound n. However, there are no analytical functions available for the relationship between the eigenvalue spectrum and the convergence rate.

2.3 Preconditioner

To make an iterative solver work efficiently, a good preconditioner is indispensable. A preconditioner is used to improve the spectrum of the original matrix; the closer the better. The ideal preconditioner of matrix A would be its inverse, A^{-1}. Then, the preconditioned matrix $A * A^{-1}$ would be the identity matrix I, which theoretically will have the iterative solver converge in only one iteration. However, it is well known that the precise inverse of a matrix is generally too difficult or too expensive to obtain, and usually much more costly than solving the matrix directly. Therefore an approximation of A^{-1} would be a usual form of the preconditioner. A good preconditioner should be evaluated in two ways. First, it should make a good approximation of A^{-1}. Second, it should not be too expensive to form. This section will introduce some commonly deployed preconditioners.

Meijerink and Van der Vorst (1977) introduced the general incomplete LU (ILU) factorization. It stated that using ILU and conjugate gradient method together can produce a stable and fast convergence for M-matrix [*]. Incomplete Cholesky conjugate gradient (ICCG) as a special case of ILU was also proposed. Results showed that ICCG(0) and ICCG(3) were preferred for large scale linear systems. The authors also had a followed up work Meijerink and Van der Vorst (1981) which demonstrated the method using practical examples, such as systems that arose from periodic boundary conditions, M-matrix with an arbitrary structure etc. Gustafsson (1978) proposed a modified ILU factorization (MILU) which employed an elimination error matrix

[*]M-matrix: Off-diagonal elements $A_{ij} \leq 0$, and diagonal elements $A_{ii} > 0$.

R to compensate the error generated in incomplete factorization. Gallivan et al. (1990) proposed to construct approximate LU and QR factorization preconditioner by ignoring the small elements during the decomposition of matrix A, and use a heuristic scheme to find a proper threshold τ to keep or discard each element. Results suggested that the dropping strategy based on such numerical dropping was often better than the dropping strategy based on position such as ILU or IC. However, the heuristic process to find the proper threshold τ was time-consuming.

It was discovered that incomplete factorization may suffer instability issues. Elman (1989) discussed the instability problem of incomplete factorization preconditioners for asymmetric linear systems and proposed stabilized incomplete factorizations in order to construct numerically stabled factorization and preconditioning. Therefore, there were some works that focused on constructing an approximate inverse of the original matrix A. This category was usually discussed together with parallel computation techniques to serve as the preconditioner. Chow and Saad (1998) presented several algorithms to construct approximated inverse by converting the preconditioner construction problem to a minimization process, which minimized the Frobenius norm for its inherited parallelism. Since it was highly possible that the inverse of a sparse matrix can be a dense matrix, this work also proposed several numerical dropping strategies so that the inverse preconditioner could stay sparse. Benzi and Tuma (1998) discussed a preconditioner construction based on the inversion of the LU factorization, and the corresponding algorithm of eliminating fill-ins was presented as well. Dehnavi et al. (2013) targeted at the minimization of $||AG - I||_F$, too. However, they had their whole set of implementation on GPU to boost the computation efficiency of construction of the preconditioner. The performance of their GPU implementation could compete with the main stream open source linear computation package ParaSail.

Another category of preconditioner is polynomial preconditioner. Polynomial preconditioner is based on the approximation theory. As a result, there were two major methods to approximate the inverse of the original matrix. First category is based on Chebyshev polynomial to uniformly approximate the inverse of A. For hermitian positive definite matrix, the Chebyshev polynomial could be applied Ashby et al. (1989). Dag and Semlyen (2003) introduced conjugate gradient with Chebyshev preconditioner into power system application. The second category is based on the least square minimization approximation which minimizes the norm of the difference between the preconditioned matrix and the unitary matrix. Johnson et al. (1983) discussed a generalized optimal polynomial preconditioner based on the quadratic norm of the residual polynomial. They presented a large class of weight functions which were positive definite. Recurrence relations for both the Chebyshev polynomial and the least square polynomial were given. Saad (1985) and Ashby et al. (1992) discussed polynomial preconditioner based on least square polynomials for symmetric positive definite matrices and also presented a comparison between Chebyshev polynomial preconditioner and least square preconditioner. Ashby et al. (1989) extended the work to hermitian indefinite matrices, and proposed an adaptive method to construct the preconditioner. Their experiments showed at best a 50% performance improvement over original conjugate gradient method. Liang et al. (2002) proposed a generalized least square polynomial preconditioning for symmetric indefinite system. The preconditioner was applied with flexible generalized minimized residual (FGMRES) solver. Zhang and Zhang (2013) implemented a least square polynomial preconditioner on GPU for the practical linear systems that arose from elasticity finite element equations. Results shown GPU-based LS preconditioned CG outperformed the CPU implementation 7-9 times speedup.

2.4 Integration of GPU in linear system solving

GPU as an efficient accelerator in scientific computation has been applied to both power system applications and other engineering applications. This section will present the works integrating GPU and related algorithms to accelerate the linear computations.

Helfenstein and Koko (2012) proposed a GPU-based parallel implementation of preconditioned conjugate gradient method to solve the generalized Poisson equation. The preconditioner used in this work was an approximated inverse matrix from a symmetric successive order-relaxation (SSOR) preconditioner and was named as SSOR-AI. The experiment showed that the proposed preconditioner implemented on GPU could gain a speedup between 8 and 10 over the corresponding implementation on CPU with test matrices sized from 26,000 to 2,100,000. Buatois et al. (2009) implemented a general sparse linear system solver using the conjugate gradient method and Jacobi preconditioner to perform mesh smoothing in image processing. It also presented the results of BLAS computation on CPU, AMD-ATI and NVIDIA GPU. The results shown that every processor had its own strength.

The most computationally intensive computation in CG method is sparse matrix-vector multiplication (SpMV). Gui and Zhang (2012) proposed a novel storage format for SpMV named modified diagonal storage format (mDIA) to access the elements more efficiently. The Jacobi iterative method with incomplete Cholesky was implemented to explore the parallelism. A speedup more than 7 was reported for matrices sized from 22,000 to 304,000 and generated from the practical Poisson equation. Zhang and Zhang (2013) employed a least-squares polynomial method as the preconditioner and solved the finite element computation with conjugate gradient solver. This work also proposed a sliced block ELLPACK storage format to store the sparse matrix more efficiently. Based on the storage, an efficient sparse matrix vector multiplication kernel was implemented. Results showed that it can solve their specific application more efficiently than the standard libraries. This implemented a mixed precision polynomial preconditioned conjugate gradient solve as well. The SBELL sparse storage and mixed precision conjugate gradient could reach over 7x speedup over CPU implementation for different meshes.

Chapter 3

Fundamentals of Power Flows and GPU Computations

Power flow is a fundamental computation for power system analysis. Many power system applications such as optimal power flow, contingency, operation and planing will first need to perform power flow computation. To solve power flow, Newton-Raphson method will be deployed first to transform the system from non-linear to linear. GPU as a newly developed hardware platform can help to enhance the computation efficiency of linear systems. This chapter starts with the introduction of background of power flow computations, and is followed by the introduction of solving a linear system. Finally, GPU for general purpose computation will be introduced.

3.1 Power system computations

Power system is modeled as a large set of non-linear equations. Other than some non-linear systems in very specific form which can be solved directly, most non-linear equations will be transformed into a set of linear equations to gradually get the numerical solution of the original non-linear system. The most common method used to convert the non-linear to linear systems is the Newton-Raphson method (also known as Newton's method). At this point, power flow computations such as power flow analysis and fast decoupled power flow can be solved by solving a sequence of linear equations. This section will introduce the computations in power system analysis and the involved linear system computations.

3.1.1 AC power flow

Given the loads and the generation and transmission network, a power flow problem solves the system bus voltage and line flows. For each bus in the system, based on Kirchoff's law, the sum of the power entering a bus should be equal to the power leaving the same bus. In other words, the injection should be equal to the consumption. Each bus has two equations, one for active power, and one for reactive power. Therefore, the power flow for bus i can be formulated as equation 3.1 shows.

$$\begin{aligned} P_i^{inj} - V_i \sum_{j=1}^{N_{bus}} V_j Y_{ij} cos(\delta_i - \delta_j - \phi_{ij}) = \Delta P = 0 \\ Q_i^{inj} - V_i \sum_{j=1}^{N_{bus}} V_j Y_{ij} sin(\delta_i - \delta_j - \phi_{ij}) = \Delta Q = 0 \\ i = 1, 2, ..., N_{bus} \end{aligned} \tag{3.1}$$

N_{bus} is the number of buses in the system. P^{inj} and Q^{inj} are the active and reactive injections for bus i, respectively. V_i is the load voltage magnitude on bus i. δ_i is the phase angle at bus i. Y_{ij} and ϕ_{ij} come from admittance matrix Y_{bus}. Y_{ij} is the magnitude and ϕ_{ij} is the angle of the admittance between bus i and bus j. The mismatch ΔP and ΔQ measure the difference between the injection and calculated power values, which should be 0. The unknowns in equation 3.1 are the phase angle δ and the voltage magnitude V. Y_{bus}, and injections P_{inj} and Q_{inj} are already given.

To solve the nonlinear model in equation 3.1, Newton-Raphson method is applied to transform them into linear equations and then to iteratively solve a sequence of linear equations with similar form as equation 3.2 to get the solution of the original nonlinear equations.

$$Jx = -F; x = \begin{bmatrix} \Delta\delta \\ \Delta V \end{bmatrix} \tag{3.2}$$

The Jacobian matrix can be rewritten as J_1, J_2, J_3 and J_4 four parts, listed from equation 3.4 to 3.11.

$$J = \begin{bmatrix} J_1 & J_2 \\ J_3 & J_4 \end{bmatrix} = \begin{bmatrix} \frac{\partial \Delta P}{\partial \delta} & \frac{\partial \Delta P}{\partial V} \\ \frac{\partial \Delta Q}{\partial \delta} & \frac{\partial \Delta Q}{\partial V} \end{bmatrix} \tag{3.3}$$

$$J_1 \begin{cases} \dfrac{\partial \Delta P_i}{\partial \delta_i} = V_i \displaystyle\sum_{j=1}^{N_{bus}} V_j Y_{ij} sin(\delta_i - \delta_j - \phi_{ij}) + V_i^2 Y_{ii} sin\phi_{ii} & (3.4) \\ \dfrac{\partial \Delta P_i}{\partial \delta_j} = -V_i V_j Y_{ij} sin(\delta_i - \delta_j - \phi_{ij}) & (3.5) \end{cases}$$

$$J_2 \begin{cases} \dfrac{\partial \Delta P_i}{\partial V_i} = -\displaystyle\sum_{j=1}^{N_{bus}} V_j Y_{ij} cos(\delta_i - \delta_j - \phi_{ij}) - V_i Y_{ii} cos\phi_{ii} & (3.6) \\ \dfrac{\partial \Delta P_i}{\partial V_j} = -V_i Y_{ij} cos(\delta_i - \delta_j - \phi_{ij}) & (3.7) \end{cases}$$

$$J_3 \begin{cases} \dfrac{\partial \Delta Q_i}{\partial \delta_i} = -V_i \displaystyle\sum_{j=1}^{N_{bus}} V_j Y_{ij} cos(\delta_i - \delta_j - \phi_{ij}) + V_i^2 Y_{ii} cos\phi_{ii} & (3.8) \\ \dfrac{\partial \Delta Q_i}{\partial \delta_j} = V_i V_j Y_{ij} cos(\delta_i - \delta_j - \phi_{ij}) & (3.9) \end{cases}$$

$$J_4 \begin{cases} \dfrac{\partial \Delta Q_i}{\partial V_i} = -\displaystyle\sum_{j=1}^{N_{bus}} V_j Y_{ij} sin(\delta_i - \delta_j - \phi_{ij}) + V_i Y_{ii} sin\phi_{ij} & (3.10) \\ \dfrac{\partial \Delta Q_i}{\partial V_j} = -V_i Y_{ij} sin(\delta_i - \delta_j - \phi_{ij}) & (3.11) \end{cases}$$

After solving equation 3.2, the mismatch $\Delta\delta$ and ΔV will be added to the value from last iteration as equation 3.12 shows.

$$\begin{aligned} \delta^k &= \delta^{k-1} + \Delta\delta \\ V^k &= V^{k-1} + \Delta V \end{aligned} \tag{3.12}$$

The Jacobian matrix J is the linear system that we will solve. It has the following characteristics: 1) very sparse; 2) symbolic symmetric; 3) numerically asymmetric. Sparsity is determined by how the power transmission network is connected. Symbolic symmetry is derived from how power flow equation is formulated. Numerical asymmetry can be concluded from the equations above.

The Jacobian matrix and the right hand side $\Delta\delta$ and ΔV will be updated every Newton-Raphson iteration until the largest magnitude of mismatch vector x is smaller than a predefined value.

Solving power flow take up a significant part of power system analysis execution time. In order to improve the analysis computation efficiency, power flow computations should be considered first.

3.1.2 Fast decoupled power flow

Power flow, as the most computationally intensive part in power system analysis, has its various methods used to simplify the computation.

Since transmission networks usually have very small resistance value (r), ϕ_{ij} is usually close to $\pm 90^o$. Also, the buses close to each other tend to have smaller phase angle difference, which leads to small $(\delta_i - \delta_j)$. Then $cos(\delta_i - \delta_j - \phi_{ij}) \approx 0$. With such approximation, J_2 and J_4 can be ignored as zero matrices. Then, the power flow can be simplified to equation 3.13, which is named as decoupled newtown method Stott and Alsac (1974). The power flow equation could be solved by solving equation 3.13.

$$\begin{bmatrix} J_1^k & 0 \\ 0 & J_4^k \end{bmatrix} \begin{bmatrix} \Delta\delta^k \\ \frac{\Delta V^k}{V} \end{bmatrix} = \begin{bmatrix} \Delta P^k \\ \Delta Q^k \end{bmatrix} \tag{3.13}$$

Equation 3.13 has different J_1 and J_4 every Newton-Raphson iteration. It can be further simplified to equation 3.14 with fixed iteration matrix B' and B''. To get to this formulation, we assume that 1) the real part of admittance matrix is close to 0, which means that $g_{ij} \approx 0$; 2) all voltage magnitudes are equal to 1 p.u.

$$\begin{bmatrix} B' & 0 \\ 0 & B'' \end{bmatrix} \begin{bmatrix} \Delta\delta^k \\ \Delta V^k \end{bmatrix} = \begin{bmatrix} \Delta P^k \\ \Delta Q^k \end{bmatrix} \tag{3.14}$$

The matrix B' and matrix B'' are defined in equation 3.15 to 3.18. x_{ij} is the reactance of each bus and b_{ij} gives the susceptance between bus i and bus j. Please note that the formulation in equation 3.15 to 3.18 is based on XB version of fast decoupled power flow Stott and Alsac (1974).

$$B'_{ij} = -\frac{1}{x_{ij}} \tag{3.15}$$

$$B'_{ii} = -\sum_{j \neq i} B'_{ij} \tag{3.16}$$

$$B''_{ij} = -b_{ij} \tag{3.17}$$

$$B''_{ii} = -b_i - \sum_{j \neq i} B''_{ij} \tag{3.18}$$

3.1.3 Linearized power flow

Linearized power flow is a linear model used to describe the power system. It can generate a power flow result faster than AC power flow, but it is usually less accurate. It ignores the reactive power and line conductance, and assumes that all the voltages are 1 p.u.. With these assumptions, the equation describing linearized power flow is given in equation 3.19.

$$B\delta = -P \tag{3.19}$$

Equation 3.20 and 3.21 give definitions for linearized power flow. x_{ij} is the reactance between bus i and bus j.

$$B_{ij} = -\frac{1}{x_{ij}} \tag{3.20}$$

$$B_{ii} = -\sum_{j \neq i} B_{ij} \tag{3.21}$$

3.1.4 Positive definiteness of FDPF and DCPF

The Jacobian matrix in equation 3.3 from AC power flow is symbolic symmetric, numerically asymmetric. However, the B' and B'' (equation 3.14) from fast decoupled power flow and matrix B (equation 3.19) are all symmetric. This section will discuss the positive definiteness of matrix B', B'' and B.

We will start with discussions on the positive definiteness of B from linearized power flow. Except for very rare cases, x_{ij} will usually be positive, $x_{ij} > 0$, when $i \neq j$. Therefore, all of the off-diagonal elements $B_{ij} < 0$. The diagonal elements B_{ii} is equal to the negative sum of all off-diagonal elements of row i, which implies equation 3.22. From equation 3.21, we have equation 3.23.

$$B_{ii} > 0 \tag{3.22}$$

$$|B_{ii}| = \sum_{j=1, j \neq i}^{N_{bus}} |B_{ij}| \tag{3.23}$$

Equation 3.22 and 3.23 give the formulation before the elimination of reference bus. Assume bus k is the reference bus, then row k will be eliminated during computation or the matrix will be singular. Without loss of generality, we assume bus l is one of the buses connected to bus k. Then the row for bus l is shown in equation 3.24.

$$B_{ll} = -\sum_{j=1,j\neq l}^{N_{bus}} B_{lj} \begin{cases} > -\displaystyle\sum_{j=1,j\neq l,j\neq k}^{N_{bus}} B_{lj} & \text{if } B_{lk} \neq 0 \qquad (3.24a) \\ = -\displaystyle\sum_{j=1,j\neq l,j\neq k}^{N_{bus}} B_{lj} & \text{if } B_{lk} = 0 \qquad (3.24b) \end{cases}$$

Then, for bus l we have:

$$|B_{ll}| \geq \sum_{j=1,j\neq l,j\neq k}^{N_{bus}} |B_{lj}| \tag{3.25}$$

There exists as least one reference bus l in each system so that the equation 3.23 after reference bus elimination will be as equation 3.26 shows. Please note that this conclusion still holds when there are several reference buses in the system. Since matrix B is symmetric, the conclusion also automatically holds for any column.

$$|B_{ii}| \geq \sum_{j=1,j\neq i}^{N_{bus}} |B_{ij}| \tag{3.26}$$

Equation 3.22 and 3.26 qualify the matrix B to a diagonally dominant matrix, which is always positive semidefinite. Since our discussion is based on stable system status, singular case is not within the scope of this work. Therefore we will consider matrix B to not have a zero eigenvalue. As a result, matrix B is positive definite as equation 3.27 shows. The notation "$>$" with a matrix will be used to mark that the matrix is positive definite throughout this work.

$$B > 0 \tag{3.27}$$

Matrix B' from fast decoupled power flow is defined in equation 3.15 and 3.16, which has exactly the same formulation as equation 3.20 and 3.21. Therefore, with similar deduction as above, we can conclude that matrix B' from FDPF is positive definite if all the reactance is positive.

$$B' > 0 \tag{3.28}$$

Matrix B'' from fast decoupled power flow is slightly different from matrix B and matrix B' because of the influence of susceptance. The susceptance b_{ij} in equation 3.17 and equation 3.18 is typically positive, $b_{ij} > 0$. b_i is the shunt susceptance of all the shunt branches connected to bus i Crow (2002). Please note that b_i is usually much smaller than $\sum_{j \neq i} b_{ij}$.

Assume that the reference buses and PQ buses are in set $\mathcal{R}$. S_{off} is the sum of the absolute value of all the off-diagonal elements after the reduction of reference buses and PQ buses, or in other words, the absolute value of all of the off-diagonal elements in the set of PV buses, then

$$\begin{aligned} |B''_{ii}| &= \sum_{j \neq i} |B''_{ij}| - b_i \\ &= \sum_{j \neq i, j \notin \mathcal{R}} |B''_{ij}| + \sum_{j \in \mathcal{R}} |B''_{ij}| - b_i \\ &= S_{off} + (\sum_{j \in \mathcal{R}} |B''_{ij}| - b_i) \end{aligned} \tag{3.29}$$

Since $b_i << B_{ij}$

$$\sum_{j \in \mathcal{R}} |B''_{ij}| - b_i > 0 \tag{3.30}$$

From equation 3.29 and equation 3.30 :

$$|B''_{ii}| > S_{off} > 0 \tag{3.31}$$

Therefore, the absolute value of diagonal elements of B'' is greater than the sum of the absolute value of all off-diagonal elements. Hence, matrix B'' is a diagonally dominant matrix. Besides, $\forall i \in N_{bus}$, $B''_{ii} > 0$. Matrix B'' is a positive semidefinite matrix too if there is no negative reactance in the system. If we don't consider the cases that system is singular, then

$$B'' > 0 \tag{3.32}$$

As a short conclusion for this subsection, Jacobian matrix J from AC power flow is a numerically asymmetric, but is a symbolic symmetric matrix; matrix B from linearized power flow, matrix B' and matrix B'' from fast decoupled power flow are all positive definite ($B > 0, B' > 0, B'' > 0$) if $\forall j \in N_{bus}$, $x_{ij} > 0$.

3.2 Solving linear systems

Including power system applications, many engineering problems are nonlinear. However, most of them could be solved in linear forms. The discussions in this section will be built on a general linear system formulated as equation 3.33. The size of the linear system A is $n \times n$, and the size of right hand side vector b is $n \times 1$. The vector x is the unknown to be solved.

$$Ax = b \tag{3.33}$$

Generally, there are two categories of method to solve linear systems: direct methods and iterative methods. The direct methods will get to the precise solution through a finite number of arithmetic operations. Once a component in the solution vector is computed in direct method, it will be a part of the final solution. There will not be any further refinement of it. However, iterative methods will generate a set of solutions (for example, $\mathbf{x^0}$, $\mathbf{x^1}$, $\mathbf{x^2}$, ..., $\mathbf{x^k}$, please note that each $\mathbf{x^k}$ is a vector), and each set of the solution is expected to get closer to the real solution $\mathbf{x}^*$. Therefore the solution given in every iteration is an approximation to the precise solution. The iterative methods will stop with respect to a predefined error threshold if they can converge, or they will reach the maximum iteration and then quit the iterations.

This section will briefly introduce the common direct methods and iterative methods. Please note that the discussions in this work are all based on full rank matrix.

The choice between direct methods and iterative methods, along with which direct method or which iterative method to use, is varied among the system to be solved. Some methods are more favorable to some systems than the others. However, the general rule is to use iterative methods for large scale sparse systems, and to use direct methods for the rest.

3.2.1 Direct method

Gaussian elimination is the first branch in this category. It will include using the right hand side to form a augmented matrix (Figure 3.1b), then arithmetic operations will follow to transform the original general matrix A (Figure 3.1a) to a upper triangular matrix (Figure 3.1c), named the forward elimination, and at last a backward substitution to reach the solution for the linear system $Ax = b$.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}$$

(a) Original linear system

$$\left[\begin{array}{ccc|c} a_{11} & a_{12} & a_{13} & b_1 \\ a_{21} & a_{22} & a_{23} & b_2 \\ a_{31} & a_{32} & a_{33} & b_3 \end{array}\right]$$

(b) Augmented system

$$\left[\begin{array}{ccc|c} t_{11} & t_{12} & t_{13} & b'_1 \\ 0 & t_{22} & t_{23} & b'_2 \\ 0 & 0 & t_{33} & b'_3 \end{array}\right]$$

(c) Triangular format

Figure 3.1: Gaussian elimination

In many practical engineering problems, the common case used to solve equations leaves with matrix part unchanged, however the right hand side keeps updating. Methods based on decomposition isolate the right hand side and have gradually become the most widely used methods. LU decomposition is the most immediately development from Gaussian elimination. It will decompose the matrix A into two triangular matrices: the lower triangular part L and the upper triangular part U, then $A = LU$. With the introduction of a dummy vector y, the original equation can be solved by one forward substitution to get $Ly = b$ and then one backward substitution to get solution x by solving $Ux = y$.

There are other forms of decomposition that can solve the equations, but they usually have other focuses with the price of increased computation complexity. For example, QR decomposition will factorize the matrix A into matrix Q and R, where Q is a orthogonal matrix while R is an upper triangular matrix. This can be used to solve the system, but it is considered more as a procedure to calculate the eigenvalue and eigenvector of the matrix. Similarly, singular value decomposition (SVD) will give much more information of the matrix such as singular values, a set of the matrix's regular basis, and it can be used for non-square matrix. Cholesky decomposition will factorize the matrix into two identical triangular matrices L that $A = L^T L$, but it can only be used for symmetric positive definite (SPD) matrix.

When the matrix under discussion is sparse, which all the power system applications are, direct methods such as LU and Cholesky usually require a reordering step to reduce the potential fill-ins during the decomposition procedure.

Generally, the computation complexity for direct method is $\mathcal{O}(n^3)$ while the memory requirement would be $\mathcal{O}(n^2)$ due to the storage needs for permutation matrix P, and decomposed matrix L and U etc. The quadratic increasing rate of memory requirement of direct method is its major drawback. Additionally, the algorithm itself is intrinsically serial. These two points together make direct methods less ideal for large scale systems.

3.2.2 Iterative method

Instead of getting the solution directly, another direction is to generate a set of solutions and the set from each iteration is expected to get closer to the actual solution; hence the name iterative methods. Conjugate gradient method is one of the most prominent one if can be applied Crow (2002).

Conjugate Gradient Method

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 equation 3.34 shows:

$$f(x) = \frac{1}{2}x'Ax - b'x + c \tag{3.34}$$

In equation 3.34, A is a symmetric positive definite matrix and b is a vector. The derivative of equation (2) is $f(x) = Ax - b$. Therefore the x which provides the minimum value of equation 3.34 satisfies $Ax - b = 0$. Thus, it is the solution of $Ax = b$ as well.

An intuitive way to find out the minimum value of equation 3.34 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. Therefore the first advantage of conjugate gradient method is the iterations for convergence is bounded to n.

The convergence rate of CG is shown in equation 3.35. Assume that x_*, x_0 and x_m are the precise solution, the starting and the current solution vector, respectively. κ is the condition number. Therefore, the upper bound of error between current and the precise solution after m iteration is reduced superlinearly from the initial error. Equation 3.35 shows that smaller condition numbers could lead to a tighter boundary for the current error. As a result, a narrow eigenvalue range can have the

```
1: Initialization
2: $r = b - Ax_0; x = x_0, p_1 = r_0;$
3: while $||r_k||$ >tolerance do
4:     $\alpha_k = r_{k-1}^T r_{k-1} / p_k^T A p_k$        $(4n + nnz \times n)$ FLOPS
5:     $x_k = x_{k-1} + \alpha_k p_k$                     $(2n)$ FLOPS
6:     $r_k = r_{k-1} - \alpha_k A p_k$                   $(2n)$ FLOPS
7:     $\beta_{k+1} = r_k^T r_k / r_{k-1}^T r_{k-1}$      $(2n)$ FLOPS
8:     $p_{k+1} = r_k + \beta_{k+1} p_k$                  $(2n)$ FLOPS
9: end while
```

Program 3.1: Algorithm of general conjugate gradient method

error dropped faster and hence improve the convergence rate to make the CG iterative solver more efficient. Transforming the eigenvalue spectrum calls for a preconditioner.

$$||x_* - x_m||_A \leq 2(\frac{\sqrt{\kappa}+1}{\sqrt{\kappa}-1})^m ||x_* - x_0||_A \tag{3.35}$$

The other advantage of conjugate gradient solver is its low memory requirement. A detailed algorithm is in Program 3.1. It shows that the storage of conjugate gradient method requires only the matrix A itself and four vectors (x_k, r_k, p_k and Ap_k). Since matrix A is commonly stored as sparse format, the total memory requirement of conjugate gradient is $\mathcal{O}(n)$. In this case, the memory requirement will not increase dramatically when the system size becomes large. In other words, the conjugate gradient method is quite scalable.

Other than the low memory requirement, the FLOPS of every steps needs are shown in Program 3.1. The computation complexity of each iteration as Program 3.1 shows is $(12n + nnz \times n)$. Also, assume that the number of iterations for the algorithm to converge is c, then the total computation complexity would be $(12n + nnz \times n) \times c$. Assume $nnz = \tau n$, τ is usually a small integer and much less than n. Then the total computation complexity is $(12n + \tau n \times n)c = \mathcal{O}(\tau n^2)c = \mathcal{O}(c\tau n^2)$. Since CG method is guaranteed to converge with in n iterations, the worst case of c is n. However, when matrix A is sparse, it is common that $c \ll n$. Then the computation complexity of CG method could be estimated to be $\mathcal{O}(n^2)$ for sparse matrix. According to section 3.2.1, the computation complexity of LUD is usually estimated to be $\mathcal{O}(n^3)$. This discussion states that the conjugate gradient method is scalable from both space and time complexity.

3.2.3 Preconditioner

Preconditioner transforms the original matrix A to unitary matrix by linear transformations. However, it is well known that the calculation of the precise inverse of a matrix, especially large scale matrix, is quite time-consuming. Therefore the practical preconditioner is usually an approximation of A^{-1} so that G can be constructed without too much computational overhead but can serve the purpose of transforming the preconditioned matrix close to an unitary matrix. A brief introduction of different preconditioners is followed.

Jacobi preconditioner

Jacobi preconditioner has the simplest form among all the widely used preconditioners. If we assume M is the matrix to approximate original matrix A, equation 3.36 shows how matrix M is formulated. It is mainly applied for matrices with diagonal dominance.

$$M = \begin{cases} A_{ij} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{3.36}$$

Naturally, the preconditioner G could be constructed as followed.

$$G = \begin{cases} \frac{1}{A_{ij}} & \text{if } i = j \\ 0 & \text{if } i \neq j \end{cases} \tag{3.37}$$

Incomplete factorization

Incomplete factorization will decompose the matrix A to two triangular matrices which share *similar* non-zero patterns as the original matrix. Incomplete LU (ILU) and incomplete Cholesky (IC) factorization are two common methods of incomplete factorization. Since IC can be considered as a special case of ILU, we will take ILU as the example for discussion.

ILU will decompose the original matrix A into two matrices: L_I and U_I, and then $M = L_I U_I$. Matrix M will be close to A that $M \approx A$. Assume that the complete decomposition of A is $A = LU$, then L_I and U_I from incomplete factorization are much sparser than L and U from complete factorization, and hence take much less memory space. If the factorization of L_I and U_I follows exactly the same non-zero pattern of matrix A without introduction of any extra storage, it is named ILU(0). If the factorization of L_I and U_I follows the sparsity pattern of A^2, it is named ILU(1). Generally, ILU(k) follows the non-zero pattern of $A^{(k+1)}$. If Cholesky factorization is deployed instead of LU factorization following the same method above, they are named as IC(0), IC(1) and IC(k) accordingly. IC can be considered as a special case of ILU when $L = U$, which needs only half of the storage of ILU methods. However, IC requires that matrix A to be symmetric positive definite.

The deeper degree of ILU or IC can lead to a more accurate approximation of A. However, the memory required to store the triangular matrices will increase accordingly.

Other than the commonly seen ILU(k) and IC(k), there are also some other variants of incomplete factorization, please refer to section 2.3 or following references for more information. Meijerink and Van der Vorst (1977), Gustafsson (1978), Meijerink and Van der Vorst (1981), Axelsson and Lindskog (1986), Elman (1989), Notay (1994) Gallivan et al. (1990) etc.

Sparse approximate inverse

ILU methods have an major issue; it may yield very ill-conditioned factorization Van der Vorst (1981) Elman (1986) Chow and Saad (1997). This shortcoming coupled with its sequential essence prevents its popularity in solving large scale sparse matrices. Since the goal of preconditioner is to approximate the inverse of matrix A, developing an approximate inverse matrix is intuitive.

To define the *closeness* of matrix G to A^{-1}, equation 3.38 is applied. r measures how close the product of A and G is to the unitary matrix I. Ideally, if $G = A^{-1}$, r would be 0. The norm could be 2-norm, F-norm or infinity norm, decided by computational overhead or availability. Naturally, the construction of matrix G turns into a minimization process of r.

$$r = ||AG - I|| \tag{3.38}$$

Another advantage of sparse approximate inverse is that it can be applied to cases that can not be associated. Factorization-based preconditioners usually require the association of the factorization matrices with the original matrix at some point during the computation, while approximate inverse can be plugged in directly with the form $GAx = Gb$ and then solve it. One example of computation scenario is interval power flow. Chow and Saad (1998) Benzi and Tuma (1998) provided detailed discussion on how to generate approximate inverse matrix parallel.

Polynomial preconditioner

Polynomial preconditioner has several advantages. First is that a polynomial of a matrix commutes with the matrix itself. Therefore, $P(A)A$ is still hermitian matrix if the original matrix A is hermitian. In our discussion, matrix A is real, therefore $P(A)A$ will always be symmetric if A is symmetric. Second is that it requires only matrix vector multiplications and vector additions which are easier to parallelize.

If A can be normalized to the form $A = I - B$ and spectrum radius of B is less than 1 ($||B|| \leq 1$). The Neumann series is a power series of G with unit coefficient as equation 3.39 shows.

$$A^{-1} = (I - B)^{-1} = I + B + B^2 + B^3 + \cdots = P_n(B) \tag{3.39}$$

Equation 3.39 can be considered as a special case of equation 3.40. The preconditioner G is a polynomial form of B, therefor is a polynomial form of A too. d in equation 3.40 is the degree of the polynomials for truncating the polynomial purpose or there will be infinite factors in the polynomial.

$$G = \sum_{j=0}^{d} \gamma_j A^j \equiv s(A) \tag{3.40}$$

To this point, the purpose of polynomial preconditioning has been to minimize $||I - GA||$. Equation 3.41 turns the matrix norm to polynomial formulation based on eigenvalues λ of matrix A.

$$||I - GA|| = ||I - s(A)A|| = \max |1 - \lambda s(\lambda)| \tag{3.41}$$

From the polynomial perspective, minimization of $||I - GA||$ is equivalent to making polynomial G approximate A as close as possible. Therefore, approximation theory can be applied here. The first type of method is to minimize the upper bound of $||I - GA||_\infty$, called min-max approximation, and the second type of method is least squared approximation, which minimize $||I - GA||_2$.

Chebyshev polynomial of the first kind gives an approximation that is close to the polynomial of the best approximation to a continuous function under infinity norm. Therefore, Chebyshev polynomial can be applied to solve the min-max problem of $||1 - \lambda s(\lambda)||$ to get the uniform approximation.

The second category minimizes the L_2 norm of $||1 - \lambda s(\lambda)||$. In order to intervene in the influence of some small or unfavored eigenvalues, a weight function $w(\lambda)$ is introduced. Least squared approximation is defined in equation 3.42.

$$||1 - \lambda s(\lambda)||_w = \int (1 - \lambda s(\lambda))^2 w(\lambda)\, \mathrm{d}\lambda \tag{3.42}$$

The polynomial $s(A)$ will be constructed either by Chebyshev polynomial or by L_2 norm minimization. Then $s(A)A$ forms the preconditioned matrix of A.

3.3 GPU computation

3.3.1 GPU and CUDA

Graphic Processing Unit (GPU) for general purpose computations has been recently widely deployed. GPU was originally designed for graphic processing, which requires intensive floating point computations. Before the release of CUDA, there were some 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 the 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 to 512 cores, while the mainstream CPU has 12 to 16 cores on chip due to the power and cooling limitations. GPU inherits the parallel computing advantage it has as a graphi CUDAc 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 Top500.org (2014). 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. 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.

3.3.2 CUBLAS and CUSPARSE

Basic Linear Algebra Subroutine (BLAS) Anderson et al. (1987) is a commonly used linear algebra library. CUDA Basic Linear Algebra Subroutines (CUBLAS) 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.

3.3.3 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) 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.

Chapter 4

Using Conjugate Gradient Method and Chebyshev Preconditioner in Power System Applications

To solve a linear system, one can either use direct method based on decomposition such as LU or Cholesky, or apply iterative solvers with considerations of scalability and parallelism. This chapter will discuss solving positive definite systems, such as the linearized DC power flow, and fast decoupled power flow. In this chapter, a polynomial preconditioner Chebyshev preconditioner will be implemented with graphic processing unit (GPU) and integrated with a GPU-based conjugate gradient solver for linearized DC power flows.

As Section 3.1 introduced, 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 the power flow of large systems in MATPOWER. The results show that about 40% to 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. On one hand, iterative methods have been adapted to power system computation in various aspects. Pai et al. (1992) have implemented the Generalized Minimal Residual (GMRES) method on a Cray machine for dynamic power system simulation. Pai and Dag (1997) further applied several iterative solvers including conjugate gradient and GMRES to dynamic power flow simulation and state estimation. On the other hand, the 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. GPU as a co-processor helps a commodity server deliver more computational throughput.

Section 4.1 takes a closer look at the polynomial preconditioner Chebyshev preconditioner. Section 4.2 presents the algorithm that this work uses and the corresponding GPU-based implementation. Computational experiments are shown in Section 4.3. A further discussion is extended in Section 4.5. Section 4.6 closes this chapter.

4.1 Iterative solver and preconditioner

4.1.1 Iterative solver

Iterative solver is getting more attention in processing large scale power system models because of its scalability and low memory requirement as section 3.2.2 discussed. Conjugate gradient method is one of the iterative solvers with promising convergence property. The algorithm of iterative solver can be found in Table 3.1. Conjugate gradient method require symmetric positive definite systems, therefore based on the power system application discussions in Section 3.1.2, 3.1.3 and their positive definiteness discussion in 3.1.4, linearized DC power flow and fast decoupled power flow can be solved by conjugate gradient solvers.

4.1.2 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 to converge. Ideally, the preconditioning matrix G would be the 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 equation 4.1 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 equation 4.2. T_k is the recurrence formulation of Chebyshev polynomial as equation 4.3 shows. These polynomials are orthogonal. c_k and constant q are defined as equation 4.4 and equation 4.5, 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 and Semlyen (2003).

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

$$A^{-1} = \frac{c_0}{2} I + \sum_{k=1}^{k=\infty} c_k T_k(Z) \tag{4.1}$$

$$Z = \frac{2}{\beta - \alpha}[A - \frac{\beta + \alpha}{2} I] \tag{4.2}$$

```
1: **Initialization:**
2: $\beta$=max(eig($A$));
3: $\alpha = \frac{\beta}{ratio}$;
4: $\boldsymbol{Z = \frac{2}{\beta-\alpha}A - \frac{\beta+\alpha}{\beta+\alpha}I}$;
5: $q = (1 - \sqrt{\frac{\alpha}{\beta}})/(1 + \sqrt{\frac{\alpha}{\beta}})$;
6: $c_0 = \frac{(-q)^0}{\sqrt{\alpha\beta}}; c_1 = \frac{(-q)^1}{\sqrt{\alpha\beta}}$
7: $T_{p0} = I; T_{p1} = Z$;
8: $\boldsymbol{G = \frac{1}{2}c_0 I + c_1 T_{p1};}$;
9: **for** $i = 2$ to $r$ **do**
10:     $\boldsymbol{T = 2ZT_{p1} - T_{p0}}$;
11:     $c = \frac{(-q)^i}{\sqrt{\alpha\beta}}$;
12:     $\boldsymbol{G = G + cT}$;
13:     $T_{p0} = T_{p1}; T_{p1} = T$;
14: **end for**
```

Figure 4.1: Algorithm of Chebyshev preconditioner

$$\begin{cases} T_0 = I \\ T_1 = Z \\ T_k = 2ZT_{k-1}(Z) - T_{k-2}(Z) \end{cases} \tag{4.3}$$

$$c_k = \frac{1}{\sqrt{\alpha\beta}}(-q)^k \tag{4.4}$$

$$q = \frac{1 - \sqrt{\frac{\alpha}{\beta}}}{1 + \sqrt{\frac{\alpha}{\beta}}} \tag{4.5}$$

4.2 Implementation

4.2.1 Chebyshev preconditioner algorithm

Chebyshev preconditioner algorithm is presented in Figure 4.1. 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 discussed how to choose *ratio* and r in detail Dag and Semlyen (2003). Matrix G is the approximation of $A's$ inverse. The output of Chebyshev preconditioner algorithm is matrix G. Bold lines (line 4, 8, 10, 12) 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 with smaller condition numbers so that the system can converge faster in the iterative solver step.

4.2.2 Conjugate gradient algorithm

Figure 4.2 shows the algorithm of the conjugate gradient method. x_0 is the initial value of the solution of $Ax = b$. If there is no preknowledge of x_0, it can be set to all 0. If there is, a cultivated x_0 it 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, bold lines (line 4, 8, 10, 11, 13, 14, 16) are implemented either by CUBLAS or CUSPARSE. It can be seen that a majority of the computations can be ported to GPU for computation.

4.2.3 Hardware platform

The experiments are carried out on a server equipped with NVIDIA GPU Tesla M2070. M2070 is a Fermi architecture product. It has 14 stream multiprocessors and

```
Initialization:
$r = b - Ax_0; x = x_0;$
$p = r; k = 1;$
$\boldsymbol{r_1 = r' * r}$;
while $r_1 >$ tolerance and iteration $<$ max iter do
    if $k > 1$ then
        $\beta = r_1/r_0;$
        $\boldsymbol{p = r + \beta * p}$;
    end if
    $\boldsymbol{Ap = A * p}$;
    $\boldsymbol{temp = p' * Ap}$;
    $\alpha = \frac{r_1}{temp};$
    $\boldsymbol{x = x + \alpha * p}$;
    $\boldsymbol{r = r - \alpha * p}$;
    $r_0 = r_1;$
    $\boldsymbol{r_1 = r' * r}$;
end while
```

Figure 4.2: Algorithm of conjugate gradient method

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.

4.2.4 Software implementation

The test cases are power system examples from Matrix Market Boisvert et al. (1997), MATPOWER Zimmerman et al. (2011), and a sample case from UCTE Zhou and Bialek (2005). Test matrices from Matrix Market are 494-bus, 662-bus, 685-bus, 1138-bus. Test matrices from MATPOWER are case2383wp, case2736sp, case2737sop. The maximum allowed error is 1×10^{-3}. The upper limit for iteration is 1000 iterations. The sample case from UCTE in summer 2002 has 1253 buses.

The bold lines in Figure 4.1 and Figure 4.2 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 vectors, CUBLAS functions will be called for speedup purpose. We implement Chebyshev preconditioner based on Figure 4.1 and integrated 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 Dag and Semlyen (2003).

4.3 Experiment

This section will present experiment results begining 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.

4.3.1 Degree of Chebyshev preconditioner

Chebyshev preconditioner can effectively reduce condition number. Condition number is generally considered as a good indicator of matrix attribute. The smaller the condition number, the less iterations the matrix needs to converge. A larger degree of Chebyshev preconditioning can further reduce the condition number. 30-bus

Table 4.1: Condition number comparison of practical systems

Degree	IEEE30	IEEE118	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

and 118-bus from IEEE standard bus system, 494-bus, 662-bus and 1138-bus from MatrixMarket, are selected as examples of small scale computation, medium scale computation and large scale computation, respectively. Table 4.1 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. The condition number is clearly dropped when the larger degree is set to Chebyshev preconditioner as Table 4.1 indicates.

Figure 4.3 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. Figure 4.3 shows that deeper degrees can always lead to a significant iteration number reduction in iterative solving.

Figure 4.4 compares the performance of Chebyshev preconditioner C/CUDA implementation with GPU and with Matlab on CPU. The IEEE 30-bus and the IEEE 118-bus systems are the smallest test cases. The GPU implementation is less efficient

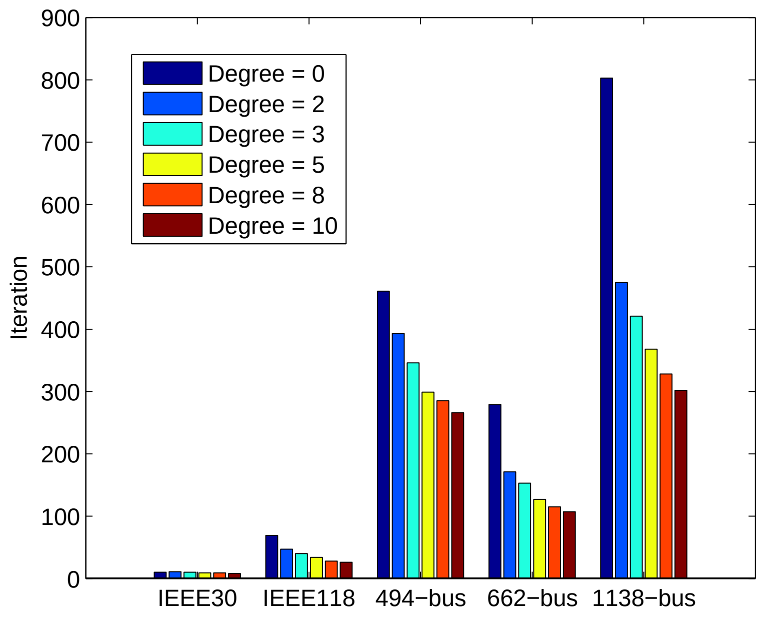


Figure 4.3: Iteration comparison of Chebyshev preconditioner with various degrees.

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 4.1 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. Figure 4.5 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

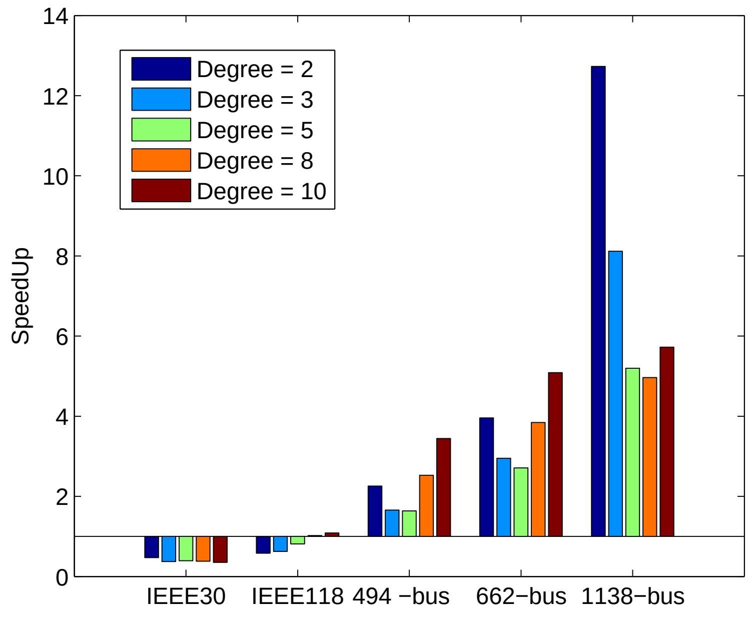


Figure 4.4: Speedup comparison of Chebyshev preconditioner on GPU over Matlab.

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

4.3.2 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 4.2 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 fifth column. The runtime of Chebyshev

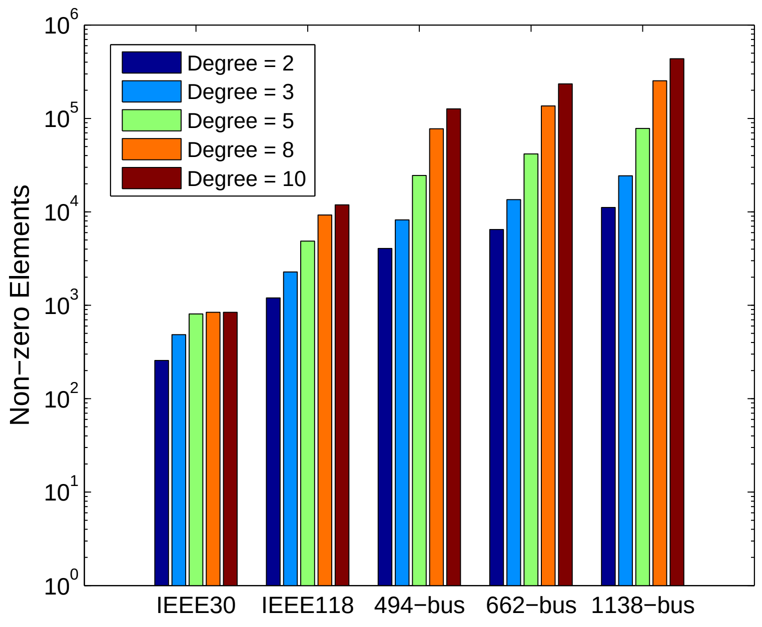


Figure 4.5: Non-zero elements increase with deeper degree.

preconditioner on Matlab increases significantly when 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 46x speedup and almost 200 ms absolute runtime saving for the largest system case2736sp. Table 4.2 shows consistent results as Figure 4.4. Chebyshev preconditioner on GPU can hardly improve computation performance for smaller systems, but it is able to gain significant runtime saving when the test systems are larger. The reason is that there is enough data to better utilize the computation capability of the GPU and offset computation overhead like data copy.

Table 4.3 shows the runtime of Matlab implementation and GPU implementation of the conjugate gradient method. Corresponding speedup in the fifth column shows GPU's advantages in a large system. The runtime of conjugate gradient

Table 4.2: Chebyshev preconditioner performance (T_{CP}) comparison between CPU and GPU implementation of practical systems

System	Size	CPU(ms)	GPU(ms)	SpeedUp	Time Saved(ms)
IEEE30	29 by 29	0.737	1.537	0.48	-0.800
IEEE57	56 by 56	0.530	1.538	0.34	-1.008
IEEE118	117 by 117	0.914	1.579	0.58	-0.665
IEEE300	299 by 299	3.472	3.254	1.07	0.218
494-bus	494 by 494	8.273	3.338	2.48	4.935
662-bus	662 by 662	15.328	3.407	4.50	11.921
685-bus	685 by 685	16.652	3.350	4.97	13.302
1138-bus	1138 by 1138	42.885	3.419	12.54	39.466
UCTE	1253 by 1253	52.565	3.389	15.51	49.176
case2383wp	2382 by 2382	152.813	4.117	37.12	148.696
case2736sp	2735 by 2735	199.502	4.263	46.80	195.238

Table 4.3: Conjugate gradient performance (T_{CH}) comparison between CPU and GPU implementation of practical systems

System	Size	CPU(ms)	GPU(ms)	SpeedUp	Time Saved (ms)
IEEE30	29 by 29	1.259	1.781	0.71	-0.522
IEEE57	56 by 56	2.100	2.966	0.71	-0.866
IEEE118	117 by 117	6.058	7.844	0.77	-1.786
IEEE300	299 by 299	12.522	13.075	0.96	-0.553
494-bus	494 by 494	52.705	47.757	1.10	4.948
662-bus	662 by 662	36.449	28.106	1.30	8.343
685-bus	685 by 685	35.601	24.925	1.43	10.676
1138-bus	1138 by 1138	128.215	75.054	1.71	53.161
UCTE	1253 by 1253	8.554	4.600	1.86	3.954
case2383wp	2382 by 2382	202.052	71.539	2.82	130.513
case2736sp	2735 by 2735	124.189	25.727	4.83	98.462

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 4.3, it is shown that the runtime of 662-bus is shorter than 494-bus. The maximum GPU implementation speedup of conjugate gradient is 4.83x for case2736sp in our experiments. The absolute runtime saving is around 100 ms for one solving in the same case.

Table 4.4 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, 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 10.79x. Last column shows the absolute time saving. The absolute runtime saving for the two largest systems are almost 300 ms for case2736sp for one solution.

4.3.3 Improvement on Chebyshev preconditioner and conjugate gradient method

The runtime breakdown of Chebyshev preconditioner and conjugate gradient is shown in Figure 4.6. 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 occupies more than 50% of the total execution time. Therefore, in order to further improve the overall performance, the conjugate gradient computation needs to be enhanced, too.

Computations involved in conjugate gradient contain a lot of matrix vector multiplications. Blocked Compressed Sparse Row (BSR) provides data reuse for

Table 4.4: Conjugate gradient with Chebyshev preconditioner performance ($T_{CP} + T_{CH}$) comparison between Matlab and GPU implementation

System	Size	CPU(ms)	GPU(ms)	SpeedUp	Time Saved(ms)
IEEE30	29 by 29	1.996	3.318	0.60	-1.322
IEEE57	56 by 56	2.630	4.504	0.58	-1.874
IEEE118	117 by 117	6.972	9.423	0.74	-2.451
IEEE300	299 by 299	15.994	16.329	0.98	-0.335
494-bus	494 by 494	60.978	51.095	1.19	9.883
662-bus	662 by 662	51.777	31.513	1.64	20.264
685-bus	685 by 685	52.253	28.275	1.85	23.978
1138-bus	1138 by 1138	171.100	78.473	2.18	92.627
UCTE	1253 by 1253	61.119	7.989	7.65	53.13
case2383wp	2382 by 2382	354.685	75.656	4.69	279.029
case2736sp	2735 by 2735	323.690	29.990	10.79	293.700

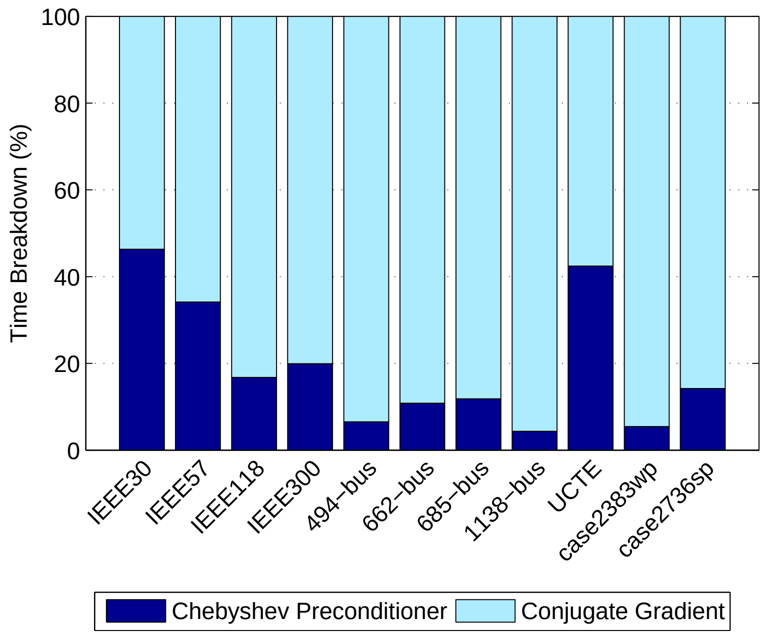


Figure 4.6: Runtime breakdown of conjugate gradient method with Chebyshev Preconditioner.

Table 4.5: Conjugate Gradient Performance Comparison Between CPU, CSR and BSR Based GPU Implementation of Vairous Systems

System	Size	CPU (ms)	G_{CSR} (ms)	G_{BSR} (ms)	CG SpeedUp		Total SpeedUp	
					G_{BSR} vs CPU	G_{BSR} vs G_{CSR}	G_{BSR} vs CPU	G_{BSR} vs G_{CSR}
IEEE30	29	1.259	1.781	1.567	0.71	0.80	0.60	0.64
IEEE57	56	2.100	2.966	2.536	0.71	0.83	0.58	0.64
IEEE118	117	6.058	7.844	6.681	0.77	0.91	0.74	0.84
IEEE300	299	12.522	13.075	11.429	0.96	1.10	0.98	1.09
494-bus	494	52.705	47.757	41.116	1.10	1.28	1.19	1.37
662-bus	662	36.449	28.106	26.026	1.30	1.40	1.64	1.76
685-bus	685	35.601	24.925	21.331	1.43	1.67	1.85	2.12
1138-bus	1138	128.215	75.054	68.201	1.71	1.88	2.18	2.39
UCTE	1253	8.554	4.600	4.411	1.86	1.94	7.65	7.82
case2383wp	2382	202.052	71.539	73.438	2.82	2.75	4.69	4.57
case2736sp	2735	124.189	25.727	26.774	4.83	4.64	10.79	10.45

matrix vector multiplication. Our improved conjugate gradient implementation uses BSR based GPU matrix-vector multiplication. The result is shown in Table 4.5. 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.

4.4 Convergence and eigenvalue discussions

This section will firstly demonstrate the CG solver with Chebyshev preconditioner and then discuss the changes that a preconditioner brings to the system. The following figures show the relative residual from each iteration when CG iterative solver is applied to the IEEE-118 system (figure 4.7), 685-bus system (figure 4.8) and UCTE system (1254 buses) (figure 4.9). The convergence criterion of the CG solver in this and the following sections is that the relative residual is smaller than

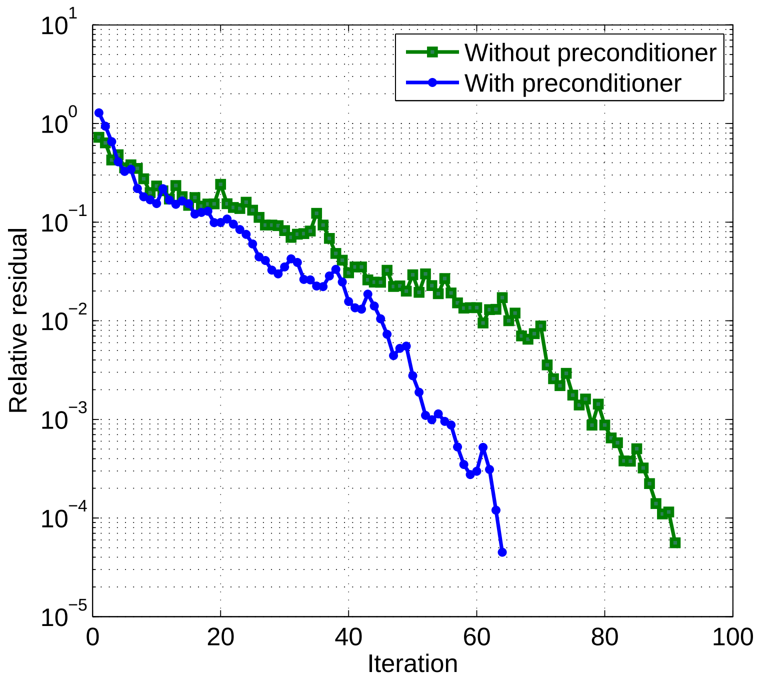


Figure 4.7: Relative residual of IEEE-118 system

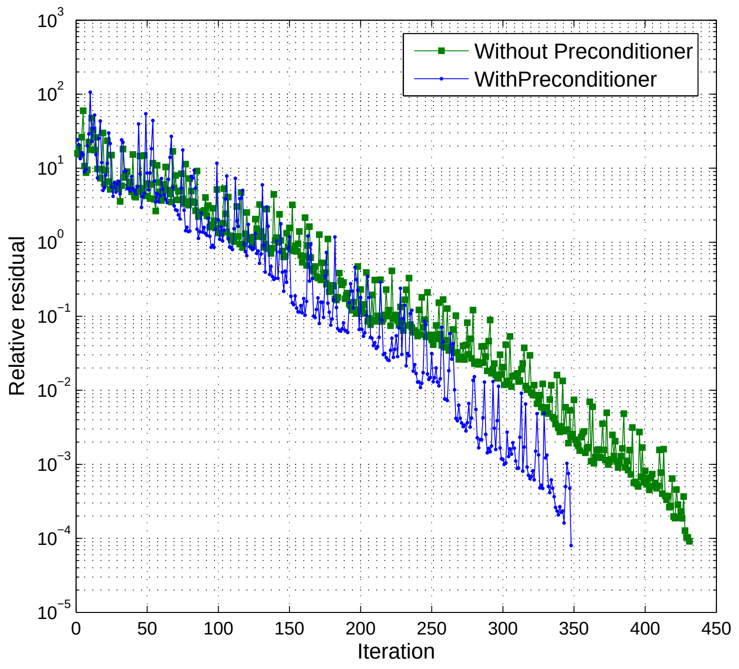


Figure 4.8: Relative residual of 685-bus system

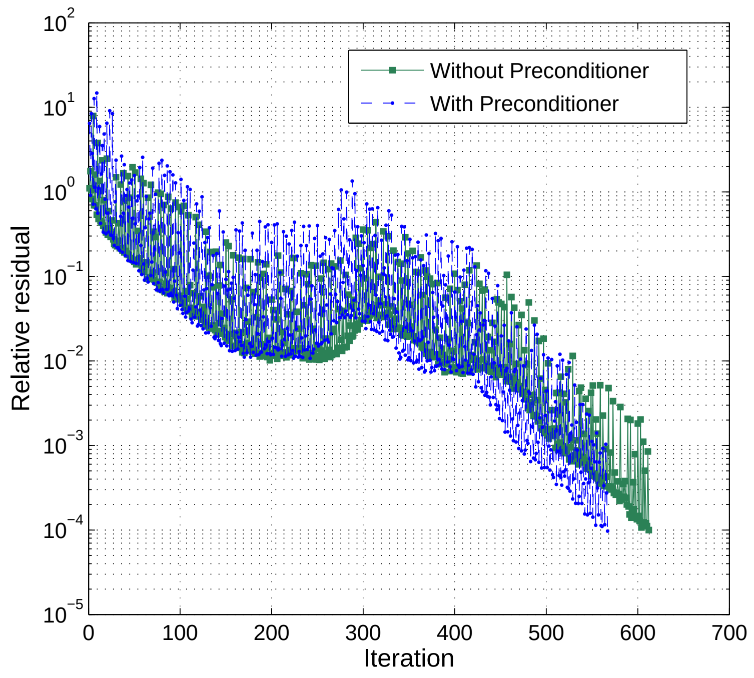


Figure 4.9: Relative residual of 1254-bus UCTE system

10^{-1}. The original system with CG method without preconditioner always take more iterations to converge in 91 iterations for IEEE-118, 431 iterations for 685-bus, and 613 iterations for the 1254-bus UCTE, while the preconditioned system with Chebyshev preconditioner degree as 2 needs less iterations to converge, 64 iterations for IEEE-118, 348 iterations for 685-bus and 568 iterations for the 1254-bus UCTE, respectively. These figures show that the relative residual drops more sharply when the Chebyshev preconditioner is applied, and hence lead to a fast convergence if compared with the case without preconditioner. It should be noted that such trend exists for all the test systems examined, but only three of them are selected to represent the small, medium and large systems due to the space limitation.

The eigenvalue spectrum has a significant influence on the convergence rate of iterative solution such as CG Jennings (1977). Figure 4.10 shows the eigenvalue spectrum shifting of the IEEE-118 bus system as an example. From top to bottom, each subfigure shows the eigenvalue distribution of the normalized linearized DC

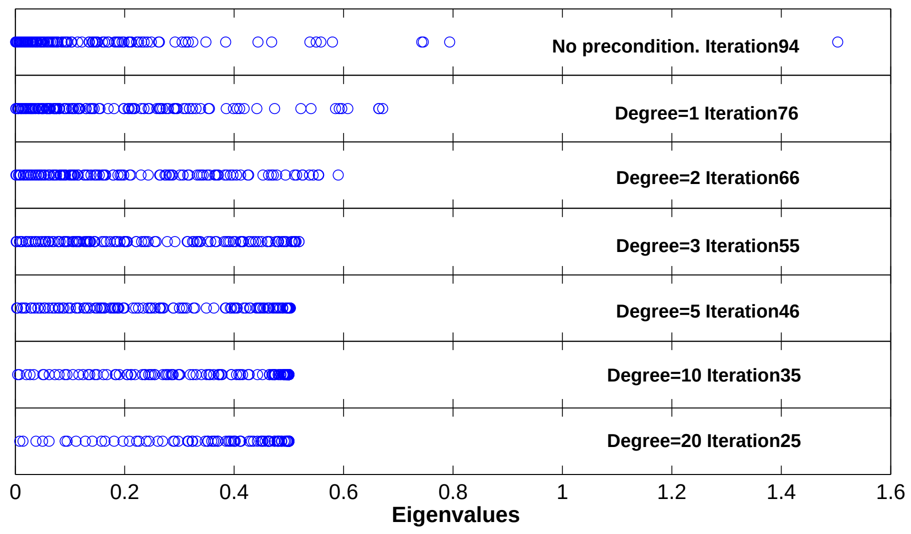


Figure 4.10: Preconditioning shifts the eigenvalue spectrum and hence reduces required iterations

power flow matrix with deeper degree of Chebyshev preconditioner. The figure shows clearly that from no preconditioner to preconditioner with degree 20, the iteration required for the iterative solver to converge is significantly reduced. When the degree is 20, the right part of the spectrum is packed very close, and the largest eigenvalue is kept being shifted towards a smaller value. The left parts have already shown clustering effects, and the smallest eigenvalue is also obviously shifted away from 0. Such shifting and clustering effects make it take only 25 iterations for the CG solver to converge when the degree is 20. This observation validates equation 3.35 that the closer the eigenvalues are packed, the less iteration the iterative solver needs to converge. This is a significant performance enhancement if compared with the top subfigure which requires 94 iterations without preconditioner for the same system. The number of iterations in Figure 4.10 demonstrates that a narrow eigenvalue spectrum will lead to a faster convergence. Our simulation experiments with other systems show the same trend as well.

4.5 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 Pai and Dag (1997) Dag and Semlyen (2003) Idema et al. (2012). Preconditioner plays an important role in 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 parallel. The Chebyshev preconditioner, a polynomial preconditioner, is a parallel method. The conjugate gradient method is one of the iterative solvers. It has been introduced to power system applications for its potential parallelism to speedup the power flow Garcia (2010).

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 $A^T Ax = A^T b$.

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.

4.6 Conclusion

Power system applications such as power system optimization, control and analysis require intensive computational ability Green et al. (2011). 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.

Our work targets at solving the fundamental computation of power system and sparse linear systems. Table 4.5 shows that the maximum overall speedup can reach 10.79% with the case2736sp system; Table 4.2 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. Figure 4.6 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.

Chapter 5

Estimation of the Largest Eigenvalue for Chebyshev Preconditioner

The discussion in Chapter 4 reveals that the maximum and minimum eigenvalues of a linear system are required in order to shift the spectrum of the original matrix to the [-1, 1] interval to construct the Chebyshev preconditioner. Solving characteristic equation to obtain eigenvalues is a time-consuming process Larson (2012), which limits the possibility to apply Chebyshev preconditioner practically. Iterative solvers such as CG require the system to be symmetric positive definite. Therefore, this work will focus on the B matrix in linearized DC power flow, as well as the B' and B'' matrices in fast decoupled power flow (FDPF), since they are all symmetric positive definite linear systems Van Amerongen (1989) Stott and Alsac (1974).

This Chapter will first discuss an estimation method for the largest eigenvalue (β) for equation 4.2 using the B matrix in DC power flow as an example. Then, the conclusion will be extended to matrix B' and B'' in fast decoupled power flow.

5.1 Range of eigenvalues

In this section, the matrix B from linearized DC power flow will be analyzed for estimating the maximum eigenvalue β. Here, the pre-estimation step will normalize the matrix first. The maximum value b_{max} of the matrix B will be selected, and then the entire matrix will be divided by b_{max} element-wise, such that the value range of the matrix will be shifted to $(-1, 1]$. Note, the maximum value of the linear system can only appear in diagonal elements for the power flow calculations discussed in this work. Finding the maximum element of a system requires only one traverse of the diagonal elements, so the complexity of this step is $O(n)$. It is not related to the off-diagonal, non-zero elements of the system, which is typically 3 to 4 times more than the system size in power system applications due to the sparsity of power systems. Thus, even with the increase of the system scale, the overhead of finding the maximum element will be increased only linearly.

It should be noted that the B matrix discussed in solving linearized power flow has its reference buses eliminated. We will name $\tilde{B}$ as the matrix before the elimination of the reference bus. Assume that the set including all the system buses is Φ, and Ψ is the set for PQ and PV buses. We assume that there is only one reference bus throughout the paper. If there are n non-reference buses (PQ and PV buses) in the system, the total bus number of the system is $(n + 1)$. Therefore, the size of set Ψ is n and the size of set Ψ is $(n + 1)$. The indices i and j are the numberings for the B matrix after the reference bus is eliminated, while the indices k and l are the numberings for the $\tilde{B}$ matrix, we have

$$\tilde{B}_{kl} \in (-1, 0]; l \neq k, l, k \in \Phi \tag{5.1}$$

$$\tilde{B}_{kk} = -\sum_{l=1, l\neq k}^{(n+1)} \tilde{B}_{kl} = \sum_{l=1, l\neq k}^{(n+1)} |\tilde{B_{kl}}| \tag{5.2}$$

Besides, we have

$$0 < \tilde{B}_{kk} \leq 1; k \in \Phi \tag{5.3}$$

If we assume the off-diagonal element $\tilde{B}_{kl}$ in the $\tilde{B}$ matrix corresponds to element B_{ij} which is the corresponding element after the reference bus elimination, we have equations 5.4 and 5.5. Note that $\tilde{B}_{kl}$ to B_{ij} represents a one-to-one mapping as equation 5.6 shows. q is the index of the reference bus.

$$B_{ij} = \tilde{B}_{kl} \in (-1, 0]; j \neq i; k \neq l; i, j \in \Psi, k, l \in \Phi \tag{5.4}$$

$$B_{ii} = \tilde{B}_{kk}; i \in \Psi, k \in \Phi \tag{5.5}$$

$$k = \begin{cases} i; & i < q \\ i-1; & i > q \end{cases} \quad l = \begin{cases} j; & j < q \\ j-1; & j > q \end{cases} \tag{5.6}$$

To simplify the derivation, we may define

$$\gamma_i = \sum_{j=1, j\neq i}^{n} |B_{ij}| \tag{5.7}$$

We will use the row and column information in the $\tilde{B}$ matrix to perform further calculations for every element in the B matrix. If we change the numbering to $\tilde{B}$ matrix-based, equation 5.7 could be given by

$$\gamma_i = \sum_{j=1, j\neq i}^{n} |B_{ij}| = \sum_{l=1, l\neq k, l\neq q}^{(n+1)} |\tilde{B}_{kl}| \tag{5.8}$$

Based on Gershgorin circle theorem Stewart (2001), all the eigenvalues of the B matrix lies in the union of the discs centered at B_{ii} with radius γ_i.

$$|\lambda - B_{ii}| \leq \gamma_i \tag{5.9}$$

Substituting the $\tilde{B}$ matrix numberings to B matrix, we have

$$|\lambda - \tilde{B}_{ii}| \leq \gamma_i \tag{5.10}$$

This can be rewritten as:

$$-\gamma_i \leq \lambda - \tilde{B}_{kk} \leq \gamma_i \tag{5.11}$$

$$-\gamma_i + \tilde{B}_{kk} \leq \lambda \leq \gamma_i +_{kk} \tag{5.12}$$

Substitute equation 5.2 and 5.8 to equation 5.12, we have

$$-\sum_{l=1,l\neq k,l\neq q}^{(n+1)} |\tilde{B_{kl}}| + (\sum_{l=1,l\neq k}^{(n+1)} |\tilde{B_{kl}}|) \leq \lambda \leq \sum_{l=1,l\neq k,l\neq q}^{(n+1)} |\tilde{B_{kl}}| + (\sum_{l=1,l\neq k}^{(n+1)} |\tilde{B_{kl}}|) \tag{5.13}$$

The left hand side can be written as

$$-\sum_{l=1,l\neq k,l\neq q}^{(n+1)} |\tilde{B_{kl}}| + (\sum_{l=1,l\neq k}^{(n+1)} |\tilde{B_{kl}}|) = |\tilde{B}_{kq}| \geq 0 \tag{5.14}$$

The right hand side can be written as equation 5.15 considering equation 5.3.

$$\sum_{l=1,l\neq k,l\neq q}^{(n+1)} |\tilde{B_{kl}}| + (\sum_{l=1,l\neq k}^{(n+1)} |\tilde{B_{kl}}|) \leq 2(\sum_{l=1,l\neq k}^{(n+1)} |\tilde{B}_{kl}|) = 2\tilde{B}_{kk} \in (0,2] \tag{5.15}$$

Therefore, equation 5.13 will be

$$0 \leq \lambda \leq 2 \tag{5.16}$$

Since the no-solution case of power flow is not within the scope of this work, the B matrix is not singular. That is, in this work, we discuss only the systems which are power flow solvable. Therefore, there will be no zero eigenvalue. Thus, equation 5.16 can be further simplified to equation 5.17, i.e., the range of eigenvalues of the B matrix is $(0, 2]$.

$$0 < \lambda \leq 2 \tag{5.17}$$

5.2 Estimation of the largest and the smallest eigenvalues

With the range $(0, 2]$ of eigenvalues, the maximum eigenvalue will be estimated in this section. Since each matrix has been normalized to the largest elements, the value range of each element is $(-1, 1]$. For power flow matrices, the diagonal elements are special since their absolute values are much larger than off-diagonal elements. Although we can hardly build a direct correlation between the larger elements and the largest eigenvalue, they can still serve as guidance on estimating the largest eigenvalue. The value of 1 exists for every matrix after normalization; hence, the largest element must always be 1. The second largest element will be employed to fit a linear function to estimate the largest eigenvalue β.

In order to study the relationship between the second largest element in the system (namely, η) and the largest eigenvalue λ_m, we started with six test systems: the IEEE 30-bus system, the IEEE 57-bus system, and the IEEE 118-bus system from standard IEEE test systems; case2383wp and case2736sp from MATPOWER Zimmerman et al. (2011); and a UCTE system with 1253 buses Zhou and Bialek (2005). Then, we modified each of the original systems to create a large amount of new test systems. The modification is performed by randomly scaling up/down, but within the range of [75%, 125%] of the original value, for one of the three values (r, x or b) of the branch Π model for a specific branch, and this process is repeated for 20% branches for each of the six test systems. Also, each system will go through the experiment for 1000 times to create 1000 new systems to better study the underlying rules. Therefore, there are 6000 data sets shown in Figure 5.1. The second largest element η in each matrix is shown in the x axis, and the largest eigenvalue λ_m is shown in the y axis

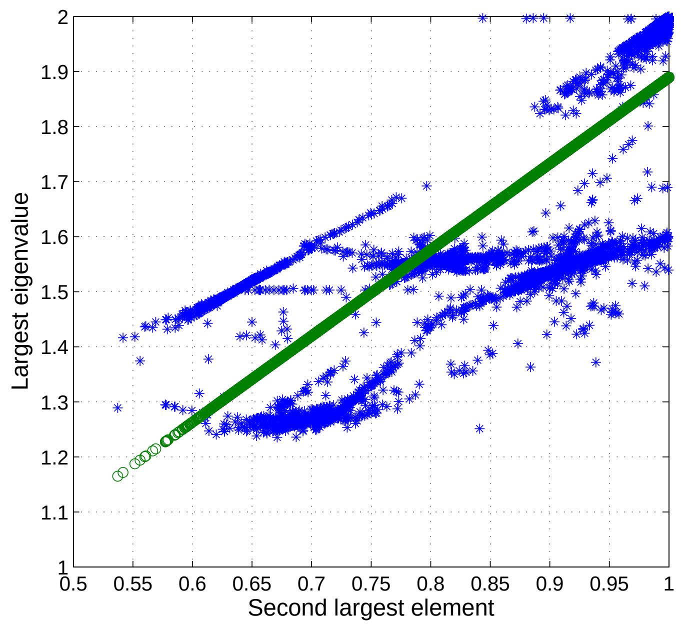


Figure 5.1: Second largest element (η) and the largest eigenvalue (λ_m) of each randomly generated systems in blue stars. Green circles show the fitted results

in Figure 5.1. The green circles form a bold green line which represents the fitted function based on the minimum least square estimation.

Since the estimated eigenvalue β is used to construct precondition matrix to reduce the iterations needed in the iterative solving process only, it does not have to be as precise as the actual largest eigenvalue λ_m, as long as β can be estimated in a short time and does not lead to a significant iteration increase. With the fitted function given in equation 5.18, the most computational intensive part is to find the second largest element, η, in the matrix. Since the formulation of the linearized DC power flow and FDPF ensures that the diagonal elements are the largest elements in each row or column, traversing the diagonal elements only will be enough to find the second largest element of the system. Therefore, β can be reached with the computation complexity $\mathcal{O}(n)$, which is very scalable. Equation 5.18 below shows the fitted function for β.

$$\beta = 1.5660 \times \eta + 0.3234 \tag{5.18}$$

After obtaining the estimated β, the smallest eigenvalue α can be estimated. Here, the artificial conditioner number is used to estimate the smallest eigenvalue (α) based on the largest eigenvalue. Since the condition number can be defined as the ratio between the largest eigenvalue and the smallest eigenvalue, this ratio that is used to estimate the smallest eigenvalue α based on the largest eigenvalue β is called the artificial condition number. The artificial condition number ω is suggested in Dag and Semlyen (2003) to be 5 for power system applications. With the estimated value of β and equation 5.19, the estimated value of α can be achieved.

$$\alpha = \beta/\omega \tag{5.19}$$

5.3 Extension to fast decoupled power flow cases

Chapter 5.1 and 5.2 discuss the proposed method to estimate the largest eigenvalue of B matrix in linearized DC power flow. In this subsection the FDPF will be discussed. Here, the matrices B' and B'' in FDPF are slightly different from the B matrix in DC power flow because of the branch susceptance. It should be noted that although there are various version of FDPF with minor difference, our discussion is based on the MATPOWER?s XB version of FDPF Zimmerman and Murillo-Sanchez (2014).

Consider the matrix B' first. We still assume bus q is the reference bus. In FDPF, for the $\tilde{B}'$ matrix before reference bus elimination and with the renumbering defined as in equation 5.20, since $r_{ij} \ll x_{ij}$ is commonly assumed for FDPF, we have

$$\tilde{B}'_{kl} = -\frac{1}{x_{kl}}; \tilde{B}'_{kk} = \sum_{l=1, l\neq k, l\neq q}^{(n+1)} \frac{1}{x_{kl}} \tag{5.20}$$

The construction of B' is similar to that of the B matrix in DC power flow. Therefore, we have

$$\tilde{B}'_{kl} \in (-1, 0]; i \neq j \tag{5.21}$$

$$\tilde{B}'_{kk} = -\sum_{l=1,l\neq k}^{(n+1)} \tilde{B}'_{kl} \quad and \quad 0 < \tilde{B}'_{kk} \leq 1 \tag{5.22}$$

Following the similar deductions in equations 5.4 to 5.17, we can reach the following conclusion:

$$0 < \lambda_{B'} \leq 2 \tag{5.23}$$

Therefore, the estimation process for the B matrix in Chapter 5.2 still holds for the B' matrix in FDPF.

Next, regarding the B'' matrix, we assume that there are n_{pv} PV buses. The $\tilde{B}''$ matrix before reference bus elimination is constructed as

$$\tilde{B}''_{kl} = -b_{kl}; \quad l \neq k \tag{5.24}$$

$$\tilde{B}''_{kk} = -b_k - \sum_{l=1,l\neq k}^{(n_{pv}+1)} \tilde{B}''_{kl} + \sum_{l=1,l\neq k}^{n_{pv}+1} b_{kl} \tag{5.25}$$

where b_{kl} is the susceptance between bus i and bus j and $b_{kl} > 0$; and b_i is the shunt susceptance at bus i, i.e., the sum of susceptance of all the shunt branches connected to bus i Crow (2002). It should be noted that b_i is usually smaller than $\sum_{j\neq i} b_{ij}$. Since $\tilde{B}''_{kk}$ is the diagonal element, the following equation still holds:

$$0 < \tilde{B}''_{kk} \leq 1 \tag{5.26}$$

For the B'' matrix after reference bus elimination, similar to equation 5.4, if we assume that Ψ is the bus set after elimination, i.e., all the PV buses, Φ is the set including all the buses, we have

$$B''_{ij} = \tilde{B}''_{kl} \in (-1, 0]; \quad j \neq i; k \neq l; i, j \in \Psi, k, l \in \Phi \tag{5.27}$$

$$B''_{ii} = \tilde{B}''_{kk}; \quad i \in \Psi, k \in \Phi \tag{5.28}$$

$$\gamma_i = \sum_{j \neq i}^{n_{pv}} |B''_{ij}| = \sum_{l=1, l \neq k, l \neq q}^{n_{pv}+1} |\tilde{B}''_{kl}| \tag{5.29}$$

If we assume $\varepsilon = min(b_i)$ and $\xi = max(b_i)$, then

$$\varepsilon \leq b_i \leq \xi \tag{5.30}$$

With Gershgorin circle theorem, we have equation 5.31, and with equation 5.28 we have equation 5.32.

$$|\lambda_{B''} - B''_{ii}| \leq \gamma_i \tag{5.31}$$

$$-\gamma_r + \tilde{B}''_{kk} \leq \lambda_{B''} \leq \gamma_i + \tilde{B}''_{kk} \tag{5.32}$$

Substitute equations 5.29 and 5.25 into equation 5.32, we have

$$-\sum_{l=1, l \neq k, l \neq q}^{n_{pv}+1} |\tilde{B}''_{kl}| + (-b_i + \sum_{l=1, l \neq k}^{n_{pv}+1} |\tilde{B}''_{kl}|) \leq \lambda_{B''} \leq \sum_{l=1, l \neq k, l \neq q}^{n_{pv}+1} |\tilde{B}''_{kl}| + (-b_i + \sum_{l=1, l \neq k}^{n_{pv}+1} |\tilde{B}''_{kl}|) \tag{5.33}$$

The left side of equation 5.33 can be written as

$$-\sum_{l=1, l \neq k, l \neq q}^{n_{pv}+1} |\tilde{B}''_{kl}| + (-b_i + \sum_{l=1, l \neq k}^{n_{pv}+1} |\tilde{B}''_{kl}|) = |\tilde{B}''_{kq}| - b_i \tag{5.34}$$

Since b_i is usually a small value if compared with $\tilde{B}''_{kq}$, equation 5.34 can be simplified to

$$|\tilde{B}''_{kq}| - b_i > 0 \tag{5.35}$$

The right side of equation 5.33 can be written as

$$\begin{aligned} \sum_{l=1, l \neq k, l \neq q}^{n_{pv}+1} |\tilde{B}''_{kl}| + (-b_i + \sum_{l=1, l \neq k}^{n_{pv}+1} |\tilde{B}''_{kl}|) &\leq 2 \sum_{l=1, l \neq k}^{n_{pv}+1} |\tilde{B}''_{kl}| - b_i \\ &= 2(-b_i + \sum_{l=1, l \neq k}^{n_{pv}+1} |\tilde{B}''_{kl}|) + b_i = 2\tilde{B}''_{kk} + b_i \end{aligned} \tag{5.36}$$

Based on equation 5.26 and 5.30, equation 5.36 can be further deducted to

$$2\tilde{B}''_{kk} + b_i \leq 2 + \xi \tag{5.37}$$

Therefore equation 5.32 can be written as

$$0 < \lambda_{B''} \leq 2 + \xi \tag{5.38}$$

Back to our assumption that ξ is a small value, we can consider that $\lambda_{B''}$ is still in the range of (0, 2], approximately. Once again, the largest eigenvalue is used to precondition the matrix for iterative solution, so it does not have to be precise. In summary, this section concludes that all the eigenvalues in the matrices of linearized DC power flow or fast decoupled power flow follow equation 5.39 after normalization.

$$0 < \lambda \leq 2 \tag{5.39}$$

5.4 Computational experiments

In this section, the performance of Chebyshev preconditioner and iterative solver will be compared using accurately calculated maximum eigenvalue and the estimated maximum eigenvalue to precondition the matrix. The example cases include IEEE 30-bus system, IEEE 57-bus system, and IEEE 118-bus system from standard IEEE test systems; 494-bus, 662-bus, 685-bus and 1138-bus from MatrixMarket Boisvert et al. (1997); case2383wp and case2736sp from MATPOWER Zimmerman et al. (2011); and one UCTE system with 1253 buses Zhou and Bialek (2005). All the experiments are based on their linearized power flow matrices. All experiments are based on Matlab implementation, but please note that once the estimation equation 5.18 is acquired, it can be integrated with any implementation (such as parallel implementation in GPU etc.) in real applications. The convergence or stop criterion for iterative solver is that the relative residual is smaller than 10^{-4} of the nodal power mismatch in per unit. The degree of Chebyshev preconditioner is selected as 2 based on recommendation from Chapter 4. Note that there are four systems in this computational experiment, 494-bus, 662-bus, 685-bus and 1138-bus from MatrixMarket, are not used in the previous section to obtain the empirical equation 5.18 because the data from MatrixMarket has only the matrices instead of the full system data (i.e., branches and buses). However, these four systems are applied here to test the generality and applicability of equation 5.18 and the overall approach.

5.4.1 Demonstration of estimated eigenvalues

The precise maximum eigenvalue should yield close-to-minimum iterations in the CG iterative solution step. The column $\beta_{est-max}$ in Table 5.1 is the estimated largest eigenvalue β based on equation 5.18. The fourth column shows the calculated largest eigenvalue λ_m. The iterations the CG solver needs when using the estimated eigenvalue and the calculated eigenvalue for Chebyshev preconditioner are listed as well. Table 5.1 shows that although the estimated eigenvalue cannot always precisely reach the largest eigenvalue, it will not cause a significant iteration increase. Instead of obtaining the precise largest eigenvalue, our goal here is to quickly estimate a reasonable large eigenvalue to 1) ensure that the iterative solution converges quickly, and 2) save the runtime spent on the eigenvalue calculation. Note that there are cases that using $\beta_{est-max}$ takes less iteration that using λ_m to get the iterative solver converged. The reason is that it has been observed that the iterative solver may take less iterations to converge when a value slightly smaller than the actual largest eigenvalue is applied Jennings (1977). However, there is no quantitative formulation or proof available regarding this observation.

5.4.2 Performance improvement

As previously mentioned, the construction of Chebyshev preconditioner requires the maximum eigenvalue of the original matrix. However, it is usually time consuming to

Table 5.1: Iterations Comparison with Estimated and Calculated Largest Eigenvalue for Various Systems

System	$\beta_{est-max}$	Iterations	λ_m	Iterations
IEEE30	1.426	25	1.270	23
IEEE57	1.587	33	1.556	33
IEEE118	1.320	63	1.503	66
494-bus	1.123	690	1.500	698
662-bus	1.877	288	1.988	293
685-bus	0.938	352	1.000	350
1138-bus	1.876	1045	1.494	1008
UCTE	1.889	571	1.998	567
case2383wp	1.883	937	1.831	937
case2736sp	1.889	616	1.976	641

calculate eigenvalues. Using the maximum eigenvalue estimation technique provided in this paper can save runtime spent on eigenvalue calculation with no significant iteration increase. This section will compare the execution time including eigenvalue calculation or estimation, and CG iterative solving process.

The total execution time of Chebyshev preconditioner can be summed as equation 5.40. T_{β_est} is the time consumed for estimating the largest eigenvalue β. T_{β_cal} stands for the runtime if the calculated maximum eigenvalue λ_m is used for β. T_{CG} is the runtime of the CG solver. T_{CP} is the execution time spent on Chebyshev preconditioning.

$$\begin{aligned} T_{cal} &= T_{\beta\_cal} + T_{CP} + T_{CG} \\ T_{est} &= T_{\beta\_est} + T_{CP} + T_{CG} \end{aligned} \tag{5.40}$$

Figure 5.2 shows the runtime breakdown of these three parts. The left bar (i.e., the T_{est} bar) in each subpanel shows T_{β_est}, while the right bar (i.e., the T_{cal} bar)

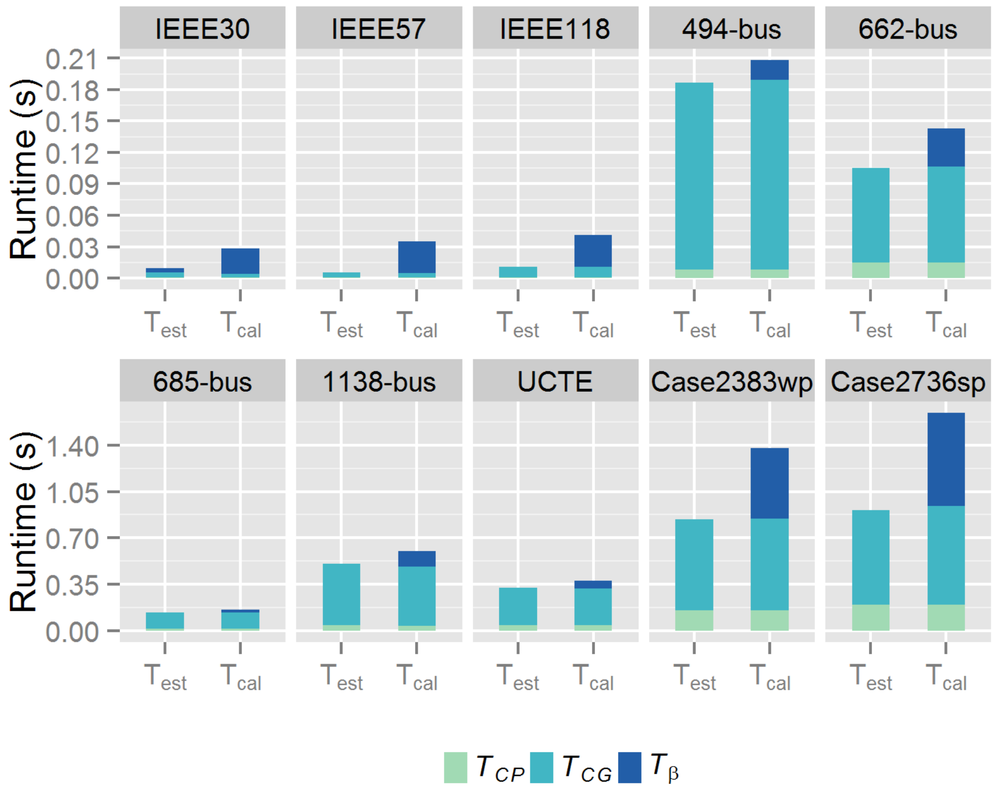


Figure 5.2: Runtime breakdown of various systems

shows T_{β_cal}. T_{β_est} is negligible if compared to the total runtime for all the cases, but T_{β_cal} increases significantly when the system is much larger. The middle section in each stacked bar shows T_{CG}. Previously, it was discussed that beta-estimation may cause the number of iterations increase in Chapter 5.4.1. However, the minor increase of iteration numbers will not lead to a significant runtime delay. If we take the 1138-bus case as an example, the iteration number increases from 1008 to 1045 if the estimated eigenvalue is used instead of the calculated eigenvalue. Hence, the time spent on the CG solver is marginally increased, while the significant time saved on estimating eigenvalues still makes T_{β_est} less than T_{β_cal} as Figure 5.2 shows. The bottom bar in each subpanel shows T_{CP}. Note, there is no significant difference in T_{CG} and T_{CP} in terms of execution time no matter estimated or precise eigenvalue is applied for Chebyshev preconditioner.

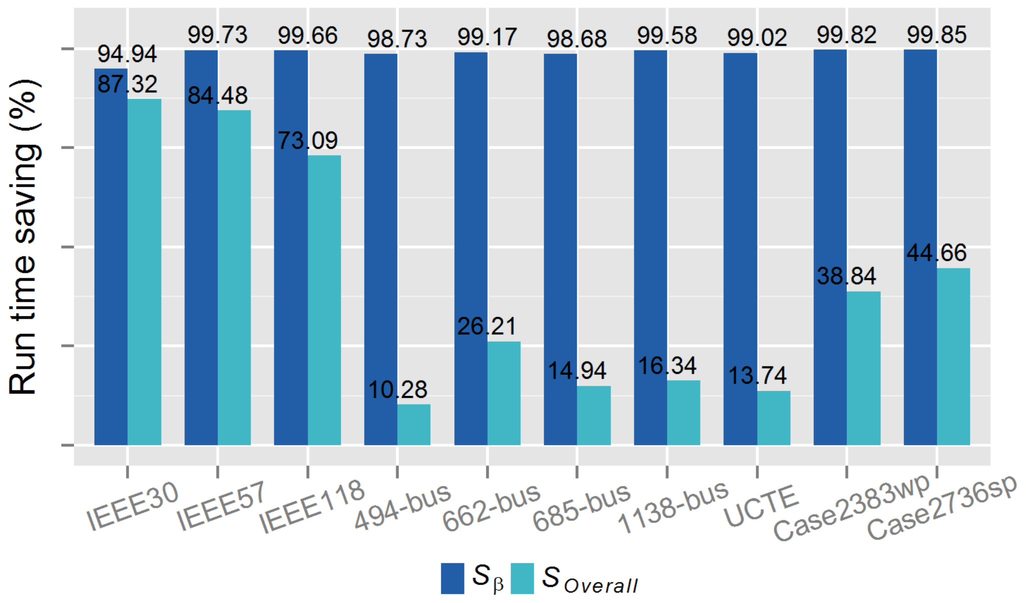


Figure 5.3: Runtime saving for eigenvalue estimation and the whole execution.

Figure 5.3 shows the saved execution time in percentage for obtaining β by estimation as opposed to calculation. The left bar in each subpanel gives S_β, the run time saving percentage of T_β by estimating β over calculating β. S_β is is calculated as follows.

$$S_\beta = \frac{T_{\beta\_cal}}{(T_{\beta\_cal} - T_{\beta\_est})} \times 100\% \tag{5.41}$$

The right bar in each subpanel shows $S_{Overall}$, the overall saving percentage of the total run time which includes the time for obtaining eigenvalue (T_{β_est} or T_{β_cal}), Chebyshev preconditioning (T_{CP}), and the final CG solution (T_{CG}). $S_{Overall}$ is calculated by equation 5.42.

$$S_{Overall} = \frac{T_{cal}}{(T_{cal} - T_{est})} \times 100\% \tag{5.42}$$

Further, in terms of the overall performance enhancement, the right bars in Figure 5.3 shows that $S_{Overall}$ (the overall running time reduction) is in the range of 10.28% to 87.32%. The average $S_{Overall}$ is 40.99%, and the $S_{Overall}$ for the two largest systems each having 2000-plus buses are 38.84% and 44.66%. Those numbers show that the proposed approach is very promising for large scale systems.

5.4.3 Discussions

The eigenvalue range $(0, 2]$ proven in Chapter 5.1 is a sufficiently good estimation of the eigenvalue range for power system applications as linearized DC power flow (DCPF) and fast decoupled power flow (FDPF) after normalized to the largest element. With the guidance from the second largest element η in the system, a straightforward and effective approach for estimating the largest eigenvalue is proposed. It may not achieve the largest eigenvalue precisely, and may increase the number of iteration that CG solver needs to converge. However, it well serves the goal to save the time spent on eigenvalue estimation, meanwhile without a significant increase of iterations in the CG iterative solving step. The total execution time including estimating the largest eigenvalue, preconditioning the matrix and iteratively solving the linear system is significantly reduced as the computational experiment shows.

5.5 Conclusion

Iterative solvers as alternatives to LU-based direct solvers have gained increasing interests due to their easy implementation in parallel computation since iterative solvers such as the Conjugate Gradient (CG) method have weak data dependency and less memory requirements. A preconditioner is always a must for a successful iterative solver. Chebyshev preconditioner is a polynomial method popularly discussed in the literature. However, Chebyshev preconditioner requires the maximum eigenvalue in order to effectively obtain the approximated inverse of the matrix. The calculation of eigenvalues is usually time-consuming. This work first proves that the maximum eigenvalue of many power system applications like linearized DC power flow and fast decoupled power flow ranges in (0, 2] after normalization, and then proposes a maximum eigenvalue estimation method based on those specific features in power system applications such that a fast and effective estimation of the maximum eigenvalue can be achieved. The average saving of the execution time for eigenvalue calculation is 98.92%, and the average saving of overall execution time is 40.99% based on computational experiment on ten sample power systems with the largest system of nearly 3000 buses. This makes the CG method and Chebyshev preconditioner more promising for further performance improvement in power system computation.

Chapter 6

Using Two-Step Preconditioning for Conjugate Gradient Method in Linear Computations in Power Flow

With aforementioned development of iterative solvers and its preconditioners, as well as the widely deployment of GPU implementation in power system computations, this chapter will discuss using GPU-based conjugate gradient method with three preconditioners, a Jacobi-like preconditioner, a polynomial Chebyshev preconditioner, and a two-step preconditioner with Jacobi-like preconditioner first and then Chebyshev preconditioner to accelerate the linear system solving in power flow computation, especially for large scale systems.

6.1 Two-step preconditioning

An iterative solver does not require the reordering as direct solvers to reduce the potential fill-in elements; however, it does require the preconditioning step to obtain a faster convergence in iterative solving. The preconditioner is a matrix, too. It linearly transforms the original matrix to another one which has a narrow eigenvalue spectrum. The narrower the eigenvalue spectrum is, the less iteration it needs to converge in iterative solving step. An ideal preconditioner is the matrix's inverse, and then the preconditioned matrix would be unitary matrix which has all the eigenvalue as 1. However, the precise inverse is too expensive to obtain for most cases. Therefore, a good preconditioner should consider a trade-off between the easiness to obtain and close approximation of the matrix inverse.

This work will integrate two preconditioners to precondition the original matrix. The first of them is a Jacobi-like preconditioner, and the second one is the Chebyshev polynomial preconditioner. Jacobi preconditioner will take the diagonal elements of the original matrix first, and then use the inverse of each element to form the preconditioner as equation 6.1 shown below. This preconditioner usually works for matrices which the values of diagonal elements are dominant. The linear systems in power flow computations are within such category. The diagonal elements are larger or equal to the sum of the absolute value of off-diagonal elements. Therefore, Jacobi preconditioner is selected as the first preconditioner.

$$J_{ii} = 1/A_{ii} \tag{6.1}$$

In order to keep the symmetry of the preconditioned matrix, the final preconditioner will be as equation 6.2 shows. The preconditioned matrix is as equation 6.3 states. After the Jacobi-like preconditioner is applied, the diagonal elements of would be all 1. In order to simplify the description, we will still call this Jacobi-like preconditioner as Jacobi preconditioner in this work.

$$J_{ii} = 1/\sqrt{A_{ii}} \tag{6.2}$$

$$A_p = JAJ \tag{6.3}$$

Chebyshev preconditioner is as section 4.1.2 introduces. In this chapter, we keep the G notation for Chebyshev preconditioner. The final preconditioned matrix is shown in equation 6.4. One of the advantages of Chebyshev polynomial is that it will retain the symmetry of the original matrix. Therefore, only a left-precondition will be performed. The two-step preconditioner in equation 6.4 is named as *JaCh*.

$$A_{pp} = GA_p = GJAJ \tag{6.4}$$

As a result, the solution of linear equation $Ax = b$ would be alternatively solved through equation 6.5 and then use equation 6.6 to reach the solution x.

$$GJAJy = GJb \tag{6.5}$$

$$x = Jy \tag{6.6}$$

6.2 Implementations

The conjugate gradient solver, the Jacobi-like preconditioner and the Chebyshev preconditioner are all scalable methods, and can be implemented with the parallel computation platform, graphic processing unit (GPU). As previously mentioned, the computational kernels in conjugate gradient method, Jacobi-like preconditioner or Chebyshev preconditioner involve mainly sparse matrix/vector operations. The implementation in this work will port the matrix and vector related operations to GPU. The vector related operations use the functions from CUBLAS NVIDIA (2012a), and sparse matrices operations use CUSPARSE NVIDIA (2012b).

The test systems are 57-bus and 118-bus systems from standard IEEE test systems; 494-bus, 685-bus, and 1138-bus from MatrixMarket Boisvert et al. (1997); and case2383wp, case2736sp, case2737wop from MATPOWER Zimmerman et al.

(2011). We use the matrices from DC power flow to demonstrate our proposed two-step preconditioner and iterative solver, but the same method can be applied to the linear matrices in fast decoupled power flow. Please note that the whole set of implementation can be considered as guidelines to other applications that are also positive symmetric definite, and have dominant diagonal elements.

The experiments are carried on a server equipped with NVIDIA Tesla M2070 GPU, which has 14 stream multiprocessors and each multiprocessor has 32 CUDA cores. The server has 8-core Xeon E5607 2.27GHz CPU and 24 GB memory. The CUDA driver version is 5.0 and GCC version 4.7.3. Operating system is Ubuntu 12.04. In order to keep the same precision as Matlab, all C/CUDA implementation are based on double-precision floating point operations. The stop criterion for iterative solver is relative residual smaller than $1e^{-4}$. The degree for Chebyshev preconditioner is selected as 2 with the consideration of preconditioning effects and the extra non-zero fill-ins that a deeper degree of Chebyshev preconditioner could bring as Chapter 4 indicates.

6.3 Computational results

In this section, the computational results will be presented when the proposed implementation is applied to practical test systems. The condition number reduction will be shown first, followed by the iteration reduction. The performance comparison between Matlab implementation on CPU and C/CUDA implementation on GPU will be provided at last.

6.3.1 Condition number reduction

Condition number is considered as a good indicator of how fast the iterative solver can converge. Therefore, for a specific system, if the condition number is reduced, the number of iteration can be expected to decrease. Table 6.1 shows the condition

Table 6.1: Comparison of Condition Number

System	**Original**	**W/Jacobi**	**W/Chebyshev**	**W/*JaCh***
IEEE-57	1.63E+03	5.37E+02	3.32E+02	1.28E+02
IEEE-118	4.85E+03	1.88E+03	9.41E+02	3.70E+02
494-bus	3.89E+06	4.04E+05	6.49E+05	8.47E+04
685-bus	5.31E+05	6.11E+04	8.85E+04	1.49E+04
1138-bus	1.23E+07	2.46E+06	2.07E+06	4.75E+05
Case2383wp	7.51E+05	2.58E+05	1.51E+05	4.88E+04
Case2736sp	1.09E+06	2.31E+05	2.30E+05	4.52E+04
Case2737sop	1.09E+06	2.32E+05	2.30E+05	4.54E+04

number comparison among the original system, system with Jacobi preconditioner, system with Chebyshev preconditioner, and system with the two-step preconditioner. The first column of Table 6.1 shows the systems under experiments. The second column gives the condition number of the original systems. The third and fourth column show the condition number when the original system is preconditioned by the Jacobi preconditioner and by the Chebyshev preconditioner, respectively. The last column shows the condition number when the original matrix is preconditioned by the Jacobi preconditioner first and then by Chebyshev preconditioner, i.e. the two-step preconditioner *JaCh*. The results in Table 6.1 show clearly that the original systems without any preconditioning have the greatest condition number value. For most of the systems, Chebyshev preconditioner can result in smaller condition number than the simple Jacobi preconditioner. However, for 494-bus and 685-bus, the Jacobi preconditioner can perform better. The two-step preconditioner *JaCh* can always provide the smallest condition number.

Table 6.2: Comparison of Number of Iteration

System	Original	W/Jacobi	W/Chebyshev	W/***JaCh***
IEEE-57	54	40	26	17
IEEE-118	106	68	55	29
494-bus	1083	395	412	166
685-bus	497	205	220	84
1138-bus	2007	900	603	375
Case2383wp	1848	926	449	391
Case2736sp	2580	766	613	316
Case2737sop	2513	725	450	299

6.3.2 Iteration comparison

Table 6.2 shows the number of iteration which the conjugate gradient solver needs with different preconditioners. For most cases, Chebyshev preconditioner can make the iterative solver converge faster. But, similar to Table 6.1, 494-bus and 685-bus need less number of iterations when preconditioned by Jacobi preconditioner. However, when the two preconditioners are combined together, the number of iteration the conjugate gradient solver needs is reduced most significantly.

6.3.3 Performance improvement

Conjugate gradient solver, Jacobi preconditioner and Chebyshev preconditioner all have parallel potentials. Therefore this section will show the performance improvement if they are implemented with parallel hardware GPU. In an iterative solving process, the preconditioner will be generated first, and then the preconditioned iterative solver will achieve the solution of the linear equations. Table 6.3 and Table 6.4 show the computation time including the time of constructing preconditioners and the time for conjugate gradient iterative solver to converge on CPU and GPU platform respectively. There are three preconditioners under discussion for each

Table 6.3: Performance Result of CPU Implementation

System	**W/Jacobi(s)**	**W/Chebyshev(s)**	**W/*Jach*(s)**
IEEE-57	0.0056	0.0031	0.0033
IEEE-118	0.0098	0.0070	0.0063
494-bus	0.0689	0.0768	0.0460
685-bus	0.0605	0.0674	0.0486
1138-bus	0.2292	0.2050	0.1675
Case2383wp	0.4598	0.3526	0.3935
Case2736sp	0.7297	0.7136	0.5624
Case2737sop	0.7190	0.5898	0.5456

implementation, Jacobi, Chebyshev, and the two-step preconditioner *JaCh*. It can be seen from Table 6.3 and Table 6.4 that the time of the CPU implementation increases more significantly than the time the GPU implementation consumes with the increase of the system size. This conclusion is true for all the three preconditioners, and proves the advantage of the parallel implementation of the conjugate gradient solver and the three preconditioners chosen here.

Table 6.5 gives the performance improvement of GPU implementation over CPU implementation for each preconditioner. GPU can hardly achieve any performance improvement for smaller systems, i.e. the IEEE-57 and IEEE-118 systems. The reason is that the speedup from parallel implementation is not able to offset the time overhead of data movement between CPU and GPU. For system larger than IEEE-118, the GPU implementation can always perform better than Matlab implementation on CPU. The conjugate gradient solver with the two-step preconditioner *Jach* accelerates the computation better than the other two preconditioners. The speedup can reach up to 8.9x for the largest test system here, while the speedup can only reach around 6x if any of the two preconditioners is applied alone.

Table 6.4: Performance Result of GPU Implementation

System	W/Jacobi(s)	W/Chebyshev(s)	W/*Jach*(s)
IEEE-57	0.0064	0.0060	0.0053
IEEE-118	0.0109	0.0111	0.0075
494-bus	0.0658	0.0753	0.0347
685-bus	0.0328	0.0431	0.0203
1138-bus	0.1450	0.1099	0.0728
Case2383wp	0.1518	0.0877	0.0789
Case2736sp	0.1237	0.1181	0.0662
Case2737sop	0.1174	0.0875	0.0613

Table 6.5: Performance Improvement of GPU Implementation Over CPU Implementation

System	W/Jacobi(s)	W/Chebyshev(s)	W/*Jach*(s)
IEEE-57	0.88	0.52	0.62
IEEE-118	0.90	0.63	0.84
494-bus	1.05	1.02	1.32
685-bus	1.84	1.56	2.40
1138-bus	1.58	1.87	2.30
Case2383wp	3.03	4.02	4.99
Case2736sp	5.90	6.04	8.50
Case2737sop	6.12	6.74	8.90

6.4 Conclusion

With the development of computation technology and modern power system, the ability of simulating or analyzing large scale system will be a trend. The efficiency of linear computations is critical to the performance of power system applications. Large scale computations call for scalable algorithms and implementations. This work discusses using conjugate gradient iterative solver and three preconditioners, Jacobi, Chebyshev, and the two-step *Jach* preconditioner to present a scalable method to solve the linear computations in power flow. The results show that the two-step preconditioner *Jach* can significantly improve the convergence rate of conjugate gradient solvers for power system applications. Besides, this work implements the iterative solver and preconditioners on the GPU platform, and hence presents a performance study of the conjugate gradient method with the three different preconditioners between Matlab implementation on CPU and CUDA implementation on GPU. Results show that the overall speedup of GPU implementation over corresponding Matlab implementation can reach up to 8.9x for the largest test system with the *Jach* preconditioner. This work demonstrates great potential for using preconditioned iterative solvers and GPU implementation to accelerate linear computations in power system applications.

Chapter 7

Using GPU-based Iterative Solver and Two-Step Preconditioning with Inexact Linear Solution in Fast Decoupled Power Flow

There are many power system applications that rely on the computation of power flow. For example, contingency screening Huang et al. (2009) Yuan and Li (2015) Fang et al. (2015) will include many power flow calculations to assess the influence if one or more system components are lost. Online security assessment asks the power flow computation to be finished as fast as possible so that there will be sufficient time left for the control unit or the operator to act. Additionally, the integration of renewable energy brings uncertainty to the power grid. Multiple power flows may need to be finished to understand the activities of different expected renewable events Huang et al. (2015b) Huang et al. (2015a) Angelis-Dimakis et al. (2011). Besides, there are other power system applications that need to use Monte Carlo method which may include repeated power flow computation Torquato et al. (2014) Xu et al. (2013). Therefore, a computationally efficient power flow is of great importance, especially for large scale systems. In this situations, the whole electric power grid becomes more and more complex due to the penetration of renewable energy, the integrations of distributed energy storage Abbey and Joos (2009) Huang et al. (2011) Xiao et al. (2014a) Xiao et al. (2014b) Bai et al. (2015), electrical vehicle Kempton et al. (2001) Cui et al. (2015) and advanced control scheme Amin and Wollenberg (2005) Xu et al. (2015) etc.

As introduced in Section 3, fast decoupled power flow (FDPF) offers an alternative way to solve power flow. Chapter 6 discussed that the GPU-based two-step preconditioner $JaCh$ is effective in improving the performance of the iterative solver. This chapter will integrate such implementation with fast decoupled power flow to provide a complete FDPF solver chain. The performance comparison with the original MATPOWER version of fast decoupled power flow will be followed. An inexact linear solution method will also be introduced in this Chapter, and related performance improvement will be presented as well.

7.1 Algorithm level introduction of fast decoupled power flow

Fast decoupled load flow makes certain simplifications and can provide a simpler and more reliable solution than Newton's power flow Stott and Alsac (1974). Unlike Newton-Raphson method, which keeps updating the Jacobian matrix in every iteration, FDPF will use the same matrix B' and B''. As a result, it has a lower memory requirement.

Figure 7.1 is the flowchart for how FDPF completes the computation. Compared with Newton Raphson load flow, fast decoupled load flow decouples the system with the assumption that buses close to each other tend to have small differences in phase angle, and does not update the Jacobian matrix every iteration. Such simplifications bring many advantages especially from computational considerations.

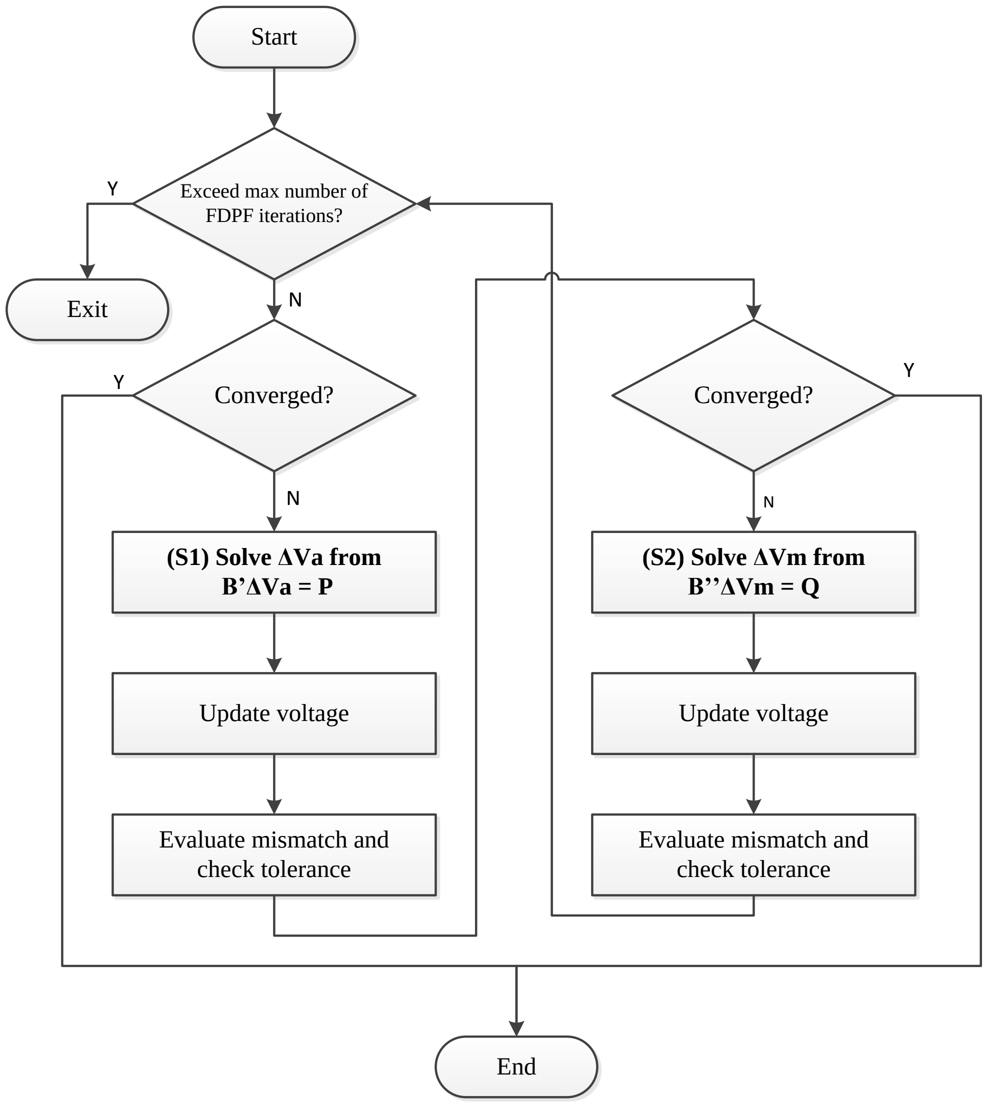

Start
Exceed max number of FDPF iterations?
Y
N
Exit
Converged?
Y
N
(S1) Solve ΔVa from B'ΔVa = P
Update voltage
Evaluate mismatch and check tolerance
Converged?
Y
N
(S2) Solve ΔVm from B''ΔVm = Q
Update voltage
Evaluate mismatch and check tolerance
End

Figure 7.1: Flowchart of fast decoupled power flow

- The solution of the original linear system is broken down to the solution of two independent linear systems. Each of the new linear system is about half the size of the original one.
- Storage space requirement is reduced because two sub-matrices (namely, J_2 and J_3) in the Jacobian matrix are approximated to zero matrices, and hence will be ignored during computation.
- The use of constant matrix B and B throughout the computation provides sufficient chances for data reuse.
- The computational time spent on updating Jacobian matrix is saved.

Besides, the algorithm shown in Figure 7.1 also employs partial updates for the convergence checking. In another word, after the computation of either ΔVa or ΔVm, the algorithm will check whether the current solution is precise enough. If so, the algorithm will stop and exit with the valid solution. If it cannot be converged within a preset maximum number of iterations, it will exit without a valid solution. This method may reduce one PQ iteration compared with checking mismatch of P and Q once every PQ iteration.

Please note that since fast decoupled power flow itself is an iterative method as well, we refer to the Newton iteration in FDPF as *PQ* iteration, and use *iteration* dedicatedly for the iterations of iterative CG solver in this chapter.

7.2 Inexact linear solution in FDPF

Fast decoupled power flow is an iterative method to reach the solution of power flow. The target of the inner linear solving is to gradually approach the solution of the non-linear equations through the outer iteration, i.e. PQ iteration for FDPF. Since the outer iteration itself is an approaching process too, the solution of the inner equations does not have to be very precise. In this case, the iterative linear solve can bring another benefit, controllable precision, while the direct method can only generate precise solutions.

Therefore, an inexact inner linear solution in fast decoupled load flow is proposed in this chapter. The iterative solvers will put the vector-to-solve $x(i)$ back into the linear equation $Ax = b$ to check the norm of $Ax(i) - b$, which is the relative residual as defined in equation 7.1. The iterative solver stops when the relative residual is smaller than a preset threshold (τ). We will use different τ as the stop criterion of the linear solving loop and compare both the number of iterations and the performance of the GPU-FDPF with inexact linear solution (ILS).

$$rr = \frac{||Ax(i) - b||}{||b||} \tag{7.1}$$

The relative residual of conjugate gradient method will generally keep a decreased trend. In other words, a smaller tolerance will lead to more iterations while a larger tolerance is expected to use less iteration; a smaller tolerance (τ) will usually guarantee that the outer loop has no delay to reach, while a larger tolerance may make the outer loop take more iterations to converge because of the lower precision of the inner loop. With such features, we can adjust the relative residual (see line 5 of Figure 4.2) to balance the iterations needed by both the outer loop and the inner loop to improve the overall performance.

7.3 Implementation

7.3.1 Fast decoupled power flow

The proposed GPU-based fast decoupled power flow is integrated with MATPOWER Zimmerman et al. (2011). Because FDPF does not have to update matrix B' and B'' every iteration as Newton Raphson method updates the Jacobian matrix, and considering the overhead of GPU memory transferring, our proposed implementation will move all matrices, including matrix B', B'' and their preconditioners to GPU

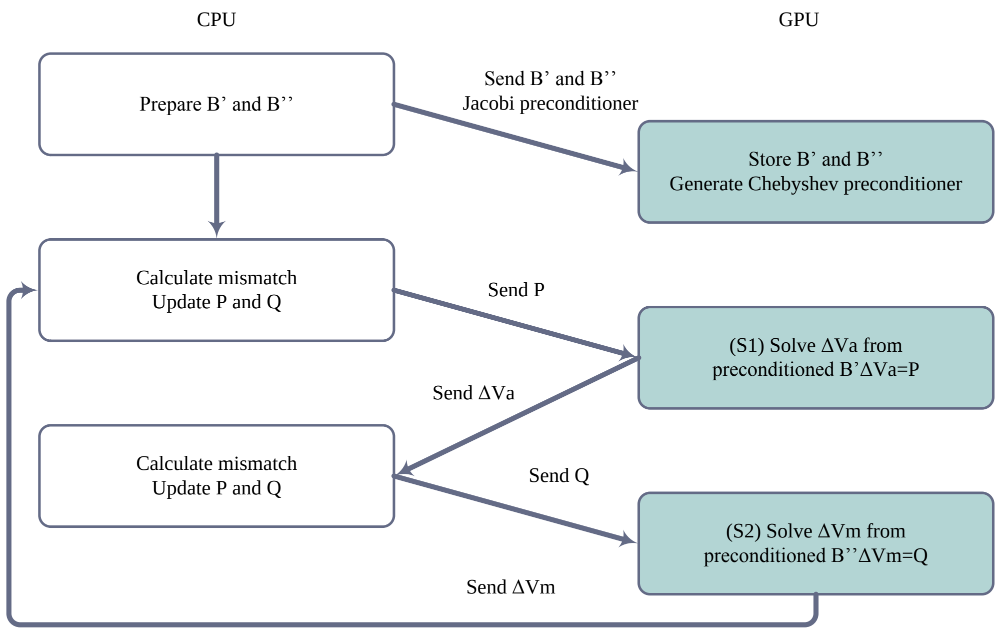


Figure 7.2: How GPU works with MATPOWER

before entering the PQ loop. All the matrices will be staying on GPU for the entire computation. The linear computations (S1) and (S2) in Figure 7.1 are carried out on GPU. The other parts involve less linear computation and, hence, will still be finished by MATPOWER on CPU. Therefore, the recurring traffic between GPU and CPU under such implementation will be limited to the updates of vector ΔVa, ΔVm and the mismatch, while the copy of matrix B', B'', and their preconditioners will be a one-time copy only.

Figure 7.2 shows how GPU-based linear solvers and preconditioners work with MATPOWER on CPU. MATPOWER will first read the system case file and prepare the matrix B' and matrix B''. After matrices B' and B'' are ready, they will be sent to GPU directly. Since the construction of Jacobi preconditioner needs the diagonal elements and hence will involve intensive index comparison work, the Jacobi preconditioner is formed on CPU and then sent to GPU. After GPU receives B' and B'', it will begin to allocate memory for B' and B'', their preconditioners, and some recurring intermediate parameters. Because Chebyshev preconditioner needs no other information but the matrix itself, GPU will formulate the Chebyshev preconditioner after it receives B' and B''. The steps above are one time preparation for one fast decoupled power flow and can be considered as the initialization. Please note that once GPU receives B', B'' and their preconditioners, all these matrices will not be transferred around between CPU and GPU. They will be staying on the GPU for the whole life time of one FDPF computation. The most time-consuming part of GPU computation is memory-related operations, and such implementation makes sure that there will be no unnecessary memory movement between CPU and GPU.

MATPOWER on CPU will calculate the mismatch and send current active power P to GPU so that GPU can solve the equation $B'\Delta V_a = P$ to get ΔV_a as step (S1) shows. The two-step preconditioner discussed in Chapter 6 will be deployed using the conjugate gradient method to solve the linear equations on GPU. After GPU gets the solution ΔV_a, it will be sent back to CPU, MATPOWER will use this updated ΔV_a to calculate current mismatch and get the new value of P and Q. Then new reactive power Q will be sent to GPU to get the solution of ΔV_m from preconditioned $B''\Delta V_m = Q$ as step (S2) shows. As previously mentioned, the preconditioned system is solved by $JaCh$ preconditioner and the conjugate gradient iterative solver. With the new Q value, updated ΔV_m will be sent back to CPU and a new PQ iteration will begin.

With that all been discussed, it is clear that after entering the loop, all the memory movement is limited to vector transferring only, i.e. vector P, Q, ΔV_a and ΔV_m. Compared with copying back and forth matrices, the vector copy is a light-weighed operation. Because of the recurring usage of the copied matrices, the copying overhead is well offset, too. The elimination of unnecessary data copies and abundant data reuse (matrix B', B'' and their preconditioners) make such implementation promising in improving the performance.

7.3.2 GPU-based iterative solver

The conjugate gradient solver is a scalable method and can be implemented with the parallel computation platform, graphic processing unit (GPU). Its algorithm can be found in Figure 4.1. The computational kernels in the conjugate gradient method involve mainly sparse matrix/vector operations. The implementation in this work will port the matrix and vector related operations to GPU. The vector related operations use the functions from CUBLAS NVIDIA (2012a), and sparse matrices operations use CUSPARSE NVIDIA (2012b).

The experiments are carried on a server equipped with NVIDIA Tesla M2070 GPU, which has 14 stream multiprocessors and each multiprocessor has 32 CUDA cores. The server has 8-core Xeon E5607 2.27GHz CPU and 24 GB memory. The CUDA driver version is 5.0 and GCC version 4.7.3. The operating system is Ubuntu 12.04. In order to keep the same precision as Matlab, all C/CUDA implementation are based on double-precision floating point operations. The stop criterion τ for iterative solver is relative residual smaller than $1e^{-3}$. The degree for the Chebyshev preconditioner is selected as 2 with the consideration of preconditioning effects and the extra non-zero fill-ins that a deeper degree of Chebyshev preconditioner could bring as Chapter 4 indicates.

7.3.3 GPU-based preconditioner

As Chapter 6 discussed, the $JaCh$ preconditioner can significantly improve the performance of conjugate gradient method. Therefore, in this Chapter, GPU-based $JaCh$ and the iterative solver will be integrated with MATPOWER to implement a GPU-based fast decoupled load flow. The construction of preconditioner $JaCh$ is as Chapter 6.1 describes.

The estimation of the largest eigenvalue to construct the preconditioner will use the method proposed in Chapter 5. The first step preconditioning uses a Jacobi-like preconditioner as equation 6.2 shows. The matrix preconditioned by a Jacobi-like

Table 7.1: Test Systems Based on Polish System

System	Origins
case2383wp	Test system in MATPOWER, Polish system
case3012wp	Test system in MATPOWER, Polish system
case10790	case3012wp $\times$ 2 and case2383wp $\times$ 2
case13173	case3012wp $\times$ 2 and case2383wp $\times$ 3

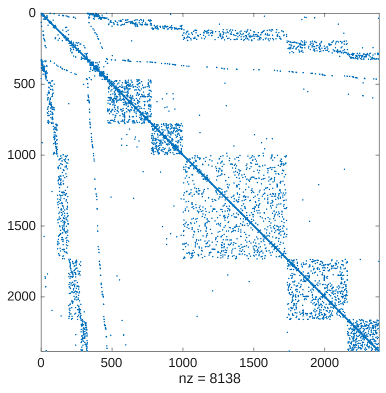
Table 7.2: Test Systems Based on Pan-European System

System	Origins
case1354pegase	Test system in MATPOWER, Pan-European system
case2869pegase	Test system in MATPOWER, Pan-European system
case5738	case2869pegase $\times$ 2
case9241pegase	Test system in MATPOWER, Pan-European System
case11624	case9241pegase and case2383wp

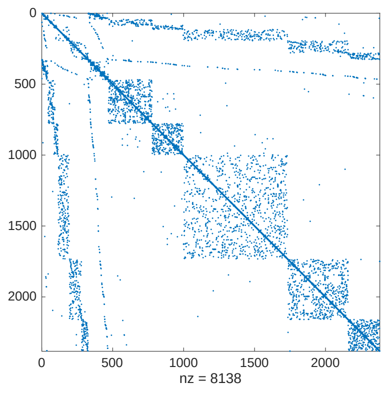
preconditioner will become a symmetric matrix which has all the diagonal elements equal to 1. Therefore, based on the discussion in Chapter 5.3, if all the elements in matrix B' and B'' are from $(-1, 1]$, then the range of all the eigenvalues will be $0 < \lambda \leq 2$. Hence we use the estimated $\lambda_{max} = 2$ to construct the second step preconditioner, the Chebyshev preconditioner. There is no need to spend extra computational time calculating the largest eigenvalue.

7.3.4 Experiments setup

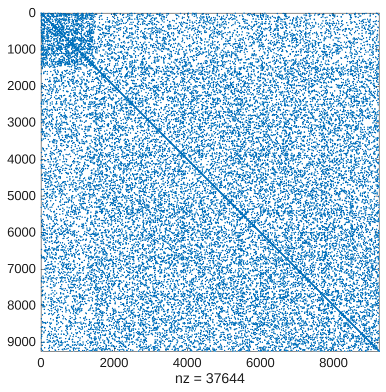
The test systems are categorized into two groups based on their different adjacent features. The first group is based on the Polish system from MATPOWER as Table 7.1 shows. We choose case2383wp and case3012wp as the base case, and use them to construct two synthetic systems, case10790 and case13173. The second group is based on the Pan-European grid as Table 7.2 shows. Case1354pegase, case2869pegase and case9241pegase are the original cases from MATPOWER. Two more synthetic



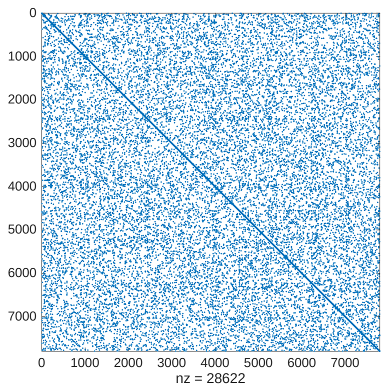
(a) case2383wp B'



(b) case2383wp B''



(c) case9241pegase B'



(d) case9241pegase B''

Figure 7.3: Sparsity pattern of case2383wp and case9241pegase.

systems case5738 and case11624 are formed to expand the test cases as well. We choose XB version of FDPF in MATPOWER.

Figure 7.3 shows the sparsity layout of B' and B'' of the typical cases from each category. Case2383 is based on the Polish system which shows several groups of locally connected subsystems, while the Pan-European system shows a more tightly coupled large system. Their different connection patterns yield different computational results, and we therefore categorize them into two groups and will discuss each separately in following sections. The sparsity pattern of all the test systems can be found in appendix A.

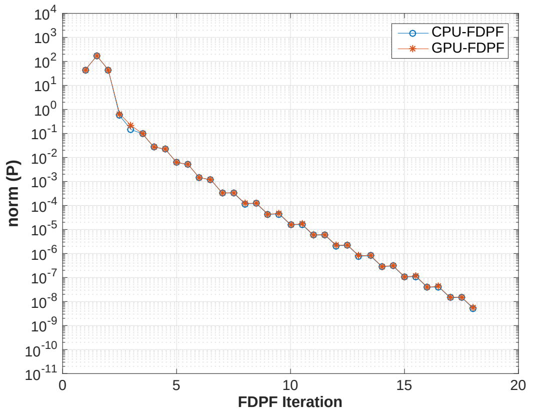
7.4 Computational results of GPU-Based FDPF

7.4.1 Precision and convergence

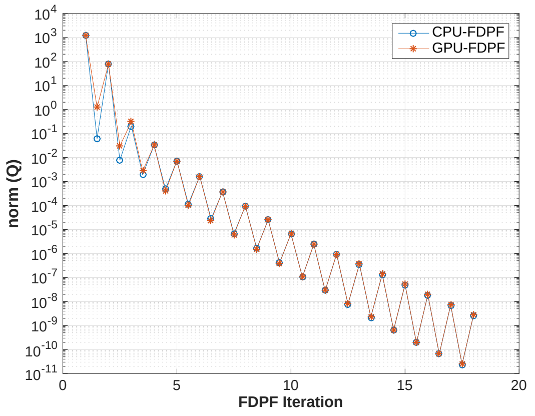
Section 6 concludes that CG iterative solver with $JaCh$ can provide the best performance among no preconditioner and the other two standalone preconditioners. Therefore CG with $JaCh$ will be used to solve the linear equations in the GPU-based fast decoupled power flow in this chapter, and then the results will be compared with the original MATPOWER.

Figure 7.4 and Figure 7.5 show the convergence process of the CPU-based and GPU-based FDPF from MATPOWER with the two groups of example cases. All of the convergence traces of GPU-FDPF match that of the CPU-FDPF very well, except the Q iteration of case9241pegase and case11624. However, even for these two cases, the deviation only begins when the norm of Q is smaller than 10^{-6}. Additionally, all of the test cases converge with the same number of PQ iterations as that of the CPU-FDPF. In other words, the inner linear iterative solver does not influence the outer PQ iteration at all. Therefore, it can be concluded that our GPU-FDPF has similar precision and convergence property as the original CPU-based FDPF.

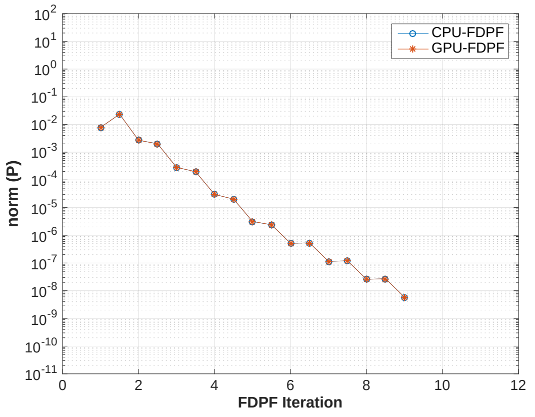
Figure 7.4: Convergence comparison between GPU-based and CPU-based FDPF for Polish systems variations



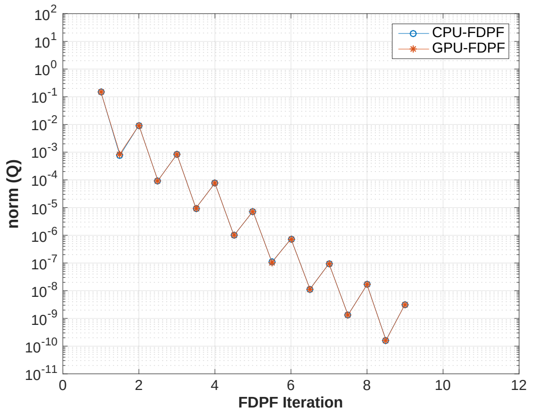
(b) case2383wp P



(c) case2383wp Q

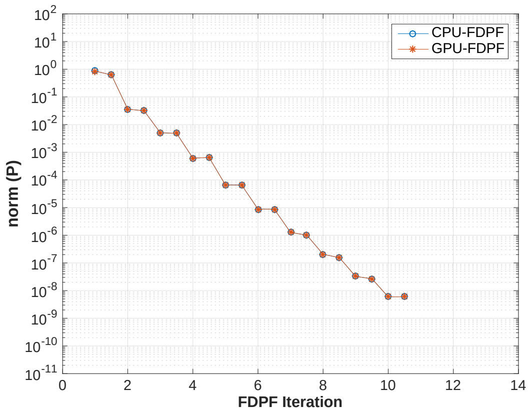


(d) case3012wp P

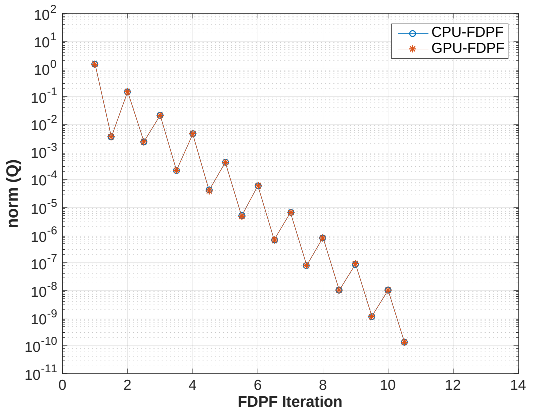


(e) case3012wp Q

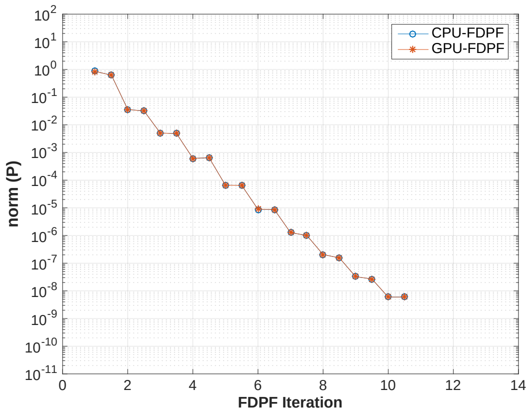
Figure 7.4: Continued



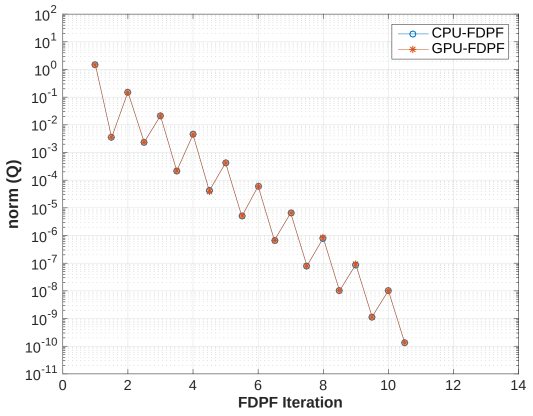
(f) case10790 P



(g) case10790 Q



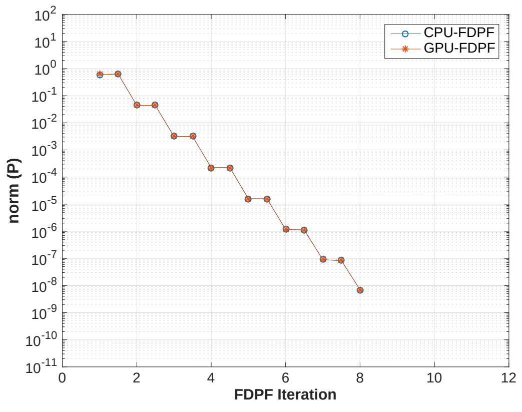
(h) case13173 P



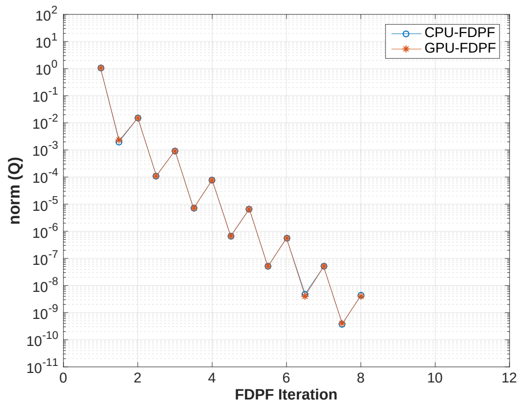
(i) case13173 Q

Figure 7.4: Continued

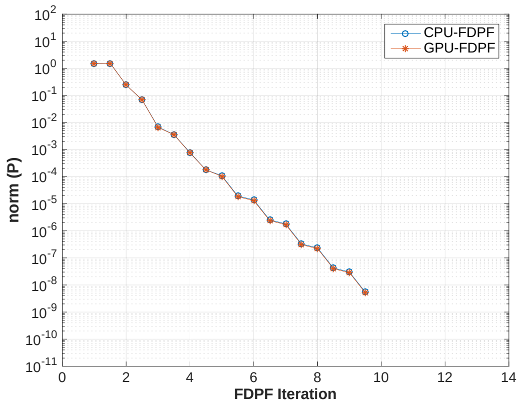
Figure 7.5: Convergence comparison between GPU-based and CPU-based FDPF for Pan-European system variations



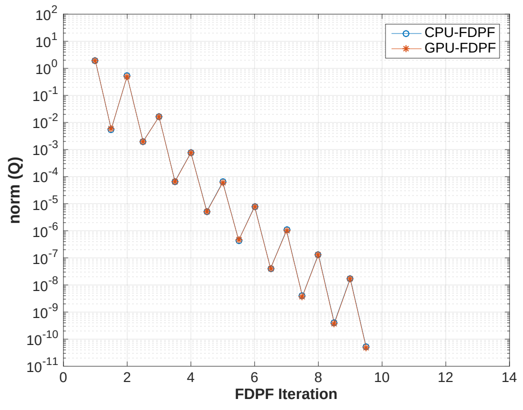
(b) case1354pegase P



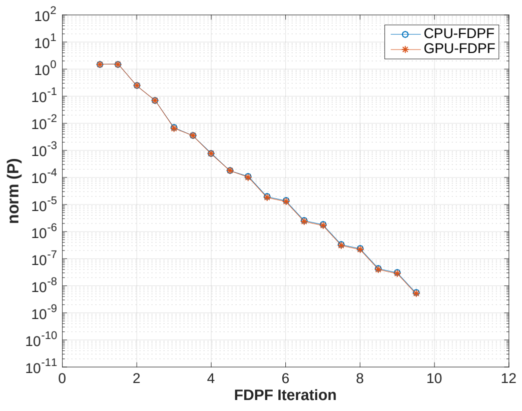
(c) case1354pegase Q



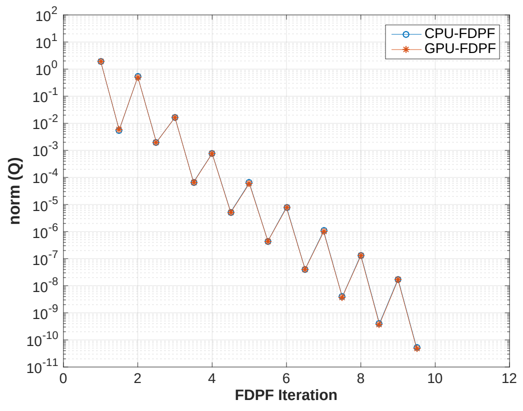
(d) case2869pegase P



(e) case2869pegase Q

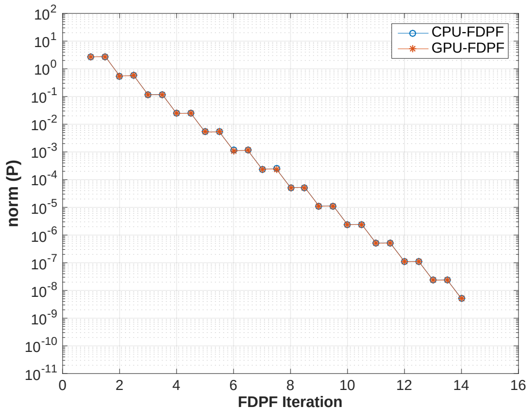


(f) case5738 P

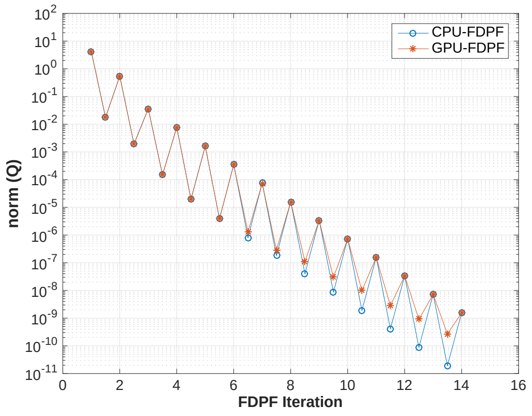


(g) case5738 Q

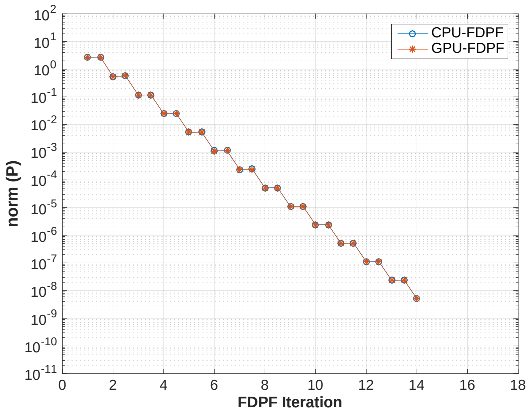
Figure 7.5: Continued



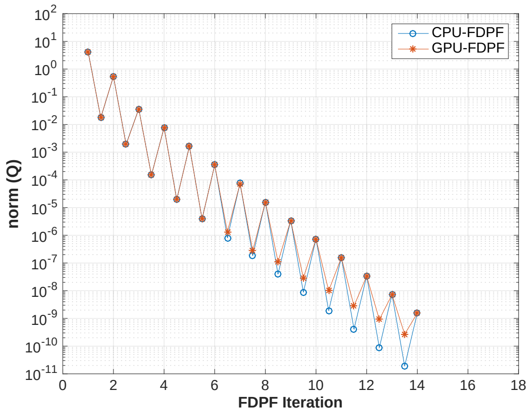
(h) case9241pegase P



(i) case9241pegase Q



(j) case11624 P



(k) case11624 Q

Figure 7.5: Continued

Table 7.3: Performance Comparison

System	CPU-FDPF (s)	GPU-FDPF (s)	Speedup
case2383wp	0.56	2.03	0.28
case3012wp	0.54	2.91	0.19
case10790	1.79	1.93	0.93
case13173	2.25	1.98	1.14
case1354pegase	0.14	0.38	0.38
case2869pegase	0.53	0.79	0.67
case5738	1.05	0.82	1.29
case9241pegase	8.79	8.47	1.04
case11624	9.37	8.37	1.12

7.4.2 Performance

Table 7.3 compares the performance of the CPU-FDPF and the GPU-FDPF. It can be seen that for the test systems that are around 10000-bus scale, GPU-FDPF can provide better performance. For Pan-European systems, because they have a denser connection (Figure 7.3 and appendix A) than the Polish system, the matrices B' and B'' are denser too. Hence, the data and computations involved are enough to better drive the GPU's parallel data processing ability. As a result, GPU-FDPF performs better than CPU-FDPF for Pan-European system starting from system larger than 5000-bus scale.

7.5 Computational results of GPU-based FDPF with inexact linear solution

7.5.1 Precision and convergence

The results in previous section show that the GPU-FDPF can provide similar computational results and better performance compared with CPU-FDPF for larger

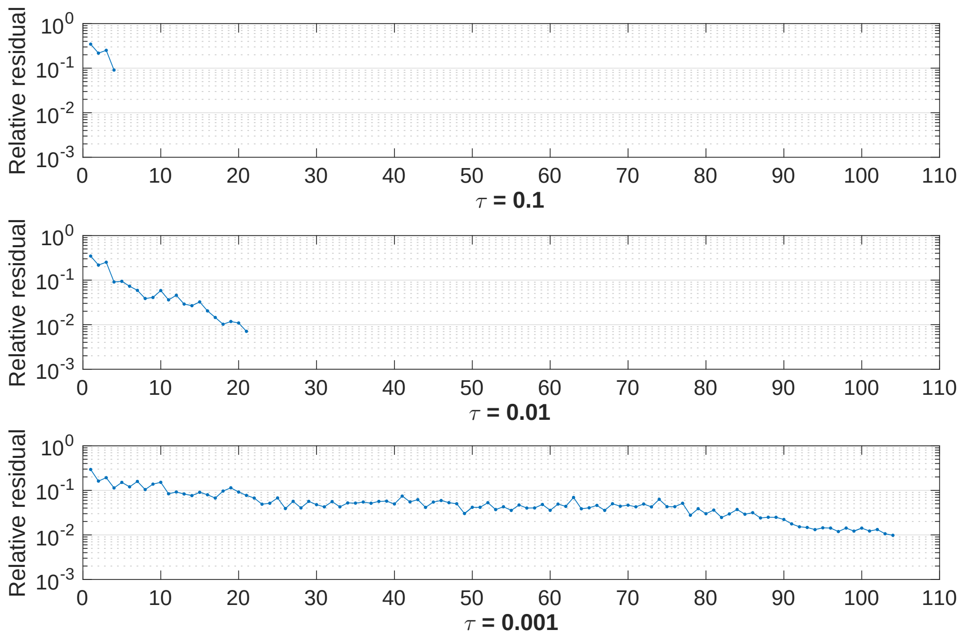


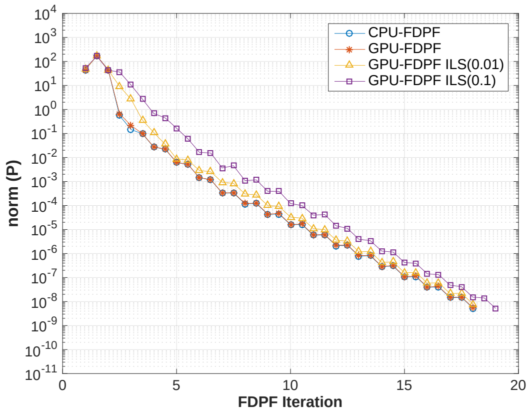
Figure 7.6: Conjugate gradient convergence with relative residual as 0.1, 0.01 and 0.001 for the solution of the first ΔV_a of FDPF for case1354pegase.

scale systems. This section will discuss applying the inexact linear solution (ILS) into fast decoupled power flow to further improve the performance. We choose the stop criterion τ to be 0.01 and 0.1 to relax the precision of the iterative linear solution inside the PQ iterations. A less precise stop criterion will always lead to a smaller number of iterations for conjugate gradient method to converge, and hence cost less runtime. Figure 7.6 shows the convergence process for the first solution of ΔV_a with test system case1354pegase. It shows clearly that when the stop criterion are set differently, the number of iterations changes significantly. For the GPU-FDPF discussed in the last section, the tolerance for relative residual is set to $1e^{-3}$, and it takes over 100 iterations to converge to $1e^{-3}$. If we set the stop criterion to 0.01, as the middle subplot shows, the iterative solver will be converged around 20 iterations. The top subplot even converges within 10 iterations when the stop criterion is set to 0.1. The x-axis of the three subplots is set to the same scale to compare the number of iteration reduction. Obviously the first subplot can provide more performance improvement. We name the stop criterion for relative residual 0.1 and 0.01 as ILS (inexact linear solution) 0.1 and ILS 0.01 in following discussions for the consideration of simplicity. We will continue to call the one with stop criterion $1e^{-3}$ GPU-FDPF for consistency.

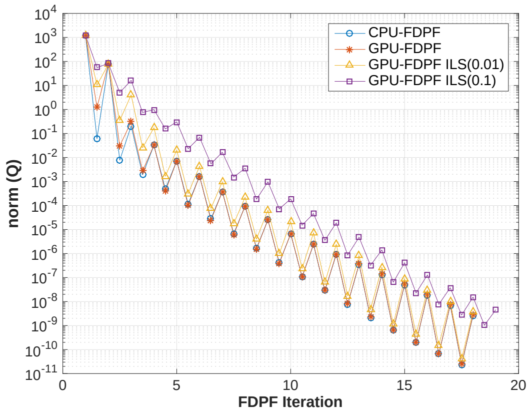
Figure 7.7 and Figure 7.8 show the convergence process in terms of every half PQ iteration for two groups of example cases with CPU-FDPF, GPU-FDPF, GPU-FDPF ILS (0.1) and GPU-FDPF ILS (0.01). GPU-FDPF can track the trace of CPU-FDPF precisely for most cases as discussed in Chapter 7.4. GPU-FDPF ILS (0.01) keeps relative similar converge process as CPU-FDPF, but there are some exceptions such as case2383wp, case1354pegase etc. It still can keep a similar number of PQ iterations as CPU-FDPF.

GPU-FDPF ILS (0.1) is the one with largest stop criterion ($\tau = 0.1$), i.e. the most imprecise one. The inner linear solving generally requires less iterations to converge as Figure 7.6 shows. The loss of precision in the inner loop will cause the increase of the outer loop as every example case in Figure 7.7 and 7.8 have shown. Besides, the convergence process of GPU-FDPF ILS(0.1) has an obvious gap from CPU-FDPF and other implementations of GPU-FDPF. However, GPU-ILS(0.1) can always manage to converge to the preset tolerance of PQ iteration, and hence solve the fast decoupled power flow.

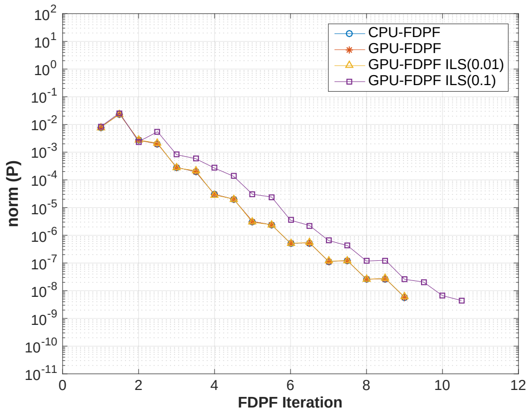
Figure 7.7: Convergence comparison between GPU-based and CPU-based FDPF for Polish systems variations



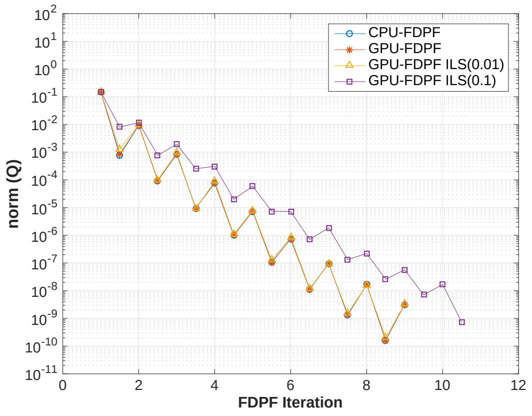
(b) case2383wp P



(c) case2383wp Q

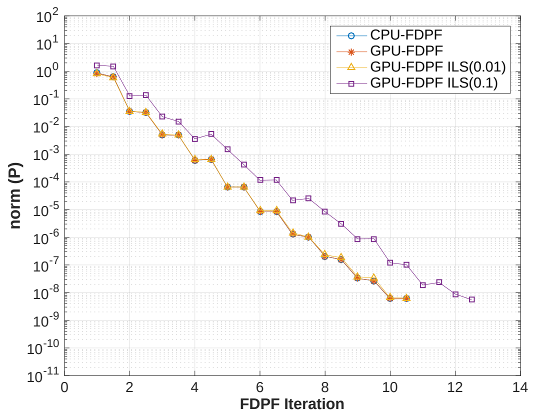


(d) case3012wp P

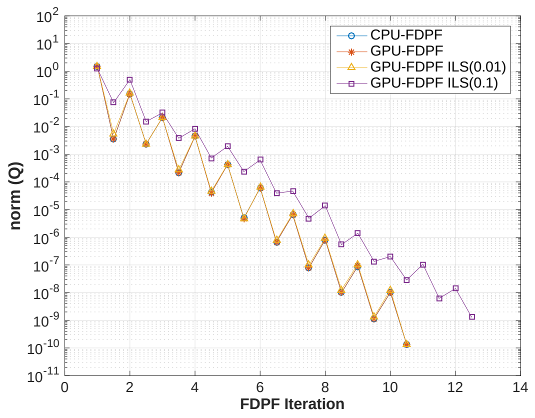


(e) case3012wp Q

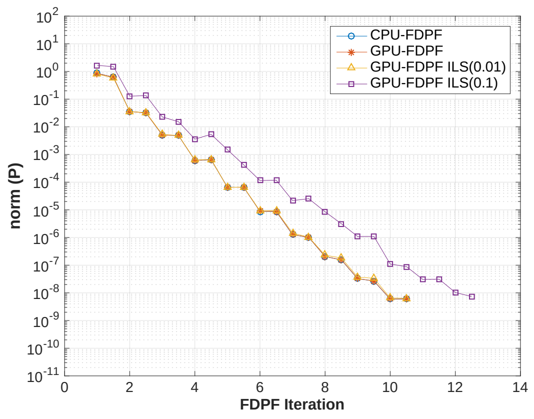
Figure 7.7: Continued



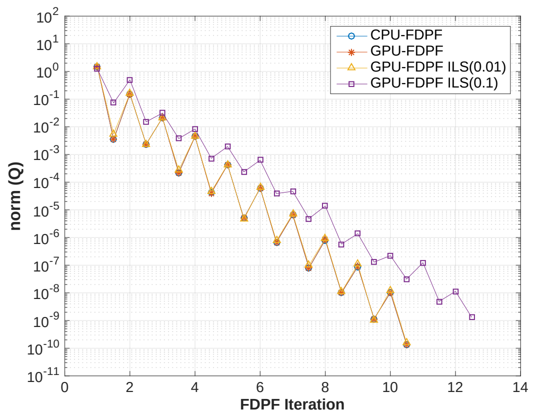
(f) case10790 P



(g) case10790 Q



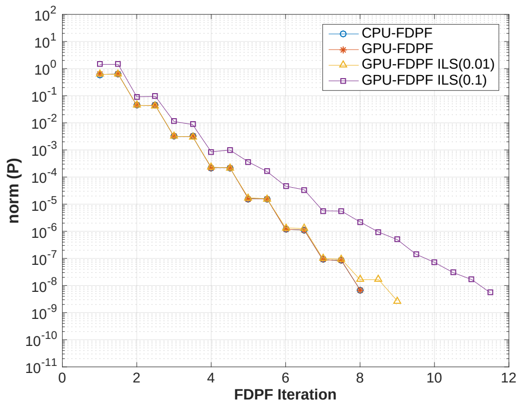
(h) case13173 P



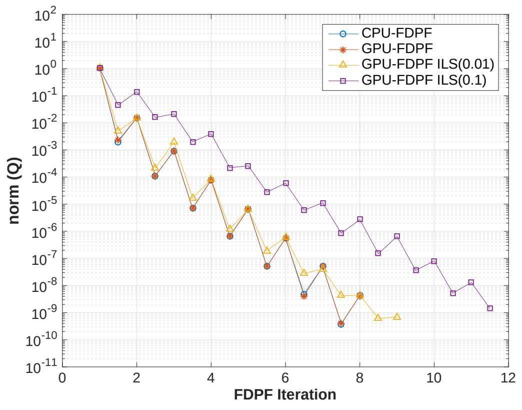
(i) case13173 Q

Figure 7.7: Continued

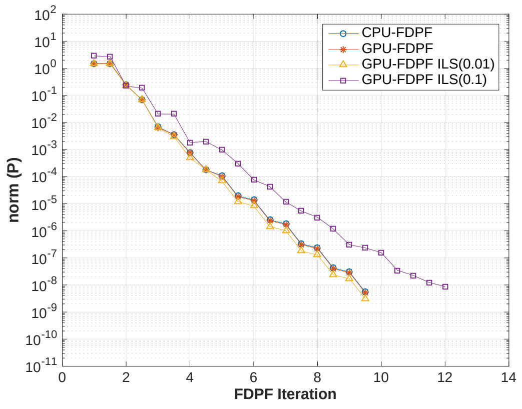
Figure 7.8: Convergence comparison between GPU-based and CPU-based FDPF for Pan-European system variations



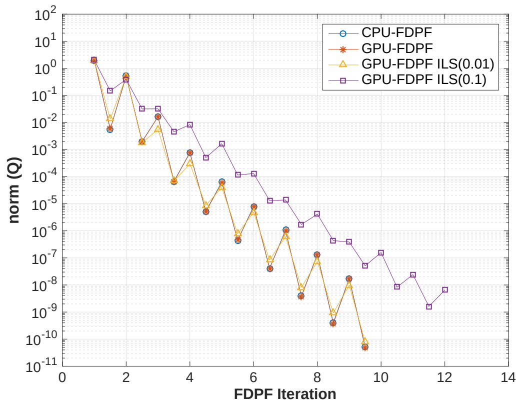
(b) case1354pegase P



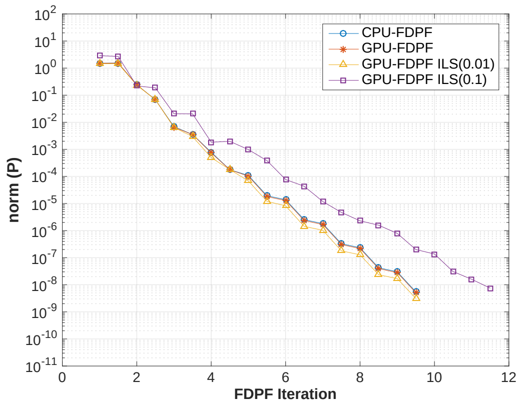
(c) case1354pegase Q



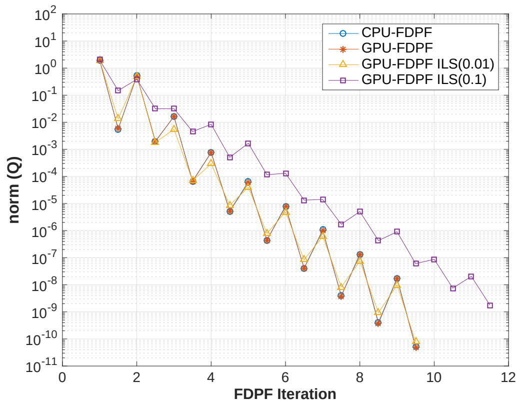
(d) case2869pegase P



(e) case2869pegase Q

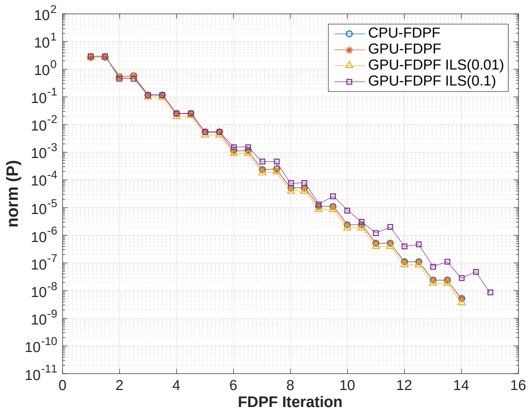


(f) case5738 P

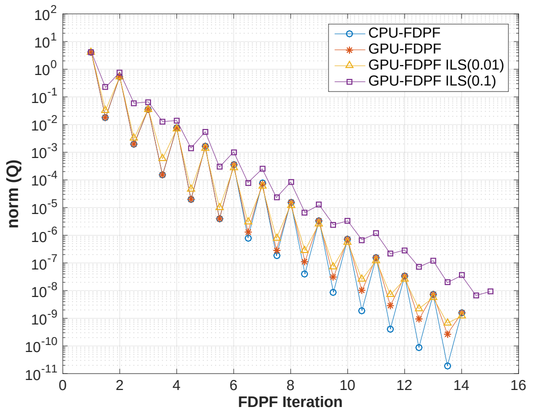


(g) case5738 Q

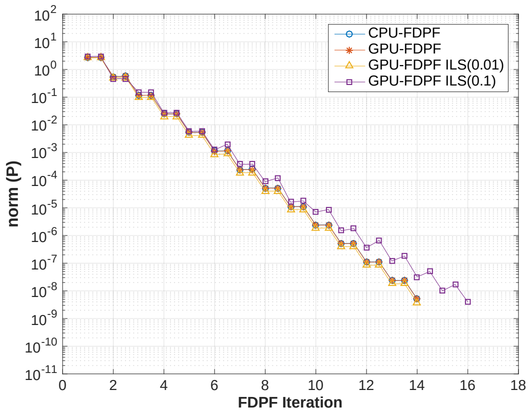
Figure 7.8: Continued



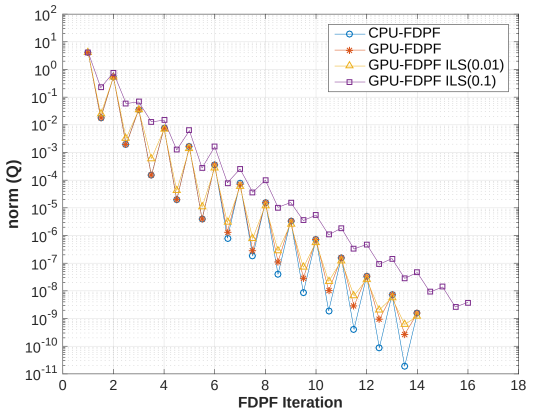
(h) case9241pegase P



(i) case9241pegase Q



(j) case11624 P



(k) case11624 Q

Figure 7.8: Continued

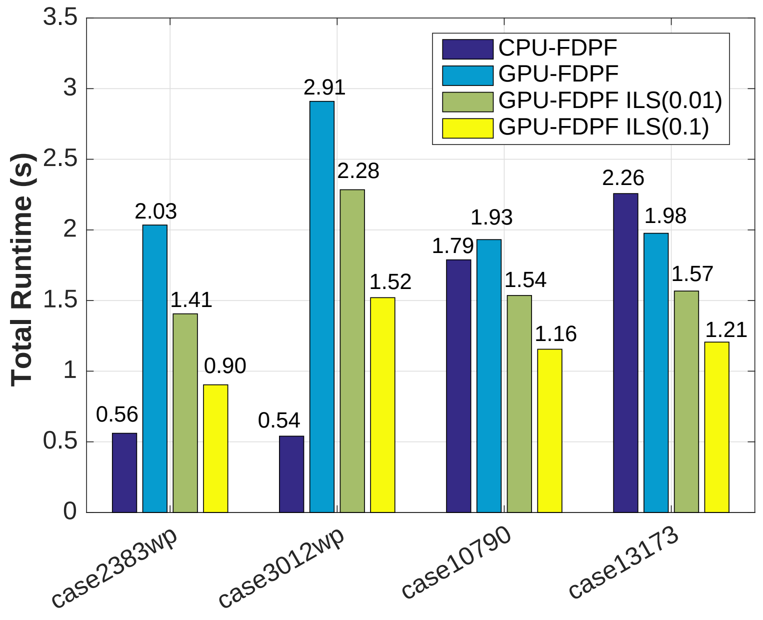
Table 7.4: Comparison of FDPF Iterations

System	CPU-FDPF		GPU-FDPF		GPU-FDPF ILS (0.01)		GPU-FDPF ILS (0.1)	
	P	Q	P	Q	P	Q	P	Q
case2383wp	18	17	18	17	18	17	19	18
case3012wp	9	8	9	8	9	8	10	10
case10790	10	10	10	10	10	10	12	12
case13173	10	10	10	10	10	10	12	12
case1354pegase	8	7	8	7	9	8	11	11
case2869pegase	9	9	9	9	9	9	12	11
case5738	9	9	9	9	9	9	12	11
case9241pegase	14	13	14	13	14	13	15	14
case11624	14	13	14	13	14	13	16	15

7.5.2 Performance

Table 7.4 shows that there are generally two or more PQ iterations for each increase of GPU-FDPF ILS (0.1) because of the introduction of inexact linear solutions. Figure 7.9 shows the total runtime of the FDPF from MATPOWER with the two groups of test systems. As mentioned in the last section, GPU-FDPF ILS(0.1) may incur a larger number of iterations compared with other implementations. However, because of the inner iteration reduction, although there may be increase of the number of PQ iteration, the GPU-FDPF ILS(0.1) provides the best performance among all the three GPU-FDPF. It can perform better than CPU-FDPF for system staring from 3000-bus scale as Figure 7.9 shows.

Table 7.5 provides a comparison of the speedup that different implementations can bring based on the results from Figure 7.9. It gives a more intuitive way to see that more speedup can be reached for the inner iterative linear with a less precision.



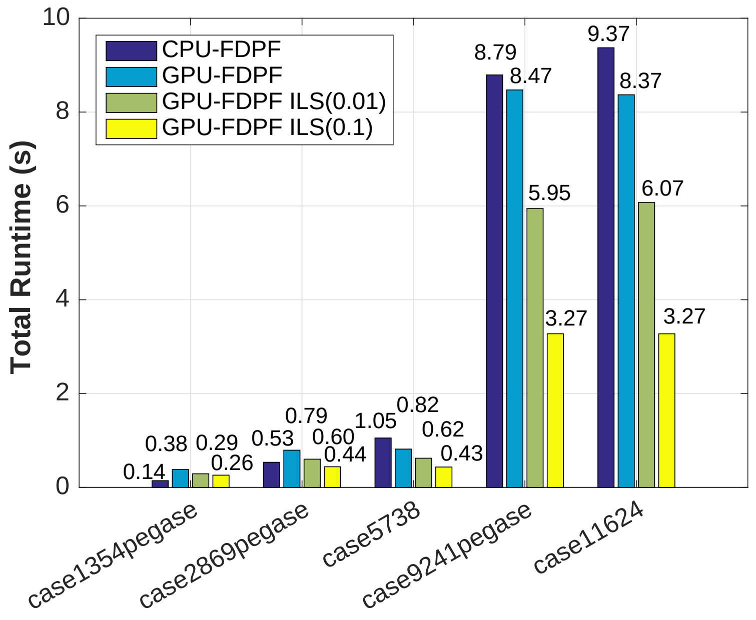


Figure 7.9: Performance comparison between the original CPU-FDPF, GPU-FDPF, GPU-FDPF ILS(0.01) and GPU-FDPF ILS(0.1)

Table 7.5: Speedup Comparison

System	CPU-FDPF (Baseline)	GPU-FDPF	GPU-FDPF ILS(0.01)	GPU-FDPF ILS(0.1)
case2383wp	1	0.28	0.40	0.62
case3012wp	1	0.19	0.24	0.36
case10790	1	0.93	**1.16**	**1.55**
case13173	1	**1.14**	**1.44**	**1.87**
case1354pegase	1	0.38	0.49	0.55
case2869pegase	1	0.67	0.89	**1.22**
case5738	1	**1.29**	**1.69**	**2.43**
case9241pegase	1	**1.04**	**1.48**	**2.68**
case11624	1	**1.12**	**1.54**	**2.86**

GPU-FDPF ILS(0.1) can always provide the most speedup compared with GPU-FDPF ILS(0.01) and GPU-FDPF. The maximum speedup that GPU-FDPF ILS(0.1) can reach is 2.86 for case11624.

Considering the FDPF iteration results in Table 7.4, more FDPF iterations will not necessarily lead to more execution time. Although all the inexact iterative linear solution (GPU-FDPF ILS(0.1) and GPU-FDPF ILS(0.01)) makes FDPF need one or several extra FDPF iterations, they can provide an overall performance improvement for systems that are sufficiently large.

Finally, Figure 7.10 shows the runtime comparison between the original MATPOWER FDPF on CPU with GPU FDPF with ILS (0.1) and also CPU FDPF ILS (0.1). CPU-FDPF uses the original back slash solver in Matlab to solve the linear equations **S1** and **S2** as in Figure 7.1. GPU-FDPF ILS(0.1) is as we discussed previously. CPU-FDPF ILS(0.1) uses the exact same algorithm as GPU-FDPF ILS(0.1), i.e. with two-step preconditioner and conjugate gradient solver, to solve the linear equations of step **S1** and **S2** with Matlab, and the ILS threshold is selected to be 0.1, which has proven to be the most efficient one.

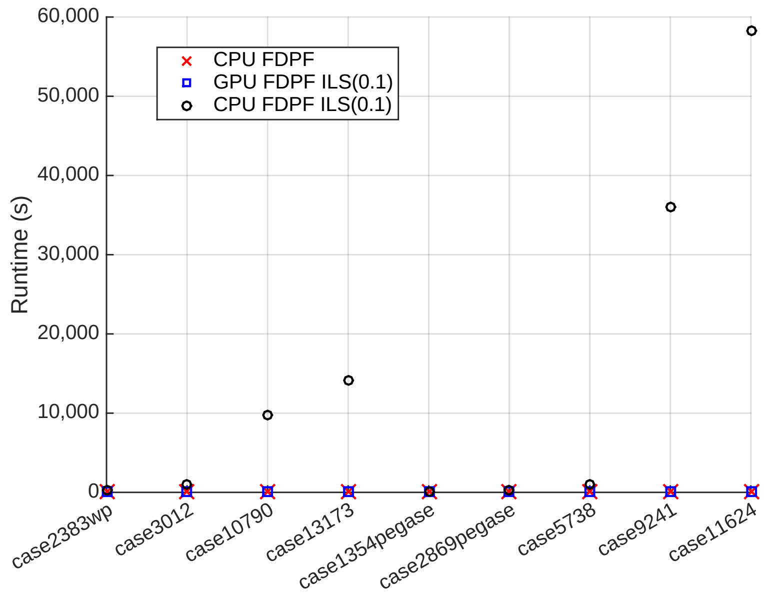


Figure 7.10: Comparison of FDPF with CPU, GPU ILS(0.1) and CPU ILS(0.1)

In Figure 7.10, CPU-FDPF and GPU-FDPF ILS(0.1) take about the same time to finish the whole fast decoupled power flow. However, CPU-FDPF ILS(0.1) takes much more time than the first two. For case11624, CPU-FDPF ILS(0.1) takes almost 60,000 seconds to finish, while CPU-FDPF and GPU-FDPF ILS (0.1) take only 9 and 3 seconds to finish the computation respectively. This figure demonstrates that there is no universal algorithm for every hardware platform. The algorithm has to closely match the features of the computation hardware to better serve the goal of efficient computation. CPU itself can finish the FDPF within an acceptable time frame. However, if an improper algorithm is chosen, the performance will be significantly degraded.

7.6 Scalability discussion

The background of this work is that LU does not scale well with the increase of problem size, therefore we try to explore the possibility of iterative solvers, which are

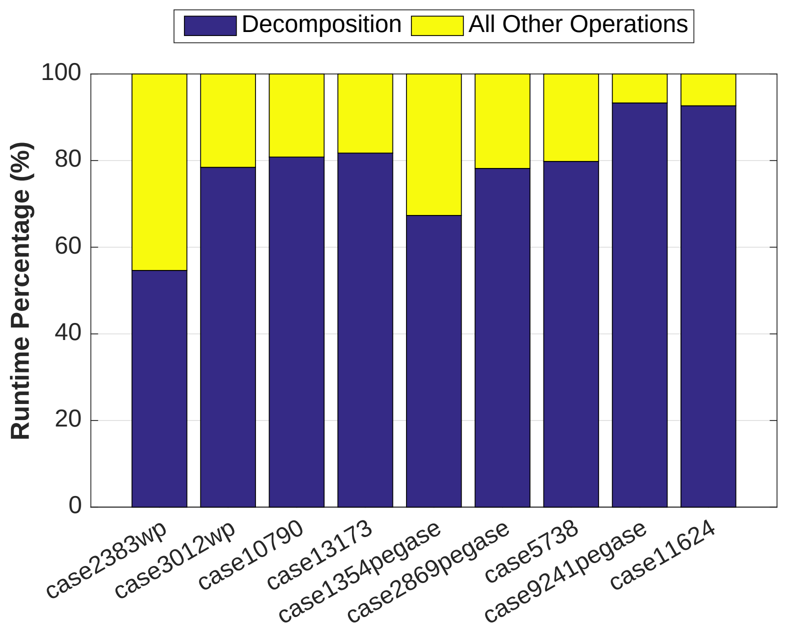


Figure 7.11: Runtime breakdown of LU decomposition and other operations.

scalable method for solving linear systems. Our experiments demonstrate this point. Figure 7.11 shows the runtime breakdown of the CPU-FDPF for different systems. We separate the LU decomposition from all the other operations of one FDPF, and show the percentage of time that the LU decomposition occupies within one FDPF calculation. It can be seen from the figure that over half of the total runtime is used for LU decomposition in all of the cases. For case9241pegase and case11624 in particular, over 90% of the total runtime is spent on LU decomposition. Therefore, when system scale is large enough, LU decomposition could be expensive, and the iterative solver becomes a better choice for solving the linear systems because of its scalability.

7.7 Conclusion

This section discusses the GPU-based FDPF with a two-step preconditioner and 1) iterative linear solution and 2) two inexact linear solution method. The GPU-FDPF is able to provide performance for several 10000-bus systems and case5738. The GPU-FDPF with inexact linear solution ($\tau = 0.01$) and stop criterion can provide speedup for one extra system (case10790).

The maximum speedup with GPU-FDPF ILS(0.1) can reach 2.86x for test system case11624 compared with the FDPF in MATPOWER. GPU-FDPF ILS(0.1) can provide performance improvement for case10790, case13173, case2869pegase, case5738 and case9241pegase, too. The performance improvement comes from several aspects. The first is that the inexact linear solving is effective in reducing the time cost for inner loop iterations. The second one comes from the two-step preconditioner, which sufficiently reduces the number of CG solver iterations. The third improvement could be contributed to the GPU parallel implementation. The forth is from the scalability issues direct method can not avoid when a system is exceptionally large, while the iterative solver and certain preconditioners can provide a scalable solution that performs well for large scale systems.

Previous GPU integrated power flow computations focus more on the performance improvement of one step of linear computation. This work integrates the GPU-based preconditioned iterative linear solver with fast decoupled power flow and demonstrates the performance improvement over standard Matlab implementations. The power system applications require intensive power flow solutions, such as N-1 contingency screen, online security assessment, and simulation with renewable energy, and they can benefit from this accelerated GPU-based fast decoupled power flow with inexact linear solution.

Chapter 8

Conclusions

The tasks for research on power system computations should be two-fold. The first task is to get more information about the system, which requires the newly developed method to be scalable enough to process large scale systems. The second task is to determine how to finish the processing of these information on time, which requires the newly developed method to be computationally efficient. Both of these require computation techniques or algorithms that are scalable, especially with penetration of renewable energy and usage of distributed energy storage, etc, which complicate the whole system. Power system itself is modeled as a nonlinear system. However, they will be transfered to a series of linear systems to be solved. How to solve these linear equations efficiently is critical to accelerating the computations in power systems. This work tries to integrate parallel computation methods, especially those can be implemented with the newly developed hardware graphic processing unit (GPU), to current power system linear computations, with the expectation to improving the computation ability and capacity of current power system computations.

This work presented a complete study from the basic linear solution to the whole process of fast decoupled power flow. The discussion about linear solution included using conjugate gradient method with Chebyshev preconditioner, Jacobi preconditioner and a two-step preconditioner JaCh. The maximum speedup for GPU-based conjugate gradient solver and Chebyshev preconditioner over corresponding CPU implementation reaches up to 10.79 times for system case2736sp. The GPU-based $JaCh$ preconditioner and conjugate gradient solver can achieve 8.9 times speedup compared with corresponding CPU implementation. Supporting discussion includes a quick estimation of the largest eigenvalue of the linear system, which is an essential parameter for the Chebyshev preconditioner. This quick estimation reduces to negligible the time spend on the computation of the largest eigenvalue.

The discussion about fast decoupled power flow integrates the most computationally efficient iterative solver and preconditioner from the first part of discussion into MATPOWER-based fast decoupled power flow. An inexact linear solving with relaxed stop criterion further improves the performance of the GPU-based iterative solver. The speedup of the GPU-FDPF with inexact linear solution can be almost 2 times faster than the native MATPOWER implementation for test system case11624.

Another point the author would like to emphasize is the scalability limitation of the LU-based direct linear solving method. Our discussion on comparison of the GPU-FDPF and MATPOWER-FDPF gives a figure (Figure 7.11) showing the percentage of the LU decomposition time out of the total runtime. When the system is getting larger, say, thousands buses scale, the time spent on LU decomposition rather than the PQ iterations actually dominates the whole computation. Therefore, the author would like to suggest considering an iterative linear solver or other scalable method to do such fundamental computations like linear solving when a large scale power system or other type of linear system is under discussion.

This dissertation explores the integration of GPU and related scalable algorithms as well as software implementation to solve the linear equations in power system applications more efficiently, so as to deal with the rapidly growing complexity of modern power systems. The experiments for basic linear solving and the integration of fast decoupled power flow demonstrate great potential for the application of GPU-based scalable methods into more power system applications.

Bibliography

Abbey, C. and Joos, G. (2009). A stochastic optimization approach to rating of energy storage systems in wind-diesel isolated grids. *IEEE Trans. Power Syst.*, 24(1):418–426. 86

Ablakovic, D., Dzafic, I., and Kecici, S. (2012). Parallelization of radial three-phase distribution power flow using gpu. In *Innovative Smart Grid Technologies (ISGT Europe), 2012 3rd IEEE PES International Conference and Exhibition on*, pages 1–7. 5

Alves, A. B., Asada, E. N., and Monticelli, A. (1999). Critical evaluation of direct and iterative methods for solving ax=b systems in power flow calculations and contingency analysis. *IEEE Trans. Power Syst.*, 14(2):702–708. 12

Amano, M., Zecevic, A. I., and Siljak, D. D. (1996). An improved block-parallel newton method via epsilon decompositions for load-flow calculations. *IEEE Trans. Power Syst.*, 11(3):1519–1527. 8

Amin, S. M. and Wollenberg, B. F. (2005). Toward a smart grid: power delivery for the 21st century. *Power and Energy Magazine, IEEE*, 3(5):34–41. 86

Anderson, E., Bai, Z., Bischof, C., Blackford, S., Demmel, J., Dongarra, J., Du Croz, J., Greenbaum, A., Hammarling, S., and McKenney, A. (1987). *LAPACK Users' guide*, volume 9. Society for Industrial Mathematics. 38

Angelis-Dimakis, A., Biberacher, M., Dominguez, J., Fiorese, G., Gadocha, S., Gnansounou, E., Guariso, G., Kartalidis, A., Panichelli, L., and Pinedo, I. (2011). Methods and tools to evaluate the availability of renewable energy sources. *Renewable and Sustainable Energy Reviews*, 15(2):1182–1200. 85

Arnoldi, W. E. (1951). The principle of minimized iterations in the solution of the matrix eigenvalue problem. *Quarterly of Applied Mathematics*, 9(1):17–29. 2

Ashby, S., Manteuffel, T., and Otto, J. (1992). A comparison of adaptive chebyshev and least squares polynomial preconditioning for hermitian positive definite linear systems. *SIAM Journal on Scientific and Statistical Computing*, 13(1):1–29. 17

Ashby, S., Manteuffel, T., and Saylor, P. (1989). Adaptive polynomial preconditioning for hermitian indefinite linear systems. *BIT*, 29(4):583–609. 17

Axelsson, O. (1972). A generalized ssor method. *BIT Numerical Mathematics*, 12(4):443–467. 2

Axelsson, O. and Lindskog, G. (1986). On the eigenvalue distribution of a class of preconditioning methods. *Numerische Mathematik*, 48(5):479–498. 33

Bai, L., Hu, Q., Li, F., Tao, D., and Sun, H. (2015). Robust mean-variance optimization model for grid-connected microgrids. In *2015 IEEE Power and Energy Society General Meeting*, pages 1–5. 86

Benzi, M. and Tuma, M. (1998). A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM J. Scientific Comput*, 19(3):968–994. 3, 16, 34

Boisvert, R. F., Pozo, R., Remington, K., Barrett, R., and Dongarra, J. J. (1997). Matrix market: a web resource for test matrix collections. *Quality of Numerical Software, Assessment and Enhancement*, pages 125–137. 45, 69, 78

Buatois, L., Caumon, G., and Levy, B. (2009). Concurrent number cruncher: a gpu implementation of a general sparse linear solver. *Int. J. Parallel Emerg. Distrib. Syst.*, 24(3):205–223. 18

Chazan, D. and Miranker, W. (1969). Chaotic relaxation. *Linear Algebra and its Applications*, 2(2):199 – 222. 2

Chen, S.-D. and Chen, J.-F. (2000). Fast load flow using multiprocessors. *International Journal of Electrical Power and Energy Systems*, 22(4):231–236. 9

Chen, Y., Jin, S., Rice, M., and Huang, Z. (2013). Parallel state estimation assessment with practical data. In *2013 IEEE Power and Energy Society General Meeting*, pages 1–5. 14

Chow, E. and Saad, Y. (1997). Experimental study of ilu preconditioners for indefinite matrices. *Journal of Computational and Applied Mathematics*, 86(2):387–414. 34

Chow, E. and Saad, Y. (1998). Approximate inverse preconditioners via sparse-sparse iterations. *SIAM Journal on Scientific Computing*, 19(3):995–1023. 16, 34

Crow, M. (2002). *Computational methods for electric power systems*. CRC Press, Boca Raton, FL. 1, 2, 26, 29, 67

Cui, H., Li, F., Fang, X., and Long, R. (2015). Distribution network reconfiguration with aggregated electric vehicle charging strategy. In *Power and Energy Society General Meeting, 2015 IEEE*, pages 1–5. 86

Dag, H. and Alvarado, F. L. (Aug. 1997). Toward improved uses of the conjugate gradient method for power system applications. *IEEE Trans. Power Syst.*, 12(3):1306–1314. 10

Dag, H. and Semlyen, A. (Nov 2003). A new preconditioned conjugate gradient power flow. *IEEE Trans. Power Syst.*, 18(4):1248–1255. 3, 17, 42, 44, 46, 58, 66

de Leon, F. and Semlyen, A. (Jul. 2002). Iterative solvers in the newton power flow problem: preconditioners, inexact solutions and partial jacobian updates. *IEE Proc. Gen., Transm. and Distrib.*, 149(4):479–484. 3, 10

Decker, I. C., Falcao, D. M., and Kaszkurewicz, E. (1996). Conjugate gradient methods for power system dynamic simulation on parallel computers. *IEEE Trans. Power Syst.*, 11(3):1218–1227. 11

Dehnavi, M. M., Fernandez, D. M., Gaudiot, J., and Giannacopoulos, D. D. (2013). Parallel sparse approximate inverse preconditioning on graphic processing units. *Parallel and Distributed Systems, IEEE Transactions on*, 24(9):1852–1862. 16

Dennis Jr, J. E. and Turner, K. (1987). Generalized conjugate directions. *Linear Algebra and its Applications*, 88-89(0):187–209. 2

Elman, H. (1989). Relaxed and stabilized incomplete factorizations for non-self-adjoint linear systems. *BIT*, 29(4):890–915. 16, 33

Elman, H. C. (1986). A stability analysis of incomplete lu factorizations. *Mathematics of Computation*, 47(175):191–217. 34

Fang, X., Li, F., Wei, Y., Azim, R., and Xu, Y. (2015). Reactive power planning under high penetration of wind energy using benders decomposition. *IET Generation, Transmission and Distribution*. 85

Flueck, A. J. and Hsiao-Dong, C. (1998). Solving the nonlinear power flow equations with an inexact newton method using gmres. *IEEE Trans. Power Syst.*, 13(2):267–273. 10

Galiana, F. D., Javidi, H., and McFee, S. (May. 1994). On the application of a pre-conditioned conjugate gradient algorithm to power network analysis. *IEEE Trans. Power Syst.*, 9(2):629–636. 9

Gallivan, K., Sameh, A., and Zlatev, Z. (1990). Solving general sparse linear systems using conjugate gradient-type methods. *SIGARCH Comput. Archit. News*, 18(3b):132–139. 16, 33

Garcia, N. (2010). Parallel power flow solutions using a biconjugate gradient algorithm and a newton method: A gpu-based approach. In *IEEE Power and Energy Society General Meeting*, pages 1–4. 4, 10, 58

Golub, G. H. and van Van Loan, C. F. (1996). *Matrix computations*. The Johns Hopkins University Press, Baltimore, MD. 2

Gopal, A., Niebur, D., and Venkatasubramanian, S. (2007). Dc power flow based contingency analysis using graphics processing units. In *IEEE Power Tech*, pages 731–736. 4

Green, R. C., Wang, L., and Alam, M. (July 2011). High performance computing for electric power systems: Applications and trends. In *IEEE Power and Energy Society General Meeting*, pages 1–8. 59

Gui, Y. and Zhang, G. (2012). An improved implementation of preconditioned conjugate gradient method on gpu. *Journal of Software*, 7(12):2695–2702. 18

Guo, C., Jiang, B., Yuan, H., Yang, Z., Wang, L., and Ren, S. (2012). Performance comparisons of parallel power flow solvers on gpu system. In *IEEE 18th Int. Conf. Embedded and Real-Time Computing Systems and Applications*, pages 232–239. 4

Gustafsson, I. (1978). A class of first order factorization methods. *BIT*, 18(2):142–156. 15, 33

Helfenstein, R. and Koko, J. (2012). Parallel preconditioned conjugate gradient algorithm on gpu. *Journal of Computational and Applied Mathematics*, 236(15):3584–3590. 17

Huang, A. Q., Crow, M. L., Heydt, G. T., Zheng, J. P., and Dale, S. J. (2011). The future renewable electric energy delivery and management (freedm) system: The energy internet. *Proceedings of the IEEE*, 99(1):133–148. 86

Huang, C., Fangxing, L., and Jin, Z. (2015a). Maximum power point tracking strategy for large-scale wind generation systems considering wind turbine dynamics. *IEEE Trans. Industrial Electronics*, 62(4):2530–2539. 85

Huang, C., Li, F., Ding, T., Jin, Z., and Ma, X. (2015b). Second-order cone programming-based optimal control strategy for wind energy conversion systems over complete operating regions. *IEEE Trans. Sustain. Energy*, 6(1):263–271. 85

Huang, Z., Chen, Y., and Nieplocha, J. (2009). Massive contingency analysis with high performance computing. In *IEEE Power and Energy Society General Meeting*, pages 1–8. 85

Idema, R., Lahaye, D. J. P., Vuik, C., and Van der Sluis, L. (Feb 2012). Scalable newton-krylov solver for very large power flow problems. *IEEE Trans. Power Syst.*, 27(1):390–396. 58

Jalili-Marandi, V. and Dinavahi, V. (2009). Large-scale transient stability simulation on graphics processing units. In *IEEE Power and Energy Society General Meeting*, pages 1–6. 4

Jalili-Marandi, V. and Dinavahi, V. (2010). Simd-based large-scale transient stability simulation on the graphics processing unit. *IEEE Trans. Power Syst.*, 25(3):1589–1599. 4

Jalili-Marandi, V., Zhiyin, Z., and Dinavahi, V. (2012). Large-scale transient stability simulation of electrical power systems on parallel gpus. *IEEE Trans. Parallel and Distributed Systems*, 23(7):1255–1266. 4

Jennings, A. (1977). Influence of the eigenvalue spectrum on the convergence rate of the conjugate gradient method. *IMA Journal of Applied Mathematics*, 20(1):61–72. 14, 56, 70

Johnson, O. G., Micchelli, C. A., and Paul, G. (1983). Polynomial preconditioners for conjugate gradient calculations. *SIAM Journal on Numerical Analysis*, 20(2):362–376. 17

Karimipour, H. and Dinavahi, V. (2013). Accelerated parallel wls state estimation for large-scale power systems on gpu. In *North American Power Symposium*, pages 1–6. 4, 14

Kempton, W., Tomic, J., Letendre, S., Brooks, A., and Lipman, T. (2001). Vehicle-to-grid power: Battery, hybrid, and fuel cell vehicles as resources for distributed electric power in california. 86

Khaitan, S. K. and McCalley, J. D. (2010). A class of new preconditioners for linear solvers used in power system time-domain simulation. *IEEE Trans. Power Syst.*, 25(4):1835–1844. 11

Koester, D. P., Ranka, S., and Fox, G. (1993). Parallel block-diagonal-bordered sparse linear solvers for electrical power system applications. In *Scalable Parallel Libraries Conference, 1993., Proceedings of the*, pages 195–203. IEEE. 9

Larson, R. (2012). *Elementary linear algebra*. Cengage Learning. 60

Li, Z., Donde, V. D., Tournier, J. C., and Yang, F. (2011). 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 Power and Energy Society General Meeting*, pages 1–8. 10

Li, Z., Zhu, J., and Yang, F. (2014). How far is the gpu technology from practical power system applications? In *IEEE Power and Energy Society General Meeting*, pages 1–5. 4

Liang, Y., Weston, J., and Szularz, M. (2002). Generalized least-squares polynomial preconditioners for symmetric indefinite linear equations. *Parallel computing*, 28(2):323–341. 17

Liu, X.-X., Wang, H., and Tan, S. X. D. (2013). Parallel power grid analysis using preconditioned gmres solver on cpu-gpu platforms. In *Computer-Aided Design (ICCAD), 2013 IEEE/ACM International Conference on*, pages 561–568. 4

Meijerink, J. A. and Van der Vorst, H. A. (1977). An iterative solution method for linear systems of which the coefficient matrix is a symmetric. *Mathematics of computation*, 31(137):148–162. 15, 33

Meijerink, J. A. and Van der Vorst, H. A. (1981). Guidelines for the usage of incomplete decompositions in solving sets of linear equations as they occur in practical problems. *Journal of Computational Physics*, 44(1):134–155. 15, 33

Mori, H., Tanaka, H., and Kanno, J. (1995). A preconditioned fast decoupled power flow method for contingency screening. In *IEEE Power Industry Computer Application Conference*, pages 262–268. 13

Nieplocha, J., Marquez, A., Tipparaju, V., Chavarria-Miranda, D., Guttromson, R., and Huang, H. (2006). Towards efficient power system state estimators on shared memory computers. In *IEEE Power Engineering Society General Meeting*, pages 1–5. 13

Notay, Y. (1994). Dric: A dynamic version of the ric method. *Numerical Linear Algebra with Applications*, 1(6):511–532. 33

NVIDIA (2012a). Cublas — nvidia developer zone. 78, 92

NVIDIA (2012b). Cusparse library. 78, 92

NVIDIA (2014). Gpu-accelerated applications. url=http://www.nvidia.com/content/tesla/pdf/gpu-accelerated-applications-for-hpc.pdf. 3

Pai, M. A. and Dag, H. (Dec 1997). Iterative solver techniques in large scale power system computation. In *Proc. of the 36th IEEE Conf. on Decision and Control*, volume 4, pages 3861–3866 vol.4. 41, 58

Pai, M. A., Sauer, P. W., and Kulkarni, A. Y. (1992). Conjugate gradient approach to parallel processing in dynamic simulation of power systems. In *American Control Conference*, pages 1644–1647. 40

Pai, M. A., Sauer, P. W., and Kulkarni, A. Y. (1995). A preconditioned iterative solver for dynamic simulation of power systems. In *IEEE International Symposium on Circuits and Systems*, volume 2, pages 1279–1282 vol.2. 11

Paige, C. and Saunders, M. (1975). Solution of sparse indefinite systems of linear equations. *SIAM J. Numerical Anal.*, 12(4):617–629. 2

Rakai, L. and Rosehart, W. (2014). Gpu-accelerated solutions to optimal power flow problems. In *47th Hawaii International Conference on System Sciences*, pages 2511–2516. 4

Saad, Y. (1981). Krylov subspace methods for solving large unsymmetric linear systems. *Mathematics of computation*, 37(155):105–126. 2

Saad, Y. (1985). Practical use of polynomial preconditionings for the conjugate gradient method. *SIAM Journal on Scientific and Statistical Computing*, 6(4):865–881. 17

Saad, Y. and Schultz, M. H. (1986). Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Scientific Statistical Comput.*, 7(3):856–869. 2

Shewchuk, J. R. (1994). An introduction to the conjugate gradient method without the agonizing pain. 14

Stewart, G. W. (1973). Conjugate direction methods for solving systems of linear equations. *Numerische Mathematik*, 21(4):285–297. 2

Stewart, G. W. (2001). *Matrix Algorithms Volume 2: Eigensystems*, volume 2. SIAM. 62

Stott, B. and Alsac, O. (1974). Fast decoupled load flow. *IEEE Trans. Power App. Syst.*, PAS-93(3):859–869. 22, 23, 60, 86

Tinney, W. F. and Hart, C. E. (1967). Power flow solution by newton's method. *IEEE Trans. Power App. and Syst.*, PAS-86(11):1449–1460. 2

Tinney, W. F. and Walker, J. W. (1967). Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proc. IEEE*, 55(11):1801–1809. 2

Top500.org (2014). Features / top 1 systems — top500 supercomputer sites. http://www.top500.org/featured/top-systems/. 37

Torquato, R., Shi, Q., Xu, W., and Freitas, W. (2014). A monte carlo simulation platform for studying low voltage residential networks. *IEEE Trans. Smart Grid*, 5(6):2766–2776. 85

Van Amerongen, R. A. (May 1989). A general-purpose version of the fast decoupled load flow. *IEEE Trans. Power Syst.*, 4(2):760–770. 60

Van der Vorst, H. A. (1981). Iterative solution methods for certain sparse linear systems with a non-symmetric matrix arising from pde-problems. *Journal of Computational Physics*, 44(1):1–19. 34

Van der Vorst, H. A. (1992). Bi-cgstab: A fast and smoothly converging variant of bi-cg for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing*, 13(2):631–644. 2

Wallach, Y. (1968). Gradient methods for load-flow problems. *IEEE Trans. Power App. Syst.*, PAS-87(5):1314–1318. 9

Wang, X., Ziavras, S. G., Nwankpa, C., Johnson, J., and Nagvajara, P. (2007). Parallel solution of newton's power flow equations on configurable chips. *International Journal of Electrical Power and Energy Systems*, 29(5):422–431. 9

Xiao, J., Bai, L., Li, F., Liang, H., and Wang, C. (2014a). Sizing of energy storage and diesel generators in an isolated microgrid using discrete fourier transform (dft). *IEEE Trans. Sustain. Energy*, 5(3):907–916. 86

Xiao, J., Bai, L., Lu, Z., and Wang, K. (2014b). Method, implementation and application of energy storage system designing. *International Transactions on Electrical Energy Systems*, 24(3):378–394. 86

Xu, Y., Hu, Q., and Li, F. (2013). Probabilistic model of payment cost minimization considering wind power and its uncertainty. *IEEE Trans. Sustain. Energy*, 4(3):716–724. 85

Xu, Y., Li, F., Jin, Z., and Huang, C. (2015). Flatness-based adaptive control (fbac) for statcom. *Electric Power Systems Research*, 122:76–85. 86

Yu, Z., Shaowe, H., Shi, L., and Chen, Y. (2014). Gpu-based jfng method for power system transient dynamic simulation. In *Power System Technology (POWERCON), 2014 International Conference on*, pages 969–975. 4

Yuan, H. and Li, F. (2015). Hybrid voltage stability assessment (vsa) for n ? 1 contingency. *Electric Power Systems Research*, 122:65–75. 85

Zhang, J. and Zhang, L. (2013). Efficient cuda polynomial preconditioned conjugate gradient solver for finite element computation of elasticity problems. *Mathematical Problems in Engineering*, 2013. 17, 18

Zhou, Q. and Bialek, J. W. (May 2005). Approximate model of european interconnected system as a benchmark system to study effects of cross-border trades. *IEEE Trans. Power Syst.*, 20(2):782–788. 45, 64, 69

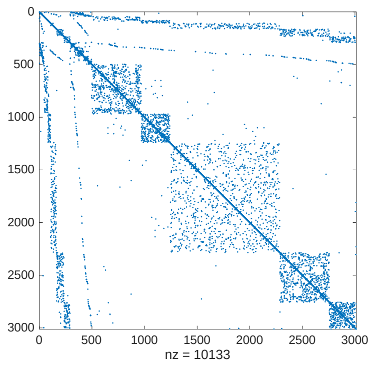
Zimmerman, R. D. and Murillo-Sanchez, C. E. (2014). Matpower 5.0 user's manual. 66

Zimmerman, R. D., Murillo-Snchez, C. E., and Thomas, R. J. (Feb 2011). Matpower: Steady-state operations, planning, and analysis tools for power systems research and education. *IEEE Trans. Power Syst.*, 26(1):12–19. 45, 64, 69, 78, 89

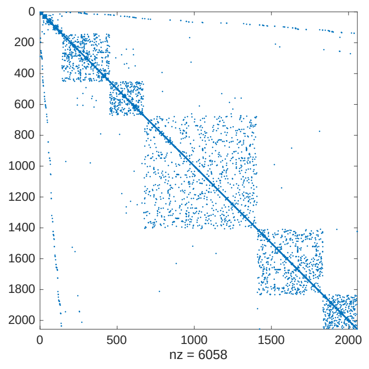
Appendix

Appendix A

Sparsity Patterns

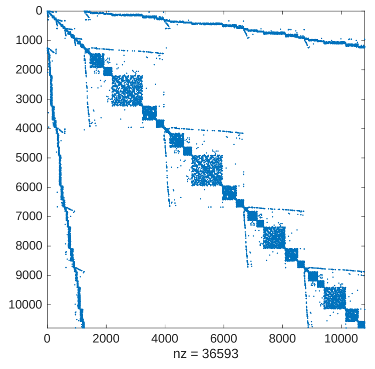


(a) case3012wp B'

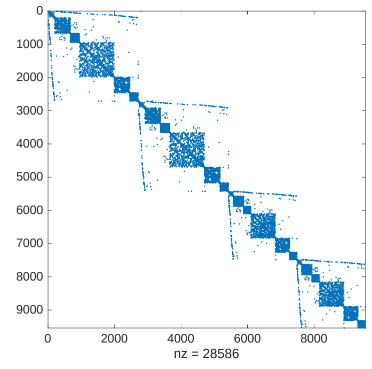


(b) case3012wp B''

Figure A.1: Sparsity pattern of case3012wp

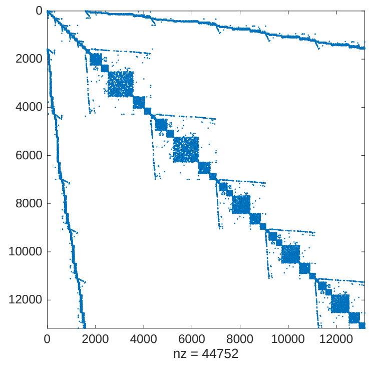


(a) case10790 B'

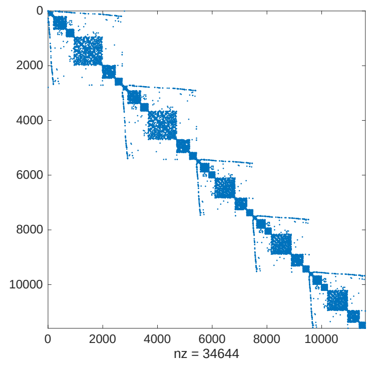


(b) case10790 B''

Figure A.2: Sparsity pattern of case10790

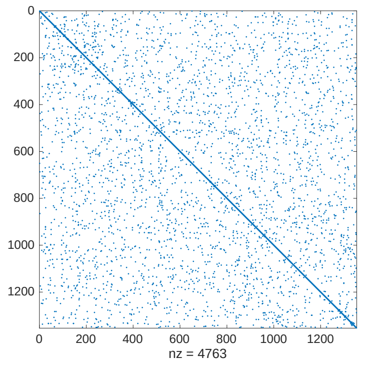


(a) case13173 B'

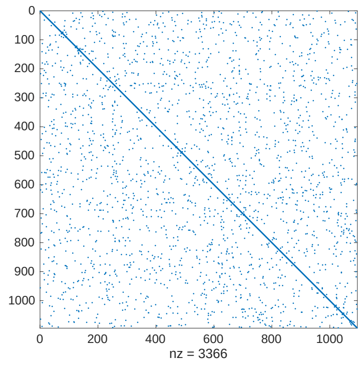


(b) case13173 B''

Figure A.3: Sparsity pattern of case13173

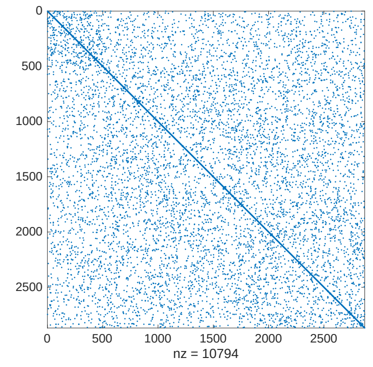


(a) case1354pegase B'

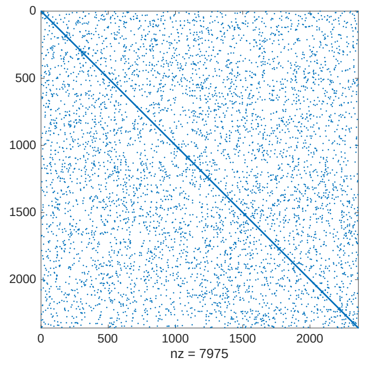


(b) case1354pegase B''

Figure A.4: Sparsity pattern of case1354pegase

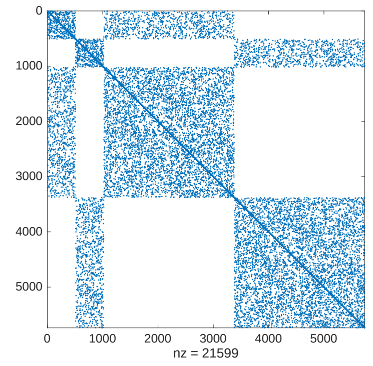


(a) case2869pegase B'

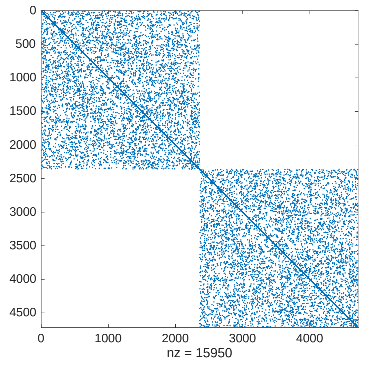


(b) case2869pegase B''

Figure A.5: Sparsity pattern of case2869pegase

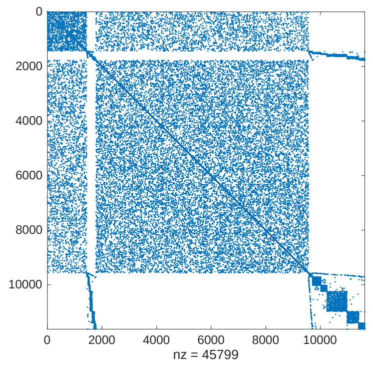


(a) case5738 B'

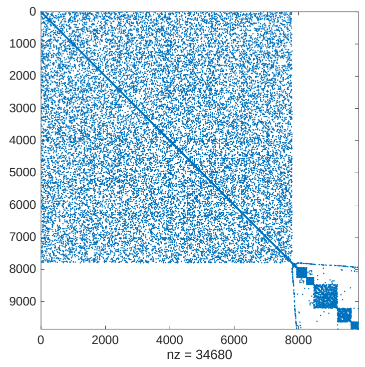


(b) case5738 B''

Figure A.6: Sparsity pattern of case5738



(a) case11624 B'



(b) case11624 B''

Figure A.7: Sparsity pattern of case11624

Vita

Xue Li received her B.S. in Computer Engineering from Northwestern Polytechnical University, Xi'an China in 2007. She received her M.S. degree in University of Tennessee, Knoxville in Computer Engineering in 2011. She started her Ph.D. studies at University of Tennessee, Knoxville from spring 2012.