# Power Grid Simulation with GPU-Accelerated Iterative Solvers and Preconditioners

**Proefschrift**

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K.C.A.M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op vrijdag 26 augustus 2011 om 10:00 uur

door **Shiming XU**,
Master in Computer Science, Tsinghua University, China

geboren te Xinxiang, China

Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. ir. A. W. Heemink

Copromotor:
Dr. ir. H. X. Lin

Samenstelling promotiecommissie:

| | |
|---|---|
| Rector Magnificus | voorzitter |
| Prof. dr. ir. A. W. Heemink | Technische Universiteit Delft, promotor |
| Dr. ir. H. X. Lin | Technische Universiteit Delft, copromotor |
| Prof.dr. W. M. Zheng | Tsinghua University, China |
| Prof.dr. C. Shen | Tsinghua University, China |
| Prof.dr.ir. C. Vuik | Technische Universiteit Delft |
| Prof.dr.ir. H. J. Sips | Technische Universiteit Delft |
| Dr.ir. M. Popov | Technische Universiteit Delft |

This thesis has been completed in partial fulfillment of the requirements of Delft University of Technology (Delft, The Netherlands) for the award of Ph.D. degree. The research described in this thesis was carried out in the Delft Institute of Applied Mathematics, Delft University of Technology. The research work was partially supported by EU Asia Link Programme under grant CN/ASIA-LINK/020, the National Natural Science Foundation of China (Grant No. 60703055) and the National Basic Research Program of China (Grant No. 2010CB951903).

Published and distributed by: Shiming Xu
E-Mail: auhgnist@gmail.com

ISBN 978-94-6186-006-4

Copyright © 2011 by Shiming Xu

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanic, including photocopying, recording or by any information storage and retrieval system, without written permission of the author.

# Summary

This thesis deals with two research problems. The first research problem is motivated by the numerical computation involved in the Time Domain Simulation (TDS) of Power Grids. Due to the ever growing size and complexity of Power Grids such as the China National Grid, accelerating TDS has become a stringent need for the online analysis of these large systems. Hence the first part of the research includes the acceleration of the TDS by means of Iterative Solvers and Preconditioners which exploit the sparsity structure of the Power Grid. The second research problem is motivated by the recent trend of using Graphics Processing Units (GPUs) in High Performance Computing (HPC). By using TDS as a sample application, the second part of research involves the design and implementation of Krylov subspace solvers and general-purpose preconditioner which can fully exploit the performance potential of GPUs.

TDS is of crucial importance to the analysis and control of Power Grids. The mathematical model of the Power Grid can be represented by a series of nonlinear Differential Algebraic Equations. With numerical time integration by an implicit scheme and linearization by Newton's Method, the major computation in TDS is the solution of a series of Jacobian matrices based linear systems. With the large size of the Power Grid and the need for fast simulation for online analysis, it is desirable to use iterative solvers and preconditioners for the solution of these linear systems. This thesis tackles the numerical problem in TDS from two aspects: design of high-performance preconditioner to the specific characterization of the TDS problem, and development of multi-step techniques for the iterative solution of a series of linear systems based on the Jacobian matrices.

We start with the analysis of the sparsity pattern of the Jacobian matrix and its relationship to the Power Network and admittance matrix. The parts in the Jacobian matrix which corresponds to the dynamic parts (i.e., the differential equations) of the Power Grid and has a block-diagonal form. The Schur complement of the dynamic part has virtually no fill-in in the algebraic part which corresponds to the connectivity of the buses in the Power Network. We then formulate a multilevel preconditioner for the Jacobian matrix based on the static sparsity pattern in the matrix and the analysis of the network topology. We show that multilevel structure based on independent sets (INDSET) can serve as an efficient preconditioners for TDS in terms of both memory efficiency and convergence property.

To accommodate matrix and right hand side changes during the simulation, we further transform the Jacobian matrix in an additive form of $Y + \Delta_Y$, where $Y$ is a static matrix and $\Delta_Y$ is a block-diagonal, low rank matrix which changes from linear system to linear system. Based on this transformation, effective preconditioner re-use, also called preconditioner updating, is derived, through dynamically adopting the changes of $\Delta_Y$ into the preconditioner initially constructed for $Y$ itself. This results in much more efficient iterative methods for TDS of Power Grid.

Furthermore, we explore the potential of matrix spectral deflation as another multi-step technique for TDS. To accommodate dynamic changes in the linear operator, GCRO-DR is used. With the additive form of $Y + \Delta_Y$, we achieve a more computationally feasible form for the updates of the deflation matrices in GCRO-DR. Spectral analysis shows that eigenvalues of both large and very small magnitude appear for the preconditioned TDS matrices, hence we propose the use of a heuristics to dynamically choose among the largest and smallest eigenvalues (or Rits values) used for deflation. The experiments show that the dynamic eigenvalue choice could greatly benefit convergence due to the dynamic changes in the matrix spectra of TDS. We also show that preconditioner updates and deflation can be used together which leads to a combined effect on the reduction of the total iteration count in TDS.

GPU based accelerated HPC systems are becoming popular due to the high performance potential and good efficiency of GPUs. Iterative Algorithm and Preconditioners are the two fundamental components of Krylov subspace solvers. However, porting them to GPU platform remain a challenge especially for preconditioners. In this thesis we target at porting of Krylov subspace solvers on GPU and the design of GPU-efficient preconditioners. Firstly we discuss the two major parts of computation involved in Krylov solvers: (1) the generation of Krylov subspace basis through Sparse Matrix-Vector multiplications (SpMV), and (2) orthogonalization by modified Gram-Schmidt method. We show that both parts can be efficiently implemented on GPU with high performance.

On GPU platform, incomplete factorization based preconditioners, such as Incomplete LU or Incomplete Cholesky, have not been successfully implemented due to the “sequentialness” and limited parallelism in their preconditioning process. While inverse form based preconditioner such as $A$-biconjugate allows higher parallelism and better performance on GPUs, it introduces too much fill-ins. We aim to design a preconditioner that can achieve high performance while maintaining good memory efficiency and convergence property of the incomplete factorizations. We use recursive multilevel structure retrieved from the elimination tree and $A$-biconjugate algorithm to achieve this. Multilevel structure is constructed based on INDSETs by symbolic analysis of the elimination tree. For the preconditioning of the last-level reduced system, we use $A$-biconjugation. The proposed preconditioner, denoted as ML-AINV, features preconditioning operation which involve a series of matrix-vector products. Through experiments with TDS problems and test matrices from various other applications, we show that ML-AINV achieves the design goal in both aspects: (1) its good convergence property is similar to incomplete factorizations, and (2) it obtains high performance by SpMV based preconditioning on GPUs.

# Samenvatting

In dit proefschrift worden twee problemen onderzocht. De eerste is het versnellen van de Tijd Domein Simulatie (TDS) van energienetwerken door middel van iteratieve methoden met preconditionering waarbij de speciale ijle matrix structuur van de energienetwerken wordt benut. Het tweede probleem is gemotiveerd door de recente trend van het gebruiken van Graphics Processing Units (GPUs) in High Performance Computing (HPC). Het betreft ontwerpen en implementeren van Krylov subspace solvers en algemene preconditionering die de prestatie potentiaal van GPUs volledig kunnen benutten, met name voor toepassingen van het TDS probleem.

TDS is van groot belang voor de analyse en operatie van energienetwerken. Het mathematische model van energienetwerken kan worden gerepresenteerd als een serie van stelsel van niet-lineaire Differentiaal Algebraïsche Vergelijkingen. Bij gebruik van de Newton linearisatie en impliciete integratie methoden is de voornaamste taak bij TDS het numerieke oplossen van een serie van lineaire systemen met Jacobiaan matrices. Vanwege de grootte van het energienetwerk en de behoefte aan snelle simulatie is het gewenst iteratieve methoden met preconditionering voor het oplossen van deze lineaire systemen te gebruiken. In dit proefschrift wordt dit probleem op twee fronten aangepakt: (1) het ontwerpen van hoge prestatie preconditioneringsmethode speciaal gericht op de karakteristieken van TDS; en (2) het ontwikkelen van multi-stappen technieken voor het efficiënt oplossen van een serie van lineaire systemen met tijd- en toestand-afhankelijke Jacobiaan matrices.

We begonnen met het analyseren van de ijle structuur van de bijbehorende Jacobiaan matrix van het elektriciteitsnetwerk. De Jacobiaan matrix bevat een deel dat correspondeert met de differentiaalvergelijkingen die het dynamische gedrag (van bijv. een generator) van het elektriciteitsnetwerk beschrijven. Dit deel heeft de vorm van blok-diagonale matrices. Het Schur-complement van dit dynamische deel levert nauwelijks vul-ins op in het algebraïsche deel van de matrix dat correspondeert met de lijn-verbindingen in het elektriciteitsnetwerk. Vervolgens hebben we een multilevel preconditioneringstechniek voor de Jacobiaan matrix gepresenteerd die gebaseerd is op het statische ijle patroon van de matrix en de analyse van de topologie van het netwerk, We laten zien dat de multilevel-structuur via een rangschikking met onafhankelijke knopenverzamelingen (INDSET) een efficiënte preconditionering in zowel geheugen-gebruik als in convergentiesnelheid oplevert.

Bij TDS moet er achter elkaar een serie van lineaire systemen worden opgelost waarbij de matrices en de rechterhand vectoren per tijdstap en vaak ook per iteratie binnen eenzelfde tijdstap veranderen. Om deze dynamische veranderingen efficiënt op te vangen hebben we de Jacobiaan matrix in een toegevoegde vorm van $Y + \Delta_Y$ herschreven, waarbij $Y$ een statische matrix is en $\Delta_Y$ een blok-diagonaal matrix van een lage rang die van lineair systeem tot lineair systeem veranderd. Gebaseerd op deze transformatie wordt er vervolgens een methode van efficiënt hergebruik van eerder berekende preconditionering, genaamd preconditionering updates, ontwikkeld. Deze methode past dynamisch aan de veranderingen in $\Delta_Y$ in de oorspronkelijk geconstrueerde preconditionering voor $Y$. Het

resultaat is een veel efficiëntere iteratieve methode voor TDS van energienetwerken.

We hebben verder de matrix spectrale deflatie als een andere multi-stappen techniek voor TDS onderzocht, hierbij wordt GCRO-DR gebruikt om de dynamische veranderingen in de lineaire operator op te vangen. De toegevoegde vorm $Y + \Delta_Y$ blijkt opnieuw een geschikte rekenkundige formulering voor updaten van de deflatie matrices in GCRO-DR. Spectrale analyse laat zien dat zowel grote als zeer kleine (in absolute waarde) eigenwaarden tegelijkertijd voorkomen in de gepreconditioneerde TDS matrices. Daarom hebben we een heuristiek voorgesteld om dynamisch de grootste en de kleinste eigenwaarden (ofwel Ritz waarden) voor deflatie te selecteren. Experimenten laten zien dat deze heuristiek voor de selectie van eigenwaarden voor deflatie zich effectief aan de dynamische verandering in de TDS matrix spectra weet aan te passen met als resultaat een grote verbetering in de convergentiesnelheid. We hebben ook aangetoond dat de technieken van updaten van preconditioneringsmatrices en deflatie geïntegreerd kunnen worden wat resulteert tot een gecombineerd effect in het verder reduceren van het totaal aantal iteraties in TDS.

HPC systemen met GPU versnellers zijn steeds populairder aan het worden vanwege het potentieel van hoge rekensnelheid en gunstige prestatie-prijs verhouding van de GPUs. Iteratieve algoritmen en preconditioneringen zijn de twee fundamentele componenten van Krylov subspace solvers. Echter, het porteren van hen op GPU platforms blijft een uitdaging vooral met betrekking tot preconditioneringen. Daarom onderzoeken we het porteren van Krylov subspace solvers naar GPU platform en ontwerpen GPU-efficiente preconditionering. Eerst analyseren we de twee belangrijkste onderdelen van de berekeningen met betrekking tot Krylov subspace solvers: (1) het genereren van een Krylov basis via ijle matrix-vector producten (SpMV), en (2) orthogonalisatie met de gemodificeerde Gram-Schmidt methode. We laten zien dat beide efficiënt kunnen worden geïmplementeerd op GPUs met een hoge prestatie.

Op GPU platforms waren preconditioneringen gebaseerd op onvolledige factorisatie, zoals Incomplete LU of Incomplete Cholesky, tot nu toe nog niet succesvol geïmplementeerd vanwege de beperkte parallellisme in het preconditioneringsproces. Hoewel preconditionering gebaseerd op inverse vorm zoals $A$-biconjugate hoge parallellisme bezit, heeft deze een groot nadeel vanwege te veel vul-ins. We mikken daarom op het ontwerpen van een preconditionering met een hoge prestatie (dat is hoge Flop/s) en tegelijkertijd de goede geheugen-efficiëntie en convergentie eigenschappen van onvolledige factorisatie worden behouden. Hiervoor gebruiken we een recursief multi-level structuur en het $A$-biconjugate algoritme. De multi-level structuur is geconstrueerd gebaseerd op INDSETs via symbolische analyse van de eliminatie boom. Voor de preconditionering van het laatste level van het recursief gereduceerde systeem gebruiken we $A$-biconjugate. De gepresenteerde preconditionering, genaamd als ML-AINV, bestaat uit een serie van matrix-vector producten. De resultaten uit de experimenten met de TDS problemen en testmatrices van verscheidene toepassingen laten zien dat ML-AINV de doelstellingen hebben gerealiseerd, namelijk: (1) convergentie eigenschappen vergelijkbaar met die van preconditioneringsmethode met onvolledige factorisaties, en (2) hoge (Flop/s) prestatie op GPUs met de SpMV gebaseerde preconditioneringen.

# Contents

**Summary** **i**

**Samenvatting** **iii**

**1 Introduction** **1**
1.1 Power Grid Simulation . . . 1
1.2 Krylov Solvers and GPU computing . . . 3
1.3 Research Motivations . . . 4
1.3.1 Time Domain Simulation . . . 4
1.3.2 GPU-based Preconditioners and Krylov Solvers . . . 4
1.3.3 Outline . . . 5

**2 Introduction to Power Grid Simulation** **7**
2.1 Introduction . . . 7
2.2 Time Domain Simulation . . . 7
2.2.1 Power Grid Components . . . 9
2.3 Jacobian Matrices . . . 9
2.3.1 Schur Complement in Jacobian Matrices . . . 12
2.4 Application of Iterative Solvers in TDS . . . 12
2.5 Simulation and Cases in TDS . . . 14

**3 Jacobian Matrices and Iterative Solvers** **17**
3.1 Introduction . . . 17
3.2 Krylov Solvers . . . 17
3.2.1 Convergence Properties of CG and GMRES . . . 19
3.2.2 Arnoldi Process, Long Recurrence and Restarts in GMRES . . . 19
3.2.3 Preconditioners . . . 20
3.3 GPU computing . . . 21
3.3.1 Introduction to CUDA . . . 23
3.3.2 CUDA Workflow . . . 24
3.4 GPU and Iterative Solvers . . . 24
3.5 Summary . . . 26

**4 Multilevel Preconditioner for TDS** **27**
4.1 Introduction . . . 27
4.2 Multilevel Preconditioner . . . 27
4.2.1 Multilevel Framework . . . 28
4.2.2 Multilevel Structure based on INDSETs . . . 29
4.3 Multilevel Preconditioner for TDS . . . 30

4.4 INDSET based on Power Network . . . 32
4.4.1 Basic Heuristics . . . 33
4.4.2 Fill-in Guidance and Large Block Sizes . . . 34
4.5 Experiments . . . 35
4.5.1 INDSET Search . . . 36
4.5.2 Characteristics of Multilevel Preconditioners . . . 38
4.6 Summary . . . 40

**5 Preconditioner Updates for TDS 43**
5.1 Introduction . . . 43
5.2 Formulation of $Y + \Delta_Y$ . . . 43
5.3 Preconditioner Update in TDS . . . 49
5.3.1 Introduction to Preconditioner Updates . . . 50
5.3.2 Updating Preconditioners in TDS . . . 51
5.4 Experiments . . . 52
5.4.1 Test Cases and Settings . . . 53
5.4.2 Comparison . . . 53
5.5 Summary . . . 55

**6 Matrix Spectra Deflation for TDS 59**
6.1 Overview . . . 59
6.2 Introduction to Deflation . . . 59
6.2.1 Computing Eigenvalues and Eigenvectors . . . 60
6.2.2 Deflation Algorithms . . . 61
6.2.3 Limitation of Deflation . . . 62
6.3 Analysis of Matrix Spectra . . . 62
6.3.1 Preconditioners to $Y$ and $Y + \Delta_Y$ . . . 63
6.3.2 $Y$ Matrices . . . 64
6.3.3 $(Y + \Delta_Y)$ Matrices . . . 64
6.4 Deflation in TDS . . . 68
6.4.1 Eigenvalue Choices . . . 72
6.5 Combining Deflation and Preconditioner Updates . . . 74
6.6 Experiments . . . 75
6.6.1 Basic Comparison with GMRES . . . 76
6.6.2 Subspace Utilization . . . 76
6.6.3 Progressive Deflation . . . 77
6.6.4 Principal Angle Analysis . . . 78
6.6.5 Deflation of Largest and Smallest Eigenvalues . . . 79
6.6.6 Deflation with Preconditioner Updates . . . 83
6.7 Summary . . . 83

**7 SpMV Optimization with CUDA 85**
7.1 Introduction . . . 85
7.1.1 Krylov Solvers and Sparse Matrix-Vector Products (SpMV) . . . 85
7.1.2 SpMV– State of the Art . . . 86
7.1.3 Outline . . . 86
7.2 GPU Cache System . . . 87
7.3 SpMV on CUDA . . . 87
7.3.1 Sparse matrix formats and SpMV . . . 87

7.3.2 Analysis of ELLPACK format . . . . . 90
7.4 Caching in SpMV . . . . . 92
7.4.1 Caching on GT-200 . . . . . 92
7.4.2 Caching Optimization on GF-100 . . . . . 93
7.5 Optimization - Bandwidth Reduction . . . . . 95
7.5.1 Locality in Accessing $x$ . . . . . 95
7.5.2 Index Compression . . . . . 96
7.6 Performance Evaluation . . . . . 98
7.6.1 Effect of Bandwidth Reduction with GT-200 . . . . . 98
7.6.2 GF-100 based Optimization . . . . . 100
7.7 Summary . . . . . 103

**8 Iterative Solver and Preconditioner on GPU 105**
8.1 Introduction . . . . . 105
8.2 Iterative Solvers on GPU . . . . . 106
8.2.1 Generation of Krylov Subspace – SpMV Operations . . . . . 108
8.2.2 Long-Recurrence Krylov Solvers – Orthogonalization . . . . . 108
8.3 Symbolic Analysis of Preconditioners . . . . . 110
8.4 AINV Preconditioner . . . . . 111
8.5 Memory-Efficient Preconditioners on GPU . . . . . 114
8.5.1 SpMV based Substitution . . . . . 115
8.5.2 Multilevel Preconditioner with AINV . . . . . 117
8.5.3 Comparison of ML-AINV and SpMV based Substitution . . . . . 121
8.6 Experiments . . . . . 122
8.6.1 Test on Jacobian Matrices from TDS . . . . . 123
8.6.2 Test of ML-AINV on Matrix Test Suite . . . . . 124
8.7 Summary . . . . . 126

**9 Conclusions and Outlook 127**
9.1 Conclusions . . . . . 127
9.2 Outlook . . . . . 129

**Acknowledgements 139**

**Curriculum Vitae 141**

# List of Figures

1.1 Power System – A Schematics . . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Affected region shown on map. . . . . . . . . . . . . . . . . . . . . . . . . 2
1.3 Satellite image of the affected region (night). . . . . . . . . . . . . . . . . . 2
1.4 From application to architecture – the scope . . . . . . . . . . . . . . . . . 3

2.1 Transmission Line – Mathematical Model . . . . . . . . . . . . . . . . . . . 9
2.2 Model for synchronous machine. . . . . . . . . . . . . . . . . . . . . . . . 10
2.3 Jacobian matrix for IEEE 39 bus system. . . . . . . . . . . . . . . . . . . . 11

3.1 Spectra of TDS Jacobian matrices – Effect of preconditioning . . . . . . . 21
3.2 Statistics of Theoretical Performance of GPU and CPU . . . . . . . . . . . 22
3.3 Performance and power usage of top 5 machines in Top-500 . . . . . . . . . 22
3.4 CUDA – GPU Architecture and Thread Hierarchy . . . . . . . . . . . . . . 24
3.5 CUDA Workflow . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

4.1 Multilevel preconditioning process. . . . . . . . . . . . . . . . . . . . . . . 28
4.2 INDSET on the network of IEEE 39 bus reference system. . . . . . . . . . . 30
4.3 Multilevel structure for the Jacobian matrix of IEEE39 system. . . . . . . 32
4.4 Effects of INDSET searching algorithms on 2K system. . . . . . . . . . . . 37
4.5 Effects of INDSET searching algorithms on 10188 system. . . . . . . . . . . 37
4.6 Effects of INDSET searching algorithms on West-US system. . . . . . . . . 37
4.7 Preconditioner memory usage and condition number relationship for 2K system. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.8 Preconditioner memory usage and iteration count relationship for 10188 system. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 41

5.1 Structure of $\Delta_Y$ matrix . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.2 Comparison of iteration counts between Preconditioner Updating strategies 56

6.1 Spectra Analysis for IEEE39 . . . . . . . . . . . . . . . . . . . . . . . . . 65
6.2 Spectra Analysis for 2K . . . . . . . . . . . . . . . . . . . . . . . . . . . 66
6.3 Heuristics function $f(\dot{a})$ on the complex plane . . . . . . . . . . . . . . . 74

7.1 Schematic comparison between cache organization on GT-200 and GF-100 87
7.2 Matrix storage formats. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
7.3 Sample matrix sparsity pattern and its storage scheme in ELLPACK. . . . 89
7.4 SpMV of Matrices in an Reduced Bandwidth form . . . . . . . . . . . . . . 95

8.1 SpMV Speedup – GPU v.s. CPU . . . . . . . . . . . . . . . . . . . . . . . 109
8.2 Performance comparison between CPU and GPU based MGS . . . . . . . 109

8.3 Elimination tree of “orsirr_1” treated with RCM and ND permutations . . . 111
8.4 ETREE– Fill-in Characterization . . . . . . . . . . . . . . . . . . . . . . . . . . 111
8.5 Sparsity of resulting matrices of $A$-biconjugate algorithm on “orsirr_1”. . . 114
8.6 ETREE– Example for Transitive Reduction . . . . . . . . . . . . . . . . . . . . 116
8.7 Sparsity pattern of 2K with ML-AINV permutation . . . . . . . . . . . . . . 123

# List of Tables

2.1 Power Networks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.2 Time Domain Simulation Cases . . . . . . . . . . . . . . . . . . . . . . 15

3.1 Quantitative comparison between GPU architectures . . . . . . . . . . . . 25

4.1 Power Networks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.2 Preconditioner's construction and solving time with iteration and memory consumption information. . . . . . . . . . . . . . . . . . . . . . . . . . . 38

5.1 TDS cases for Preconditioner Updates tests . . . . . . . . . . . . . . . . 53
5.2 Preconditioner Update – Basic Settings . . . . . . . . . . . . . . . . . . 53
5.3 Preconditioner Update results . . . . . . . . . . . . . . . . . . . . . . 53
5.4 Preconditioner Update results . . . . . . . . . . . . . . . . . . . . . . 54
5.5 Preconditioner Update results . . . . . . . . . . . . . . . . . . . . . . 55

6.1 Power systems and test cases used in deflation experiments . . . . . . . . 63
6.2 Extreme values in spectra for preconditioned augmented $Y$ matrices for 2K-STABLE case . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
6.3 Extreme values in spectra for preconditioned augmented $Y$ matrices for Dongbei-B case . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 67
6.4 Extreme values in spectra for preconditioned augmented $Y$ matrices for 10188 case . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 68
6.5 Comparison of $A$ and overhead in applying $(I - CC^H)$ . . . . . . . . . . . 71
6.6 GMRES and GCRODR – Basic Comparison . . . . . . . . . . . . . . . . . . 76
6.7 Comparison of GCRO-DR and GMRES– constant vector space size . . . . . 77
6.8 Comparison of GCRO-DR and GMRES– constant non-deflation vector space size . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
6.9 Principal Angle results – Startup Process Analysis . . . . . . . . . . . . . 80
6.10 Principal Angle results – Stationary Status . . . . . . . . . . . . . . . . 81
6.11 Effect of deflating both small and large eigenvalues . . . . . . . . . . . . 81
6.12 Choice among extreme eigenvalues . . . . . . . . . . . . . . . . . . . . 81
6.13 Cosines of the principal angles between $\mathcal{U}$ and eigenvector spaces ($\mathcal{G}_l$ and $\mathcal{H}_k$) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
6.14 Effect of combining preconditioner updates and deflation . . . . . . . . . . 83

7.1 Quantitative comparison between GPU architectures . . . . . . . . . . . . 88
7.2 Test matrices for SpMV– Characteristics. . . . . . . . . . . . . . . . . . 91
7.3 Assembly-level Timing Analysis in GT-200 . . . . . . . . . . . . . . . . . 92
7.4 Performance results of SpMV when a pseudo vector of $x$ is used for GT-200. 93
7.5 Cache line accesses in $x$ for matrices. . . . . . . . . . . . . . . . . . . . 96

7.6 Applicability of column index compression for SpMV . . . . . . . . . . . . 97
7.7 Reduction ratio in accessed memory amount for SpMV using Index Compression. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
7.8 Matrices permutated by RCM – Performance Results . . . . . . . . . . . . 99
7.9 Evaluation of Index Compression on $t_{pseudo\_x}$ . . . . . . . . . . . . . . . . . . 99
7.10 Configuration . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
7.11 Performance results on SpMV speedups . . . . . . . . . . . . . . . . . . . . 101
7.12 Performance Enhancement for GF-100 (GTX-480) GPU . . . . . . . . . . 102
7.13 Performance Comparison – GT-200 and GF-100 GPUs . . . . . . . . . . . 103

8.1 Test matrices for SpMV and Krylov subspace bases generation . . . . . . . 108
8.2 Test Platforms - CPU and GPU . . . . . . . . . . . . . . . . . . . . . . . . 123
8.3 Symbolic analysis of TDS matrices . . . . . . . . . . . . . . . . . . . . . . 123
8.4 ML-AINV characteristics on TDS Jacobian matrices . . . . . . . . . . . . . 124
8.5 Matrix test suite for ML-AINV . . . . . . . . . . . . . . . . . . . . . . . . 124
8.6 Convergence comparison between ML-AINV, ILU-C and SuperILU . . . . . 125
8.7 Performance comparison between ML-AINV and SuperILU . . . . . . . . . 125

# Chapter 1

# Introduction

## 1.1 Power Grid and Time Domain Simulation

Power Grid is one of the most fundamental infrastructures of our society. It generates power in the form of electricity and delivers it through a network to end users. A schematics of the Power Grid is shown in Fig.1.1. A Power Grid mainly consists of three parts: (1) electricity generation, (2) power transmission, i.e., the Power Network, and (3) consumption. Electricity are generated from various sources, such as thermal/hydro electric power plants. The power transmission is usually carried out by networks of power lines, voltage transformers, etc. End users include as factories, home users, etc, which consume the electricity in various ways.

In real life, huge amount of electricity is generated at any given moment and it cannot be stored effectively and must be consumed at the same time. According to [21], the averaged electricity generation in China is 4.24 $\times 10^8$ kilo-Watts in year 2009. Such amount of electricity is transferred to end users and consumed at the same time. Hence the electricity distribution should always stay at a balanced status. None of these three major components of the Power Grid is stationary and their status is subjected to changes over time. The power grid hence should be kept at a stable status, to accommodate dynamic behavior such as power line changes, electricity usage fluctuations, etc.

Dynamic changes in the power grid happen all the time. But certain changes, especially failure of major power lines, could have vast, even catastrophic aftermath. The blackout in 2003 across the eastern parts of US and Canada [2] was caused by an outburst at a power plant and cascading power line/plant failures, and affected more than 55 million people and costed billions of dollars. The study the dynamic behavior of power grids is of crucial importance to the maintenance of the Power Grid at stable status and avoidance of system-level failures. While experiments with real-life power grids is generally not possible due to very limited availability, analysis by means of simulation is the most popular and cost-effective way for understanding its behaviors.

Simulation of power systems in time domain using digital computers involves the numerical integration in time domain, based on discretization and mathematical model for components in the Power Grid. The dynamic behavior of Power Grids can be characterized in a series of Differential Algebraic Equations (DAEs). These DAEs are of highly nonlinear nature, and in numerical simulation they are usually linearized with the Newton's method. Computationally, the solution of a series of Jacobian matrix-based linear systems is the major numerical problem.

With the emerging need for fast, even online analysis and control [19] for large, es-

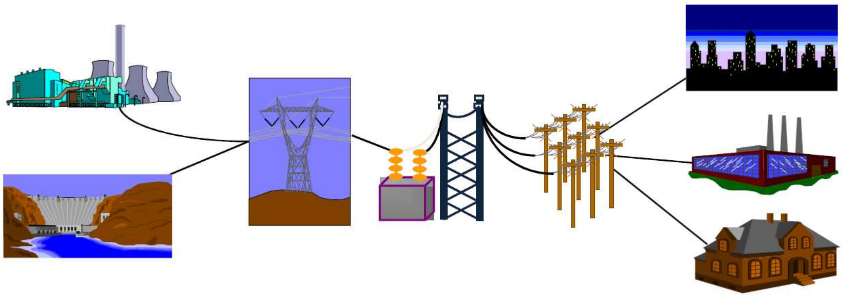


Figure 1.1: Power System – A Schematics

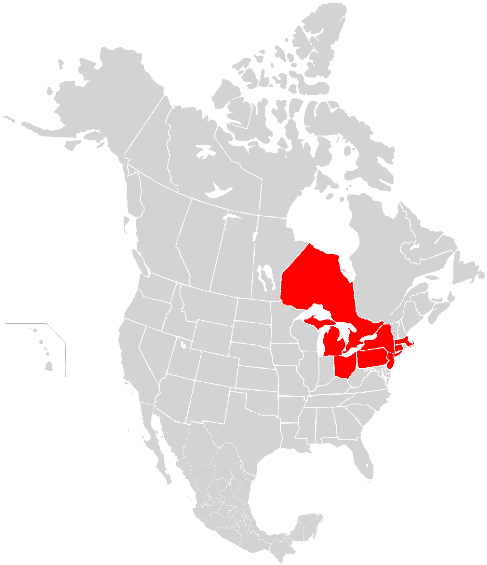


Figure 1.2: Affected region shown on map.

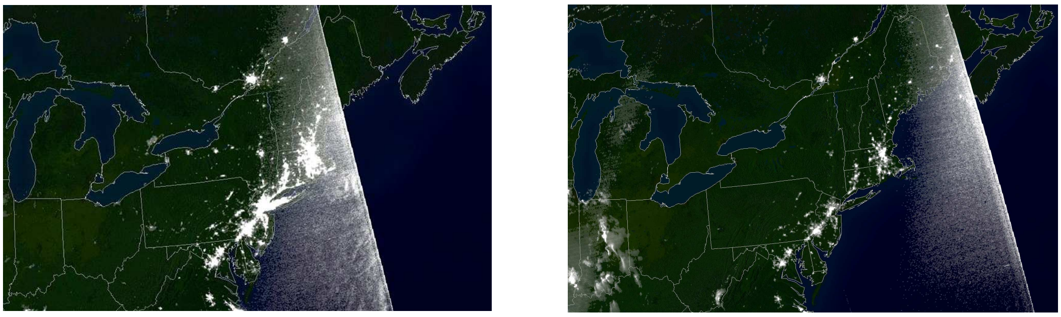


Figure 1.3: Satellite image of the affected region (night).

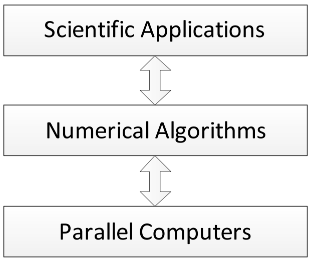


Figure 1.4: From application to architecture – the scope

pecially nation-wide Power Grids, Time Domain Simulation (TDS) has become a more important and demanding computational task. How to speed up the process of solving Jacobian based linear systems holds the key to effective Power Grid modeling and analysis. To achieve this, iterative solvers and accompanying preconditioning techniques are applied to the solution of these linear systems, as in [67, 68, 49, 41, 58, 44]. In the related research field of Power Flow study, iterative solvers are also widely adopted, as in [75, 48, 24, 93]. By using iterative solvers, the computational amount and simulation time can be potentially reduced, as compared with full factorization scheme for each and every Jacobian matrix.

## 1.2 Iterative Solvers and GPU-based Accelerated Computing

Iterative Solvers based on Krylov subspace, together with Preconditioners, represent an important category of numerical algorithms which are widely used in various scientific applications, including TDS of Power Grids. These algorithms usually reduce the norm of the error residue in the Krylov subspace. To achieve practical convergence, preconditioners are usually adopted. General-purposed preconditioners such as those based on incomplete factorization including incomplete Cholesky (IC), incomplete LU (ILU) are among the most popular.

Accelerators, especially Graphics Processing Units (GPU) are widely adopted in High Performance Computing (HPC) systems in recent years [5, 16, 15]. Compared with traditional CPU, GPU has the advantage of much higher peak floating point performance, memory bandwidth and better efficiency in terms of performance-per-watt. Contrary to traditional CPU, the parallelism exposed by GPU is fine-grained, usually data-level massive parallelism. Each GPU thread carries out relatively simple operations, while there are tens of thousands of concurrent threads running at the same time per GPU chip. The huge count of threads is necessary for hiding the latency of each thread. Unlike traditional CPU-based computation, for GPU based computation it is necessary to consider in terms of the total throughput rather than the performance/latency of each thread.

GPU has been widely used in various scientific applications [3]. Due to the fundamental difference of the throughput-oriented architecture of GPUs as compared with CPUs, scientific applications usually should be re-programmed or even re-designed to accommodate the GPU platform. This is reflected in Fig.1.4. When new parallel architecture such as GPU emerges for scientific computing, algorithms and applications have to be adapted to better utilize the potential of the new architecture. On the other side, scientific applications, as the consumer of computational power, also affect the design and evolving direction of the hardware architecture.

There have been numerous works on porting iterative solvers to the GPU platform, especially within the recent decade, such as those in [37, 36, 81, 25]. The major limitation of these works is in the lack of preconditioner algorithm which can fully exploit the computational potential of GPUs. Most frequently used preconditioners in these works are either diagonal Jacobian or block diagonal preconditioners. This is mainly due to these trivially constructed preconditioners have inner parallelism which are direct match for GPUs. But ignoring all the off-diagonal elements can be very inefficient when better convergence or more precise preconditioner is desired. Due to the fact that the preconditioning operation associated with the triangular matrices is mainly sequential in the substitution process, the traditional, general-purposed preconditioners based on incomplete factorization such as IC and ILU, have so far not been successfully ported to GPU.

## 1.3 Research Motivation and Outline

This thesis focuses on two research topics: (1) efficient solution of Jacobian-based linear systems in Time Domain Simulation of Power Grids, (2) numerical solution of linear systems on GPU-based platforms with Krylov subspace solvers and preconditioners. This section outlines the research direction on both topics and outlines the rest of the thesis.

### 1.3.1 Time Domain Simulation

The solution of a sequence of linear systems based on Jacobian matrices is the central numerical problem for TDS. To enhance the convergence of iterative solvers on these linear systems, we set out on two different tracks of study: preconditioner centric approach and iterative solver centric approaches. We develop two different categories of techniques for this problem. The first category includes preconditioner for a single Jacobian matrix-based linear system: we design multilevel preconditioner for Jacobian matrices in TDS. By analyzing the sparsity pattern of the Jacobian matrices, a multilevel structure is constructed based on off-line analysis of the Power Network topology. An algebraic multilevel preconditioner is then constructed upon this multilevel structure.

TDS involves solving a sequence of linear systems. Hence the second category of techniques include multi-step techniques to reuse information between linear solving process. Further analysis of the Jacobian matrices reveals its inner relationship with the admittance matrix. The additive formulation for the Jacobian matrix based on admittance matrices enables multi-step techniques of Preconditioner Updates and Spectra Deflation. Instead of traditional dishonest preconditioner strategy, we propose the use of preconditioner updates to accommodate dynamic changes in the Jacobian matrices. The Arnoldi process embedded in GMRES and GCR algorithms exposes extreme eigenvalues of the Jacobian matrices. Hence the eigenvectors are retrieved and used for speeding up the solution of consecutive linear systems by spectra deflation.

### 1.3.2 GPU-based Preconditioners and Iterative Solvers

Iterative Solvers and Preconditioner are used in TDS of Power Grids, as well as many other scientific applications. We target GPU accelerated iterative solvers and preconditioner as a research focus in this thesis. Krylov subspace solvers mainly rely on matrix-vector products for the generation of Krylov subspace bases. We propose optimization of Sparse Matrix-Vector Multiplication on various GPU architectures based on matrix profile reduction and cache-specific optimizations on GPU.

Incomplete factorization based preconditioners such as IC and ILU has not been successfully ported to GPUs. To fully exploit the massive parallelism on GPU while maintain good convergence properties, we design preconditioners which satisfy: (1) close relationship with incomplete factorizations in both formulation and convergence properties, and (2) high performance on GPU with inverse-based preconditioning operations. The proposed preconditioner can be applied to matrices from various applications and serve as a general-purposed preconditioning framework on GPU.

### 1.3.3 Thesis Outline

The following part of the thesis is organized as follows. Chapter 2 gives a short introduction to the mathematical model of Power Grid Simulation and the application of iterative solvers in it. Chapter 3 briefly introduces iterative solvers, preconditioners, and their relationship with GPU-based high performance computing. Chapter 4 discuss the design and implementation of multi-level preconditioner for Jacobian matrices in TDS of Power Grids. Chapter 5 and Chapter 6 cover the preconditioner-based multi-step techniques in TDS. Chapter 5 discusses preconditioner updating algorithms in TDS, while Chapter 6 discusses matrix spectra-based deflation using GCRO-DR. Chapter 7 and Chapter 8 are dedicated to the iterative solver and preconditioner on GPU. Chapter 7 focuses on the basic operation of the Sparse Matrix-Vector Multiplication (SpMV), for the generation of Krylov subspace, its optimization on GPU-based platforms. Chapter 8 proposes general-purposed preconditioners based on multilevel framework and approximate inverse. Chapter 9 gives conclusions and outlooks for future research.

# Chapter 2

# Power Grid Modeling and Simulation

## 2.1 Introduction

This chapter begins with a short introduction to the Time Domain Simulation of Power Grids in Section 2.2. Mathematical models for some basic components within the Power Grid are described, including synchronous machine, power network model, etc. The main numerical computation involved in TDS is the solution of the Jacobian matrix based linear systems. We further analyze the properties of the Jacobian matrices in Section 2.3. Analysis show that the Jacobian matrices feature sub-structures which are based on the linkage relationship in the power network. Section 2.4 surveys relevant works in the application of iterative solvers and preconditioners in Power Grid computation. Finally in Section 2.5 we introduce the Power Grid systems and cases for the experiments in following chapters.

## 2.2 Time Domain Simulation of Power Grids

The dynamic behavior of Power Grids can be characterized by a set of Differential Algebraic Equations (DAE's), as shown in Eqs.2.1 and Eqs.2.2. Eqs.2.1 characterizes the dynamic behavior of the Power Grid, while Eqs.2.2 contains the algebraic equations, which characterizes the constraints on the balanced relationship in the Power Grid, usually Kirchhoff's circuits laws. The status of the Power Grid is described by two sets of variables: $X$ and $V$. $X$ is the vector variables of dynamic components (such as states in the synchronous machines, including rotor angle, rotor speed, transient voltages at $d$-/$q$-axis of synchronous machines, etc.); $V$ is the vector of algebraic variables, usually bus voltages in the Power Network.

$$\dot{X} = f(X, V) \tag{2.1}$$

$$0 = g(X, V) \tag{2.2}$$

Typically, by discretization in time-domain by $\Delta t$ and application of implicit integration with trapezoidal rule, and application of Newton-Raphson method, we obtain the following algorithm for TDS. We use $t$ to represent the discretized time points in the simulation. The outer loop is the iteration over time steps. Based on $\begin{bmatrix} X_t \\ V_t \end{bmatrix}$, i.e, system states at time step $t$, the calculation of $\begin{bmatrix} X_{t+1} \\ V_{t+1} \end{bmatrix}$, i.e., states at step $t+1$ may involve a number of Newton iterations, indexed by the variable of $i$ (inner loop from line 5 to 15). The Newton iterations are considered to have converged if $\begin{bmatrix} \Delta X_t^{(i)} \\ \Delta V_t^{(i)} \end{bmatrix}$ is sufficiently small.

```
Input: System model for dynamic components: $f(X, V)$
       System model for balanced relationship: $g(X, V)$
       Initial system state: $X_0, V_0$
       Simulation step: $\Delta t$
       Total simulation time steps: $T_{max}$
Output: Dynamic behavior: $\begin{bmatrix} X_t \\ V_t \end{bmatrix}$ for $0 < t \leq T_{max}$
1  $t \leftarrow 0$;
2  while $t < T_{max}$ do
3      $i \leftarrow 0$ ;
4      $\begin{bmatrix} X_{t+1}^{(0)} \\ V_{t+1}^{(0)} \end{bmatrix} = \begin{bmatrix} X_t \\ V_t \end{bmatrix}$ ;
5      while True do
6          Form Jacobian matrix $A^{(i)}$ ;
7          Calculate error as $\begin{bmatrix} f_n^{(i)} \\ g(X_{t+1}^{(i)}, V_{t+1}^{(i)}) \end{bmatrix}$ ;
8          Solve: $\begin{bmatrix} f_n^{(i)} \\ g(X_{t+1}^{(i)}, V_{t+1}^{(i)}) \end{bmatrix} = -A^{(i)} \begin{bmatrix} \Delta X_{t+1}^{(i)} \\ \Delta V_{t+1}^{(i)} \end{bmatrix}$ ;
9          if $\begin{bmatrix} \Delta X_{t+1}^{(i)} \\ \Delta V_{t+1}^{(i)} \end{bmatrix}$ is sufficiently small then
10             $\begin{bmatrix} X_{t+1} \\ V_{t+1} \end{bmatrix} = \begin{bmatrix} X_{t+1}^{(i)} \\ V_{t+1}^{(i)} \end{bmatrix}$ ;
11             break ;
12         end
13         $\begin{bmatrix} X_{t+1}^{(i+1)} \\ V_{t+1}^{(i+1)} \end{bmatrix} = \begin{bmatrix} X_{t+1}^{(i)} \\ V_{t+1}^{(i)} \end{bmatrix} + \begin{bmatrix} \Delta X_{t+1}^{(i)} \\ \Delta V_{t+1}^{(i)} \end{bmatrix}$
14         $i \leftarrow i + 1$ ;
15     end
16     $t \leftarrow t + 1$ ;
17 end
18 return $\begin{bmatrix} X_i \\ V_i \end{bmatrix}$ for $0 < i \leq T_{max}$;
```

**Algorithm 1**: Time Domain Simulation

TDS algorithm involves two levels of iterations. The outer iteration is governed by the simulation time. It carries on at the time step of $\Delta_t$ until the simulation duration is reached. At each time step, a non-linear problem has to be solved. Hence Newton's method is applied and an iterative process is used to compute the system status for the next time step.

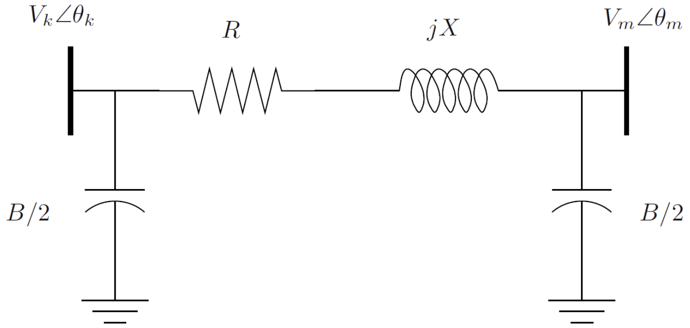


Figure 2.1: Transmission Line – Mathematical Model

The solution of the linear system on line 8 usually consumes more than 70% of the computational time during the whole simulation. Hence the major numerical problem involved in TDS is the solution of a sequence of Jacobian matrix based linear systems.

### 2.2.1 Power Grid Components and Models

The distribution of electricity in the Power Grid is through Power Network. The Power Network composes of inter-linked electrical buses [4], transmission lines, transformers, etc. Electrical bus (or 'bus' for short) serves as conceptual node in the Power Network. All other devices are connected to buses. Electrical generators and loads (for consumption of electricity) are connected to a bus. Buses are inter-liked together by transmission lines, transformers, etc. The Power Network can be viewed as a randomly sparse graph, with buses as vertices and transmission lines as edges. The circuit-level model of transmission line is shown in Fig.2.1.

The Power Network then can be abstracted as an admittance matrix. Each bus corresponds to one row/column of the matrix. The element in the matrix is non-zero if and only if the two corresponding buses are connected.

Electrical generators are the most important dynamic component in the Power Grid simulation. For electrical generators, we consider the Park-Concordia model [59] for synchronous machines. The basic scheme for the synchronous machine is shown in Fig.2.2. Depending on the details for modeling of generators, various model orders [*] are adopted. The experiments in this thesis involve generator order from 2 to 5. Normally models with order 4 or higher are usually used. The detailed models for the generator is beyond the discussion of the thesis. Please refer to [59, 63] for more details.

## 2.3 Analysis of Jacobian Matrices

Following the implicit integration and trapezoidal rule, the Jacobian matrix formed in Line 6 of the TDS algorithm at time step $t$ in Newton iteration $i$, i.e., $J^{(i)}$, and the part of the right hand side (i.e., $f_n^{(i)}$) are defined as follows:

---

[*] Generator orders denote the simplification levels to the characterization of the dynamic process of the generator, with higher orders include more details and less simplification.

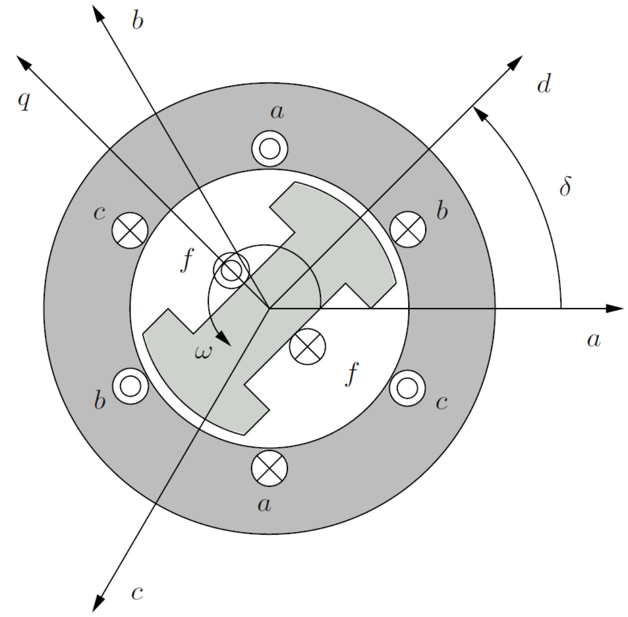


Figure 2.2: Model for synchronous machine.

$$J^{(i)} = \begin{bmatrix} I_n - \frac{\Delta t}{2}(\frac{\partial f}{\partial X}|^{(i)}_{t+1}) & -\frac{\Delta t}{2}(\frac{\partial f}{\partial V}|^{(i)}_{t+1}) \\ (\frac{\partial g}{\partial X}|^{(i)}_{t+1}) & (\frac{\partial g}{\partial V}|^{(i)}_{t+1}) \end{bmatrix} \tag{2.3}$$

$$\begin{aligned} f_n^{(i)} =& X^{(i)}_{t+1} - X_t - \\ & \frac{\Delta t}{2}\{f(X^{(i)}_{t+1}, V^{(i)}_{t+1}) + f(X_t, V_t)\} \end{aligned} \tag{2.4}$$

In our system we use phase angle and amplitude of bus voltages for the algebraic variable $\dot{V}$: $\theta$ and $v$, where $\theta$ is the vector of angles and $v$ is the vector of amplitudes at buses. Then $V$ in Eqs.2.1 and Eqs.2.2 is defined as $[\theta_1, \theta_2, \cdots, \theta_n, v_1, v_2, \cdots, v_n]^T$ where $\dot{V}_i$ is the voltage at bus with index $i$. In Eqs. 2.2 we use the active power $P$ and reactive power $Q$ for describing the balanced relationship within the system.

Then in Eqs.2.3, the Jacobian matrix can be divided into four sub-matrices: $J_{1,1}$, $J_{1,2}$, $J_{2,1}$ and $J_{2,2}$. $J_{1,1}$ has the following characteristics:

- The relationship among these components, i.e., generators, excitors, etc, are local. Each component corresponds to a block structure on the diagonal of $J_{1,1}$.

- The size of each block depends on the specific component type. For example, a 4-order synchronous machine, i.e., generator corresponds to a $4 \times 4$ block. The diagonal blocks are usually very small, and dense.

- Inter-block connection is sparse. This is due to that dynamic components are rarely inter-connected. Synchronous machines are not inter-connected. Actually in the models used in our experiments, the only possible connection exists between the generator and its associated components such as excitor. In this case, it does not break the general statement that the sparsity pattern of $J_{1,1}$ is block-diagonal. Hence

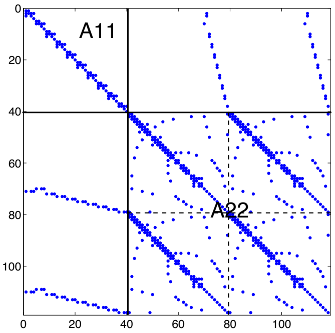


Figure 2.3: Jacobian matrix for IEEE 39 bus system.

the factorization or inverse of $J_{1,1}$ can be computed with very low computational overhead.

$J_{2,2}$ marks the differentials related to the active/reactive power to the algebraic variables, i.e., bus voltage angles and amplitudes. Hence the structure of $J_{2,2}$ is closely related to the topology of the Power Network. Non-zero elements only exist when the two buses are connected. If two buses are not connected in the network, the value corresponding to the power variable and the voltage variable should be 0. We further divide $J_{2,2}$ into four subparts. Based on definitions above, $J_{2,2}$ can be formulated as:

$$J_{2,2} = \frac{\partial g}{\partial V} = \begin{bmatrix} \frac{\partial P}{\partial \theta} & \frac{\partial P}{\partial v} \\ \frac{\partial Q}{\partial \theta} & \frac{\partial Q}{\partial v} \end{bmatrix} \tag{2.5}$$

From the analysis above, we know that each subpart of $J_{2,2}$ has the sparsity pattern defined by the graph of the Power Network. We can also arrange the vector of active/reactive power and $V$ in an bus-wise way, i.e. : $V = [\theta_1, v_1, \theta_2, v_2, \cdots, \theta_n, v_n]$. In this case, matrix $J_{2,2}$ would be composed of blocks of size $2 \times 2$, and the connection between the blocks is the same as the Power Network topology.

$J_{1,2}$ and $J_{2,1}$ mark the inter-relationship between dynamic components and buses in the network, the structure of these two matrices are coherent with the actual connection between physical entities such as generators, buses, HVDC devices, etc.

- When a single-ended device, such as a generator is associated, there will only be non-zero elements between the rows/columns corresponding to that device, and the rows/columns corresponding to the bus to which the device is connected.

- A double-ended dynamic device, such as an HVDC device is connected to two buses. Then the non-zero elements only appear on the rows/columns that are related to the device and the buses it connects.

As an example, Fig.2.3 shows the Jacobian matrix structure for a reference system from IEEE with 39 buses (dynamic components only include 4-order generators). The sub-blocks of the matrix are shown. Note the block-diagonal formulation of $J_{1,1}$, the identical substructures in $J_{2,2}$, and the inter-relationship between these two parts.

### 2.3.1 Schur Complement in Jacobian Matrices

If we take the Schur complement of $J_{1,1}$ in $J_{2,2}$, the resulting matrix would be : $J_{2,2} - J_{2,1}J_{1,1}^{-1}J_{1,2}$. Here we present an important sparsity pattern analysis of the Jacobian matrix.

**Theorem 1.** *The fill-in caused by Schur complement of $J_{1,1}$ in $J_{2,2}$ is virtually nil.*

*Proof.* For the $r$-th row in $J_{2,1}$, if the bus it corresponds to is not linked with any dynamic components, then the row is zero. If the bus it corresponds to is connected with a dynamic component, then this row has only non-zero in the columns that corresponds to the dynamic components in $J_{1,1}$. Similarly in $J_{1,2}$, the $c$-th column in $J_{1,2}$ will have non-zeros i.f.f. the bus it corresponds to is connected to a dynamic component, and only in the rows that corresponds to the components in $J_{1,1}$. Without losing the generality, we assume that the inverse of $J_{1,1}$ is a block diagonal matrix with dense diagonal blocks. Then the Schur complement of $J_{1,1}$ in $J_{2,2}$ is by definition:

$$S = J_{2,1}J_{1,1}^{-1}J_{1,2} \tag{2.6}$$

and an element in $S$:

$$S_{u,v} = R_u \times J_{1,1}^{-1} \times C_v \tag{2.7}$$

$R_u$ is the $u$-th row of $J_{2,1}$ and $C_v$ the $v$-th column of $J_{1,2}$. $S_{u,v} \neq 0$ i.f.f. bus associated with $u$ and bus associated with $v$ are connected to the same dynamic component. If this dynamic component is single-ended such as a generator, then the non-zero elements in S will be on the diagonal parts of each of the 4 sub-parts in $J_{2,2}$ (each featuring the same graph of the network topology). If this dynamic component is double-ended such as a HVDC component, then the ensuing non-zero elements in $S$ will be on both the diagonal of the sub-parts in $J_{2,2}$ and those positions in the sub-parts of $J_{2,2}$ that corresponds to the buses the component is connected to in the network. Either way, $S$ does not contain any element in $J_{2,2}$ that is not included in the sparsity pattern defined by the Power Network topology. Hence the theorem is proved. □

We will apply the theorem above to the construction of multilevel preconditioner for the Jacobian matrices and it also serves as a foundation of the further discussion in this thesis.

## 2.4 Application of Iterative Solvers in TDS

Traditionally the solution of Jacobian matrix-based linear systems are solved using direct solvers ([59, 76, 40]). The associated computation accounts for more than 70% of the total computation in the whole TDS process. Due to the fact that the Jacobian changes from iteration to iteration, the factorization has to be carried out for each matrix. With more demanding simulation requirements such as the introduction of more precise models, simulation of large-scale Power Grids, multi-step simulation techniques as in [74, 92], larger Jacobian matrices would ensue, resulting in even higher computational cost with direct solving techniques.

To tackle the growing computation demand with direct solving schemes, several works ([49, 41, 58, 44]) have been focusing on applying iterative solvers such as GMRES [72] and accompanying preconditioning techniques to TDS and related problems. Due to the important role that preconditioners play in the convergence of the iterative solving process, many research works on designing efficient preconditioning techniques have been carried out recently. It is shown in [58] that with ILU and dishonest preconditioner technique, iterative methods could achieve 66.4 times speedup against LU-based direct methods, and for TDS problems GMRES is the most robust among GMRES, BiCG, CGSquared, and CG-Stab. In [41] GMRES-E and normalization preconditioning has been presented to deal with ill-conditioned systems. This work includes the initial idea of reusing matrix spectrum information across linear system solving processes by deflation. But it has the drawback that the change of Jacobian is not well perceived. In [48], inexact Newton's method is accompanied by GMRES with varying precision to reduce computation time; (block) diagonal and ILU preconditioners are adopted. Chebyshev polynomials are used to speed up preconditioners in [44] and parallelization issues are also addressed.

Several previous works ([67, 68, 58]) utilizes iterative solvers with the dishonest preconditioning strategies. The preconditioner constructed for the Jacobian matrix at certain iteration, and is reused for further Jacobian matrices. This approach is called "dishonest preconditioners". Although this is an immediate way for reusing information and save overall computation cost, the effect of preconditioner might be compromised due to the fact that the preconditioner constructed based on previous Jacobian matrices does not effectively reflect up-to-date information in the Jacobian matrices, resulting in higher iteration counts. To alleviate this problem, some preconditioner reconstruction schemes are proposed, in which the preconditioner is re-built from time to time. Popular reconstruction scheduling schemes include: periodical reconstruction, iteration count-based heuristics, etc. However, although the optimality in the scheduling is of crucial importance, the proposed schemes are usually empirical and lacks both theoretical and technical support.

In other aspects of Power Grid computation, such as Power Flow, iterative solvers and preconditioning have also been applied. Similar to TDS, in Power Flow problem, a nonlinear problem is solved with Newton's method, and with each Newton step a Jacobian based linear system is to be solved. The main motivation of the use of iterative methods is the computational concern of direct methods, especially for large scale Power Grid analysis. [75, 48, 24, 93] uses iterative solvers to the solution of a sequence of Jacobian matrix-based linear systems which arise from Power Flow computation. In [93], iterative solvers are used to solve Power Flow problems for system size of up to 20000 buses. Preconditioned iterative solvers are effective solution techniques for these problems.

Iterative solvers and preconditioners have been successfully applied to Power Grid analysis, including TDS problems. But the current works are limited in two aspects. First, iterative solvers and preconditioners are simply applied to TDS problems as replacement for direct methods, but they do not exploit the specific characteristics of TDS problem, such as characteristics of the Jacobian matrices, etc. Second, there are very limited work on the multi-step techniques for TDS. Despite the simplicity of the popular approach of "dishonest preconditioner", it is potentially problematic in that there is no guarantee of the quality of preconditioner when applied to each and every Jacobian matrix. In this thesis we tackle the limitation of existing works by designing Jacobian matrix-specific preconditioners and multi-step techniques to reuse information more effectively across linear systems in TDS.

| System Name | Bus Count | Line Count | Generator Count | Matrix Size | $nnz$ |
|---|---|---|---|---|---|
| 2K [a] | 1165 | 1408 | 216 | 3194 | 21108 |
| Dongbei [b] | 488 | 683 | 64 | 1104 | 7208 |
| 10188 [c] | 10188 | 11905 | 1072 | 25308 | 160877 |
| West-US [d] | 5330 | 8271 | N/A | | |

---

[a] Detailed model of Hebei province, China; 4-order generator model without excitor or governer

[b] Simplified model for northeast provinces in China; 2-order generator model without exciter or governer

[c] Highly simplified model of Chinese national grid; 5-order generator model without excitor or governer

[d] Simplified Model for western United States grid; network analysis only, no information of generators available

Table 2.1: Power Networks

## 2.5 Simulation and Cases in TDS

One major issue with carrying out simulation for Power Grids is the lack of large scale models which are from real-life and hence satisfy requirements such as it should accept a stationary status in Power Flow. New simulation techniques should also be verified with actual measurements and reference waveform from the standard simulation process, such as that achieved by a commercial software as PSS.

In this thesis we mainly consider the simulation of large scale Power Grids. These include two regional Power Grid and a simplified national Power Grid of China. A reference system from the IEEE library is used: IEEE39. All the power systems used for static network analysis is listed in Tab.2.1.

We denote an instance of scenario which is simulated as a *simulation case*, or *case* for short. A *case* corresponds to a sequence of events which could potentially happen for the Power Grid. A case have a certain simulation duration. For example, for a 5 second simulation duration, the dynamic behavior of the Power Grid for 5 seconds is simulated. A standard set of simulation cases shown in Tab.2.2 are derived from the Power Grids in Tab.2.1. By default, the simulation step size, i.e., $\Delta_t$ is taken as 0.01 second by default, if not stated otherwise.

These simulation cases are used in this thesis in Chapter 4, Chapter 5 and Chapter 6 for the evaluation of multilevel preconditioner and various multi-step techniques such as preconditioner updates and matrix spectra deflation.

| Case Name | System | Case Description |
|---|---|---|
| 2K-STABLE | 2K | A three-phase fault on a single 220kV bus; the fault occurred at 1s and cleared at 1.5s; 10s simulation duration |
| 2K-UNSTABLE | 2K | A three-phase fault on a single 220kV bus; the fault occurred at 1s and cleared at 1.2s; the simulation was terminated at 2.8s due to damping |
| 10188 | 10188 | A three-phase fault on a single 500kV bus; the fault occurred at 0s and cleared at 0.06s; the fault resistance was 0+0.1j; 5s simulation duration |
| Dongbei-B | Dongbei | A single 220kV branch tripped at 1.0s; 10s simulation duration |
| Dongbei-F | Dongbei | A three-phase fault on a single 220kV bus; the fault occurred at 0s and cleared at 0.06s; the fault resistance was 0+0.1j; 10s simulation duration |

Table 2.2: Time Domain Simulation Cases

# Chapter 3

# Iterative Solvers and Accelerated Computing with GPUs

## 3.1 Introduction

The problem of solving linear systems based on large sparse matrices occurs in various scientific applications. Krylov subspace solvers are the most important algorithms for this problem. The performance and convergence properties of iterative solvers and accompanying preconditioning techniques are hence of great importance to the solution of scientific problems.

General-Purposed computing with Graphics Processor Units (GP-GPU) is a trend for High Performance Computing (HPC) in recent years. GPU poses great interest to scientific computation due to its potential for higher performance and better efficiency. Yet programming on GPU, especially when high performance is desired, poses special problem as compared with conventional CPU platforms. It is important to port and design iterative solvers and preconditioning techniques which are suitable for GPU architecture so that they can be effectively accelerated by GPU platforms.

In this chapter we give a brief survey into the Krylov subspace iterative solvers, preconditioners and GPU computation. Introduction to iterative solvers and preconditioning framework also serves as a basis for the preconditioner design and multi-step simulation techniques in following chapters. Section 3.2 covers introduction to the iterative solver framework, GMRES algorithm, preconditioners, and other relevant issues. Section 3.3 introduces General-Purposes computing on GPU (GP-GPU) and NVIDIA CUDA platform. Section 3.4 gives a brief survey of the state-of-the-art work on the application of GPU in numerical linear algebra. Design and implementation of iterative solvers and preconditioners on GPU platform is included. Section 3.5 summarizes the chapter.

## 3.2 Krylov Iterative Solvers

Solution of large sparse matrix based linear systems of form $Ax = b$ usually require iterative solvers based on Krylov subspaces. Due to the large size of the matrices, complete factorization based direct solvers may incur overhead which is inhibitively high or computationally intractable so that iterative solvers are the only practical choice. The Krylov subspace is defined as follows (with $r$ as the initial residue vector):

$$\{r, Ar, \cdots, A^m r\} \tag{3.1}$$

The Conjugate Gradient algorithm (CG , [56]) is designed to solve Symmetric Positive Definite matrix based linear systems by minimizing the $A$ norm of the residue: $\|r\|_A$. For general non-symmetric matrices, algorithms such as GMRES [73] and BiCG [30] can be applied. GMRES minimizes the residue norm in the Krylov subspace. Here we list two algorithms: GMRES and CG .

```
Input: Sparse matrix $A$
       Right hand side $\vec{b}$
       Initial guess $\vec{x}_0$
Output: Solution $\vec{x}$
$\vec{r}_0 \leftarrow \vec{b} - A\vec{x}_0$ ;
$\vec{v}_1 \leftarrow$ normalized $r_0$ ;
for $i \leftarrow 1$ to $m$ do
    $\vec{w}_i \leftarrow A\vec{v}_i$ ;
    Orthogonalize $\vec{w}_i$ against $\vec{v}_j$, for $1 \le j \le i$ ;
    if $\| \vec{w}_i \|_2 < \epsilon$ then
        Break ;
    end
    Update Hessenberg matrix $\bar{H}$ ;
    $\vec{v}_{i+1} \leftarrow$ normalized $\vec{w}_i$ ;
end
$\vec{y}_m \leftarrow \arg\min_{\vec{y}} \|\beta \vec{e}_1 - \bar{H}\vec{y}\|_2$ ;
$\vec{x} \leftarrow \vec{x}_0 + \sum_{i=1}^{m} y_i \vec{v}_i$ ;
return $\vec{x}$ ;
```

**Algorithm 2**: GMRES

```
Input: Sparse matrix $A$
       Right hand side $\vec{b}$
       Initial guess $\vec{x}_0$
Output: Solution $\vec{x}$
$\vec{r}_0 \leftarrow \vec{b} - A\vec{x}_0$ ;
$\vec{p}_0 \leftarrow$ normalized $r_0$ ;
for $i \leftarrow 0$ to $n$ do
    $\alpha_i \leftarrow \frac{\langle \vec{r}_i, \vec{r}_i \rangle}{\langle A\vec{p}_i, \vec{p}_i \rangle}$ ;
    $\vec{x}_{i+1} \leftarrow \vec{x}_i + \alpha_i \vec{p}_i$ ;
    $\vec{r}_{i+1} \leftarrow \vec{r}_i - \alpha_i A \vec{p}_i$ ;
    $\beta_i \leftarrow \frac{\langle \vec{r}_{i+1}, \vec{r}_{i+1} \rangle}{\langle \vec{r}_i, \vec{r}_i \rangle}$ ;
    $\vec{p}_{i+1} \leftarrow \vec{r}_{i+1} + \beta_i \vec{p}_i$ ;
end
return $\vec{x}_i$ ;
```

**Algorithm 3**: CG

### 3.2.1 Convergence Properties of CG and GMRES

The convergence of CG algorithm greatly depends on the eigenvalue distribution on the real axis. Simple yet elegant theoretical convergence bounds exist, such as arguably the most popular one in [52] shown in Eqs.3.2, or the less referenced yet tighter bound shown in Eqs.3.3.

$$\frac{\|x_k - x_0\|_A}{\|x - x_0\|_A} < 2\left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1}\right)^k \tag{3.2}$$

$$\frac{\|x_k - x_0\|_A}{\|x - x_0\|_A} \leq \min_{\substack{p(0)=1 \\ deg(p)=k}} \max_{1 \leq j \leq n} |p(\lambda_j)| \tag{3.3}$$

In Eqs.3.2, $\| \cdot \|_A$ is the $A$-norm, and $\kappa(A)$ is the condition number of $A$, i.e., $\frac{\lambda_{max}}{\lambda_{min}}$ where $\lambda_{max}$ and $\lambda_{min}$ are the largest and smallest eigenvalue, respectively. For generally non-normal matrices as the Jacobian matrices in TDS, there exists convergence bound described in [73], as shown in Eqs.3.4.

$$\frac{\|r_n\|}{\|r_0\|} = \min_{\substack{p(0)=1 \\ deg(p)=k}} \frac{\|Vp(\Lambda)V^{-1}r_0\|}{\|r_0\|} \leq \kappa(V) \min_{\substack{p(0)=1 \\ deg(p)=k}} \max_{1 \leq j \leq n} |p(\lambda_j)| \tag{3.4}$$

Where $V$ is the matrix of column eigenvectors. Due to that retrieving the full $V$ matrix is usually computationally intractable, hence is the analysis of the spectrum of $V$. Because the formulation of Eqs.3.4 is complex, it is hardly used in practice for convergence analysis. In actual scenarios, as a generally applicable rule, reducing the condition number by eliminating both very large and very small (close to origin) eigenvalues in GMRES by means of preconditioning or spectra deflation can enhance convergence and reduce iteration counts.

### 3.2.2 Arnoldi Process, Long Recurrence and Restarts in GMRES

In GMRES, an Arnoldi process is embedded. Arnoldi iteration has the property of capturing the extreme eigenvalues of $A$ by similarity transformation. Through the analysis of the reduced upper Hessenberg matrix, the extreme value of the original matrix $A$ can be revealed. By analyzing the spectrum of matrix $H_m$, extreme eigenvalues of $A$ could be estimated. Eigenvalue/vector estimates to that of $A$ by $H_m$ are called Rits values and Rits vectors.

$$AV_m = V_m H_m \tag{3.5}$$

In the CG algorithm, because of the symmetry in $A$, it results in a 3 term recurrence form. The counterpart of Arnoldi process for symmetric matrices is called Lanczos process. In GMRES, since $A$ is not symmetric, the recurrence is related to all the existing bases generated. Since the bases are of length $n$, in practice the Arnoldi process, hence GMRES, requires restarts, as is in other long recurrence based algorithms such as GCR [47]. When GMRES restarts, by default the previous generated bases are discarded. This is equivalent to starting GMRES with another residue vector with a smaller norm. This process discards a lot of information and there are several ways to retain and reuse some of the information. Here we list 2 approaches to reuse information from previous restarts.

- Keeping a small number of vectors (directions) of the Krylov subspace before GMRES restart, to guide Arnoldi process after restart, as in [29]
- Retain information about Rits value and Rits vectors, and use them for spectral deflation, as in GMRES-E and GMRES-DR [65]

In this thesis, spectral deflation is considered for speeding up the solution of the sequence of Jacobian matrix-based linear systems, where a non-static Jacobian, special algorithm variants have to be used to tackle the gradual changing spectra of the matrix.

### 3.2.3 Preconditioners

Because the convergence of iterative solvers greatly depends on properties of the matrix $A$, i.e., the structure of the spectrum, preconditioning is needed to speed up the convergence by using a preconditioner to amend the spectra structure of the matrix. A preconditioner $M$ is constructed based on $A$, hence the linear system is transformed:

$$M^{-1}Ax = M^{-1}b \tag{3.6}$$

The effective matrix used for the iterations would be $M^{-1}A$. $M$ should satisfy the following criteria:

- Easy to compute based on $A$
- Constructed in such a way that the application of applying $M^{-1}$, i.e., preconditioning operation should be simple
- The effective matrix, $M^{-1}A$ should have a good spectra structure

Ideally, $M$ should be a good approximation to $A$ so that $M^{-1}A$ is close to the identity matrix. On the matrix spectra level, $M^{-1}A$ has much more condensed spectrum than $A$. $M$ can also be on the right side of $A$ or a two-sided operator on $A$:

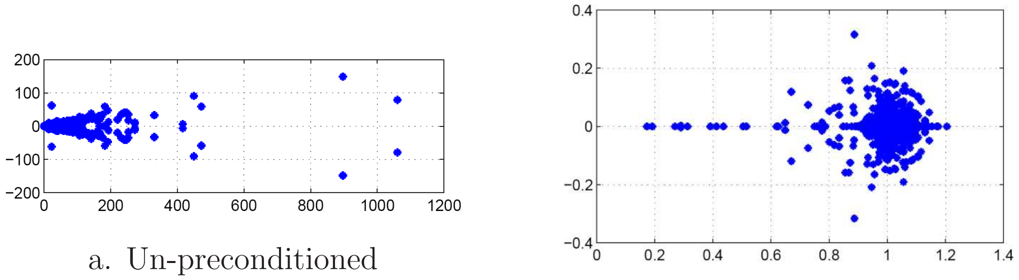
$$(AM^{-1})(Mx) = b \tag{3.7}$$

$$(M_l^{-1}AM_r^{-1})(M_r x) = M_l^{-1}b \tag{3.8}$$

We call Eqs.3.6 the left preconditioning scheme, Eqs.3.7 the right preconditioning scheme, and Eqs.3.8 the left-right preconditioning scheme. Note that the effective matrix used for iteration in these three schemes are actually different. When $A$ is a symmetric operator, such as an SPD matrix, we would expect the left-right preconditioning scheme is used and $M_l = M_r^T$, so that the symmetry of the effective matrix used for iterations is kept, so that iterative solvers such as CG still apply. This is the case when an incomplete Cholesky factorization is used for the preconditioning of an SPD matrix.

#### Incomplete Factorization based Preconditioners

A wide range of preconditioners which are popular and general-purposed are incomplete factorization based ones, for example, incomplete Cholesky factorizations (IC [30]) and incomplete LU factorization (ILU [30]). ILU preconditioners are used for the preconditioning of nonsymmetric matrices.



a. Un-preconditioned

b. Preconditioned with ILU($10^{-4}$,10)

Figure 3.1: Spectra of TDS Jacobian matrices – Effect of preconditioning

In ILU, the term "incomplete" means that the factorization process is incomplete and approximate. Suppose that the incomplete factorization is as: $A \simeq LU$. Then a non-zero error matrix exists as: $E = A - LU$.

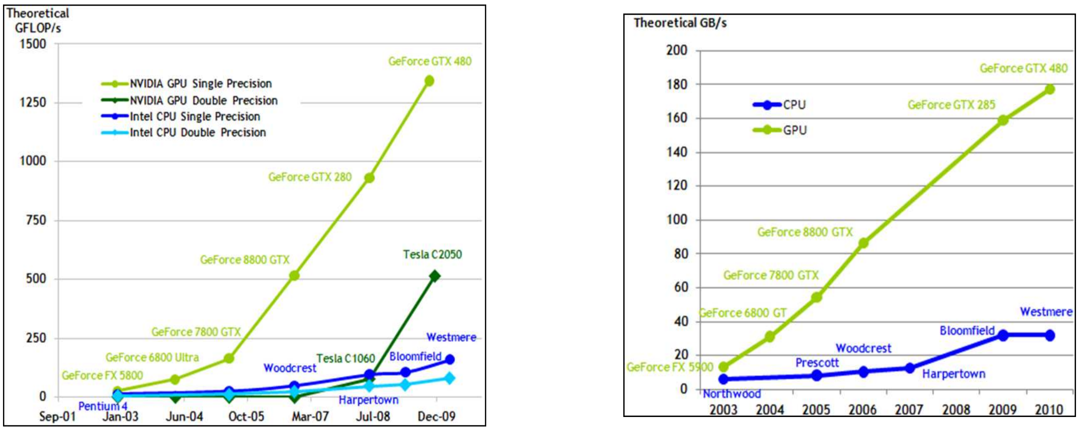
Variants of ILU preconditioner exists. High performance ILU implementations such as SuperILU [12] relies on symbolic analysis and CPU-based vectorization for high performance preconditioning.

Fig.3.1 shows the spectra of the Jacobian matrix from 2K-STABLE case in TDS (see Chapter 2 for details). Spectra of un-preconditioned matrix and that preconditioned by ILU($10^{-4}$,10) are shown. The effect of condensed spectrum in terms of iteration counts can be observed clearly: using GMRES(40), the un-preconditioned system cannot reach convergence within 500 iterations, while the preconditioned one reaches convergence in 35 iterations (here $10^{-10}$ is used as the convergence bound).

## 3.3 GPU, Accelerated Computing, and Iterative Solvers

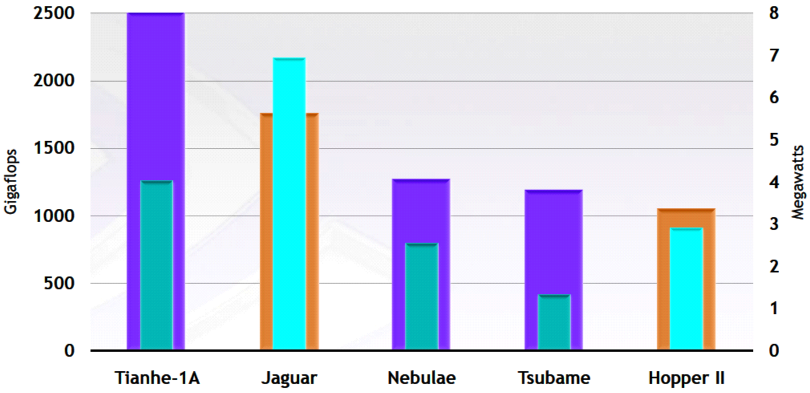
High Performance Computing (HPC) plays a crucial role in various scientific applications. With the exponential growth [8] in computational capability of microprocessors and the wide adoption of computer clusters, wide availability of HPC to ordinary researchers has become a fact. In recent years, accelerators such as GPUs have been widely used in HPC systems of various sizes [16, 15]. GP-GPU [5] is now widely used for various scientific applications. Out of the top ten HPC computers in TOP500 [17] (list in 2010-Nov [18]), four systems integrates accelerators, with three of them using Graphics Processing Units (GPUs). Another important criteria for modern HPC systems is power efficiency, i.e., Performance-per-Watt. Performance per Watt is calculated as the floating operation count that an HPC machine generate when consuming 1 watt of electricity. Green500 [14] is the popular ranking list for performance per watt among the supercomputers. On the most recent list of Green500 (2010-Nov) [13], eight of the top ten machines use accelerators as the main number crunching device. Among them five are GPU-based systems.

On the chip-level comparison between GPUs and CPUs, GPU shows much higher potential of peak floating point capability and memory bandwidth. Fig.3.2 (courtesy of NVIDIA) shows the developing trend in both peak performance and peak memory bandwidth for state-of-the-art CPU and GPU chips. For efficiency comparisons, Fig.3.3 shows the performance and power usage comparison between the top five machines in the TOP500 list (in Nov.-2010). GPU-equipped machines (including Tianhe-1A, Nebulae and Tsubame) have a very clear lead in efficiency in terms of performance per watt.



a. Peak FLOPS Comparison b. Peak Memory Bandwidth Comparison

Figure 3.2: Statistics of Theoretical Performance of GPU and CPU



* Wide bars show the peak performance, while narrow bars show the power usage.

Figure 3.3: Performance and power usage of top 5 machines in Top-500

Despite the huge potential benefit of using GPU in HPC systems, GPU poses a different scenario in terms of both architecture and programmability as compared with traditional CPU platforms. The parallelism exposed by GPU is fine-grained, usually data-level massive parallelism. Each computational thread carries relatively simple operations, while there are tens of thousands of concurrent threads running at the same time for each GPU chip. The huge amount of threads is necessary for hiding the latency of each thread. Hence unlike traditional CPU-based computations, for GPU it is necessary to consider the total throughput rather than the performance/latency of each single thread.

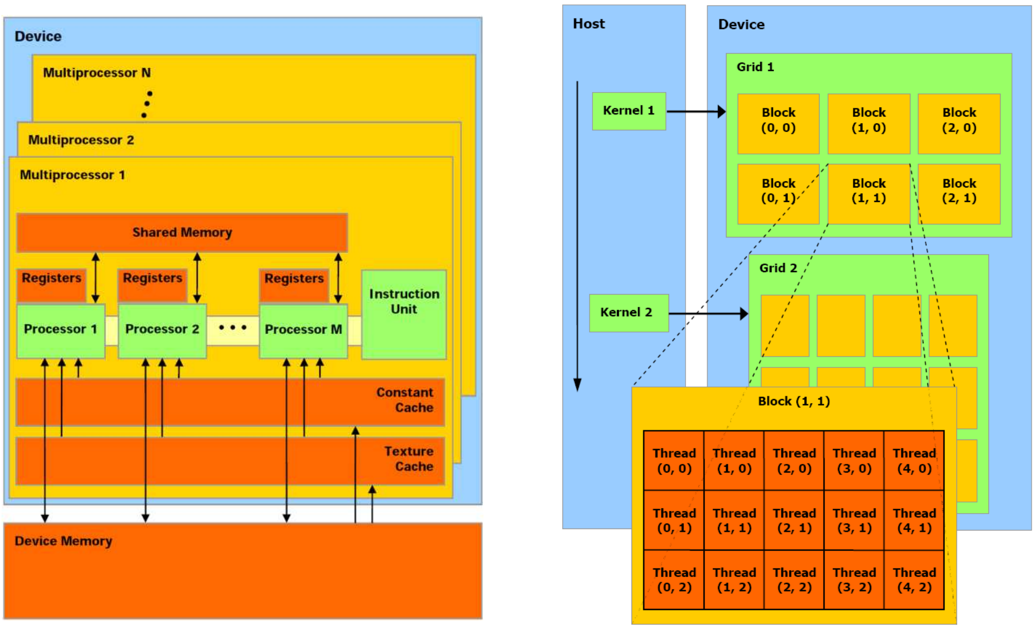
Furthermore, utilizing GPUs for scientific computing requires programmers mapping computational tasks to GPUs. This involves programming in GPU-related API [3, 9] or languages [10]. While many scattered works of GP-GPU exist as early as in 1970's (see [66] for references), wide adoption and acceptance of GP-GPU has only become a fact in recent five years, with the introduction of more general-purposed GPU design and better support for data types such as double-precision. On the hardware side, state-of-the art GPUs are capable of much higher peak performance compared with their contemporary CPU counterparts both in the single-precision and double-precision floating operation capabilities, and in memory bandwidth. On the software side, programmability is much enhanced with widely-accepted API/libraries such as CUDA[3], OpenCL[9].

In GP-GPU related research in this thesis, we use the most popular GP-GPU platform: NVIDIA CUDA, mainly based on 2 factors: (1) CUDA is arguably the most mature GP-GPU platform up to date, (2) CUDA has very good technical support and the widest adoption among GP-GPU communities.

### 3.3.1 Introduction to CUDA

In CUDA, the GPU is abstracted as a massively parallel processor. As shown in Fig.3.4.a, on the GPU chip, multiple Multiprocessors (SM) exist, each comprising of some Processors, and resources such as registers, caches, Shared Memory (ShMem). Besides the GPU chip, there is a large off-chip DRAM memory.

A CUDA-based program is composed of a main program on the CPU and several subsequent calls to GPU-based subprograms, called kernels. As shown in Fig.3.4.b, each kernel consists of a hierarchy of threads: (1) a kernel call corresponds to a thread grid, (2) a grid is a collection of blocks, and (3) a block consists of a collection of threads. Programmer specifies the task of a single thread, hence the threads are homogeneous in the logic of the computation, but different in the specific data each of them is working on. Hence the programming paradigm of CUDA is Single-Program with Multiple Data, i.e., SPMD, which is also widely used in MPI-based applications. CUDA schedules threads to multiprocessor at the granularity of blocks, i.e., threads within the same block will be executed on the same Multiprocessor. Threads can claim Multiprocessor resources such as registers, ShMem, etc. These claims are resolved at compile time and might cause resource contention and failure of compilation. When scheduled to a Multiprocessor, threads within the block will be executed until finish. Threads within the same thread block are then scheduled to the Processors at the granularity of warps, i.e., 32 threads. There is a hardware-specific upperbound on the total number of concurrent threads to be scheduled onto 1 single Multiprocessor at the same time. There may be tens of thousands of concurrent threads on the GPU. Hence CUDA on GPU is massively parallel with fine grain parallelism, in which the optimization for throughput is dominant over the performance of a specific thread.



a. GPU Architecture

b. CUDA Thread Hierarchy

Figure 3.4: CUDA – GPU Architecture and Thread Hierarchy

In Tab.3.1 we list the 2 GPU architecture used for CUDA experiments in this thesis. GT-200 is the first NVIDIA GPU to include Double-Precision support. GF-100 is the latest NVIDIA GPU (in year 2010) with a renewed design in both Multiprocessor and memory subsystem. The timing results in Tab.3.1 are based on both results in [84] and similar micro-benchmark results on GTX-480 GPU.

### 3.3.2 CUDA Workflow

When using CUDA, the programmer implements functions to be executed on CUDA devices through `.cu` files. Also the code that calls CUDA kernels on the host, i.e., CPU, has to be implemented. CUDA toolkit will compile both files and link them together to an executable file. This process is shown in Fig.3.5. Green blocks in the figure are included in CUDA toolkit. CUDA compiler compiles `.cu` files into `.ptx` files, which are assembly-level files for the CUDA kernels. PTX format based assembly code is very close to the binary code to be executed on the device. Usually it has a 1-to-1 mapping between these two instruction sets. Analyzing PTX level codes gives us more potential for detailed analysis and low-level manipulation for performance enhancements on GPU.

## 3.4 Utilizing GPU in Sparse Linear System Solvers

Linear system solvers, especially for large scale sparse linear systems, play a very important role in scientific computing and numerical linear algebra. While GPU has been proven to be effective with high performance in dense matrix-based linear algebra such as factorization, etc [78, 39, 6], so far the work on the solution of sparse matrix based linear systems with GPUs is limited.

The major problem with sparse solvers on GPU is that sparse matrix structure poses special problems for effective parallelization on GPUs, including: (1) irregularity in data

| GPU | Series | GT-200 | GF-100 |
|---|---|---|---|
| | Year | 2008 | 2010 |
| | Name | Tesla C1060 | GeForce GTX-480 |
| SP count per SM | | 8 | 32 |
| SM Count | | 30 | 15 |
| Concurrent Thread Count per SM | | 1024 | 1536 |
| Concurrent Thread Count | | 30720 | 23040 |
| Memory Bandwidth | | ~80 GB/s | ~115 GB/s |
| Shared Memory | | 16 KB per SM | 16/48 KB per SM |
| L1 Data Cache | Size | N/A | 16/48 KB per SM |
| | Hit Latency | | 80 cycles |
| L2 Data Cache | Size | N/A | 768 KB |
| | Hit Latency | | 212 cycles |
| L1 Texture Cache | Size | ~ 5KB per SM | 12 KB per SM |
| | Hit Latency | 258 cycles | 220 cycles |
| L2 Texture Cache | Size | 256 KB | None (unified Data Cache) |
| | Hit Latency | 366 cycles | 427 cycles |
| | Miss Latency | 547 cycles | 632 cycles |
| Global Memory | Size | 4 GB | 1.5 GB |
| | Latency | 506 cycles | 319 cycles |

Table 3.1: Quantitative comparison between GPU architectures

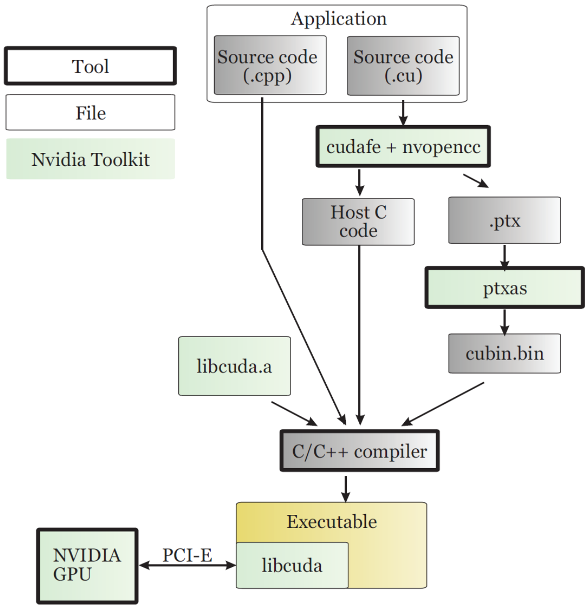


Figure 3.5: CUDA Workflow

access pattern and program branches, and (2) limited parallelism in the construction and substitution process. GPU has shown limited speedup in the factorization phase as indicated in [55]. Due to that the full factorization is used only once in the solution phase, the benefit of using GPU is therefore limited.

Unlike factorization in direct solvers, preconditioners which are based on incomplete factorizations are used for $m$ times during the solution of the linear system, where $m$ is the number of iterations required for convergence. Preconditioners that can be fully utilize the potential of GPU platforms is of crucial importance to the overall performance of the iterative solver on GPUs.

There have been several works in applying GPU to iterative solvers. They can be divided into two categories: implementation of the iterative solvers, and design of new preconditioners that can effectively utilize performance potential of GPU. Without preconditioning, the iterative solver can be directly implemented on GPU platform without much hassle. Since the algorithmic core of iterative solvers such as CG and GMRES mainly consists of level-1 BLAS operations and matrix-vector products. The implementation of the solvers is straightforward. The main problem for iterative solvers on GPU is the pursuit of higher performance through optimization. We dedicate Chapter 7 to this issue.

The preconditioner design on GPU is a major issue for effective iterative solver on GPU. Up to now the works on this topic are still rare and lacks generality. Especially for general-purposed preconditioners such as IC or ILU, there is no counterpart implementation on GPU. This is mainly due to the preconditioning operation of these preconditioners relies on substitution which lacks sufficient parallelism on GPU. The pioneering work in [37] [36] used Jacobian preconditioners which is trivial in construction and preconditioning. Work in [81] used Block ILU preconditioner to exploit block-level parallelism to match that of GPUs. This can be seen as a generalized block-Jacobian preconditioner which ignores all the off-diagonal data in the matrix. Hence these (block) Jacobian preconditioners are limited in generality and convergence properties. Work in [25] utilizes the specialized matrix formulation from Poisson equations for the construction of preconditioners on GPU. No effective work has been propose on the incomplete factorization based preconditioners on GPU as of Nov. 2010.

## 3.5 Summary

In this chapter we briefly introduced basics of iterative solvers and preconditioners. We also gave a introduction to GP-GPU and CUDA platform. Iterative Solvers can be implemented on GPU with good match for the GPU architecture. On the contrary, general purposed preconditioners such as IC or ILU face specific problem on GPUs, due to limited inherent parallelism. Until now, limited work has been done on the design of parallel incomplete factorization based preconditioners on GPU. In this thesis, we dedicate two chapters to iterative solver and preconditioners on GPU. Chapter 7 focuses on optimization on the core operation in iterative solvers, i.e., Sparse Matrix-Vector multiplication (SpMV) on GPU. Chapter 8 considers iterative solver and preconditioner on GPU, focusing on the design of GPU-efficient preconditioners based on incomplete factorizations.

# Chapter 4

# Multilevel Preconditioner for Jacobian Matrices in TDS

## 4.1 Introduction

Multilevel techniques are known to be able to solve large systems effectively with good memory efficiency ([71, 27]), especially for systems with regular structures. In this chapter we apply multi-level approach to the preconditioning of Jacobian matrices in Time Domain Simulation (TDS) of Power Grids. Multilevel preconditioners proposed in this chapter are constructed based on the algebraic structure of the Jacobian matrices. Specifically, Independent Set (IndSet) are retrieved by analyzing the matrix structure and power networks. Multilevel preconditioners are constructed by using the specific structure of the parts in the Jacobian matrix corresponding to the Dynamic components in the power network and the recursive Independent Set structure on the network.

This chapter is organized as follows. In Section 4.2 we give a short introduction to the multi-level preconditioning framework based IndSet's and algebraic structure of the TDS Jacobian matrices. In Section 4.3 we analyze the characteristics of both the dynamic parts and algebraic parts of the matrix, and apply the multilevel framework to it. We present the technique of applying IndSet search based on power system network to the construction of multilevel preconditioner. To reduce the memory usage and preconditioning overhead of the preconditioner, we propose the use of fill-in guidance in IndSet searching algorithms in Section 4.4. Experiments and results are included in Section 4.5. Section 4.6 summarizes the chapter.

## 4.2 Multi-level Preconditioners

Unlike traditional scenarios in which multilevel techniques are applied to, such as Finite Difference or Finite Element, Jacobian matrices arise in TDS composes of randomly sparse structures. Hence for TDS problem we consider the construction of algebraic multilevel structure based on the matrix sparsity pattern. In this section we examine the general algebraic structure of multilevel method and its application in preconditioning in Section 4.2.1 and give the recursive algebraic multilevel framework based on IndSet's in Section 4.2.2.

---

Part of content in this chapter has been published as conference paper [85] and accepted as journal paper [90]

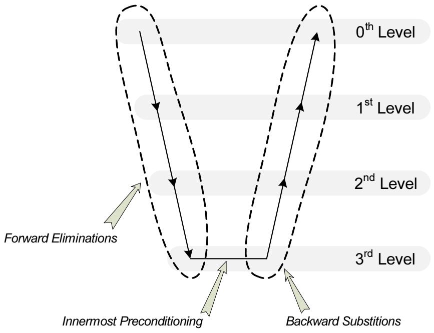


Figure 4.1: Multilevel preconditioning process.

### 4.2.1 Multilevel Preconditioning Framework

In multilevel framework, a hierarchical structure is constructed based on the original problem domain defined as $\mathcal{M}_0$. In a multilevel framework with $l$ levels, a series of subdomains are created recursively, i.e., $\mathcal{M}_{i+1}$ is retrieved from $\mathcal{M}_i$, for $i = 0, ..., l-1$. $\mathcal{M}_i$'s satisfy:

$$\mathcal{M}_0 \supset \mathcal{M}_1 \supset \mathcal{M}_2 \supset ... \supset \mathcal{M}_l \tag{4.1}$$

Each $\mathcal{M}_i$ is a direct reduced system of $\mathcal{M}_{i-1}$, and a reduced system of $\mathcal{M}_0$ itself. In a matrix form, the original matrix is $A$, and $\mathcal{M}_0$ corresponds to the set of the column/row indices in $A$. We denote the system on level $i$ as $A_i$ and let the system on the $0^{th}$ level as $A_0 = A$, then a multilevel structure can be constructed recursively. Based on $A_i$, i.e., the system on level $i$, a permutation matrix $P_i$ is constructed so that:

$$P_i A_i P_i^T = \begin{bmatrix} D_i & F_i \\ E_i & C_i \end{bmatrix} \tag{4.2}$$

$$= \begin{bmatrix} I_i & \\ E_i D_i^{-1} & I_i \end{bmatrix} \times \begin{bmatrix} D_i & F_i \\ & \tilde{A}_{i+1} \end{bmatrix} \tag{4.3}$$

In which $C_i$ corresponds to the nodes on level $i+1$, i.e., those in domain $\mathcal{M}_{i+1}$, and $D_i$ to those left on level $i$, i.e., those in $\mathcal{M}_{i+1} - \mathcal{M}_i$. Also $\tilde{A}_{i+1} = C_i - E_i D_i^{-1} F_i$, which is the Schur complement of $D_i$ in $P_i A_i P_i^T$, will serve as the system on level $i+1$.

When used for preconditioning, for the sake of fast speed and low memory usage, an approximation to $\tilde{A}_{i+1}$ is used as the system on level-$(i+1)$, denoted as $A_{i+1}$. Also the inverse of $D_i$ is usually approximated and values may be dropped in the Schur complement.

The scheme of the preconditioning process is illustrated in Fig.4.1 in a classical V-cycle form with 4 levels. The V-cycle process is carried out per preconditioning operation. On level $i$ suppose $A_i$, $P_i$, $v_i$ and $u_i$ are the matrix, the permutation, the vector to be preconditioned, and the preconditioned vector, respectively. Then we have:

$$(P_i A_i P_i^T)(P_i u_i) = P_i v_i \tag{4.4}$$

If we write this equation in block matrix form, and divide $P_i u_i$ and $P_i v_i$ into $u_i'$, $u_i''$ and $v_i'$ and $v_i''$, according to the way $P_i A_i P_i^T$ is divided into blocks, we get:

$$\begin{bmatrix} D_i & F_i \\ E_i & C_i \end{bmatrix} \begin{bmatrix} u_i' \\ u_i'' \end{bmatrix} = \begin{bmatrix} v_i' \\ v_i'' \end{bmatrix} \tag{4.5}$$

The resulted form is as:

$$\begin{bmatrix} D_i & F_i \\ & A_{i+1} \end{bmatrix} \begin{bmatrix} u_i' \\ u_i'' \end{bmatrix} = \begin{bmatrix} v_i' \\ v_i'' - E_i D_i^{-1} v_i' \end{bmatrix} \tag{4.6}$$

The forward elimination results in a system: $A_{i+1}u_i'' = v_i'' - E_i D_i v_i'$. This is the system on level $i+1$ and subjected to recursive processing. When $u_i''$ is retrieved, the backward substitution will be used for calculating $u_i'$:

$$u_i' = D_i^{-1}(v_i' - F_i u_i'') \tag{4.7}$$

In this way the preconditioned vector of $u_i$ can be retrieved and used for the preconditioning on higher level, if there is any.

On the innermost level, the preconditioned vector can be obtained by an inexact solver based on the original last level system and the vector to be preconditioned. Usually an incomplete factorization can be used, or an iterative solver with a low precision can be adopted.

The overall preconditioning process with multilevel preconditioning consists of 3 steps (suppose that the preconditioner has $l+1$ levels):

1. Transform $v$, i.e., the vector to be preconditioned, onto the innermost level to $v_l$, by recursive forward elimination by $l$ times;

2. Precondition $v_l$ on the innermost level, to retrieve the preconditioned vector $u_l$; this process usually involves an inexact solve with low precision;

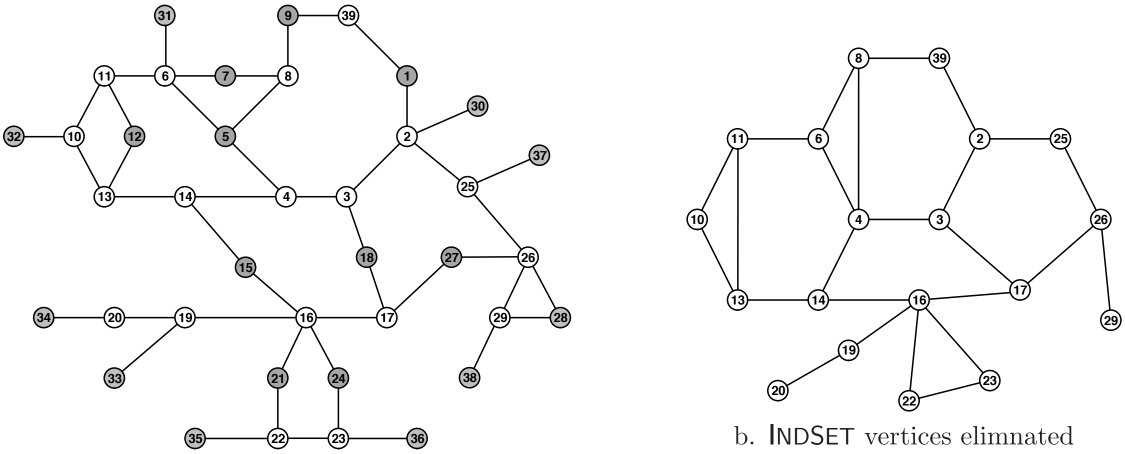
3. Transform $u_l$ back to the outmost level, by recursive backward substitution by $l$ times, to retrieve the preconditioned vector $u$.

The recursive forward elimination process will traverse each $D_i^{-1\prime}$ and $E_i$, and the backward substitution will traverse each $D_i^{-1\prime}$ and $F_i$. Hence the preconditioner has to keep record of the approximate to $D_i$'s (denoted $D_i^{-1\prime}$), $E_i$'s and $F_i$'s on each level. On the innermost level, the memory traversal scheme depends on the specific strategy adopted: (1) if an inexact direct solver is adopted, the incomplete factorization of the last level system will be traversed once only; (2) if an iterative solver is adopted, the incomplete factorization and the last level system itself will be traversed $m$ times, where $m$ is the iteration count required for convergence.

### 4.2.2 Multilevel Structure based on INDSETs

The effectiveness of multilevel methods is greatly affected by how the system size is reduced across the levels. Previous works on multilevel preconditioners mostly focus on problems with regular structures [27]. It is ideal if the size of $\mathcal{M}_{i+1}$ can be reduced exponentially as in multi-grid methods, so that a small last-level system can be attained which is easy to process. But for the problems with irregular structures such as Jacobian matrices of TDS, an algebraic structure has to be constructed.

In this thesis we consider a representative way to choose $\mathcal{M}_{i+1}$ out of $\mathcal{M}_i$ so that $D_i$ can be organized into an INDependent SET (INDSET) of small, dense supernodes [71, 1]. $\mathcal{M}_{i+1}$ will contain the vertices that are not in the INDSET. So to attain good size reduction from $\mathcal{M}_i$ to $\mathcal{M}_{i+1}$, we usually try to maximize the coverage of INDSET in $\mathcal{M}_i$. Fig.4.2a shows a maximal INDSET in grey on the network graph of IEEE 39 bus reference



b. INDSET vertices elimnated

a. Network of IEEE 39 bus reference system

Each bus corresponds to a vertex; each line corresponds to an edge in the graph. Bus numbers are shown.

Figure 4.2: INDSET on the network of IEEE 39 bus reference system.

system, which can not be augmented by adding any other vertex/bus without breaking the independency relationship. For the multilevel structure of previous section, if the vertices in the INDSET of $\mathcal{M}_i$ are eliminated from the graph, new edges should be added to the reduced system of $\mathcal{M}_{i+1}$, which is called the elimination graph in graph theory (see [70, 61]). The process of adding new edges corresponds to the introduction of non-zero elements (also called fill-in's) in the reduced system in the Schur complement operation mentioned in previous section. Fig.4.2b gives the resulting graph with the vertices in the INDSET in Fig.4.2a eliminated.

## 4.3 Multilevel Preconditioner for Jacobian Matrices based on INDSET

In this section we consider how to construct Multilevel Preconditioner for Jacobian Matrices. The preconditioner is constructed based on the sparsity pattern of the Jacobian matrix. From the analysis in Chapter 2 we know that: (1) in the Jacobian matrix, no extra fill-in occurs for the Schur complement of the dynamic part in the algebraic part of the matrix, and (2) the sparsity pattern of the algebraic part of the matrix holds tight relationship with the Power Network topology. Based on these two analysis, we construct a multilevel preconditioner for the whole Jacobian matrix.

As in Chapter 2, the Jacobian matrix is split into blocks as follows. $J_{1,1}$ corresponds to the dynamic equations, while $J_{2,2}$ to algebraic equations.

$$J = \begin{bmatrix} J_{1,1} & J_{1,2} \\ J_{2,1} & J_{2,2} \end{bmatrix} \tag{4.8}$$

By taking the Schur complement of $J_{1,1}$ in $J_{2,2}$, we have matrix $J'_{2,2} = J_{2,2} - J_{2,1} J_{1,1}^{-1} J_{1,2}$. By Theorem 1 in Chapter 2, we know that $J_{2,2}$ and $J'_{2,2}$ have the same sparsity pattern. $J_{2,2}$ has four parts, and each part corresponds the graph as defined by the network topology. Hence we can construct multilevel structure based on the static, non-changing network topology, and map the structure back to $J_{2,2}$ for the construction of a multilevel preconditioner.

A multiple level structure involves INDSET searching recursively on the network topology. On each level, a maximal INDSET is found and eliminated from the network graph. We also treat the graph as an elimination graph. When INDSET is removed from the graph, new edges are introduced to the graph. This process corresponds to the computation of Schur-complement. Fill-ins are introduced by elimination of the INDSET from the matrix. The recursive INDSET searching process represents a symbolic analysis process as in conventional Cholesky factorization [46]. The recursive INDSET searching process corresponds to that proposed in Section 4.2.1.

We map the recursive INDSET search result to $J'_{2,2}$. Note that $J'_{2,2}$ is divided into four sub-blocks, each with the same structure as the network topology. Without loss of generality, we assume an INDSET $S$ in the network topology composed of blocks. There are no connection among the blocks in $S$. Then we map each block in $S$ to a block in $J'_{2,2}$. Suppose that the block $B$ in $S$ contains buses with number $b_1$ to $b_l$ (block size of $l$). The mapping of $B$ to $J'_{2,2}$ is $B'$, which contains the following rows/columns in $J_{2,2}$ (note that the resulting block in $J'_{2,2}$ corresponds to $2l$ rows/columns in $J_{2,2}$):

$$b_1, \cdots, b_l, b_1 + n, \cdots, b_l + n \tag{4.9}$$

Each block in $S$ corresponds to a block in $J'_{2,2}$. Hence the whole set of $S$ can be mapped to $J'_{2,2}$, with each block of size $l$ in $S$ mapping to a set of rows/columns in $J'_{2,2}$ with size of $2l$. It is easy to prove that this set is an INDSET by the graph defined by $J'_{2,2}$. We denote the mapping of INDSET $S$ on the network topology in $J'_{2,2}$ as $S'$. Note that $S'$ will contain only blocks of even sizes: if $S$ composes of only single buses, $S'$ will only contain blocks with size of 2.

The structure of the multilevel preconditioner for $J$ based on INDSET's is as follows. The first level of the preconditioner for the Jacobian matrix is dedicated to dynamic components, i.e., $J_{1,1}$. The second level and beyond are constructed based on the INDSET searching results for the network topology. Hence the $i$-th level of the INDSET on the network corresponds to $(i+1)$-th level variables for the preconditioner. A block in INDSET on the topology with s buses with indices of $B_1, \ldots, B_s$ corresponds with $2s$ positions in Jacobian matrix with index of $d+B_1, d+B_2, \ldots, d+B_s, d+N+B_1, d+N+B_2, \ldots,$ and $d+N+B_s$ ($d$ is the size of $J_{1,1}$, $N$ is the bus count). As an example, the Jacobian matrix of IEEE-39 bus system for multilevel is shown in Fig.4.3, with the INDSET's permuted to the beginning at the beginning on each level. Note the independence of buses on each level of the multilevel structure.

Under the context of preconditioning, some values may be dropped in the process of computing the Schur complement. Since $D_i$'s are dense, we do not need to drop values in the diagonal blocks. Using the notation in Section 4.2.2, we calculate $D_i^{-1}$ precisely, but drop from each $E_i D_i^{-1}$ and $E_i D_i^{-1} F_i$ the values that are lower than a threshold $\delta$, compared with the 2-norm of the corresponding row. Notice that even with some values dropped, the INDSET searching result can still be applied it, since dropping of values in the matrix does not break the INDSET property. Note that due to the small size of the blocks, the computation associated with inverses of $D_i^{-1}$ is marginal. On the inner most level, we are left with a linear system with a much smaller size, for which we use an incomplete factorization as the preconditioning on the inner most level. Other preconditioner candidates are also possible, such as approximate inverse preconditioners.

The preconditioning operation for Jacobian matrix follows the routine in Section 4.2.2. Firstly, the right-hand side is transformed, combining the dynamic part into the algebraic part on the first level, and is recursively transformed into the algebraic-related multilevel,

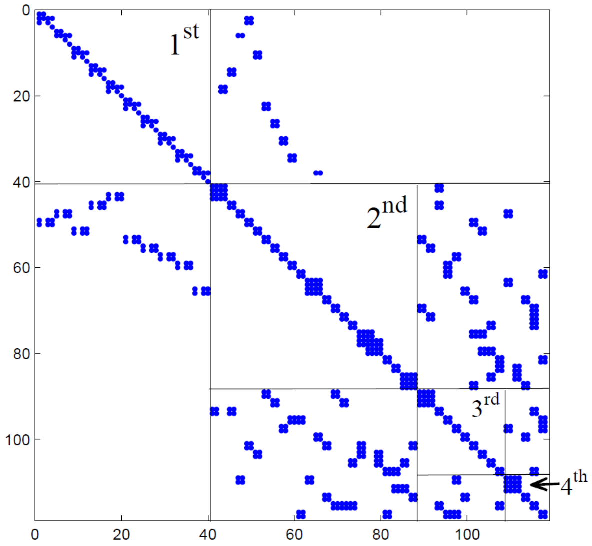


Figure 4.3: Multilevel structure for the Jacobian matrix of IEEE39 system.

until the inner most level. Secondly, on the inner most level, the preconditioning with the incomplete factorization is carried out. Lastly, the preconditioned vector is transformed recursively back through levels, to the outermost level, and the preconditioning operation is completed.

## 4.4 INDSET on Power Network – Searching Algorithms

In this section we present INDSET searching based on the power network topology. We start with the original power grid network, using it as the initial graph (denoted as $G(V, E)$) for INDSET searches. We also treat this graph as an elimination graph. When a maximal INDSET is found for $G$, the vertices in the INDSET are removed and fill-edges caused by eliminating them are added to the graph. We denote the vertices which are not in the INDSET as $V'$, and the set of new fill-edges $E'$. The resulting graph $G'$ is a subgraph of $G(V', E \bigcup E')$. We carry on recursive INDSET search based on $G'$, to retrieve an recursive algebraic multi-level structure. This structure is then mapped back to the Jacobian matrix as proposed in Section 4.2.2.

On each level of the recursive structure for multilevel, we try to maximize the size of INDSET, in order to achieve good system size reduction. For a general random network such as power networks, searching for the maximum INDSET is an NP-hard problem. Since the problem of finding the minimum reduced system size involves finding a series of maximum INDSET coverage, it is also NP-Hard. For the INDSET problem on a single level, heuristics are applied as in [71, 31, 54, 26] which all treat INDSET searching problem as a combinatorial optimization problem and try to maximize INDSET coverage and reach at a maximal solution. These algorithms are general purpose and none of them takes into account the context that the INDSET result is used for multilevel preconditioning, which implies: (1) a recursive INDSET structure is constructed, with each level corresponding to an INDSET searching problem; (2) INDSET result influences the preconditioner in computational complexity, precision, and memory usage; (3) optimal, i.e., maximum INDSET selection on a certain level does not necessarily generate a most effective reduction across levels. It is worth to mention that although in [71] the INDSET searching results are used for multilevel preconditioner, the algorithms themselves are general purposed. Given the context of multilevel preconditioning, it is important to devise good heuristics which can take into consideration of the issues above.

In this section, brief introduction to the heuristics are given in Section 4.4.1. We include 2 complementary heuristics based on the degrees of the vertices, namely Node Degree with Local Optimals (NDLO) and Vertex Cover (VC). Fill-in control and block-based INDSET are introduced in Section 4.4.2. Fill-in control provide better heuristics for lowering memory usage for tie-breaking scenarios. Using dense blocks of vertices instead of single vertices as components of the INDSET can result in more effective system size reduction for the multilevel structure.

### 4.4.1 Heuristics for INDSET Searching

In the following we introduce the simple heuristics for INDSET searching. NDLO(Node-Degree with Local Optimals) and VC(Vertex Cover) are classical INDSET searching algorithms. They are heuristics based on the degree of the vertices in the graph. NDLO starts from vertices with small degrees and put them in the INDSET. As a side-effect, the immediate neighbors of that vertex are excluded from the INDSET. "Local Optimal" in NDLO implies that the edges associated with the vertex and its neighbors are also removed from the graph hence the degrees of the vertices that are still in the graph are affected as a consequence.

VC tries INDSET searching from the opposite direction. It removes the vertex with the highest degree, along with the associated edges from it. The vertex removed is excluded from the INDSET. VC algorithm carries on until there is no edge left in the graph. Then the remaining vertices compose of an INDSET.

```
Input: Network Graph $G(V, E)$
Output: Independent Set of Nodes $S$
$S \leftarrow \phi$;
while $G$ not empty do
    $v \leftarrow$ least-connected vertices in $G$;
    $S \leftarrow S \bigcup \{v\}$;
    $N_v \leftarrow$ immediate neighbors of $v$;
    Delete $v$ and $N_v$ and related edges from $G$;
end
return $S$;
```

**Algorithm 4**: NDFLO

```
Input: Network Graph $G(V, E)$
Output: Independent Set of Nodes $S$
while $E \neq \emptyset$ do
    $v \leftarrow$ vertexes with largest degree in $G$;
    Delete $v$ from $G$ and delete edges of $v$ from $E$;
end
return remaining vertexes in $G$;
```

**Algorithm 5**: VC

NDLO and VC takes two different, somehow complementary approaches. NDLO chooses candidates with low degrees for IndSet, excluding their neighbors on the way. VC excludes vertices with higher degrees and stops when edges are all eliminated. NDLO and VC are greedy algorithms and the IndSet they return are maximal IndSet's. For Power Networks, the degree of the nodes are not high on the average. There are many tie-breaking scenarios that one has to deal with when choosing candidate vertices.

### 4.4.2 Fill-in Guidance and Large Block Sizes

Since every fill-edge on the graph potentially corresponds to an non-zero element in terms of matrix, reducing the total non-zero edge count in the multilevel structure on the network can reduce the non-zero element storage usage for the multilevel preconditioner. The high memory usage of the preconditioner translates directly to various problems, such as lower overall performance, etc. To reduce the total memory usage by the multilevel preconditioner, we design new heuristics by introducing fill-in control to NDLO and VC. We provide the fill-in control as a tie-breaking mechanism: when there are more than 1 candidates with the same of degree, choose the one which causes less fill-in's (for NDLO) or avoid the one which causes more fill-in's (for VC). We denote these two algorithms NDLOF(Node-Degree with Local Optimal and Fill-in control) and VCF(Vertex Cover with Fill-in control).

```
Input: Network Graph $G$
Output: Independent Set of Nodes $S$
$S \leftarrow \phi$;
while $G$ not empty do
    $V^* \leftarrow$ least-connected vertices in $G$;
    $v \leftarrow$ vertex in $V^*$ that introduces least fill-in's;
    $S \leftarrow S \bigcup \{v\}$;
    $N_v \leftarrow$ immediate neighbors of $v$;
    Delete $v$ and $N_v$ and related edges from $G$;
end
return $S$;
```

**Algorithm 6**: NDLOF

```
Input: Network Graph $G(V, E)$
Output: Independent Set of Nodes $S$
while $E \neq \emptyset$ do
    $V^* \leftarrow$ vertexes with largest degree in $G$;
    $v \leftarrow$ vertex in $V^*$ that introduces most fill-in's;
    Delete $v$ from $G$ and delete edges of $v$ from $E$;
end
return remaining vertexes in $G$;
```

**Algorithm 7**: VCF

Furthermore we propose the use of dense block based IndSet instead of single vertices based IndSet. In general, there are 3 benefits for using blocks based IndSet for multilevel preconditioners:

- For sparse problems like the Jacobian matrices of TDS, there is a large potential for achieving a large coverage of dense supernodes, so that a much larger system size reduction over levels can be achieved. From the formation of TDS Jacobian matrix in Chapter 2, we know that they are very sparse matrices, which can be exploited by a small, dense blocks-based multilevel scheme.

- The exact inverse of $D_i$ can be calculated in an exact way with little computational cost and memory usage, which is also beneficial to the overall preconditioner quality.
- Adopting blocks of vertices for IndSet can enhance both system size reduction and numerical stability. If only single-vertex based nodes are allowed, dense supernodes, especially cliques may pose difficulties for system size reduction; as is shown in [1, 85], allowing supernodes in IndSet selection can enlarge IndSet coverage and hence result in more effective system size reduction. Inverting a diagonal element may introduce numerical instability when its absolute value is very small; Diagonal blocks (which corresponds to an supernode of size over 1) can be introduced to avoid possible loss in precision and stability in the situation that the diagonal values are small ([1]).

The integration of block-based IndSet is straight forward for VC algorithm: instead of stopping VC when the edge set is empty, we now stop when all the sub-parts in the graph are small enough. We propose the algorithm of VCBF(Vertex Cover with Blocks and Fill-in control). Allowing blocks of vertices in IndSet's also introduce problems from two different aspects: (1) combinatorially it is infeasible to generate all candidates for IndSet, especially when the block size is large; (2) allowing a very sparse block will introduce much memory and computational overhead in inverting it, which could be unjustifiable in general. In the following section we evaluate the effect of the various IndSet algorithms. For VCBF, we only test VCBF(2). With VCBF(2), blocks are dense by themselves, and the evaluation for the VCBF stopping criteria is simple: degree of vertices be not higher than 1.

```
Input: Network Graph $G$
       Block size threshold $bsize$
Output: Independent Set of Nodes $S$
$S \leftarrow \phi$;
$V_L \leftarrow$ vertex within blocks smaller than $bsize$;
$S \leftarrow S \bigcup V_L$ ;
for vertex block $G'_i$ that's larger than bsize in $G$ do
    $V^* \leftarrow$ vertexes with largest degree in $G$ ;
    $v \leftarrow$ vertex in $V^*$ that introduces most fill-in's ;
    Delete vertex of largest degree from $G'_i$ ;
    $S \leftarrow S \bigcup$ VCBF($G'_i, bsize$) ;
end
return $S$;
```

**Algorithm 8**: VCBF

## 4.5 Experiments and Results

In this section we show the experiments and results of IndSet searching algorithms on power networks and the characteristics of the multilevel preconditioners. Our experiments are based on large, operational power grids. The power systems used for IndSet searching experiments are listed in Tab.4.1. We also include a simplified model of western United

| System Name | Bus Count | Line Count | Generator Count | Matrix Size | $nnz$ |
|---|---|---|---|---|---|
| 2K [a] | 1165 | 1408 | 216 | 3194 | 21108 |
| 10188 [b] | 10188 | 11905 | 1072 | 25308 | 160877 |
| West-US [c] | 5330 | 8271 | N/A | | |

---

[a] Detailed model of He Bei province, China; 4-order generator model without excitor or governer

[b] Highly simplified model of Chinese National Grid; 5-order generator model without excitor or governer

[c] Simplified Model for western United States grid; no information of generators available

Table 4.1: Power Networks

States power grid, to demonstrate the effect of INDSET searching algorithms, but due to the lack of the complete model such as generators and loads, there is no simulation case or Jacobian matrices available for it. For the solution based on the multilevel preconditioners, we use BILUM [1] as the standard platform for comparison.

### 4.5.1 INDSET Searching on Power Network

We implemented aforementioned INDSET searching algorithms, including NDLOF, VCF, VCB(2), and VCBF(2). VCB is the counterpart of VCBF without fill-in control. VCB(2) allows block sizes of 2. NDLO and VC are also implemented and tested as references. BILUM's built-in greedy algorithm for INDSET searching was skipped due to its poor performance.

The 6 algorithms are divided into 3 groups: NDLO and NDLOF, VC and VCF, VCB(2) and VCBF(2) (i.e., organized in a fill-in control disabled/enabled way), and tested on the systems in Tab.4.1. Fig.4.4, Fig.4.5 and Fig.4.6 show the performance of different INDSET searching algorithms on 2K, 10188, and West-US system, respectively as described in Table 4.1.

We compare different INDSET searching algorithms by two criteria: (1) system size reduction, i.e., the size of the system on certain level, and (2) total number of edges in the graph. As is indicated in Section 4.2.2, on each level of the multilevel structure, when the INDSET is eliminated from the graph, potentially fill-edges have to be added for the next level graph. To distinguish them from traditional power lines in terms of Power Network, we denote these edges as "virtual lines". We use "Total Line" to include the original power lines in the original network and the "virtual lines" introduced on each level. The number of "Total Lines" sets an upper bound for the memory usage of the multilevel structure of the preconditioner.

Among NDLO, VC and VCB(2), VCB(2) achieves the most effective system size reduction with lowest total line count. For 2K, on level 8, VCB(2) obtains a reduced system size that is 1/5 of that obtained by VC or NDLO; for 10188 and West-US, this ratio is about 1/3 and 1/2, respectively. VC performs slightly better than NDLO with marginal improvements.

For NDLO, VC, VCB(2) and their fill-in control enabled counterparts, the ones with fill-in guidance perform better by achieving a lower last-level size at the same level. For 2K the last-level size obtained by NDLOF is 30% lower than that by NDLO. The gap between last-level sizes obtained by algorithms with fill-in control and those without it

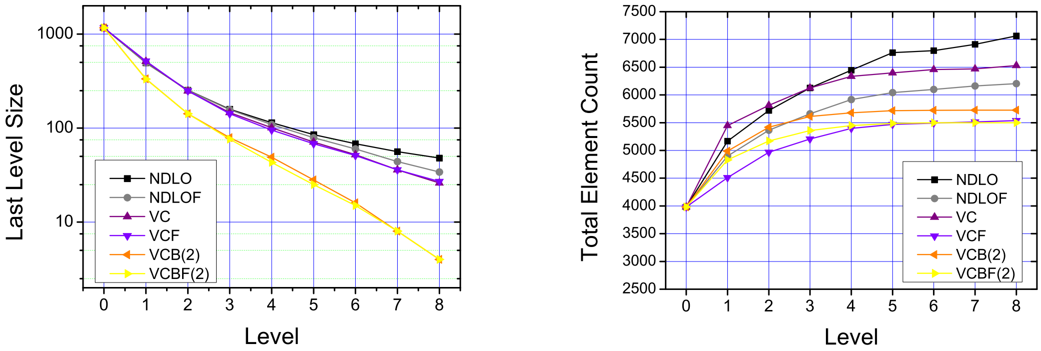


Figure 4.4: Effects of INDSET searching algorithms on 2K system.

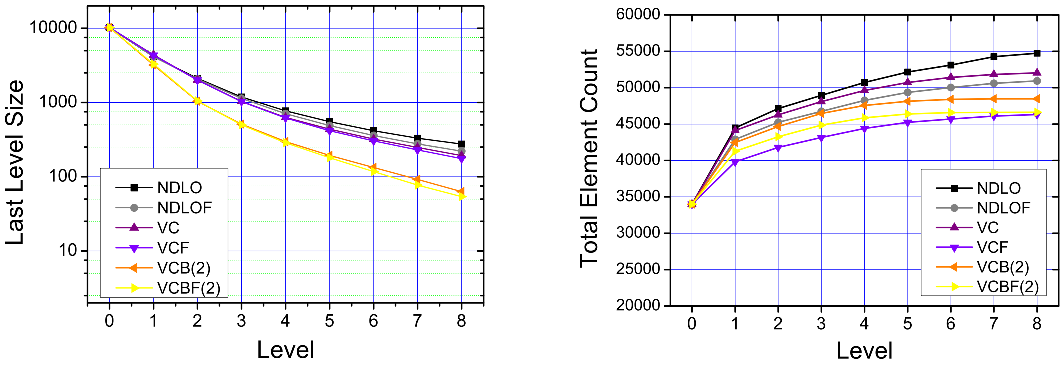


Figure 4.5: Effects of INDSET searching algorithms on 10188 system.

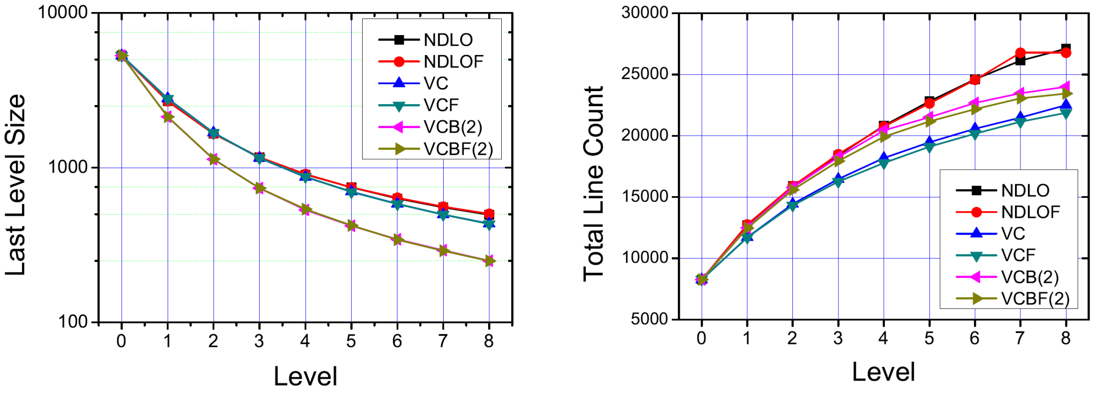


Figure 4.6: Effects of INDSET searching algorithms on West-US system.

| 2K | Construction Time (ms) | Solving Time (ms) | Iteration Count | *nnz* |
|---|---|---|---|---|
| ILU-0 | 4 | 276 | 117 | 21109 |
| ILU($10^{-4}$,10) | 40 | 108 | 35 | 39903 |
| ILU($10^{-5}$,50) | 416 | 68 | 11 | 149262 |
| ILU($10^{-6}$,100) | 1428 | 68 | 8 | 280399 |
| VCBF(2)($10^{-4}$,10) | 36 | 8 | 3 | 27757 |
| VCBF(2)($10^{-5}$,20) | 36 | 4 | 2 | 27779 |
| NDLO($10^{-4}$,10) | 52 | 12 | 4 | 34451 |
| NDLOF($10^{-4}$,10) | 44 | 12 | 4 | 31533 |

a. Result for 2K system.

| 10188 | Construction Time (ms) | Solving Time (ms) | Iteration Count | *nnz* |
|---|---|---|---|---|
| ILU-0 | 36 | No Convergence | | 160877 |
| ILU($10^{-4}$,20) | 356 | 1928 | 58 | 378845 |
| ILU($10^{-5}$,50) | 1748 | 720 | 17 | 796366 |
| ILU($10^{-6}$,100) | 6900 | 544 | 10 | 1465791 |
| VCBF(2)($10^{-4}$,10) | 320 | 688 | 26 | 212543 |
| VCBF(2)($10^{-5}$,20) | 324 | 244 | 9 | 216262 |
| NDLO($10^{-4}$,10) | 428 | 832 | 27 | 246577 |
| NDLOF($10^{-4}$,10) | 404 | 724 | 25 | 232396 |

b. Result for 10188 system.

Table 4.2: Preconditioner's construction and solving time with iteration and memory consumption information.

tends to widen across levels, which is due to the reason that adopting fill-in strategy for choosing candidates properly could benefit the sparsity of reduced systems hence resulting in more effective independent set searching in later levels. Results are consistent with the priority between heuristics by showing a marginal lead in reduction achieved by fill-in enabled algorithms across three algorithms pairs.

Notable decrease in memory usage is observed across the 3 algorithm pairs. For 2K the reduction from VC to VCF is 15%, and for 10188 10%. VCB(2) and VCBF(2) algorithm pair experienced the least reduction for both 2K and 10188: 4% and 5%, respectively. West-US has experienced lower enhancements.

Another thing to note is that moving from VC to VCB(2) (or VCF to VCBF(2)) generally results in increase in the total line count and decrease in the last level size.This implies that using large block sizes will generally introduce more fill-in's at the sake of smaller last level size. With smaller last level size, the preconditioner for the last level system could occupy less space. Yet this trade-off point of the memory usage within the multilevel structure and the last level system is unclear.

### 4.5.2 Characteristics of Multilevel Preconditioners

The convergence behavior and related information of the GMRES iterations using different preconditioners are shown in Tab.4.2 (all multilevel preconditioners have 9 levels). The timing is measured with BILUM and SPARSKIT [11]. 2K system is easy to solve: GMRES with ILU-0 can achieve convergence in 117 iterations. But for faster convergence, ILU with low dropping threshold parameters has to be adopted, and large fill-in's appear to be the key factor. For 2K ILU($10^{-6}$,100), which uses up to 10 times memory space of that used by VCBF(2), still cannot achieve comparable convergence speed. For 10188, GMRES with ILU-0 fails to give converge. When using ILU preconditioners, there is a strong correlation between convergence speed and the amount of fill-in's, while a similar correlation exists for multilevel preconditioners between the speed and the value-dropping parameter. Fill-in count threshold does not seem to be a severe constraint for convergence for multilevel preconditioners. The reasons are: (1) in Schur complements not many fill-in's are introduced when using these INDSET algorithms, and (2) the last-level system of the multilevel preconditioner is small, so for a moderate fill-in count upper bound, the convergence property is mainly decided by the value-dropping threshold. The fact that ILU preconditioners cannot achieve similar convergence speed as their multilevel counterparts within practical/comparable memory budget gives us further justification to adopting multilevel preconditioners for solving large Jacobian matrix based linear systems.

Given a set of similar parameters, e.g., the same value-dropping threshold with similar memory usage, CPU time required by multilevel preconditioners are about 1/10 that of non-multilevel ones for the 2K and about 1/3 for the 10188. Construction time for ILU preconditioners increases drastically when lowering value-dropping threshold. Besides, per-iteration time for preconditioners (both ILU and multilevel ones) is generally proportional to the amount of memory used by the preconditioner. For 10188, since NDLO and NDLOF achieves almost the same last-level size, the performance advantage of NDLOF over NDLO on a per-iteration basis is due to the difference in preconditioner sizes: the preconditioner size of NDLOF is lower than that of NDLO by 5.7%, while it performs about 6% faster per iteration. This shows that reducing the size of preconditioner can reduce the overall computation time, if the loss in preconditioner quality (if there is any) could be amortized.

Fig.4.7 shows the condition number comparison between various preconditioners for 2K. It shows a clear trade-off between the quality of the ILU preconditioner and the memory it uses. Comparatively, multilevel preconditioner achieves a condition number of the preconditioned matrix of similar magnitude with much less memory consumption. It also shows the difference in fill-in amount between NDLO and NDLOF clearly affects the total number of non-zeros in the preconditioner but not the condition number. Although condition number cannot serve as a direct evaluation for convergence, it does show the clustering of the eigenvalues of the preconditioned system to (1, 0) on the complex plane, especially when the condition number is close to 1. Fig.4.8 shows the details of trade-off between convergence iteration count and the preconditioner's memory usage for 10188. There is a clear trade-off between convergence speed and the size of the preconditioner for ILU. The convergence is very slow for iterations using ILU with fill-in count upper bound below 15 (per row for $L$ and $U$). But for all the 3 multilevel preconditioners, only 10 fill-in's per row are allowed in ILU. Lowering the value-dropping threshold will introduce extra fill-in's by a small percentage, but can boost the preconditioner quality significantly (the iteration count drops from 26 to 6); also considering the memory used by the preconditioner remains within 1.6 times that of the original matrix, we conclude that multilevel preconditioners are more memory-efficient compared with the non-multilevel counterparts. Furthermore, for NDLO and NDLOF, we can see virtually no difference in convergence speed, but a reduction of 14181 elements in the memory space used by the

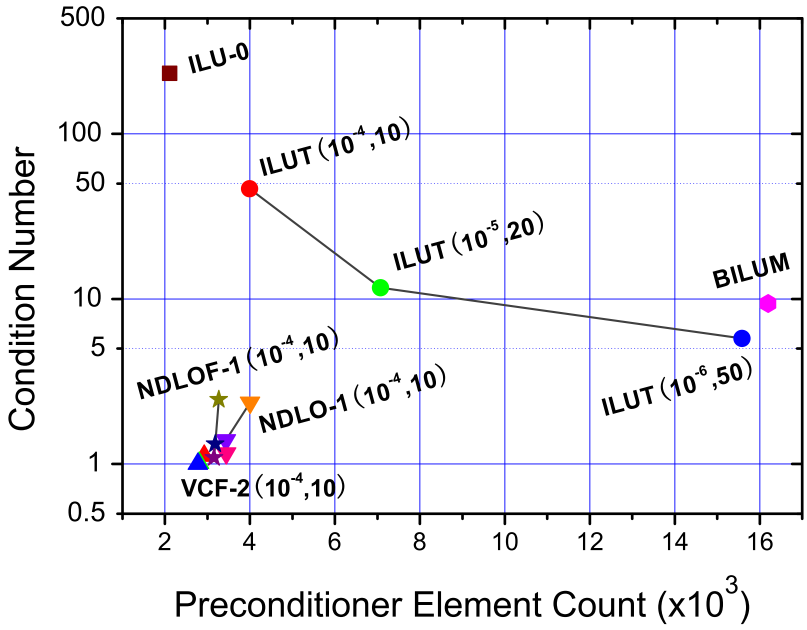


Figure 4.7: Preconditioner memory usage and condition number relationship for 2K system.

preconditioner, which is about 5.8% of the total size of the preconditioner. The marker with name BILUM corresponds to the result obtained by the built-in INDSET search algorithm of BILUM.

## 4.6 Summary

In this chapter we applied multilevel technique to the preconditioning of Jacobian matrices in TDS. Specially we use INDSET based algebraic multilevel structure, based on the structure of the Jacobian matrices. Based on the sparsity pattern of the matrix and its direct relationship with the network topology, we carry out INDSET searching based on the power network and map it to the construction of the multilevel preconditioner. For the INDSET searching, Fill-in control and block sizes beyond 1 both yield good system size reduction and memory usage. Compared with traditional ILU preconditioners, multilevel preconditioners based on INDSET have much better memory efficiency and preconditioner quality.

Multilevel preconditioners are also famous for the parallelism for preconditioning operations. Chapter 8 explores the parallelization of multilevel preconditioners based on INDSET and approximate inverse preconditioners for the last-level system.

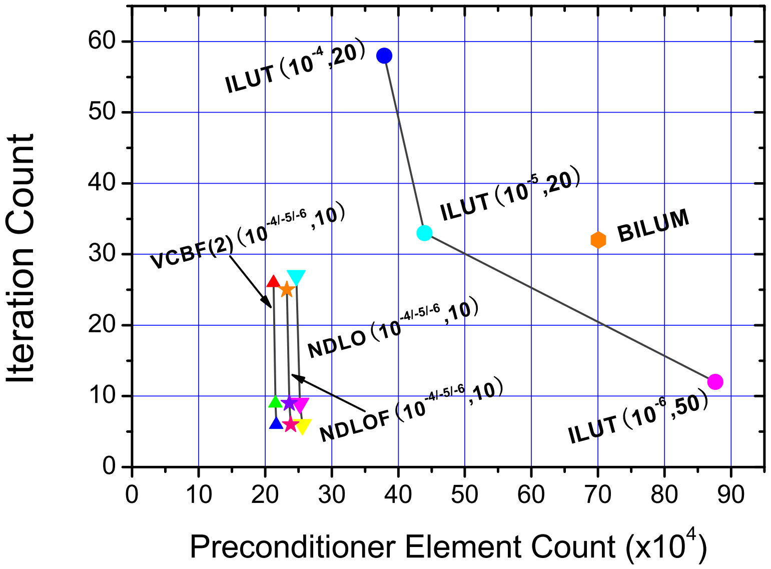


Figure 4.8: Preconditioner memory usage and iteration count relationship for 10188 system.

# Chapter 5

# Multi-Step Technique – Preconditioner Updates for TDS

## 5.1 Introduction

This chapter and the next chapter discusses applying multi-step techniques to Time Domain Simulation (TDS) of Power Grids. TDS requires the solution of a sequence of linear systems, which occur both in various Newton steps in the same time step and at multiple time steps. During this process, reusing information generated from the solving process would speedup the overall TDS process. We consider the application of preconditioner updating technology and its application in TDS. Applying preconditioner updates requires Jacobian matrices which does not undergo significant change from step to step. Most popular are additive changes to the Jacobian, in which works such as [77] can be applied.

To achieve this, firstly we prove that the formulation of TDS matrices can be transformed into the form of $Y + \Delta_Y$. $Y$ is a transformed matrix of $\dot{Y}$, i.e., the original admittance matrix of the power network and $\Delta_Y$ is a block diagonal matrix. The Jacobian-based linear system is in turn transformed as a linear system based on $Y + \Delta_Y$. This formulation gives the additive change to the Jacobian, and the applicability of preconditioner updating algorithms. It also enables more computationally feasible deflation by Krylov solvers, which provides another dimension of speedups for multi-step technique. Deflation-based techniques covered in the next chapter.

This chapter is organized as follows. Section 5.2 reformulate the conventional TDS problem into linear systems of form $Y + \Delta_Y$. Section 5.3 briefs on Preconditioner Updates, analyzes the specific case of preconditioner updates in TDS, and proposes new preconditioner updating algorithms for TDS. Section 5.4 includes experiment results and analysis of preconditioner updates in TDS. Section 5.5 concludes the chapter.

## 5.2 Formulation of $Y + \Delta_Y$ for TDS Matrices

In TDS, at each Newton step at certain time steps, a linear system is solved based on the Jacobian matrix $J$. We divide $J$ into 4 sub-blocks, in which $J_{1,1}$ corresponds to dynamic part of the power system and $J_{2,2}$ corresponds to the algebraic part.

$$Jx = b \tag{5.1}$$

$$\begin{bmatrix} J_{1,1} & J_{1,2} \\ J_{2,1} & J_{2,2} \end{bmatrix} \times \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} \tag{5.2}$$

In further discussion, we follow and extend the notations used for Power Grid models as in Chapter 2:

$\dot{V}$ : the column vector of bus voltages, i.e., $\dot{V}_i$'s. $\dot{V}_i$ is the voltage at bus $i$, for $1 \leq i \leq n$; $n$ is the total bus count.

$\dot{\mathbb{V}}$ : formulation of $\dot{V}_i$'s into a diagonal matrix, with $\dot{\mathbb{V}}(i,i) = \dot{V}_i$.

$\dot{I}$ : the column vector of bus currents, i.e., $\dot{I}_i$'s. $\dot{I}_i$ is the injected current on bus $i$, for $1 \leq i \leq n$.

$\dot{\mathbb{I}}$ : formulation of $\dot{I}_i$'s into a diagonal matrix, with $\dot{\mathbb{I}}(i,i) = \dot{I}_i$.

$v$ : the column vector of bus voltage amplitudes, i.e., $\dot{v}_i$'s. $\dot{v}_i$ is the amplitude of the voltage on bus $i$, for $1 \leq i \leq n$.

$\theta$ : the column vector of bus voltage angles, i.e., $\dot{\theta}_i$'s. $\dot{\theta}_i$ is the angle of the voltage on bus $i$, for $1 \leq i \leq n$.

$\dot{\mathbb{A}}$ : formulation of the angle of the bus voltages into a diagonal matrix, with $\dot{\mathbb{A}}(i,i) = e^{j\theta_i}$.

We use a dot to denote the value/matrix to be a complex variable, as in $\dot{V}$, $\dot{\mathbb{V}}$, etc. Variables without a dot such as $v$, $\theta$ are real variables. Furthermore, we use a bar over a complex variable to denote the complex conjugate of the variable. If the variable is a matrix rather than a scalar value, then the resulting matrix has every element as the complex conjugate of the corresponding element in the original matrix. Also by definition it holds that: $\dot{V}_i = v_i e^{j\theta_i}$ and $\bar{\dot{V}}_i = v_i e^{-j\theta_i}$. We formalize the definition as follows:

$$\dot{V} = \begin{bmatrix} \dot{V}_1 \\ \dot{V}_2 \\ \vdots \\ \dot{V}_n \end{bmatrix}, \quad \dot{\mathbb{V}} = \begin{bmatrix} \dot{V}_1 & & & \\ & \dot{V}_2 & & \\ & & \ddots & \\ & & & \dot{V}_n \end{bmatrix} \tag{5.3}$$

$$\dot{I} = \begin{bmatrix} \dot{I}_1 \\ \dot{I}_2 \\ \vdots \\ \dot{I}_n \end{bmatrix}, \quad \dot{\mathbb{I}} = \begin{bmatrix} \dot{I}_1 & & & \\ & \dot{I}_2 & & \\ & & \ddots & \\ & & & \dot{I}_n \end{bmatrix} \tag{5.4}$$

$$v = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}, \quad \mathbb{V} = \begin{bmatrix} v_1 & & & \\ & v_2 & & \\ & & \ddots & \\ & & & v_n \end{bmatrix} \tag{5.5}$$

$$\theta = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix}, \quad \dot{\mathbb{A}} = \begin{bmatrix} e^{j\theta_1} & & & \\ & e^{j\theta_2} & & \\ & & \ddots & \\ & & & e^{j\theta_n} \end{bmatrix} \tag{5.6}$$

Note that $\dot{\mathbb{V}}$, $\dot{\mathbb{I}}$, $\dot{\mathbb{A}}$ and $\mathbb{V}$ are scaling matrices. Hence we have: $\dot{\mathbb{V}} = \mathbb{V} \times \dot{\mathbb{A}} = \dot{\mathbb{A}} \times \mathbb{V}$. Also for the complex conjugate matrices, we have $\bar{\dot{\mathbb{V}}} = \mathbb{V} \times \bar{\dot{\mathbb{A}}} = \bar{\dot{\mathbb{A}}} \times \mathbb{V}$.

By circuits' law, injection of currents $\dot{I}$ is defined by the voltage and the admittance between buses. It can be calculated as ($\dot{Y}_{i,k}$ denotes the admittance between bus $i$ and bus $k$):

$$\begin{aligned}
\dot{I} &= \dot{Y}\dot{V} \qquad (5.7)\\
&= \begin{bmatrix} \dot{Y}_{1,1} & \dot{Y}_{1,2} & \dots & \dot{Y}_{1,n} \\ \dot{Y}_{2,1} & \dot{Y}_{2,2} & \dots & \dot{Y}_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \dot{Y}_{1,1} & \dot{Y}_{1,2} & \dots & \dot{Y}_{1,n} \end{bmatrix} \times \begin{bmatrix} \dot{V}_1 \\ \dot{V}_2 \\ \vdots \\ \dot{V}_n \end{bmatrix} \qquad (5.8)
\end{aligned}$$

And the power injection at bus $i$:

$$\begin{aligned}
\dot{P}_i &= \bar{\dot{V}}_i \cdot \dot{I}_i \qquad (5.9)\\
&= (v_i e^{-j\theta_i}) \sum_{1 \le s \le n} \dot{Y}_{i,s}(v_s e^{j\theta_s}) \qquad (5.10)
\end{aligned}$$

Then the derivatives for power at bus $i$ are:

$$\begin{aligned}
\frac{\partial \dot{P}_i}{\partial v_k} &= \begin{cases} (v_i e^{-j\theta_i})\dot{Y}_{i,k}(e^{j\theta_k}), & i \neq k \\ (v_i e^{-j\theta_i})\dot{Y}_{i,i}(e^{j\theta_i}) + (e^{-j\theta_i}) \sum_{1 \le s \le n} \dot{Y}_{i,s}(v_s e^{j\theta_s}), & i = k \end{cases} \qquad (5.11)\\
\frac{\partial \dot{P}_i}{\partial \theta_k} &= \begin{cases} j(v_i e^{-j\theta_i})\dot{Y}_{i,k}(v_k e^{j\theta_k}), & i \neq k \\ j(v_i e^{-j\theta_i})\dot{Y}_{i,i}(v_k e^{j\theta_i}) - j(v_i e^{-j\theta_i}) \sum_{1 \le s \le n} \dot{Y}_{i,s}(v_s e^{j\theta_s}), & i = k \end{cases} \qquad (5.12)\\
& \qquad (5.13)
\end{aligned}$$

We can re-write Eqs.5.11 and Eqs.5.12 in matrix form as:

$$\begin{aligned}
\frac{\partial \dot{P}}{\partial v} &= \bar{\dot{\mathbb{V}}} \times \dot{Y} \times \dot{\mathbb{A}} + \bar{\dot{\mathbb{A}}} \times \dot{\mathbb{I}} \qquad (5.14)\\
\frac{\partial \dot{P}}{\partial \theta} &= j\Big(\bar{\dot{\mathbb{V}}} \times \dot{Y} \times \dot{\mathbb{V}} - \bar{\dot{\mathbb{V}}} \times \dot{\mathbb{I}}\Big) \qquad (5.15)
\end{aligned}$$

Further, we have:

$$\frac{\partial \dot{P}}{\partial v} = \bar{\dot{\mathbb{V}}} \times \dot{Y} \times \dot{\mathbb{A}} + \bar{\dot{\mathbb{A}}} \times \dot{\mathbb{I}} \tag{5.16}$$

$$= \mathbb{V} \times \bar{\dot{\mathbb{A}}} \times \left( \dot{Y} + \begin{bmatrix} \frac{\dot{I}_1}{\dot{V}_1} & & & \\ & \frac{\dot{I}_2}{\dot{V}_2} & & \\ & & \ddots & \\ & & & \frac{\dot{I}_n}{\dot{V}_n} \end{bmatrix} \right) \times \dot{\mathbb{A}} \tag{5.17}$$

$$= \bar{\dot{\mathbb{V}}} \times \left( \dot{Y} + \begin{bmatrix} \frac{\dot{I}_1}{\dot{V}_1} & & & \\ & \frac{\dot{I}_2}{\dot{V}_2} & & \\ & & \ddots & \\ & & & \frac{\dot{I}_n}{\dot{V}_n} \end{bmatrix} \right) \times \dot{\mathbb{A}} \tag{5.18}$$

For $\frac{\partial \dot{P}}{\partial \theta}$, we have:

$$\frac{\partial \dot{P}}{\partial \theta} = j \left( \bar{\dot{\mathbb{V}}} \times \dot{Y} \times \dot{\mathbb{V}} - \bar{\dot{\mathbb{V}}} \times \dot{\mathbb{A}} \right) \tag{5.19}$$

$$= j\bar{\dot{\mathbb{V}}} \left( \dot{Y} + \begin{bmatrix} \frac{\dot{I}_1}{\dot{V}_1} & & & \\ & \frac{\dot{I}_2}{\dot{V}_2} & & \\ & & \ddots & \\ & & & \frac{\dot{I}_n}{\dot{V}_n} \end{bmatrix} \right) \times \dot{\mathbb{V}} \tag{5.20}$$

We denote:

$$\Delta_{\dot{Y}} = \begin{bmatrix} \frac{\dot{I}_1}{\dot{V}_1} & & & \\ & \frac{\dot{I}_2}{\dot{V}_2} & & \\ & & \ddots & \\ & & & \frac{\dot{I}_n}{\dot{V}_n} \end{bmatrix} \tag{5.21}$$

We denote $\dot{P}$ the complex power: $\dot{P} = P + jQ$, where $j$ is the complex root of -1, (i.e., unit of imaginary numbers). By definition of derivatives in complex functions and separation of real and imaginary parts, we have:

$$\begin{cases} \frac{\partial \dot{P}}{\partial v} = \frac{\partial P}{\partial v} - j\frac{\partial Q}{\partial v} \\ \frac{\partial \dot{P}}{\partial \theta} = \frac{\partial P}{\partial \theta} - j\frac{\partial Q}{\partial \theta} \end{cases} \tag{5.22}$$

The parts in 5.22 represent sub-blocks in $J_{2,2}$:

$$J_{2,2} = \begin{bmatrix} \frac{\partial P}{\partial v} & \frac{\partial P}{\partial \theta} \\ \frac{\partial Q}{\partial v} & \frac{\partial Q}{\partial \theta} \end{bmatrix} \tag{5.23}$$

Since the part corresponding to the network parts in the Jacobian matrix in Chapter 2 is as in Eqs.5.23, Eqs.5.24 shows $J_{2,2}$ with the Schur complement of $J_{1,1}$ as $\Delta$, which is formed by sub-parts of $\Delta_1$, $\Delta_2$, $\Delta_3$, and $\Delta_4$.

$$J_{2,2} + \Delta = \begin{bmatrix} \frac{\partial P}{\partial v} + \Delta_1 & \frac{\partial P}{\partial \theta} + \Delta_3 \\ \frac{\partial Q}{\partial v} + \Delta_2 & \frac{\partial Q}{\partial \theta} + \Delta_4 \end{bmatrix} \tag{5.24}$$

$$\frac{\partial P}{\partial \theta} = Real\left(\frac{\partial \dot{P}}{\partial \theta}\right) \tag{5.25}$$

$$= Real\left(-j\mathbb{V} \times \bar{\dot{\mathbb{A}}} \times (\dot{Y} + \Delta_{\dot{Y}}) \times \dot{\mathbb{V}}\right) \tag{5.26}$$

$$= Imag\left(\mathbb{V} \times \bar{\dot{\mathbb{A}}} \times (\dot{Y} + \Delta_{\dot{Y}}) \times \dot{\mathbb{V}}\right) \tag{5.27}$$

$$\frac{\partial P}{\partial v} = Real\left(\frac{\partial \dot{P}}{\partial v}\right) \tag{5.28}$$

$$= Real\left(\mathbb{V} \times \bar{\dot{\mathbb{A}}} \times (\dot{Y} + \Delta_{\dot{Y}}) \times \dot{\mathbb{A}}\right) \tag{5.29}$$

$$\frac{\partial Q}{\partial \theta} = -Imag\left(\frac{\partial \dot{P}}{\partial \theta}\right) \tag{5.30}$$

$$= -Imag\left(-j\mathbb{V} \times \bar{\dot{\mathbb{A}}} \times (\dot{Y} + \Delta_{\dot{Y}}) \times \dot{\mathbb{V}}\right) \tag{5.31}$$

$$= Real\left(\mathbb{V} \times \bar{\dot{\mathbb{A}}} \times (\dot{Y} + \Delta_{\dot{Y}}) \times \dot{\mathbb{V}}\right) \tag{5.32}$$

$$\frac{\partial Q}{\partial v} = -Imag\left(\frac{\partial \dot{P}}{\partial v}\right) \tag{5.33}$$

$$= -Imag\left(\mathbb{V} \times \bar{\dot{\mathbb{A}}} \times (\dot{Y} + \Delta_{\dot{Y}}) \times \dot{\mathbb{A}}\right) \tag{5.34}$$

$$\tag{5.35}$$

Note Theorem 1 in Chapter 2 for the sparsity pattern the Schur complement of $J_{11}$ in $J_{12}$, stating that sub-parts of $\Delta$ are diagonal matrices. Specifically, denote that the Schur complement in $\frac{\partial P}{\partial v}$, $\frac{\partial P}{\partial \theta}$, $\frac{\partial Q}{\partial v}$ and $\frac{\partial Q}{\partial \theta}$ are $\Delta_1$, $\Delta_2$, $\Delta_3$ and $\Delta_4$, respectively. Then these matrices only have non-zero elements on the main diagonals. Subsequently we can carry out the following transformations.

$$(\frac{\partial P}{\partial v} + \Delta_1) - j(\frac{\partial Q}{\partial v} + \Delta_3) = (\frac{\partial P}{\partial v} - j\frac{\partial Q}{\partial v}) + (\Delta_1 - j\Delta_3) \tag{5.36}$$

$$\bar{\dot{\mathbb{V}}}^{-1}\left((\frac{\partial P}{\partial v} - j\frac{\partial Q}{\partial v}) + (\Delta_1 - j\Delta_3)\right)\dot{\mathbb{A}}^{-1} = (\dot{Y} + \Delta_{\dot{Y}}) + \bar{\dot{\mathbb{V}}}^{-1}(\Delta_1 - j\Delta_3)\dot{\mathbb{A}}^{-1} \tag{5.37}$$

Note that $(\Delta_1 - j\Delta_3)$ is a diagonal matrix. Together with $\bar{\dot{\mathbb{V}}}^{-1}$ and $\dot{\mathbb{A}}^{-1}$ as diagonal scaling matrices, it is immediate that $\bar{\dot{\mathbb{V}}}^{-1}(\Delta_1 - j\Delta_3)\dot{\mathbb{A}}^{-1}$ is a diagonal matrix. We denote it as $\Delta_{\dot{Y},P}$.

Similarly, we have:

$$(\frac{\partial P}{\partial \theta} + \Delta_2) - j(\frac{\partial Q}{\partial \theta} + \Delta_4) = (\frac{\partial P}{\partial \theta} - j\frac{\partial Q}{\partial \theta}) + (\Delta_2 - j\Delta_4) \tag{5.38}$$

$$(j\bar{\dot{\mathbb{V}}})^{-1}\Big((\frac{\partial P}{\partial \theta} - j\frac{\partial Q}{\partial \theta}) + (\Delta_2 - j\Delta_4)\Big)\dot{\mathbb{V}}^{-1} = (\dot{Y} + \Delta_{\dot{Y}}) + (j\bar{\dot{\mathbb{V}}})^{-1}(\Delta_2 - j\Delta_4)\dot{\mathbb{V}}^{-1} \tag{5.39}$$

We denote $\Delta_{\dot{Y},Q}$ as $(j\bar{\dot{\mathbb{V}}})^{-1}(\Delta_2 - j\Delta_4)\dot{\mathbb{V}}^{-1}$. It is clear that like $\Delta_{\dot{Y}}$ and $\Delta_{\dot{Y},P}$, $\Delta_{\dot{Y},Q}$ is also a complex diagonal matrix. We specify the solution part and corresponding parts in the right-hand side as $\delta_v$, $\delta_\theta$, $\delta'_P$ and $\delta'_Q$, respectively. Then the linear system can be viewed as in Eqs.5.40. Note that Eqs.5.40 corresponds to the first level system in the multi-level structure presented in Chapter 4.

$$\begin{bmatrix} \frac{\partial P}{\partial v} + \Delta_1 & \frac{\partial P}{\partial \theta} + \Delta_3 \\ \frac{\partial Q}{\partial v} + \Delta_2 & \frac{\partial Q}{\partial \theta} + \Delta_4 \end{bmatrix} \times \begin{bmatrix} \delta_v \\ \delta_\theta \end{bmatrix} = \begin{bmatrix} \delta'_P \\ \delta'_Q \end{bmatrix} \tag{5.40}$$

$$\left(\begin{bmatrix} \frac{\partial P}{\partial v} + \Delta_1 & \frac{\partial P}{\partial \theta} + \Delta_3 \\ \frac{\partial Q}{\partial v} + \Delta_2 & \frac{\partial Q}{\partial \theta} + \Delta_4 \end{bmatrix} \times \begin{bmatrix} I & \\ & \mathbb{V}^{-1} \end{bmatrix}\right) \times \left(\begin{bmatrix} I & \\ & \mathbb{V} \end{bmatrix} \times \begin{bmatrix} \delta_v \\ \delta_\theta \end{bmatrix}\right) = \begin{bmatrix} \delta'_P \\ \delta'_Q \end{bmatrix} \tag{5.41}$$

$$\begin{bmatrix} \frac{\partial P}{\partial v} + \Delta_1 & (\frac{\partial P}{\partial \theta} + \Delta_3) \times \mathbb{V}^{-1} \\ \frac{\partial Q}{\partial v} + \Delta_2 & (\frac{\partial Q}{\partial \theta} + \Delta_4) \times \mathbb{V}^{-1} \end{bmatrix} \times \begin{bmatrix} \delta_v \\ \mathbb{V}\delta_\theta \end{bmatrix} = \begin{bmatrix} \delta'_P \\ \delta'_Q \end{bmatrix} \tag{5.42}$$

It can then be deduced that:

$$\begin{bmatrix} Real(\dot{\mathbb{V}}) & Imag(\dot{\mathbb{V}}) \\ -Imag(\dot{\mathbb{V}}) & Real(\dot{\mathbb{V}}) \end{bmatrix}^{-1} \times \begin{bmatrix} \frac{\partial P}{\partial v} + \Delta_1 & (\frac{\partial P}{\partial \theta} + \Delta_3) \times \mathbb{V}^{-1} \\ \frac{\partial Q}{\partial v} + \Delta_2 & (\frac{\partial Q}{\partial \theta} + \Delta_4) \times \mathbb{V}^{-1} \end{bmatrix} \times \begin{bmatrix} Real(\dot{\mathbb{A}}) & -Imag(\dot{\mathbb{A}}) \\ Imag(\dot{\mathbb{A}}) & Real(\dot{\mathbb{A}}) \end{bmatrix}^{-1} = \begin{bmatrix} Real(\dot{Y}) & -Imag(\dot{Y}) \\ Imag(\dot{Y}) & Real(\dot{Y}) \end{bmatrix} + \begin{bmatrix} Real(\Delta_{\dot{Y}}) + \Delta'_1 & -Imag(\Delta_{\dot{Y}}) + \Delta'_3 \\ Imag(\Delta_{\dot{Y}}) + \Delta'_2 & Real(\Delta_{\dot{Y}}) + \Delta'_4 \end{bmatrix} \tag{5.43}$$

In Eqs.5.43 $\Delta'_1$, $\Delta'_2$, $\Delta'_3$ and $\Delta'_4$ are generated by $\Delta_1$, $\Delta_2$, $\Delta_3$ and $\Delta_4$ with $\dot{\mathbb{V}}$, $\mathbb{V}$ and $\dot{\mathbb{A}}$. Due to the fact that $\dot{\mathbb{V}}$, $\mathbb{V}$ and $\dot{\mathbb{A}}$ are all scaling matrices, $\Delta'_1$, $\Delta'_2$, $\Delta'_3$ and $\Delta'_4$ are all diagonal matrices.

**Theorem 2.** *[80] The Jacobian matrix-based linear system in TDS can be transformed into the form of $(Y + \Delta_Y)x = b$, with $Y$ as a matrix which is based on the admittance of the Power System network, and $\Delta_Y$ an relatively low-rank matrix with non-zeros on 3 diagonals.*

*Proof.* We denote $Y$ and $\Delta_Y$ as:

$$Y = \begin{bmatrix} Real(\dot{Y}) & -Imag(\dot{Y}) \\ Imag(\dot{Y}) & Real(\dot{Y}) \end{bmatrix} \tag{5.44}$$

$$\Delta_Y = \begin{bmatrix} Real(\Delta_{\dot{Y}}) + \Delta'_1 & -Imag(\Delta_{\dot{Y}}) + \Delta'_3 \\ Imag(\Delta_{\dot{Y}}) + \Delta'_2 & Real(\Delta_{\dot{Y}}) + \Delta'_4 \end{bmatrix} \tag{5.45}$$

Further, we denote the "scaling" matrices as:

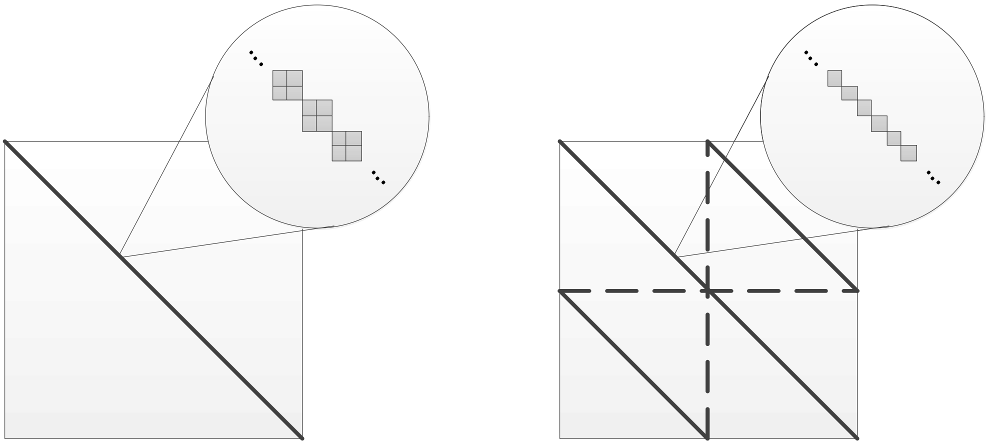

a. Blocked form b. Element-wise form

Figure 5.1: Structure of $\Delta_Y$ matrix

$$\dot{\mathbb{V}}_L = \begin{bmatrix} Real(\dot{\mathbb{V}}) & Imag(\dot{\mathbb{V}}) \\ -Imag(\dot{\mathbb{V}}) & Real(\dot{\mathbb{V}}) \end{bmatrix}^{-1} \tag{5.46}$$

$$\dot{\mathbb{A}}_R = \begin{bmatrix} Real(\dot{\mathbb{A}}) & -Imag(\dot{\mathbb{A}}) \\ Imag(\dot{\mathbb{A}}) & Real(\dot{\mathbb{A}}) \end{bmatrix}^{-1} \tag{5.47}$$

We apply $\dot{\mathbb{V}}_L$ and $\dot{\mathbb{A}}_R$ to the left and the right side of the linear system in Eqs.5.42. The result linear system is as follows:

$$\Bigg(Y + \Delta_Y\Bigg) \times \begin{bmatrix} \delta'_v \\ \delta'_\theta \end{bmatrix} = \begin{bmatrix} \delta''_P \\ \delta''_Q \end{bmatrix} \tag{5.48}$$

Hence the Jacobian-based linear system is transformed into a linear system based on $Y + \Delta_Y$, as in Eqs.5.48. $Y$ is the transformed admittance matrix of the original matrix. The sparsity pattern of $\Delta_Y$ satisfies that there is only non-zero on the main-diagonal and the $n$-th diagonal below/above the main diagonal. This completes the proof. ☐

Note that $\Delta_Y$ can also be seen as a matrix with sub-blocks as diagonal matrices. $Y$ is static and non-changing during the simulation, while $\Delta_Y$ changes from step to step. At each step, we can transform the original linear system into formulation in Eqs.5.48. By solving this transformed linear system, its solution is mapped back to the original linear system based on the full Jacobian matrix, hence the entire system is solved.

Similar to Chapter 4, we can also form the matrix in an element-wise manner rather than block-wise: we align the voltage $v$ and angle $\theta$ of each bus to be adjacent. This is equivalent to a diagonal reordering on the matrix in Eqs.5.48. In the reordered form, $\Delta_Y$ is a block-diagonal matrix, with non-zeros only present on $2 \times 2$ diagonal blocks. The structure of $\Delta_Y$ matrix in blocked format and element-wise format is shown in Fig.5.1.

## 5.3 Preconditioner Update and Application to TDS

In this section firstly a brief introduction to preconditioner updates is given in Section 5.3.1. After that, algorithm variants tailored to TDS simulation are presented in Section 5.3.2.

### 5.3.1 Introduction to Preconditioner Updates

We consider updating an existing incomplete factorization of a given matrix $A$ when $A$ undergoes an additive change of $\Delta_A$. We use $LDU$ to denote the (incomplete) factorization with $L$ and $U$ unitary on the main diagonal and $D$ is a diagonal scaling matrix. By [77] it can be deduced as:

$$A \overset{\text{❶}}{\approx} LDU \tag{5.49}$$

$$A + \Delta_A \overset{\text{❶}}{\approx} LDU + \Delta_A \tag{5.50}$$

$$LDU + \Delta_A = L(DU + L^{-1}\Delta_A) \tag{5.51}$$

$$\overset{\text{❷}}{\approx} L(DU + \Delta_A) \tag{5.52}$$

$$\overset{\text{❹}}{\approx} L(DU + triu(\Delta_A)) \tag{5.53}$$

$$LDU + \Delta_A = (LD + \Delta_A U^{-1})U \tag{5.54}$$

$$\overset{\text{❸}}{\approx} (LD + \Delta_A)U \tag{5.55}$$

$$\overset{\text{❺}}{\approx} (LD + tril(\Delta_A))U \tag{5.56}$$

The approximations due to omitting elements are labeled. Label ❶ denotes the approximation brought by incomplete factorization. When we assume that $L^{-1}$ is close to $I$, we can further take the approximation of ❷. If we want to maintain the sparsity of $DU + \Delta_A$ and contain all the non-zero in the upper triangular part, further approximation labeled as ❹ is taken, omitting the non-zero elements in the strict-lower part of $DU + \Delta_A$ by keeping only the upper-triangular part of $\Delta_A$. Approximation ❸ and ❺ are the counterparts of ❷ and ❹ respectively, by operating on $L$ rather than $U$.

We also propose the use of another form of preconditioner updates which keeps more information by considering the effects of the whole matrix of $\Delta_A$ in Eqs.5.52, rather than only the upper-triangular part of it as in Eqs.5.53. Following Eqs.5.52, we have:

$$LDU + \Delta_A \overset{\text{❷}}{\approx} L(DU + \Delta_A) \tag{5.57}$$

$$\overset{\text{❻}}{=} LL^*U^* \tag{5.58}$$

In which, $L^*$ and $U^*$ is the exact LU decomposition of $(DU + \Delta_A)$. Exact factorization implies no data loss as brought about by approximation ❹ (or similarly by ❺). We use ❻ as the label for the algorithm in Eqs.5.58. Note that this process is not an approximation and does not introduce extra information loss. Whether it is feasible for the full factorization depends on 3 factors:

- Computational overhead of the factorization
- Sparsity pattern of $L^*$, especially its non-zero element count
- Sparsity pattern of $U^*$, especially its non-zero element count as compared with $U$

If pivoting is used for incomplete factorization, then we have the following settings:

$$PA \overset{\text{❶}}{\approx} LDU \tag{5.59}$$

$$P(A+\Delta_A) \overset{\text{❶}}{\approx} LDU + P\Delta_A \tag{5.60}$$

$$LDU + P\Delta_A \overset{\text{❷}}{\approx} L(DU + P\Delta_A) \tag{5.61}$$

$$\overset{\text{❹}}{\approx} L(DU + triu(P\Delta_A)) \tag{5.62}$$

$$LDU + P\Delta_A \overset{\text{❸}}{\approx} (LD + P\Delta_A)U \tag{5.63}$$

$$\overset{\text{❺}}{\approx} (LD + tril(P\Delta_A))U \tag{5.64}$$

Note the use of pivoting matrix in Eqs.5.62. In the following part of this section, we show that without pivots, it is feasible to use Eqs.5.58 and ❻ for TDS matrices, without high computational and memory overhead. Pivoting breaks these properties and hence not considered in further experiments.

### 5.3.2 Updating Preconditioners in TDS

The formulation of the linear system for TDS in $Y + \Delta_Y$ enables the additive updates to the preconditioners as introduced in Eqs.5.53 and Eqs.5.58. For Eqs.5.53 we can directly apply to the matrix $Y + \Delta_Y$. Eqs.5.58 is subjected to more analysis in feasibility of complete factorizations for TDS matrices.

As basic settings, we construct preconditioner based on the transformed admittance matrix of the power network, i.e., $Y$ matrix, and use $\Delta_{Y_i}$, i.e., the additive change at the $i$-th linear system to update the preconditioner. Note that the preconditioner can be constructed in an "off-line" manner: it is not subjected to changes during the simulation. Hence the construction does not bring extra runtime overhead

#### Feasibility Analysis and Preconditioner Choice

We use the element-wise formulation for $Y+\Delta_Y$. Note that in the strict lower part of $\Delta_Y$, there is only non-zero elements on the diagonal with index -1, i.e., immediately close to the main diagonal, and the non-zero is only present on the rows with even indices. When applying ❻, we factorize the matrix of $DU + \Delta_A$.

Then the factorization with $DU + \Delta_A$ would entail:

- Computation of $L^*$ in Eqs.5.58
- Computation of $U^*$ in Eqs.5.58

This factorization operation would only involve the linear combination of some adjacent rows of $DU+\Delta_A$. Denote $C$ as $DU+\Delta_Y$ and $c_i$ the $i$-th row of $C$. Then in the strict lower part of $C$, the only non-zero elements would only exist at positions $(2i-1, 2i)$, with $1 \leq i \leq n$ and $n$ is the total bus number. Eliminating these values to form $L^*$ and $U^*$ involves combining $c_{2i-1}$ into $c_{2i}$. Then we have:

$$L_{i,j} = \begin{cases} 1 & \text{if } \ i = j \\ \frac{C_{i,i-1}}{C_{i-1,i-1}} & \text{if } \ i \mod 2 = 0, j = i-1 \\ 0 & \text{otherwise} \end{cases} \tag{5.65}$$

$$U_{i,j} = \begin{cases} C_{i,j} & \text{if } \ i \mod 2 = 1, i \leq j \\ C_{i,j} - \frac{C_{i,i-1}}{C_{i-1,i-1}} C_{i-1,j} & \text{if } \ i \mod 2 = 0, i \leq j \\ 0 & \text{otherwise} \end{cases} \tag{5.66}$$

It is immediate that the construction of $L^*$ and $U^*$ is easy and only involves local combination of the rows of $C$. In runtime, the computation of $L^*$ and $U^*$ only involves a little computation and this process is potentially parallel. On the memory usage side, the linear combination of the rows may introduce extra non-zero elements in the rows with even indices in $U^*$. Potentially the sparsity pattern of $U^*$ would be close to that of $C$ and $U$. We can allocate memory usage in these rows in advance to accommodate these extra fill-in's caused by full decomposition of $C$.

### Preconditioner Choice

The feasibility analysis depends on the specific ILU preconditioner we use for $Y$.

If we use preconditioner with no pivoting, such as the basic form of Crout-version of ILU [60] without pivoting, denoted ILU-C, in the strict lower triangular part of $(DU + \Delta_Y)$, only positions at $(i, i-1)$ contain non-zero elements for $\{i | i \mod 2 = 0, \ 1 < i \leq 2n\}$. Then we have the following properties:

- Sparsity pattern of $L^*$ is exactly that of the lower triangular of $(DU + \Delta_A)$
- Computing of $L^*$ and $U^*$ is straightforward
- Computing $U^*$ involves linear combination of the $(2 * i - 1)$-th row and $2i$-th row for $1 \leq i \leq n$
- Sparsity pattern of $U^*$ is close to that of $C$ or $U$

But if we use preconditioner with pivoting, such as ILU-TP. Then in the strict lower triangular part of $(DU + P\Delta_A)$, potentially non-zero elements can exist in $L^*$ at positions other than the sparsity pattern defined by the strictly lower triangular of $P\Delta_A$. The actual positions for extra non-zero elements depend on the structure of the pivoting matrix. Due to this, we only consider the use of ILU-C preconditioners for the preconditioner updating operations in this thesis.

## 5.4 Experiments

We carry out experiments on various TDS cases using Preconditioner Updating algorithms in Section 5.3. We transform the Jacobian-based linear systems in the form of Eqs.5.48. Based on $Y$, we construct preconditioners $M$ for it using incomplete factorizations. Preconditioner Updates are applied to $M$ based on $\Delta_Y$ for each step.

| Case | Matrix Size | Linear System Count |
|---|---|---|
| 2K-STABLE | 2330 | 1970 |
| 2K-UNSTABLE | 2330 | 647 |
| Dongbei-B | 976 | 3901 |
| Dongbei-F | 976 | 4010 |
| 10188 | 20376 | 1012 |

Table 5.1: TDS cases for Preconditioner Updates tests

| Case | Solver | Preconditioner |
|---|---|---|
| 2K-STABLE | GMRES(40) | ILU-C($10^{-2}$) |
| 2K-UNSTABLE | GMRES(40) | ILU-C($10^{-2}$) |
| Dongbei-B | GMRES(40) | ILU-C($10^{-2}$) |
| Dongbei-F | GMRES(40) | ILU-C($10^{-2}$) |
| 10188 | GMRES(60) | ILU-C($10^{-4}$) |

Table 5.2: Preconditioner Update – Basic Settings

### 5.4.1 Test Cases and Settings

The TDS cases used for tests are listed in Tab.5.1. The description of the cases are in Chapter 2. Tab.5.1 lists the total linear system count, i.e., total Newton step count across all the time steps. The sizes of the matrices as in Tab.5.1 now equal to $2n$, i.e., the size of $Y$ and $\Delta_Y$.

We denote various Preconditioner Update strategies as follows:

**Strategy 1** : use "dishonest preconditioner" strategy as in [68, 58], i.e.: a fixed preconditioner for $Y$ is used and no updates are involved.

**Strategy 2** : use updating algorithm ❹, based on the preconditioner for $Y$.

**Strategy 3** : use updating algorithm ❻, based on the preconditioner for $Y$.

**Strategy 4** : reconstruct preconditioner at each Newton step

### 5.4.2 Comparison

We compare the effect of dishonest preconditioner based on dishonest preconditioner and preconditioner reconstruction strategies based on the settings shown in Tab.5.2. We use the default 0.01s as the time stepping for the simulation. Restarted GMRES is used. Tab.5.3 shows the averaged iteration count for different strategies as described above.

| Case | Strategy-1 | Strategy-2 | Strategy-3 | Strategy-4 |
|---|---|---|---|---|
| 2K-STABLE | 166.20 | 46.81 | 38.33 | 22.62 |
| 2K-UNSTABLE | 153.67 | 60.09 | 57.39 | 23.98 |
| Dongbei-B | 50.93 | 34.35 | 34.26 | 31.00 |
| Dongbei-F | 51.51 | 34.53 | 34.46 | 30.99 |
| 10188 | 288.45 | 192.05 | 191.31 | 74.90 |

Table 5.3: Preconditioner Update results

| Case | Preconditioner | Strategy-1 | Strategy-2 | Strategy-3 |
|---|---|---|---|---|
| 2K-STABLE | ILU-C($10^{-2}$) | 166.20 | 46.81 | 38.33 |
| 2K-STABLE | ILU-C($10^{-3}$) | 98.80 | 44.31 | 24.05 |
| 2K-STABLE | ILU-C($10^{-4}$) | 105.19 | 113.88 | 36.00 |
| 2K-UNSTABLE | ILU-C($10^{-2}$) | 153.67 | 60.09 | 57.39 |
| 2K-UNSTABLE | ILU-C($10^{-3}$) | 94.82 | 47.03 | 30.56 |
| 2K-UNSTABLE | ILU-C($10^{-4}$) | 101.02 | 64.96 | 34.46 |
| Dongbei-B | ILU-C($10^{-2}$) | 50.93 | 34.35 | 34.26 |
| Dongbei-B | ILU-C($10^{-3}$) | 31.80 | 22.89 | 22.71 |
| Dongbei-B | ILU-C($10^{-4}$) | 29.97 | 23.88 | 23.60 |
| Dongbei-F | ILU-C($10^{-2}$) | 51.51 | 34.53 | 34.46 |
| Dongbei-F | ILU-C($10^{-3}$) | 31.85 | 23.08 | 22.74 |
| Dongbei-F | ILU-C($10^{-4}$) | 30.09 | 23.91 | 23.64 |
| 10188 | ILU-C($10^{-4}$) | 288.45 | 192.05 | 191.31 |
| 10188 | ILU-C($10^{-5}$) | 165.57 | 111.22 | 111.56 |
| 10188 | ILU-C($10^{-6}$) | 100.17 | 79.81 | 79.88 |

Table 5.4: Preconditioner Update results

Preconditioner updates achieves significant reduction in iteration count compared with dishonest preconditioner strategy. Using the basic algorithm as in ❹, reduction ratio in total iteration count ranges from 33% to 70%. When using the proposed algorithm of ❻, the iteration counts for 2K-STABLE and 2K-UNSTABLE are further reduced by 20% and 5%, respectively. Note that the iteration count for Strategy-4 is actually the lower bound for that using preconditioner updates. For Dongbei-B and Dongbei-F, the iteration counts achieved by preconditioner updates are already very close to the upperbound by Strategy-4.

Tab.5.3 lists the comparison between different strategies under basic preconditioner settings. To facilitate preconditioner updates, ILU-C with no pivoting is used. We can see that preconditioner updates can achieve significant reduction in iteration count. In 2 Dongbei related cases and 10188, reduction ratio in iteration count is about 33%. For 2K cases this ratio is over 70%. Strategy-3, i.e., preconditioner updates with ❻ benefit various cases in different way. For 2K-STABLE, the reduction ration of using Strategy-3 compared with Stratetegy-2 is about 20%, while for other cases, this ratio is below 5%.

### Effect of Preconditioner Quality

We vary the preconditioner configuration to different dropping tolerance levels, to investigate the effect of preconditioner quality on updates. Tab.5.4 shows the average iteration count per linear system for cases in Tab.5.1.

Tab.5.4 shows that: (1) when initial preconditioner dropping threshold is lowered, iteration count does not always decrease, and (2) preconditioner updates always yields lower iteration count compared with dishonest preconditioner strategy. For 2K-STABLE and 2K-UNSTABLE, lowering the dropping threshold for ILU-C from $10^{-3}$ to $10^{-3}$ results increased iteration count. For Dongbei-B and Dongbei-F, lowering the threshold results in virtually stationary iteration count. This counter-intuitive result is caused by the fact that the initial preconditioner is constructed based on $Y$. Assume the error bound for the initial incomplete factorization is $\varepsilon = Y - LDU$. Then the actual error bound is

| Case | Preconditioner | Time Step | Strategy-1 | Strategy-2 | Strategy-3 |
|---|---|---|---|---|---|
| 2K-STABLE | ILU-C($10^{-2}$) | 0.01s | 166.20 | 46.81 | 38.33 |
| | | 0.02s | 168.04 | 69.24 | 55.66 |
| | | 0.05s | 170.29 | 66.93 | 55.16 |
| 2K-UNSTABLE | ILU-C($10^{-2}$) | 0.01s | 153.67 | 60.09 | 57.39 |
| | | 0.02s | 163.71 | 61.77 | 51.79 |
| | | 0.05s | 164.17 | 60.62 | 57.13 |
| Dongbei-B | ILU-C($10^{-2}$) | 0.01s | 50.93 | 34.35 | 34.26 |
| | | 0.02s | 57.44 | 38.12 | 37.93 |
| | | 0.05s | 56.40 | 38.14 | 37.79 |
| Dongbei-F | ILU-C($10^{-2}$) | 0.01s | 51.51 | 34.53 | 34.46 |
| | | 0.02s | 57.35 | 38.05 | 37.81 |
| | | 0.05s | 56.16 | 37.99 | 37.60 |
| 10188 | ILU-C($10^{-4}$) | 0.01s | 288.45 | 192.05 | 191.31 |
| | | 0.02s | 280.10 | 189.65 | 189.16 |
| | | 0.05s | 265.37 | 179.62 | 179.96 |

Table 5.5: Preconditioner Update results

$\varepsilon_i = \varepsilon + \Delta_{Y_i}$ for the $i$-th matrix. There is no guarantee on $\|\varepsilon_i\|$ even if $\|\varepsilon\|$ decreases by lowering the dropping threshold for $Y$s. In all the cases, preconditioner updates yields lower iteration count by over 20%. For 2K-STABLE and 2K-STABLE, using ❻ yields about 40% further reduction in iteration count than ❹, while in other cases there is no obvious advantage.

### Effect of Time Stepping

We relax the time stepping of 0.01s to larger sizes of 0.02s and 0.05s. The results are shown in Tab.5.5. The reduction of iteration counts by preconditioner updates are consistent for different time step sizes.

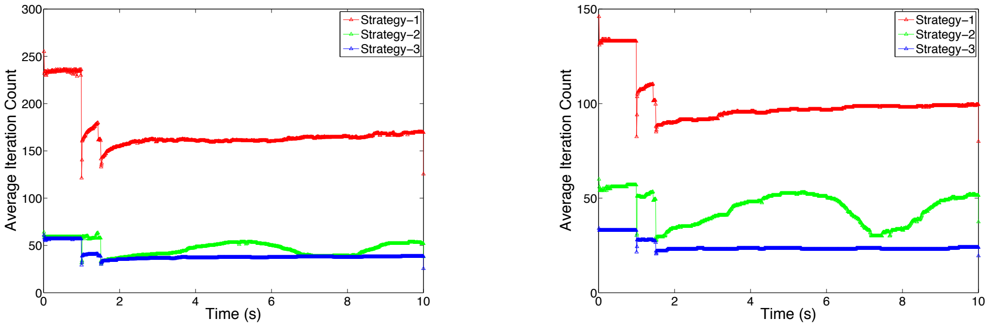
### Analysis of Simulation Process

To further investigate the effect of preconditioner updating, the average iteration count per time step is retrieved. Fig.5.2 shows the comparison between different strategies for 2K-STABLE, Dongbei-B and 10188.

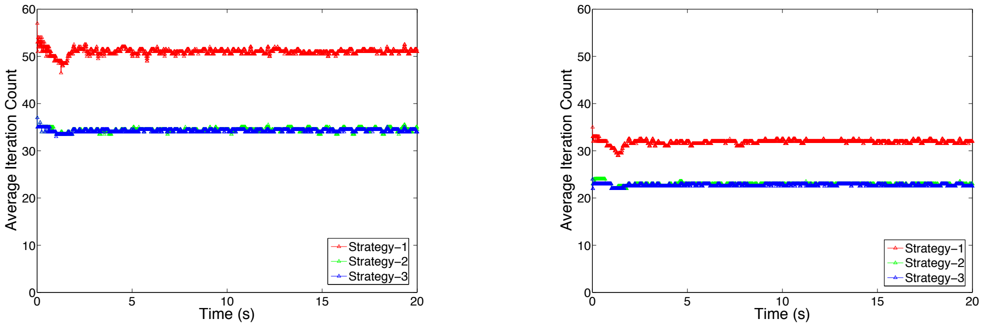
We can see that for 2K-STABLE, for certain period of the time, such as from 1s to 1.5s and the time around 5s and 9.5s, Strategy-3 performed better than Strategy-2 in a more significant way, while at other times, Strategy-3 performs similarly to Strategy-2. For Dongbei-B and 10188, the difference between the 2 strategies are never obvious. Generally speaking, Strategy-3 performs better than or on par with Strategy-2 for these cases.

## 5.5 Summary

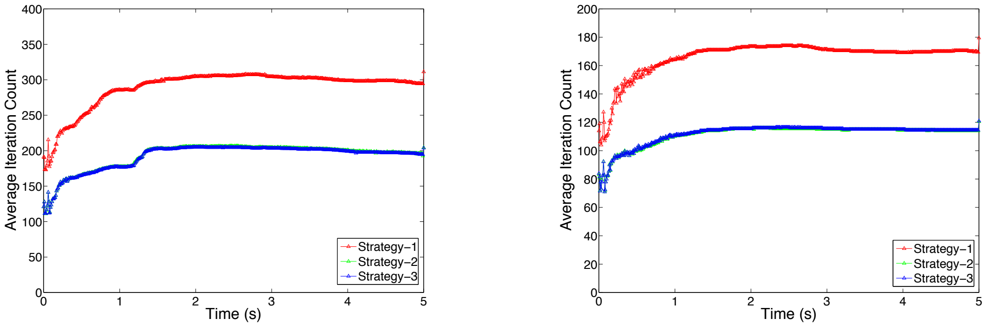
In this chapter we apply the preconditioner updating techniques to TDS. By transforming the Jacobian matrix-based linear system into a matrix with form of $Y + \Delta_Y$, we introduce



a. 2K-STABLE with $\mathsf{ILU\text{-}C}(10^{-2})$ b. 2K-STABLE with $\mathsf{ILU\text{-}C}(10^{-3})$



c. Dongbei-B with $\mathsf{ILU\text{-}C}(10^{-2})$ d. Dongbei-B with $\mathsf{ILU\text{-}C}(10^{-3})$



e. 10188 with $\mathsf{ILU\text{-}C}(10^{-4})$ f. 10188 with $\mathsf{ILU\text{-}C}(10^{-5})$

Figure 5.2: Comparison of iteration counts between Preconditioner Updating strategies

preconditioner based on the admittance matrix $Y$ and off-line analysis to the preconditioning of the transformed Jacobian matrices. Further we adopt the additive change to ILU-C preconditioner constructed based on ILU-C. The proposed algorithms to update the upper-triangular part of the incomplete factorization introduces little computational overhead to TDS. Iteration count with GMRES is greatly reduced by introducing updates to the preconditioners: compared with a previous existing preconditioner updating method, i.e., "dishonest preconditioners", the reduction ratio ranges between 20% and 70% across all preconditioner configurations and test cases. It has been shown that preconditioner updates serve as an effective multi-step technique to re-use the preconditioner information of the static analysis of the network and reduce the total computation of the time domain simulation of Power Systems.

# Chapter 6

# Multi Step Techniques in TDS – Matrix Spectra Deflation

## 6.1 Introduction

Time Domain Simulation (TDS) of Power Grids involves solving a sequence of Jacobian matrix-based linear systems. By transformations, the Jacobian matrix can be formulated as a derived matrix based on the original admittance matrix of the Power Network. Given the simulation task in TDS, it is beneficiary to reuse certain information derived from previous iterations in future linear systems, to reduce the iteration count and overall computation amount. This chapter discusses the use of matrix spectra deflation in TDS. GMRES and GCR include restarted Arnoldi process to minimize norm, which reveals extreme eigenvalues of the Jacobian matrices. The dynamically changes of the Jacobian matrices pose special problem since the matrix, hence the spectrum changes from system to system. We consider the use of GCRO-DR and dynamic selection of extreme eigenvalues to deflate both small and large eigenvalues to accommodate the dynamic matrix changes. This chapter is organized as follows. Section 6.2 gives a short introduction to spectra deflation. In Section 6.3, the analysis of the (preconditioned) Jacobian matrices based on the $Y + \Delta_Y$ formulation is given, showing that the Jacobian matrices contain both large and small eigenvalues during simulation. Section 6.4 discusses how to design effective deflation algorithm for TDS. Section 6.5 the Preconditioner Updates and Deflation is combined for better convergence of GCRO-DR. Experiments and analysis are included in Section 6.6. Section 6.7 concludes the chapter.

## 6.2 Introduction to Spectra Deflation

Matrix spectra deflation involves enhancing the convergence of iterative solvers by introducing vectors to “deflate”, i.e., remove unwanted eigenvalues and improve the spectra of the matrix. The vectors used are usually eigenvectors which are associated with the unwanted eigenvalues which are close to the origin. The main motivations for spectra deflation are:

- Spectra of the matrix affect the convergence rate of Krylov solvers. Removing extreme eigenvalues from the spectra, especially those close to the origin, will benefit convergence speed greatly.

---

This chapter is submitted as journal paper [87]

- Practical long-recurrence based algorithms such as GMRES require restarts, which cause the loss of all the Krylov subspace information periodically and result in potential stagnation. Deflation is an effective way to retain information from previous restarts.

- When solving a series of linear systems as in TDS, either the matrix of the linear systems is constant with multiple right-hand sides (RHS), or the matrix of the linear systems also changes, it is desirable to retain certain information generated from previous linear system solution processes to aid the solution of further systems. In this case, deflation could also be used.

Although spectra information can be used to speed up the convergence of Krylov solvers, computing all the eigenvalues and associated eigenvectors of a given matrix is not a computationally feasible task. Fortunately, extreme values among the eigenvalues can be retrieved without the effort of computing all the eigenvalues, by an Arnoldi process (or Lanczos process for symmetric matrices) which is embedded in solvers such as GMRES. The GMRES iteration generates a upper-Hessenberg matrix $H_m$, which is the upper $m \times m$ part of the matrix $\bar{H}_m$ in the following equation:

$$AV_m = V_{m+1}\bar{H}_m \tag{6.1}$$

With the iteration of GMRES carries on, the extreme eigenvalues of $H_m$ (i.e., the largest and the smallest in norm) converge to those of $A$. We call the eigenvalues of $H_m$ Rits values.

### 6.2.1 Computing Eigenvalues and Eigenvectors

We calculate the eigenvalues of $H_m$ and use it for deflation of $A$. We denote the eigenvalues and corresponding eigenvectors of $H_m$: $\lambda_i$ and $v_i$, for $1 \leq i \leq m$, with $|\lambda_1| \leq |\lambda_2| \leq \cdots \leq |\lambda_m|$. Those of $A$ are denoted as $\Lambda_i$ and $V_i$, for $1 \leq i \leq n$, with $|\Lambda_1| \leq |\Lambda_2| \leq \cdots \leq |\Lambda_n|$.

Normally we consider the smallest eigenvalues for deflation, but it is also mentioned that largest eigenvalues can also be retrieved and used in deflation as in [69]. There are various ways to calculate $k$ smallest eigenvalues of $A$ through $H$. Here we list two of them:

1. Denote the smallest eigenvalues of $A$ as $\Lambda_1$ to $\Lambda_k$. To compute them, we calculate the smallest $k$ eigenvalues of matrix: $(H_m + h^2_{m+1,m} H_m^{-T} e_m e_m^T)$, we denote them and their corresponding eigenvectors as $\lambda'_i$ and $v'_i$. The formulation for Arnoldi-based eigenvalue estimation is as in [51].

2. Use a general form of eigenvalue formation: $((AV_m)^H V_m)x = \frac{1}{\lambda}((AV_m)^H AV_m)x$ as used in [64], where $(AV_m)^H$ is the conjugate transpose of $AV_m$. This form tends to estimate the smallest eigenvalues of $A$ more precisely, i.e., largest values for $\frac{1}{\lambda}$. Note that $AV_m$ can be replaced by $\bar{H}_m$ and in turn the internal QR factorization of $\bar{H}_m$: $\bar{H}_m = Q_{m+1} \times \begin{bmatrix} R_m \\ 0 \end{bmatrix}$, with $R_m$ an upper-triangular matrix.

When $\lambda_i$'s and their corresponding eigenvectors $v_i$'s (also called Rits vectors) are calculated, we can calculate $V_i$ simply: $V_i = V_m \times v_i$.

## 6.2.2 Deflation Algorithms

Generally speaking, there are two approaches for deflation in GMRES. The first way is through augmenting the Krylov subspace by adding deflation vectors to it. This involves algorithms such as GMRES-E [65] and GMRES-DR [65]. The second way is through preconditioners built with deflation vectors [28, 38, 42].

### Deflation-based Preconditioning

By preconditioning, the corresponding eigenvalues are shifted from their original positions (which are close to the origin) to (1, 0) in the complex plane. Deflation based on preconditioning takes this form:

$$(I - \delta VV^H)Ax = (I - \delta VV^H)b \tag{6.2}$$

Where $V$ is the matrix consisting of deflation vectors, i.e., eigenvectors of $A$, $V$ is of size $n \times k$, given there are $k$ eigenvectors. One advantage of this approach is that we can use whatever vectors for deflation, even if they are not eigenvectors (or they are poor estimation of them). When they are not actual eigenvectors, effect on convergence is then compromised.

### Deflation Vector used in Krylov Subspace

Another way for deflation falls under Ronald Morgan's works such as GMRES-E. From GMRES-E to GMRES-DR, etc. GMRES-E tries to augment the Krylov subspace when a restart is due. Suppose that we have certain deflation vectors, $V_1$ to $V_k$. When the restart of GMRES is due, we orthogonalize the remaining vector against deflation vectors. The effective subspace used for GMRES-E is:

$$Span\{r_0, Ar_0, ..., A^{m-1}r_0, V_1, ...V_k\} \tag{6.3}$$

Alternatively, one can place those deflation vectors in the front of the Krylov subspace. Orthogonalization is carried firstly with these vectors, then with the basis generated in current GMRES run. The algorithm developed is called GMRES-DR. The effective subspace by GMRES-DR is:

$$Span\{V_1, ..., V_k, r_0, Ar_0, ..., A^{m-1}r_0\} \tag{6.4}$$

The effectiveness of both GMRES-E and GMRES-DR depends on the condition that the vectors used, i.e., $V_i$'s form an invariant subspace for $A$. If such a condition does not hold, the effect of speeding up the iterations is lost.

### Deflation in Solving Consecutive Linear Systems

There have been some (recent) works on how to use deflation to benefit the scenario that several consecutive linear systems are to be solved, with constant matrix $A$ or a gradually changing $A$. They also fall into 2 categories: (1) deflation through Krylov subspace augmentation as in [69], and (2) preconditioning by deflation as in [50].

For augmenting Krylov subspace, due to that the loss of the property of Krylov subspace will potentially introduce stagnation in convergence, GMRES-E and GMRES-DR can not be effectively adopted to integrate the eigenvectors generated from previous runs, since changing either RHS or $A$ will result in breaking the Krylov subspace property of orthogonality.

In [69], the author proposes the use of any deflation vectors (possibly eigenvectors from another matrix) in the current linear solving process, without restricting that they should form an invariant subspace of current matrix $A$. It uses a variant of GCR , due to that GCR may allow more relaxed relationship between search subspace and minimization subspace. A variant of GCR , called GCROT deals with restarting GCR with optimal truncation (i.e., restarting but keep certain subspace information). A deflation-based GCR -based algorithm, GCRO-DR is developed to deal with solving consecutive linear systems.

The work in [50] is exemplary on preconditioning by deflation for consecutive linear systems. In this section, the preconditioner is augmented, i.e., updated with more spectra information as more and more linear systems are solved. It also uses GMRES-DR for the reconstruction of spectra information. It is designed to take into consideration of the quality of the spectra information, so that inaccurate, un-converged part of the estimated spectrum is not used. However it does not consider the situation when matrix $A$ is changing. Experiments in [50] show that with spectra preconditioner of low quality can even damage the convergence speed. Therefore, we should design a suitable strategy with more accurate spectra preconditioner for a the case with a changing matrix.

### 6.2.3 Limitation of Deflation

Deflation alone usually cannot accomplish the task of improving convergence effectively, and therefore it is usually used in combination with other preconditioners such as IC, ILU. Hence deflation can be considered as a method for accelerating the convergence of Krylov solvers.

## 6.3 Spectra Analysis of Jacobian Matrices in TDS

We apply deflation to Time Domain Simulation (TDS) of Power Grids. We adopt the transformation in Chapter 5 for the additive formulation of the Jacobian matrices as the basic setting of deflation in TDS. For each Jacobian-based linear system in TDS, we transform it into the following form:

$$(Y + \Delta_Y) \times x = b \tag{6.5}$$

Where $Y$ is a transformed matrix of the original admittance matrix $Y^\circ$. $Y^\circ$ is the original admittance matrix of the Power Network, which is symmetric and complex. $Y$ is a transformed form of $Y^\circ$ and is a real valued matrix. $Y$ remains the same across the whole simulation, while both $\Delta_Y$ and $b$ change from linear system to linear system.

We first analyze the matrix's own characteristics from several aspects: (1) $Y^\circ$ be transformed in various ways, i.e., $Y = \begin{bmatrix} Re(Y^\circ) & Im(Y^\circ) \\ -Im(Y^\circ) & Re(Y^\circ) \end{bmatrix}$ (denoted as Scheme [ 1, -i; i, 1 ]), $Y = \begin{bmatrix} -Im(Y^\circ) & Re(Y^\circ) \\ Re(Y^\circ) & Im(Y^\circ) \end{bmatrix}$ (denoted as Scheme [ -i, 1; 1, i ]), or $Y = \begin{bmatrix} -Im(Y^\circ) & Re(Y^\circ) \\ Re(Y^\circ) & Im(Y^\circ) \end{bmatrix}$ (denoted as Scheme [ i, -1; 1, i ]); (2) treat $Y^\circ$ in these schemes after diagonal scaling, (3) treat $Y^\circ$ in these formulations after ILU preconditioners (either ILU-0 or ILU with the dropping threshold of $10^{-2}$), which is used in previous chapters.

| System | Bus Count | Size of $Y$ | Description |
|---|---|---|---|
| IEEE39 | 39 | 78 | Standard system as model problem |
| Dongbei | 688 | 1376 | Power grid for Dong Bei in China |
| 2K | 1165 | 2330 | Power grid for He Bei in China |
| 10188 | 10188 | 20376 | Simplified model for China power grid |

a. Power systems

| System | Case | Simulation Duration | Linear System Count | Preconditioner |
|---|---|---|---|---|
| IEEE39 | v | 0.5 s | 369 | Diagonal Jacobian based on $Y$ |
| Dongbei | B | 20 s | 3901 | ILU$(10^{-2})$ |
| 2K | STABLE | 10 s | 1970 | ILU$(10^{-2})$ |
| 10188 | | 5 s | 1012 | ILU$(10^{-4})$ |

b. Time domain simulation cases

Table 6.1: Power systems and test cases used in deflation experiments

We firstly analyze the spectra of $Y$ matrices, and their spectra after preconditioners are applied. We also analyze the augmented form of the matrix which is used in time domain simulation. Since deflation is usually accompanied with preconditioners, this analysis serves as the base ground for applying deflation to preconditioned TDS matrices. The following power grids and corresponding time domain simulation cases are used for spectra analysis and deflation experiments:

**IEEE39** : 39 bus reference system.

**Dongbei** : 488 bus system for Dong Bei of China.

**2K** : He Bei 1165 bus system.

**10188** : China electrical grid with 10188 buses.

IEEE39 is a reference system which is used as a model problem, while other systems are from our own test set which are used in previous chapters. The corresponding simulation cases are listed in Tab.6.1. Note that preconditioner configurations and the numbers of linear systems to be solved are also included.

### 6.3.1 Preconditioners to $Y$ and $Y + \Delta_Y$

We analyze $Y$ and constructing a preconditioner for it, and re-use the preconditioner for all the linear systems. We denote the left and right preconditioners $M_l$ and $M_r$. The effective system and right-hand side for Krylov solver are $M_l \backslash (Y + \Delta_Y)/M_r$ and $M_l \backslash b$.

We use two kinds of preconditioners for the experiments and evaluation: (transformed-point) Jacobian and ILU. For simple model problem, transformed point Jacobian is good enough for convergence, while larger systems usually require ILU for practical convergence behavior.

The transformed point Jacobian is similar to a traditional point Jacobian preconditioner. Here we retrieve the diagonal part of $Y^\circ$ matrix, denoted $D_{Y^\circ}$, and cast the transformation which is used for transforming $Y^\circ$ into $Y$ on $D_{Y^\circ}$, to retrieve $D_Y$. $D_Y$ is then used as the preconditioner of $Y$. $D_Y$ is in fact a point-Jacobian preconditioner for $Y^\circ$. Hence we call $D_Y$ a transformed point Jacobian preconditioner. $D_Y$ only possesses non-zero elements on three diagonals: the main diagonal and the $N$-th diagonal below/above the main diagonal, where $n$ is the bus count. The inverse of $D_{Y^\circ}$ can be calculated in $O(n)$ time, hence it only incurs very little computation to compute the multiplicative form of $D_Y^{-1}$.

### 6.3.2 Spectra Analysis of $Y$ Matrices

Here we list analysis of spectra over 3 of the selected admittance matrices. We use 3 schemes to transform admittance matrices, and investigate effects of preliminary preconditioners on the spectra. We show the spectra for IEEE39 and 2K system in Fig.6.1 and Fig.6.2, respectively.

Before any preconditioning, the spectra is widely spread, mainly due to large difference on the diagonal part of $Y$. When treated with Scheme 1 (i.e., $[1, -i; i, 1]$), the spectra is mainly on the right side of the imaginary axis. When treated with Scheme 2 (i.e., $[-i, 1; 1, i]$), due to the fact that the matrix is symmetric, the spectrum totally falls on the real axis. When using Scheme 3 (i.e., $[i, -1; 1, i]$), the spectrum is effectively transformed by: swapping real and imaginary part. Due to that the spectrum is widely spread in any scheme, it will generally cause slow convergence by iterative solvers without preconditioning.

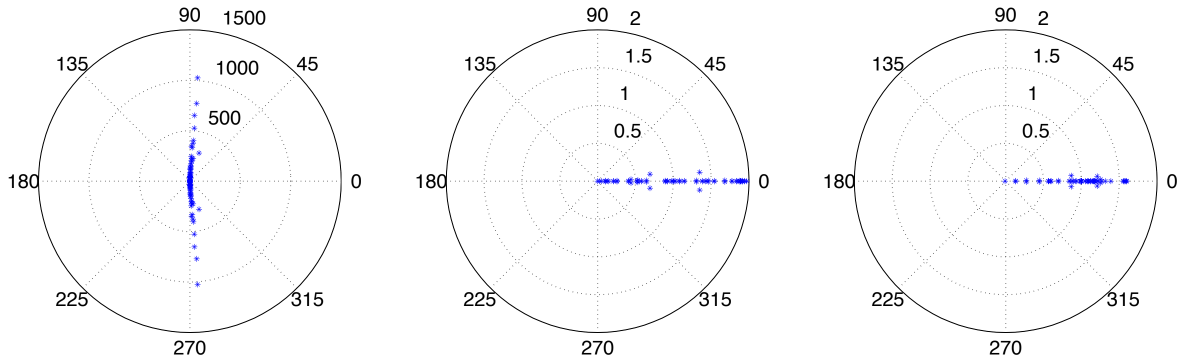
When using diagonal preconditioning, i.e., we treat admittance matrices using firstly the schemes listed above, and secondly, a block-Jacobian preconditioner (in fact, a point-Jacobian preconditioner for the complex admittance matrix). This effectively reduces the modula of the largest eigenvalues of the admittance matrices. But the smallest eigenvalues have become even smaller. Also note that the different schemes now all have little impact on the spectra.

When treating transposed matrices with the Schemes above, combined with an ILU-0 or ILU($10^{-2}$) preconditioner , the result is similar to Jacobian preconditioner. The spectra is now much condensed, with smallest eigenvalues very small. We use ILU($10^{-2}$) for 2K, mainly because very large eigenvalues appear when ILU-0 is used. These eigenvalues are very rare (their total amount is below 10 even for large systems), reflecting the (almost) singularity of ILU preconditioners.

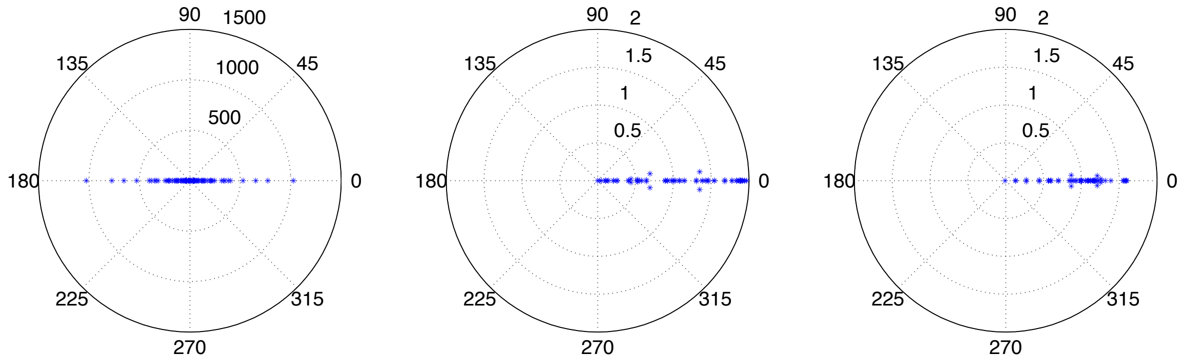
### 6.3.3 Spectra Analysis of Augmented Matrices ($Y + \Delta_Y$)

We evaluate the effect of preconditioners constructed using $Y$ on real TDS matrices, i.e., matrices with the form $Y + \Delta_Y$. We retrieve the $Y + \Delta_Y$ matrices from TDS simulation process. The matrix $Y$ is effectively transformed as: $\begin{bmatrix} Re(Y) & Im(Y) \\ -Im(Y) & Re(Y) \end{bmatrix}$. Aforementioned preconditioners are used in together with GCRO-DR(or GMRES) for TDS cases. The used cases are listed in Tab.6.1 (0.01s time stepping is used for all cases).

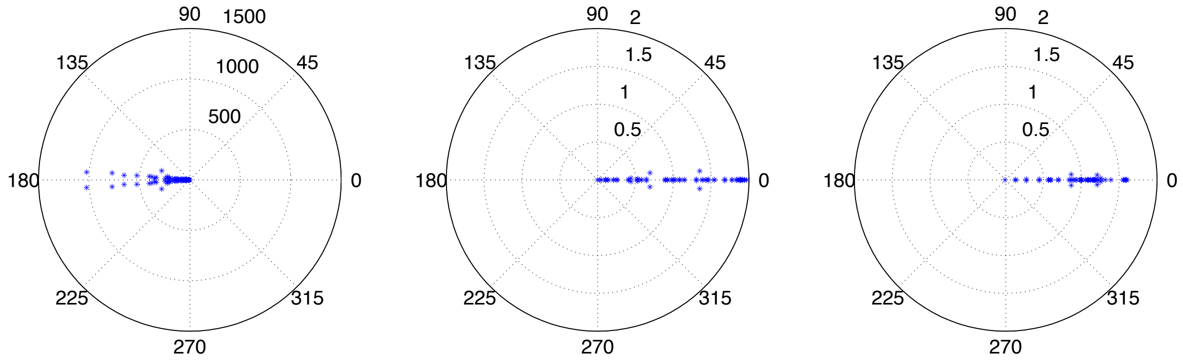
Tab.6.2, Tab.6.3 and Tab.6.4 show the largest and smallest eigenvalues (in magnitude) of selected augmented $Y$ matrices treated with ILU preconditioner derived from $Y$, for case 2K-STABLE, Dongbei-B and 10188, respectively. Unlike the preconditioned $Y$ matrices which are not augmented with $\Delta_Y$, they usually contain eigenvalues large in magnitude. This is due to that the augmented part, $\Delta_Y$ changes some elements largely, especially



a. Scheme [ 1, -i; i, 1 ]

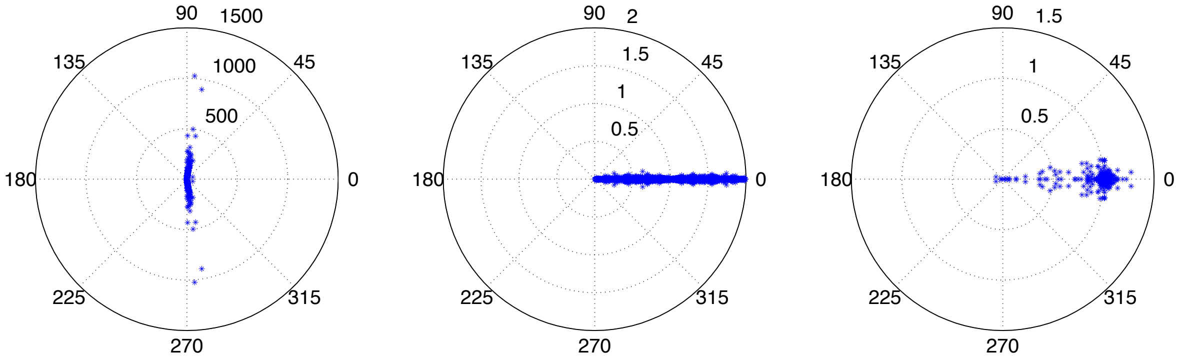


b. Scheme [ -i, 1; 1, i ]

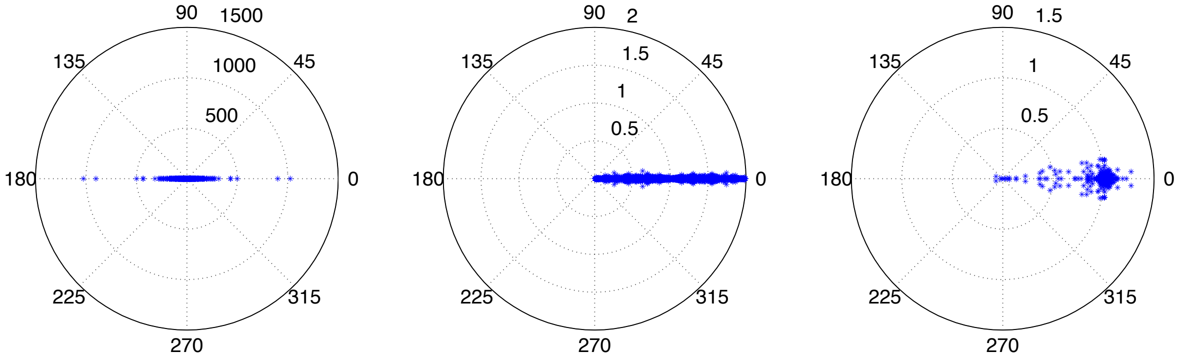


c. Scheme [ i, -1; 1, i ]

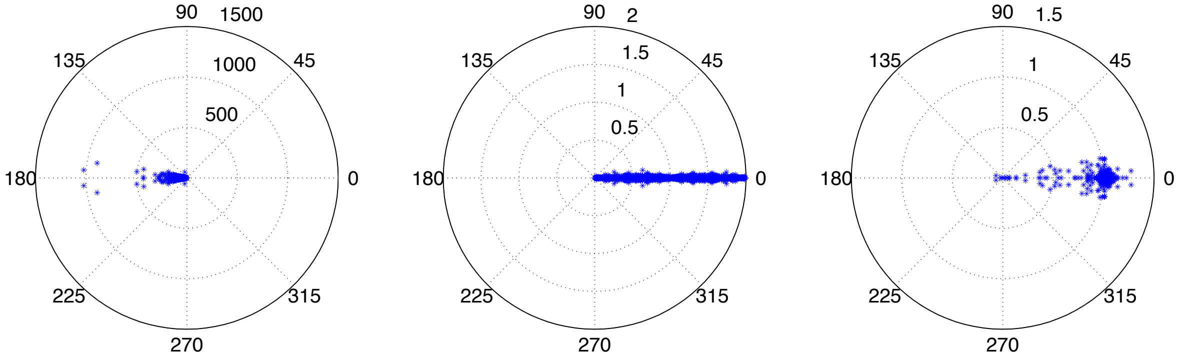
Figure 6.1: Spectra Analysis for IEEE39



a. Scheme [ 1, -i; i, 1 ]



b. Scheme [ -i, 1; 1, i ]



c. Scheme [ i, -1; 1, i ]

Figure 6.2: Spectra Analysis for 2K

| Linear System | 1-st | 200-th | 1500-th |
|---|---|---|---|
| 10 Largest Eigenvalues | 1.500E+2 | 7.574E+3 | 2.024E+6 |
| | 1.414E+2 | 1.500E+2 | -9.480E+5 |
| | 1.224E+2 | 1.414E+2 | 1.504E+2 |
| | 1.056E+2 | 1.224E+2 | 1.392E+2 |
| | 7.257E+1 | 1.056E+2 | 1.227E+2 |
| | 6.214E+1 | 7.257E+1 | 1.060E+2 |
| | 5.917E+1 | 6.214E+1 | 7.257E+1 |
| | 5.889E+1 | 5.917E+1 | 6.251E+1 |
| | 5.417E+1 | 5.889E+1 | 6.101E+1 |
| | 5.061E+1 | 5.417E+1 | 5.916E+1 |
| 10 Smallest Eigenvalues | 5.652E-1 ± 4.612E-2 $i$<br>5.456E-1 ± 5.044E-2 $i$<br>4.672E-1 ± 9.115E-3 $i$<br>3.401E-1 ± 5.580E-4 $i$<br>3.142E-1<br>3.109E-1 | 5.652E-1 ± 4.615E-2 $i$<br>5.466E-1 ± 5.134E-2 $i$<br>4.672E-1 ± 9.129E-3 $i$<br>3.402E-1 ± 3.661E-4 $i$<br>3.142E-1<br>3.110E-1 | 6.122E-1<br>5.790E-1<br>5.229E-1 ± 3.786E-3 $i$<br>4.565E-1<br>4.056E-1<br>3.881E-1<br>3.667E-1<br>2.967E-1<br>2.645E-1 |

Table 6.2: Extreme values in spectra for preconditioned augmented $Y$ matrices for 2K-STABLE case

| Linear System | 1-st | 200-th | 2000-th |
|---|---|---|---|
| 10 Largest Eigenvalues | 2.746E+1 ± 1.771E+1 $i$ | 2.745E+1 ± 1.771E+1 $i$ | 2.745E+1 ± 1.771E+1 $i$ |
| | 4.619E+0 ± 8.715E-1 $i$ | 4.619E+0 ± 8.715E-1 $i$ | 4.623E+0 ± 8.451E-1 $i$ |
| | 3.844E+0 ± 1.128E-1 $i$ | 3.844E+0 ± 1.127E-1 $i$ | 3.844E+0 ± 1.147E-1 $i$ |
| | 3.689E+0 ± 1.045E-1 $i$ | 3.689E+0 ± 1.046E-1 $i$ | 3.689E+0 ± 9.871E-2 $i$ |
| | 3.391E+0 ± 5.235E-1 $i$ | 3.391E+0 ± 5.235E-1 $i$ | 3.392E+0 ± 5.215E-2 $i$ |
| 10 Smallest Eigenvalues | 7.189E-1 ± 1.405E-1 $i$ | 7.189E-1 ± 1.405E-1 $i$ | 7.189E-1 ± 1.405E-1 $i$ |
| | 7.215E-1 ± 5.886E-2 $i$ | 7.215E-1 ± 5.886E-2 $i$ | 7.215E-1 ± 5.866E-2 $i$ |
| | 5.291E-1 ± 1.201E-1 $i$ | 5.291E-1 ± 1.201E-1 $i$ | 5.291E-1 ± 1.201E-2 $i$ |
| | 4.648E-1 ± 6.673E-2 $i$ | 4.648E-1 ± 6.673E-2 $i$ | 4.648E-1 ± 6.659E-2 $i$ |
| | 3.285E-1 ± 5.655E-2 $i$ | 3.285E-1 ± 5.655E-2 $i$ | 3.285E-1 ± 5.643E-2 $i$ |

Table 6.3: Extreme values in spectra for preconditioned augmented $Y$ matrices for Dongbei-B case

| Linear System | 1-st | 200-th | 1000-th |
|---|---|---|---|
| 10 Largest Eigenvalues | 1.624E+2 ± 8.180E+1 $i$ | 1.625E+2 ± 8.069E+1 $i$ | 1.622E+2 ± 8.130E+1 $i$ |
| | -0.513E+1 | -5.088E+1 | -5.088E+1 |
| | 0.439E+1 | 4.365E+1 | 4.365E+1 |
| | 0.305E+1 | 3.038E+1 | 3.038E+1 |
| | 0.224E+1 | 2.223E+1 | 2.223E+1 |
| | 0.195E+1 ± 6.712E+0 $i$ | 1.946E+2 ± 6.687E+0 $i$ | 1.946E+1 ± 6.687E+0 $i$ |
| | 0.194E+1 ± 1.600E+0 $i$ | 1.937E+2 ± 1.591E+0 $i$ | 1.937E+1 ± 1.592E+0 $i$ |
| 10 Smallest Eigenvalues | 7.342E-2 ± 1.447E-2 $i$ | 7.329E-2 ± 1.463E-2 $i$ | 7.329E-2 ± 1.470E-2 $i$ |
| | 1.087E-1 ± 1.611E-2 $i$ | 1.086E-1 ± 1.618E-2 $i$ | 1.086E-1 ± 1.631E-2 $i$ |
| | 1.276E-1 ± 1.373E-2 $i$ | 1.275E-1 ± 1.376E-2 $i$ | 1.275E-1 ± 1.413E-2 $i$ |
| | 1.669E-1 ± 2.583E-2 $i$ | 1.669E-1 ± 2.588E-2 $i$ | 1.669E-1 ± 2.587E-2 $i$ |
| | 2.004E-1 ± 2.949E-2 $i$ | 2.003E-1 ± 2.962E-2 $i$ | 2.003E-1 ± 2.961E-2 $i$ |

Table 6.4: Extreme values in spectra for preconditioned augmented $Y$ matrices for 10188 case

diagonal ones. Besides, many other eigenvalues are changed too, but in a less significant way, as shown in the difference between the eigenvalues which can be easily matched across matrices. For example, for 2K-STABLE, some large eigenvalues appear in the first matrix, and 1 or 2 very large eigenvalues appears in the 200th and the 1500th matrix. Especially for the 1500th matrix, a very large and negative eigenvalue appears. The smallest eigenvalues in the augmented matrices also change, but in these systems, none of the eigenvalues are very close to 0. Similar scenario happens for Dongbei-B and 10188 too, with eigenvalues of extreme magnitude appears for (preconditioned) $Y + \Delta_Y$ matrices.

Changed eigenvalues may have negative effect on the convergence of iterative solvers, especially when the extreme ones are very large or small in magnitude. In this sense, $Y$ matrix before augmentation may be well preconditioned by off-line analysis, but when augmented with $\Delta_Y$, its convergence properties may be compromised due to spectra changes.

## 6.4 Deflation in TDS

As mentioned before, both GMRES-E and GMRES-DR require a constant matrix $A$ in the linear systems, otherwise convergence properties would be compromised. For TDS, we have both the matrix and the right hand side changing from iteration to iteration. We expect limited applicability of GMRES-E and GMRES-DR to our problem. Hence we adopt the GCRO-DR [69] to tackle both changing matrix and right-hand sides. As a reference, we list the outline of GCRO-DR algorithm. We use $A$ to represents the preconditioned linear system.

```
Input: Matrix $A$, RHS $b$
       Parameter $n$, $m$, and $k$
       Deflation vectors $U$ (undefined for the first linear system)
Output: Solution of the linear system, $x$
        Deflation vectors $U$ for the next system
$r = b$
if $U$ is defined then
    QR factorize matrix $AU$ ;
    $C = Q$ ;
    $U = U/R$ ;
    $x = UC^H r$ ;
    $r = r - CC^H r$ ;
else
    Carry out GMRES on $Ax = b$ with $m$ as restart value ;
    Retrieve new solution and residue: $x$ and $r$ ;
    Solve eigenvalue problem for matrix $(H_m + h^2_{m+1,m} H_m^{-H} e_m e_m^H)$ ;
    Find eigenvectors of the $k$ smallest eigenvalues, denote $P$ ;
    $V = V_m P$ ;
    QR factorize matrix $\bar{H}_m V$ ;
    $C = V_{m+1} Q$ ;
    $U = V/R$ ;
end
$i = 0$ ;
$r_0 = r$ ;
$x_0 = x$ ;
while $\|r\|_2 > \epsilon$ do
    $i = i + 1$ ;
    Perform Arnoldi process for $m - k$ steps, with matrix $(I - CC^H)A$, and $v_1 = \frac{r_0}{\|r_0\|_2}$, to
    retrieve $V_{m-k+1}$, $\bar{H}_{m-k}$, and $B_{m-k} = C^H A V_{m-k}$ ;
    Normalize columns of $U$ to retrieve $\tilde{U}$ and scaling matrix $D$: $\tilde{U} = UD$ ;
    $V_m = \begin{bmatrix} \tilde{U} & V_{m-k} \end{bmatrix}$ ;
    $W_{m+1} = \begin{bmatrix} C & V_{m-k+1} \end{bmatrix}$ ;
    $\bar{G}_m = \begin{bmatrix} D & B_{m-k} \\ & \bar{H}_{m-k} \end{bmatrix}$ ;
    Solve $min\|W_{m+1}^H r_{i-1} - \bar{G}_m y\|_2$ for $y$ ;
    $x_i = x_{i-1} + V_m y$ ;
    $r_i = r_{i-1} - W_{m+1} \bar{G}_m y$ ;
    Solve a generalized eigenvalue problem: $(\bar{G}_m^H \bar{G}_m)x = \lambda(\bar{G}_m^H W_{m+1}^H V_m)x$ ;
    Find eigenvectors of the $k$ smallest eigenvalues, store in $P$ ;
    $V = V_m P$ ;
    QR factorize matrix $\bar{G}_m P$ ;
    $C = W_{m+1} Q$ ;
    $U = V/R$ ;
end
Return solution $x$ and deflation vector set $U$ ;
```

**Algorithm 9**: GCRODR

Unlike GMRES-E or GMRES-DR which require the deflation vectors to be an invariant subspace of the matrix, GCRO-DR stores 2 matrices $U$ and $C$ and has the following requirements:

$$\begin{aligned} A \times U &= C \\ C^H \times C &= I \end{aligned} \tag{6.6}$$

$U$ is a collection of (column) deflation vectors. $U$ and $C$ are both of size $n \times k$, where $k$ is the deflation vector count.

When deflation vectors are not available, such as the scenario of solving the first linear system, an Arnoldi process is carried out by solving an GMRES process with only one restart, to retrieve basic information about Rits values and vectors (line 9 to 16). Otherwise when such deflation vectors are available as $U$ matrix, it is updated and the corresponding matrix of $C$ is constructed to maintain the requirements in Eqs.6.6 (line 3 to 7). As the main body, GCRO-DR has an outer-inner iteration scheme like FGMRES. The inner iteration is an Arnoldi process (usually in the form of GMRES) with a constant restart value (line 23). At each restart of the inner GMRES, Hessenberg matrix $\bar{H}_{m-k}$ on line 23 is retrieved to form a larger, general eigenvalue problem. Also newly generated Krylov subspace bases recorded in $V_{m-k+1}$ on line 23, is combined with old deflation vectors ($U$). Finally Rits values and vectors are generated based on the combined matrix of $\bar{G}_m$ (defined on line 27) by solving a generalized eigenvalue problem. Also the constraint defined in Eqs.6.6 is kept.

GCRO-DR is mainly governed by 3 parameters: (1) $m$, the total count of the vectors in the outer iteration of GCRO-DR, where deflation vectors and newly generated bases are combined as $V_m$ in line 25, (2) $k$, the count of the deflation vectors, and (3) $s$, the size of the vector space dedicated to newly generated Krylov subspace bases, i.e., $V_{m-k}$ on line 23 and 25. Note that $m = k + s$.

### Updates of $U$ and $C$ at Startup

When calling GCRO-DR with deflation vector $U$ available from previous runs, one has to make sure that the relations in Eqs.6.6 still hold. If the linear operator $A$ does not change, there is no need to change $U$ or $C$. Otherwise if $A$ is changed since the solution of last system with GCRO-DR, one has to go through the lines of 3 to 7 in GCRO-DR algorithm.

If new operator $A^*$ can be written in a form as additive change to previous operator $A$: $A^* = A + \Delta$, the calculation of line 3 can be greatly simplified. Now we have:

$$A^*U = AU + \Delta \times U = C + \Delta \times U \tag{6.7}$$

If $\Delta$ has much lower rank/sparsity, computing of $A^*U$ can be much easier.

In the TDS simulation we formulate each linear system as: $Y + \Delta_Y^{(i)}$, in which $i$ denote the $i$-th linear system. Hence when solving the $(i+1)$-th linear system, the matrix can be written as:

$$Y + \Delta_Y^{(i+1)} = (Y + \Delta_Y^{(i)}) + (\Delta_Y^{(i+1)} - \Delta_Y^{(i)}) \tag{6.8}$$

Also preconditioners are always applied, with left preconditioner denoted as $M_l$, and right preconditioner $M_r$. Hence the $\Delta \times U$ in the right side of Eqs.6.7 can be written as:

$$M_l^{-1} \times (\Delta_Y^{(i+1)} - \Delta_Y^{(i)}) \times M_r^{-1} \times U \tag{6.9}$$

In TDS, $\Delta_Y^{(i)}$ is very sparse and not full rank. Yet it is definitely not very low in rank (due to the fill-in's caused by dynamic parts). We choose to combine the operations in Eqs.6.9 from right to left: firstly calculating $M_r \backslash U$, then applying $(\Delta_Y^{(i+1)} - \Delta_Y^{(i)})$, and

| | Configuration of $A$ (based on $A^0$) | | Configuration of $C$ |
|---|---|---|---|
| Data Type | Real | | Complex |
| Preconditioner | None | ILU | N/A |
| Operator | SpMV | SpMV + fwd/bwd substitution | MV |
| Operation Count | $2 \cdot nnz_A$ | $2 \cdot (nnz_A + nnz_L + nnz_U)$ | $12kn$ |

Acronyms:

**SpMV** : SParse Matrix-dense Vector multiplication

**MV** : dense Matrix-dense Vector multiplication

$nnz_A$ : Number of Non-Zero elements in sparse matrix $A$

Table 6.5: Comparison of $A$ and overhead in applying $(I - CC^H)$

finally applying $M_l^{-1}$. The reason for the ordering is that we expect the structural rank of $(\Delta_Y^{(i+1)} - \Delta_Y^{(i)})$ to be much higher than the column count of $U$. Computation order starting from $U$ would benefic overall computation amount.

### Implementation Details – Inner Arnoldi Process

Note that the inner Arnoldi process is carried out on the matrix of $(I - CC^H)A$, which is a more complex linear operator than the original $A$, which might imply preconditioning operations. Suppose that left preconditioner $(M_l)$ and right preconditioner $(M_r)$ are both used, while the original unpreconditioned matrix is $A^0$. Hence $A = M_l \backslash A^0 / M_r$. The iterations with $(I - CC^H)A$ should avoid generating the matrix of $CC^H$ due to memory usage concerns. Then generating the next Krylov subspace basis based on vector $v$ should be carried out in the following steps:

- Apply $M_r$ to $v$
- Apply $A^0$ to $M_r \backslash v$
- Apply $M_l$ to $A(M_r \backslash v)$, denote the outcome as $u$
- Apply $C^H$ to $u$
- Apply $C$ to $C^H u$, denote the outcome as $w$
- Compute $u - w$, which is $(I - CC^H)Av$

### Computational Characteristics compared with (deflated) GMRES

The change of linear operator from $A$ to $(I - CC^H)A$ incurs per-iteration overhead. If $A$ is not preconditioned and $A$ has a lower per-row non-zero element count (, e.g., comparable to $k$), then deflation will incur a comparatively high overhead, since data in $C$ would be accessed twice. In this case, in order to lower the total amount of computation, deflation should achieve reduction of iteration count by a large extent to offset the overhead it causes. Otherwise if $A$ contains non-trivial preconditioning operations such as ILU, the inherent computational complexity of operator $A$ is already high, hence potential problem of the overhead introduced by deflation is mitigated.

### 6.4.1 Eigenvalue Choices

On line 12 and 32 in GCRO-DR, the $k$ eigenvalues of the smallest magnitude are chosen for deflation. This is based on the observation that in many cases, especially for SPD matrices, "smallest" eigenvalues usually cause convergence problems.

From the analysis above for augmented matrices used in TDS, we know that "large" eigenvalues always appear for the preconditioned system, despite the fact that the preconditioned system using the non-augmented transformed admittance always have eigenvalues within a certain, small magnitude. These eigenvalues might be on the negative half of the complex plane, and their effect on convergence could be much more severe than the ones with the same magnitude but on the positive axis. Hence for the augmented matrices, we choose to deflate both smallest and largest eigenvalues.

In this subsection we first briefly review the classical convergence theory of Krylov iterative solvers, and point out the problem we face with non-normal matrices in TDS. Next we design heuristics for the selection of Rits values for deflation.

#### Convergence Behavior versus Matrix Spectra

For classical problems such as Hermitian positive definite matrices, simple yet elegant theoretical convergence bounds exist, such as arguably the most popular one in [52] shown in Eqs.6.10, or the less referenced yet tighter bound shown in Eqs.6.11.

$$\frac{\|x_k - x_0\|_A}{\|x - x_0\|_A} < 2\left(\frac{\sqrt{\kappa(A)} - 1}{\sqrt{\kappa(A)} + 1}\right)^k \tag{6.10}$$

In Eqs.6.10, $\|\cdot\|_A$ is the $A$-norm, and $\kappa(A)$ is the condition number of $A$, i.e., $\frac{\lambda_{max}}{\lambda_{min}}$. Using this boundary, we can simplify the problem of Rits value choices by evaluating the minimum condition number as a result of deflation. Suppose that $k$ Rits values are chosen from $l$: $\lambda_1 \leq \cdots \leq \lambda_l$. Then we have $k+1$ candidate configurations for deflation to choose from, with the $i$-th configuration is to deflate eigenvalues corresponding to the Rits value set: $\{\lambda_1 \leq \cdots \leq \lambda_{i-1}, \lambda_{l-k+i} \leq \cdots \leq \lambda_l\}$. The effective condition number for the $i$-th configuration is: $\kappa_A^{(i)} = \frac{\lambda_{l-k+i-1}}{\lambda_i}$. We have to find the configuration, i.e., number of $i$ to minimize $\kappa_A^{(i)}$, which involves very simple evaluation of $k+1$ arithmetic divisions and comparison operations. Since Eqs.6.10 is an upperbound for the more strict bound as defined in Eqs.6.11, the heuristics listed above is not guaranteed to be optimal in this sense.

$$\frac{\|x_k - x_0\|_A}{\|x - x_0\|_A} \leq \min_{\substack{p(0)=1 \\ deg(p)=k}} \max_{1\leq j\leq n} |p(\lambda_j)| \tag{6.11}$$

For generally non-normal matrices as the augmented admittance matrices used in TDS, there exists convergence bound described in [73], as shown in Eqs.6.12.

$$\frac{\|r_n\|}{\|r_0\|} = \min_{\substack{p(0)=1 \\ deg(p)=k}} \frac{\|Vp(\Lambda)V^{-1}r_0\|}{\|r_0\|} \leq \kappa(V) \min_{\substack{p(0)=1 \\ deg(p)=k}} \max_{1\leq j\leq n} |p(\lambda_j)| \tag{6.12}$$

Where $V$ is the eigenvector matrix. Due to that retrieving the full $V$ matrix is intractable, hence is the analysis of the spectrum of $V$. Due to the complex formulation of Eqs.6.12, it is hardly used in practice for convergence analysis. Especially when Arnoldi process only generates several candidate Rits values, we have to introduce heuristics for choosing among them for deflation, by taking into consideration of the spectra of non-normal matrices.

### Criteria in Choosing Deflation Eigenvalues

Given a non-normal, full rank real matrix, such as the (preconditioned) matrix $Y + \Delta_Y$ in TDS, the eigenvalues might exist on any place on the complex plane except the origin. The only restriction is that the spectrum is symmetric with respect of the real axis. Also we do not take any assumption of the right hand sides. We use $r$ to denote the magnitude of the complex number ($r > 0$), and $\theta$ the angle of it ($-\pi < \theta \leq \pi$). Then the heuristics for choosing the eigenvalues should satisfy the following criteria:

- Give eigenvalues that are close to the origin ($r \ll 1$) high priority
- Give eigenvalues that are very large in magnitude ($r \gg 1$) high priority
- Ignore eigenvalues those are close to (1,0) on the complex plane

We define an evaluation function $f$ which takes a complex number ($\dot{a}$) as parameter, and yields a real number for the evaluation of the preference when $\dot{a}$ is considered for deflation. Then $f$ should satisfy the following criteria:

- $\arg\min\limits_{\dot{a}} f(\dot{a}) = 1 + 0i$
- $f(\dot{a}) > f(\dot{b})$, when: $r(\dot{a}) > r(\dot{b}) \gg 1$
- $f(\dot{a}) > f(\dot{b})$, when: $r(\dot{a}) < r(\dot{b}) \ll 1$

These criteria are formal definitions for previous descriptions for the standards for choosing deflation eigenvalues.

### Heuristics for Eigenvalue Selection

Based on the criteria above. Here we design the heuristics for choosing eigenvalues based on eigenvalue positions on the complex plane:

$$f(\dot{a}) = f(r, \theta) = \left(\frac{1}{r} + r^{\frac{1}{3}}\right)\left(\sin\frac{|\theta|}{2} + (r^{\frac{1}{3}} - 1)^2\right) \tag{6.13}$$

It is easily verifiable that: (1) $f(r, \theta)$ is a continuous, differentiable function on the complex plane, and (2) it satisfies the formal requirements mentioned before.

When $|r| \ll 1$, $f(r, \theta)$ approaches that of $\frac{1}{r}$. When $|r| \gg 1$, $f(r, \theta)$ approaches $r$, which is caused by the exponent of $\frac{1}{3}$. The purpose of assuring that $f(r, \theta)$ approaches $r$ when $r$ is large is to locate a proper trade-off between very small eigenvalues and very large ones. Note that the definition of condition number for SPD matrices is $\frac{\lambda_n}{\lambda_1}$ where $\lambda_n$ and $\lambda_1$ are the largest and the smallest eigenvalue, respectively. Note the most famous convergence bound for CG methods is defined over condition number. We expect the behavior of $f(\dot{a})$ is coherent with the definition of condition number, hence the classical theory for CG convergence bound. The topology of the heuristics on the complex plane is as plotted in Fig.6.3. Note that the axis for the value of $f(\dot{a})$ is in logarithmic scale.

We apply this heuristics to the choice of Rits values for deflation for the preconditioned $Y + \Delta_Y$ matrices arisen in TDS.

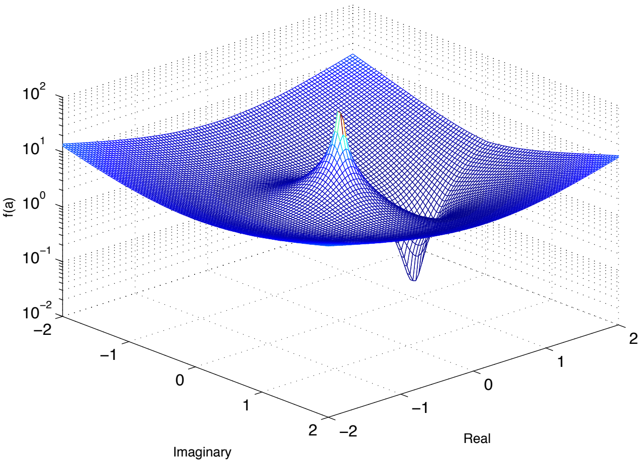


Figure 6.3: Heuristics function $f(\dot{a})$ on the complex plane

## 6.5 Combining Deflation and Preconditioner Updates

Preconditioner updates in Chapter 5 and Deflation in this chapter are both separate techniques for speeding up the iteration of solving a sequence of linear systems. However, they represent approaches from different perspectives. Potentially they can have a combined effect on the reduction of iteration counts. In this section we propose the use of both in TDS. The preconditioner update is restated as follows ($L^*U^*$ is the $LU$ decomposition of $(U + \Delta_Y)$):

$$Y \overset{❶}{\approx} LU \tag{6.14}$$

$$Y + \Delta_Y \overset{❶}{\approx} LU + \Delta_Y \tag{6.15}$$

$$LU + \Delta_Y = L(U + L^{-1}\Delta_Y) \tag{6.16}$$

$$\overset{❷}{\approx} L(U + \Delta_Y) \tag{6.17}$$

$$\overset{❻}{\approx} LL^*U^* \tag{6.18}$$

The difference when preconditioner update is introduced is that the linear operators change with a different formulation. More complex computation is involved when these two techniques are used together. Note that for each GCRO-DR iteration, if a static, non-changing preconditioner is used, the process of computing $C_{new} = AU_{old}$ can be simplified to an additive form:

$$C_{new} = M_l^{-1}\Delta_A M_r^{-1}U_{old} + C_{old} \tag{6.19}$$

But with updated preconditioners, i.e., updated $M_l$ and/or $M_r$, we have:

$$C_{new} - C_{old} = M_{l_{new}}^{-1} A_{new} M_{r_{new}}^{-1} U_{old} - M_{l_{old}}^{-1} A_{old} M_{r_{old}}^{-1} U_{old} \tag{6.20}$$

$$\overset{❻}{=} M_l^{-1}\Big((A + \Delta_{new})(M_r + \Delta_{new})^{-1} - (A + \Delta_{old})(M_r + \Delta_{old})^{-1}\Big)U_{old} \tag{6.21}$$

Note that we use $M_l$ and $M_u$ to denote left and right preconditioners, instead of the conventional notation of $L$ and $U$, mainly to distinguish from the notation of $U$ as deflation vectors in GCRO-DR.

We can see that instead of the simple form in Eqs.6.19, we have a non-trivial form of Eqs.6.20, in which: $(M_r + \Delta_{old})^{-1}U_{old}$, $(M_r + \Delta_{old})^{-1}U_{new}$ have to be carried out with the application of both $A + \Delta_{old}$ and $A + \Delta_{new}$. This is mainly due to the difference between the inverse operator of $(M_r + \Delta_{old})^{-1}$ and $(M_r + \Delta_{new})^{-1}$. This brings extra computation overhead if preconditioner updates are used in combination with deflation.

Instead of the additive form of Eqs.6.20, we shall use the formulation as $C_{new} = AU_{old}$, as shown in Eqs.6.22.

$$C_{new} = M^{-1}_{l_{new}} A_{new} M^{-1}_{r_{new}} \times U_{old} \tag{6.22}$$

$$\overset{\text{❻}}{=} M_l^{-1}\Big((A + \Delta_{new})(M_r + \Delta_{new})^{-1}\Big)U_{old} \tag{6.23}$$

Compared with Eqs.6.21, Eqs.6.23 is both simpler in formulation and computation. Hence we use Eqs.6.23 for implementation.

## 6.6 Experiment Results

In this section we test the effect of deflation by using GCRO-DR on the augmented $Y$ matrices with right hand sides retrieved from the TDS simulation process of each test case listed in Tab.6.1. To evaluate the effect on deflation, we compare it with GMRES based solving process. Preconditioner configuration used for GCRO-DR and GMRES are kept the same.

For GCRO-DR, 2 parameters have to be decided: $m$ and $k$. Parameter $m$ is the total vector space size, including both deflation vector space and newly generated Krylov subspace, while $k$ is the size of the deflation vector space. We use $s$ to denote $m - k$, which is the size of the newly generated Krylov subspace, i.e., the restart value of the inner GMRES/Anorldi process.

We evaluate the basic setting for $m$ and $k$ in Section 6.6.1 by comparing GCRO-DR and GMRES with the same vector space size. Extra investigation include experiments with 2 scenarios: (1) when the total vector space size is fixed, i.e., $m$ is fixed, we test the trade-off between $k$ (the size of the deflation vector space size) and $s$ (the space size for newly generated Krylov subspace bases); (2) when the newly generated Krylov subspace size $s$ is fixed, we test the effect of introducing more deflation vectors, i.e., increasing $k$ hence $m$. In these tests, we only deflate the eigenvalues of the smallest magnitude. We denote the first scenario as Subspace Utilization test covered in Section 6.6.2, and the second one as Progressive Deflation test covered in Section 6.6.3.

To further investigate the effect of deflation, we use Principle Angle Analysis, to analyze the relationship between deflation vector space and the actual eigenvector space. By computing the cosine value of the Principal Angles between these two spaces, we grasp the actual effect on the convergence of Rits values/vectors. Section 6.6.6 covers the Principal Angle analysis results for a specific TDS case.

Section 6.6.5 includes the results for deflating both large and small eigenvalues by the heuristics introduced in Section 6.4.1. Thorough the simulation process, the choice for the eigenvalues are not static. Principle Angle analysis shows the deflation process

| Case | Solver | SpMV count | Averaged SpMV Count |
|---|---|---|---|
| IEEE39-v | GMRES(40) | 37690 | 102.1 |
| IEEE39-v | GCRO-DR(40,20) | 12651 | 34.3 |
| Dongbei-B | GMRES(40) | 173828 | 44.6 |
| Dongbei-B | GCRO-DR(40,20) | 122084 | 31.3 |
| 2K-STABLE | GMRES(40) | 246471 | 125.1 |
| 2K-STABLE | GCRO-DR(40,20) | 208759 | 106.0 |
| 10188 | GMRES(40) | 312409 | 308.7 |
| 10188 | GCRO-DR(40,20) | 729206 | 720.6 |
| 10188 | GMRES(60) | 230878 | 228.1 |
| 10188 | GCRO-DR(60,20) | 204501 | 202.1 |

Table 6.6: GMRES and GCRODR – Basic Comparison

achieves locating accurate eigenvectors on both ends of the spectrum. Finally, Section 6.6.6 includes the evaluation of Preconditioner Updates (PU) and deflation. These two techniques have a combined effect on convergence. Lower iteration count is achieved when compared with PU or deflation alone.

### 6.6.1 Basic Comparison with GMRES

The restart value for GMRES is set as the total vector space size used for GCRO-DR. For all the cases, we use 40 as the $m$ value for GMRES and GCRO-DR, we use 20 as the $k$ value for GCRO-DR. For 10188 case, we also test larger $m$ values. The results are shown in Tab.6.6. For IEEE39-v, the convergence is drastically improved, mainly due to the small size of the system and the count of extreme eigenvalues are small compared to the value of $k$ and $s$ in GCRO-DR. Dongbei and 2K are both mid-sized power grid models. For their corresponding cases, i.e., Dongbei-B and 2K-STABLE, GCRO-DR can reduce the total iteration count by $13 \sim 19$ per linear system, which account for about 30% and 15% of the total iteration counts, respectively.

When $m$ is 40, except for 10188, other cases all experienced a drop in iteration count when GCRO-DR is used instead of GMRES. The reason why the 10188 case shows a rise in iteration count when using GCRO-DR(40,20) compared with GMRES(40) is that with 20 as the restart value for the inner Arnoldi process for GCRO-DR, stagnation is much more significant compared with GMRES(40) which has double size in Arnoldi process. Even if deflation is used, slowing down due to the bad spectrum for the effective deflated system results in large iteration count. When using larger spaces by setting $m$ to 60, even if the same value of $k$ is used, compared with GMRES(60), GCRO-DR(60,20) shows a 10% reduction in iteration count.

### 6.6.2 Subspace Utilization

With a fixed size for total subspace size, $m$, increasing the subspace size used for deflation will in effect decrease the size of the subspace used for new bases. With a small value of $k$, too few eigenvalues are deflated to have a profound effect on decreasing the iteration count. But with a value of $k$ which is too large, there is not enough space to contain newly generated bases, resulting in potential slowdown or even stagnation. The results in Tab.6.7 are consistent with the analysis.

| Case | Solver | SpMV count | Averaged SpMV Count |
|---|---|---|---|
| IEEE39-v | GCRO-DR(40,5) | 13579 | 36.8 |
| IEEE39-v | GCRO-DR(40,10) | 10578 | 28.7 |
| IEEE39-v | GCRO-DR(40,20) | 6580 | 17.8 |
| IEEE39-v | GCRO-DR(40,30) | 4915 | 13.3 |
| IEEE39-v | GCRO-DR(40,35) | 5917 | 16.0 |
| Dongbei-B | GCRO-DR(40,5) | 130292 | 33.4 |
| Dongbei-B | GCRO-DR(40,10) | 123774 | 31.7 |
| Dongbei-B | GCRO-DR(40,20) | 122084 | 31.3 |
| Dongbei-B | GCRO-DR(40,30) | 138891 | 35.6 |
| Dongbei-B | GCRO-DR(40,35) | 204365 | 52.4 |
| 2K-STABLE | GCRO-DR(40,5) | 217106 | 110.2 |
| 2K-STABLE | GCRO-DR(40,10) | 200425 | 101.7 |
| 2K-STABLE | GCRO-DR(40,20) | 208759 | 106.0 |
| 2K-STABLE | GCRO-DR(40,30) | 369860 | 187.8 |
| 2K-STABLE | GCRO-DR(40,35) | 799484 | 405.8 |
| 10188 | GCRO-DR(60,5) | 221356 | 218.7 |
| 10188 | GCRO-DR(60,10) | 196743 | 194.4 |
| 10188 | GCRO-DR(60,20) | 204501 | 202.1 |
| 10188 | GCRO-DR(60,30) | 225819 | 223.1 |
| 10188 | GCRO-DR(60,40) | 138622 | 137.0 |
| 10188 | GCRO-DR(60,50) | 297387 | 293.9 |

Table 6.7: Comparison of GCRO-DR and GMRES– constant vector space size

For further comparison and analysis. We carry out experiments on how the size of the deflation vector space affects overall iteration count. We use GCRO-DR(40,k) to evaluate the iteration count, by varying the value of $k$ from 5 to 35. We expect that with a smaller $k$, average iteration count would be high due to that not much info is kept between iterations, and the scenario will be similar to the GMRES run without deflation. We also expect that with a very large $k$ (close to $m$, which is 40 here), the iteration count would be also high, due to that even if much info is kept from previous runs, the Arnoldi process will have to restart very frequently, resulting in a higher convergence bound, and even stagnation. We expect to witness an “optimal” value for the size of deflation vector space, which yields the lowest iteration count. From the test results shown in Tab.6.7 we can see for IEEE39-v, Dongbei-B, and 2K-STABLE, such optimal value exists. The larger the system is, the smaller the optimal value is. For IEEE39-v, the value is between 20 and 35, around 30. For Dongbei-B, the values are between 10 and 30, close to 20. For 2K-STABLE, the value is between 5 and 20, around 10.

### 6.6.3 Progressive Deflation

We fix GCRO-DR restart value $s$, i.e., the size of the newly generated Krylov subspace, and vary the size of the deflation space. Hence $m = m' + k$. The purpose of this experiment is to observe the effect of introducing more deflation vectors on convergence behavior.

Another experiment is carried out to evaluate the effect of adding extra deflation vector space on iteration count. We use IEEE39-v, Dongbei-B and 2K-STABLE and test iteration count of GCRO-DR(20+$k$,$k$) by varying the value of $k$ from 5 to 35. We

| Case | Solver | SpMV count | Averaged SpMV Count |
|---|---|---|---|
| IEEE39-v | GCRO-DR(25,5) | 15040 | 40.8 |
| IEEE39-v | GCRO-DR(30,10) | 10989 | 29.8 |
| IEEE39-v | GCRO-DR(40,20) | 6580 | 17.8 |
| IEEE39-v | GCRO-DR(50,30) | 5647 | 15.3 |
| IEEE39-v | GCRO-DR(55,35) | 5505 | 14.9 |
| Dongbei-B | GCRO-DR(25,5) | 150076 | 38.5 |
| Dongbei-B | GCRO-DR(30,10) | 135422 | 34.7 |
| Dongbei-B | GCRO-DR(40,20) | 122084 | 31.3 |
| Dongbei-B | GCRO-DR(50,30) | 118829 | 30.5 |
| Dongbei-B | GCRO-DR(55,35) | 118718 | 30.4 |
| 2K-STABLE | GCRO-DR(25,5) | 289983 | 147.2 |
| 2K-STABLE | GCRO-DR(30,10) | 244178 | 124.0 |
| 2K-STABLE | GCRO-DR(40,20) | 208759 | 106.0 |
| 2K-STABLE | GCRO-DR(50,30) | 201768 | 102.4 |
| 2K-STABLE | GCRO-DR(55,35) | 200184 | 101.6 |

Table 6.8: Comparison of GCRO-DR and GMRES– constant non-deflation vector space size

expect diminishing iteration count with larger value of $k$. The extreme of $k$ being $n$ is that the iteration will converge in 1 iteration, but with prohibitively high overhead of computing and storing the eigenvectors. Experiments give coherent results, as shown in Tab.6.8. But the effect of some extra deflation subspace diminishes as the total size of the deflation subspace grows. Introducing too many deflation vectors would introduce too much overhead per iteration due to inner products, which is a performance issue and affects overall execution time even if iteration count is lowered.

### 6.6.4 Principal Angle Analysis

The effect of deflation depends on how well the predicted eigenvectors match the actual eigenvectors. If they match each other, their associated eigenvalues are in effect deflated from the linear system used for Krylov iterations.

One major criterion for evaluating the accuracy of eigenvectors is through observing directly the relationship between the space consisting of the deflation vectors and that of the desired eigenvectors, by calculating the principal angles [51] between these two spaces.

The subspace defined by deflation vectors in $U$ is denoted as $\mathcal{U}$, while that formed by the eigenvectors of the $l$ smallest eigenvalues is denoted as $\mathcal{G}_l$. We observe the principal angles between space $\mathcal{U}$ and $\mathcal{G}_l$. We evaluate these angles by their cosine values. With larger $l$, when the cosine value shows visible decrease from 1, the quality of $\mathcal{U}$ as an approximation to $\mathcal{G}$ is lost. We can also estimate how many eigenvalues are effectively removed by deflation.

#### Start-up Process

When GCRO-DR is carried out on the first linear system, a boot-up process with GMRES is carried out to set the initial values for Rits value and Rits vectors. Initial values of $U$ and $C$ are set. Afterwards, GCRO-DR iterations are carried out. At each restart of the inner Arnoldi process of GCRO-DR, $U$ and $C$ are updated. Initially, the quality of Rits values and $U$ are poor as estimations to the eigenvalues and eigenvectors to the matrix. This can be reflected by large principal angles between $\mathcal{U}$ and $\mathcal{G}_l$. As GCRO-DR progresses, $\mathcal{U}$ gradually becomes better estimation of subspace generated by eigenvectors of the smallest eigenvalues.

Here we record the cosine of the principal angles between $\mathcal{U}$ and $\mathcal{G}_l$ at each restart of the Arnoldi process for GCRO-DR. 2K-STABLE is used as an example. GCRO-DR configurations are: $m = 40$, $k = 20$, ILU-C($10^{-2}$) preconditioner for $Y$ is used. Results are shown in Tab.6.9.

For simplicity, the cosine values of principal angles up to $l = 5$ are shown. With each restart, the principal angles decrease. Before calling GCRO-DR, only a GMRES process is carried out for the initial estimation of eigenvectors. Hence the cosine of the principal angle between $\mathcal{U}$ and $\mathcal{G}_1$ before any GCRO-DR iterations is as low as 0.634. With each restart of inner iteration of GCRO-DR, the cosine of the principal angle between $\mathcal{U}$ and $\mathcal{G}_1$ gradually approaches 1, while those for $\mathcal{G}_i$ with $i > 1$ also increases.

#### Stationary Status

In this subsection we investigate the principal angles at stationary status during the simulation. When the principal angles approach 0, less change would be witnessed for the principal angles due to the fact that deflation is stable. Due to the large number of total linear systems, most of the simulation would be under stationary status.

Here we record the cosine of the principal angles when the deflation enters stationary status. The results are shown in Tab.6.10.

We only show the cosine of the last 5 principal angles between $\mathcal{U}$ and $\mathcal{G}_l$, since the quality of $\mathcal{U}$ is mainly reflected by the value of $l$ when the principal angle grows evidently above 0. Note that we use GCRO-DR(40,20) for all 3 cases shown in Tab.6.10, i.e., $k = 20$. We can see that for none of the 3 cases, deflation manage to grasp the full 20 smallest eigenvalues. But deflation does manage to estimate the eigenvectors which correspond to the smallest eigenvalues, resulting in significant decrease in iteration count as in Tab.6.6.

### 6.6.5 Deflation of Largest and Smallest Eigenvalues

Overall effect of deflating both largest and smallest eigenvalues based on Heuristics is list in the experiment results in Tab.6.11.

#### Analysis – Choice Distribution among Extreme Eigenvalues

The choice among largest and smallest eigenvalues of each case is listed in Tab.6.12. GCRO-DR configurations are the same as in Tab.6.11.

The reason that not all chosen eigenvalues are among the extreme ones in the spectra is that the heuristic does not work solely on the magnitude of the eigenvalues. This is true especially for eigenvalues around $(1 + 0i)$, and when eigenvalues exist on the left side of the imaginary axis, i.e., with negative real parts.

#### Principal Angle Analysis

Hereby we analyze the principal angles of between the subspace corresponding to large/small eigenvalues and the deflation vector space $\mathcal{U}$. Suppose that by Heuristics, $k_1$ is the count of the smallest eigenvalues to be deflated, and $k_2$ that of the largest ones. $k_1$ and $k_2$ are

| $l$ | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Before GCRO-DR | 0.63356 | 0.93953<br>0.59870 | 0.99603<br>0.67325<br>0.18759 | 0.99933<br>0.67811<br>0.19086<br>0.14840 | 0.99951<br>0.70298<br>0.21374<br>0.16182<br>0.07515 |
| 1st restart | 0.88727 | 0.98999<br>0.80252 | 0.99793<br>0.96081<br>0.10734 | 0.99996<br>0.96770<br>0.15461<br>0.06493 | 0.99998<br>0.97541<br>0.20954<br>0.14655<br>0.03897 |
| 2nd restart | 0.91317 | 0.99946<br>0.81993 | 0.99950<br>0.99214<br>0.11787 | 1.00000<br>0.99796<br>0.18100<br>0.09991 | 1.00000<br>0.99874<br>0.45216<br>0.17328<br>0.04538 |
| 3rd restart | 0.91636 | 0.99995<br>0.82403 | 0.99997<br>0.99368<br>0.15190 | 1.00000<br>0.99985<br>0.18643<br>0.12609 | 1.00000<br>0.99993<br>0.63895<br>0.17327<br>0.11791 |
| 4th restart | 0.95359 | 0.99998<br>0.90276 | 1.00000<br>0.99389<br>0.65199 | 1.00000<br>0.99999<br>0.65331<br>0.16130 | 1.00000<br>1.00000<br>0.86897<br>0.50758<br>0.13789 |
| 5th restart | 0.99415 | 0.99999<br>0.98737 | 1.00000<br>0.99415<br>0.97086 | 1.00000<br>1.00000<br>0.97202<br>0.16827 | 1.00000<br>1.00000<br>0.98439<br>0.79548<br>0.13259 |
| 6th restart | 0.99811 | 1.00000<br>0.99560 | 1.00000<br>0.99846<br>0.99266 | 1.00000<br>1.00000<br>0.99814<br>0.16160 | 1.00000<br>1.00000<br>0.99834<br>0.88554<br>0.14035 |

Table 6.9: Principal Angle results – Startup Process Analysis

| $l$ | 2 | 5 | 10 | 14 | 20 |
|---|---|---|---|---|---|
| IEEE39-v | 1.00000 | 1.00000 | 0.99988 | 0.99978 | 0.99956 |
| | 0.99999 | 1.00000 | 0.99985 | 0.99966 | 0.99934 |
| | | 0.99993 | 0.99981 | 0.99957 | 0.99922 |
| | | 0.99989 | 0.99971 | 0.99944 | 0.99878 |
| | | 0.99977 | 0.99947 | 0.99914 | 0.99803 |
| 2K-STABLE | 1.00000 | 1.00000 | 1.00000 | 0.99999 | 0.99456 |
| | 1.00000 | 1.00000 | 1.00000 | 0.99995 | 0.98672 |
| | | 1.00000 | 0.99999 | 0.99871 | 0.96481 |
| | | 1.00000 | 0.99971 | 0.99757 | 0.93734 |
| | | 0.99981 | 0.99789 | 0.98383 | 0.84372 |
| Dongbei-B | 1.00000 | 1.00000 | 1.99996 | 0.99964 | 0.99411 |
| | 1.00000 | 1.00000 | 1.99987 | 0.99899 | 0.98841 |
| | | 1.00000 | 0.99961 | 0.99789 | 0.98551 |
| | | 1.00000 | 0.99751 | 0.99414 | 0.97744 |
| | | 0.99910 | 0.99521 | 0.98674 | 0.93427 |

Table 6.10: Principal Angle results – Stationary Status

| Case | GCRO-DR Configuration | Averaged Iteration Count (smallest eigenvalue only) | Averaged Iteration Count (with heuristics) | Reduction Ratio |
|---|---|---|---|---|
| IEEE39-v | GCRO-DR(40,20) | 17.9 | 16.1 | 10.1% |
| 2K-STABLE | GCRO-DR(40,20) | 106.0 | 85.6 | 19.2% |
| Dongbei-B | GCRO-DR(40,20) | 31.3 | 32.7 | - 4.5% |
| 10188 | GCRO-DR(60,30) | 223.1 | 185.9 | 28.1% |

Table 6.11: Effect of deflating both small and large eigenvalues

| Case | Phase | Eigenvalues | | |
|---|---|---|---|---|
| | | Largest | Smallest | Other |
| IEEE39-v | 1 | 6 | 13 | 1 |
| | 2 | 2 | 18 | 0 |
| | 3 | 0 | 20 | 0 |
| | 4 √ | 2 | 16 | 2 |
| 2K-STABLE | None | 20 | 0 | 0 |
| Dongbei-B | None | 16 | 2 | 2 |
| 10188 | 1 | 30 | 0 | 0 |
| | 2 | 29 | 1 | 0 |
| | 3 √ | 28 | 2 | 0 |

Table 6.12: Choice among extreme eigenvalues

| $l$ | 5 | 10 | 14 | 20 |
|---|---|---|---|---|
| | 0.99999 | 0.99994 | 0.99984 | 0.99930 |
| | 0.99998 | 0.99991 | 0.99967 | 0.99908 |
| | 0.99995 | 0.99988 | 0.99947 | 0.99871 |
| | 0.99989 | 0.99945 | 0.99934 | 0.00055 |
| | 0.99975 | 0.99930 | 0.99904 | 0.00016 |

a. IEEE39-v: analysis between $\mathcal{U}$ and $\mathcal{G}_l$

| $k$ | 1 | 2 | 5 |
|---|---|---|---|
| | 0.99960 | 0.99969 | 0.99969 |
| | | 0.99957 | 0.99957 |
| | | | 0.20346 |
| | | | 0.19420 |
| | | | 0.09384 |

b. IEEE39-v: analysis between $\mathcal{U}$ and $\mathcal{H}_k$

| $l$ | 5 | 10 | 14 | 20 |
|---|---|---|---|---|
| | 1.00000 | 1.00000 | 0.99996 | 0.99996 |
| | 1.00000 | 0.99996 | 0.99995 | 0.99995 |
| | 1.00000 | 0.99996 | 0.99893 | 0.99893 |
| | 0.99993 | 0.99993 | 0.99927 | 0.99927 |
| | 0.99763 | 0.99764 | 0.99763 | 0.99763 |

c. 2K-STABLE: analysis between $\mathcal{U}$ and $\mathcal{G}_l$

| $k$ | 5 |
|---|---|
| | 0.20737 |
| | 0.18136 |
| | 0.05325 |
| | 0.04028 |
| | 0.03240 |

d. 2K-STABLE: analysis between $\mathcal{U}$ and $\mathcal{H}_k$

| $l$ | 1 | 2 | 5 |
|---|---|---|---|
| | 1.00000 | 1.00000 | 1.00000 |
| | | 1.00000 | 1.00000 |
| | | | 0.14716 |
| | | | 0.12705 |
| | | | 0.04813 |

e. Dongbei-B: analysis between $\mathcal{U}$ and $\mathcal{G}_l$

| $k$ | 5 | 10 | 14 | 20 |
|---|---|---|---|---|
| | 1.00000 | 1.00000 | 1.00000 | 0.99986 |
| | 1.00000 | 1.00000 | 1.00000 | 0.99930 |
| | 1.00000 | 1.00000 | 1.00000 | 0.99514 |
| | 1.00000 | 1.00000 | 0.99916 | 0.09522 |
| | 1.00000 | 1.00000 | 0.99678 | 0.05970 |

f. Dongbei-B: analysis between $\mathcal{U}$ and $\mathcal{H}_k$

Table 6.13: Cosines of the principal angles between $\mathcal{U}$ and eigenvector spaces ($\mathcal{G}_l$ and $\mathcal{H}_k$)

decided by the Rits values and the Heuristics. The ideal case is that $\mathcal{U}$ fully contains both the subspace defined by the eigenvectors of $k_1$ smallest eigenvalues and the subspace defined by the eigenvectors of $k_2$ largest ones.

Following the definition of $\mathcal{G}_l$ as the subspace defined by the eigenvectors of the $l$ smallest eigenvalues, we define $\mathcal{H}_k$ as the subspace defined by the eigenvectors of the $k$ largest eigenvalues. We evaluate at stationary status in the simulation, the cosine values of: (1) the principal angles between $\mathcal{U}$ and $\mathcal{G}_l$, and (2) the principal angles between $\mathcal{U}$ and $\mathcal{H}_k$. By varying the values of $l$ and $k$, the precision of $\mathcal{U}$ hence the quality of deflation vectors can be retrieved. Results are shown in Tab.6.13. Note that similar to Tab.6.10, we only show the value for the last 5 of the principal angles, to demonstrate the number of eigenvalues deflated.

We show that principal angle analysis in Tab.6.13 is coherent with the smallest/largest eigenvalue configuration listed in Tab.6.12. Tab.6.13.a and Tab.6.13.b show that for IEEE39-v, exactly 2 smallest eigenvalues are deflated, and around 17 largest eigenvalues are deflated. This is coherent with the setting in Tab.6.12 in which for IEEE39-v has 2 smallest, 16 largest, and 2 other eigenvalues chosen for deflation. The results for 2K-STABLE and Dongbei-B are consistent in both tables under similar examinations.

| Case | Preconditioner Configuration | None | PU of ❻ | Deflation | Both |
|---|---|---|---|---|---|
| 2K-STABLE | ILU-C($10^{-2}$) | 166.20 | 38.33 | 111.71 | 33.78 |
| 2K-UNSTABLE | ILU-C($10^{-2}$) | 153.67 | 57.39 | 104.53 | 37.45 |
| Dongbei-B | ILU-C($10^{-2}$) | 50.93 | 34.26 | 35.41 | 23.12 |
| Dongbei-F | ILU-C($10^{-2}$) | 51.51 | 34.46 | 35.44 | 23.55 |
| 10188 | ILU-C($10^{-4}$) | 288.45 | 191.31 | 133.37 | 109.53 |

For 2K-STABLE, 2K-UNSTABLE, Dongbei-B and Dongbei-F, GMRES(40) and GCRO-DR(40,20) are applied. For 10188, GMRES(60) and GCRO-DR(60,30) are applied.

Table 6.14: Effect of combining preconditioner updates and deflation

### 6.6.6 Deflation with Preconditioner Updates

In Tab.6.14 we show the results of combining preconditioner updates and deflation with GCRO-DR. Note that we use non-pivoting ILU-C preconditioners now to utilize the preconditioner update strategy ❻ proposed in Chapter 5.

Overall, we use: (1) form ❻ of preconditioner updates, and (2) deflation based on heuristics of Rits values, which may be largest and smallest in magnitude. We consider two more TDS cases, i.e., 2K-UNSTABLE and Dongbei-F for comparison. Tab.6.14 show that Preconditioner Updates and Deflation can both reduce iteration count drastically. When used together, Preconditioner Update and Deflation have a combined effect in reducing the overall iteration count.

## 6.7 Summary

Deflation can be adopted to accelerate the simulation of power systems. By formulating the TDS into a problem of solving a sequence of linear systems with additive changes to the matrix. Proper choice among deflation algorithms is necessary, in which GCRO-DR suites our needs best in that it can both handle changes to the linear operator and right-hand side vectors. Deflation should be used in combination with preconditioners, in that deflation cannot handle too many extreme eigenvalues due to high memory usage when $m$ is impractically large. Since deflation vector and associated Rits value and eigenvalue of the original matrix has a 1-to-1 matching relationship, too many extreme eigenvalues would hamper the effects of deflation. With the treatment of preconditioners, extreme eigenvalues do appear for TDS problem, but at a very low quantity. We show that when used with a static preconditioner derived from admittance matrix, deflation results in up to 30% reduction in iteration count, compared with GMRES.

Additive form of linear operators in TDS simulation as proposed in Chapter 5 also enables more computationally feasible deflation with GCRO-DR. With a non-changing preconditioner, when GCRO-DR reuses information from previous runs, computation could be further reduced by calculating the additive part in $AU$ based on the additive form of $A$ as proposed in Chapter 5.

We also show through detailed analysis of the preconditioned matrices in TDS: when static preconditioner is applied to augmented linear systems, eigenvalues with large magnitude tend to appear. Based on this, we proposed heuristics which can handle extreme eigenvalues which are either small or large in magnitude. Experiments show that iteration count could be further reduced by up to 30%, compared with GCRO-DR which only deflates smallest eigenvalues.

In Chapter 5 and this chapter, multi-step techniques have been proposed to speed up the solution of a series linear systems based on dynamically changing Jacobian matrices. They represent different approaches in multi-step simulation, with the approach in Chapter 5 targeting at preconditioner quality and deflation targeting at iterative solving algorithm. These two techniques both depend on the formulation of $Y + \Delta_Y$ of the Jacobian matrices for the use of dishonest preconditioner and computationally feasible GCRO-DR algorithm. Also, when combined, Preconditioner Updates and Deflation have a joint effect on further reduction of the iteration count in TDS process. However, as a side-effect, higher overhead in each GCRO-DR startup process would ensue due to preconditioner changes. How to combine these two techniques in a more computationally-more effective way remains a challenge and serves a future research direction beyond this thesis.

# Chapter 7

# Sparse Matrix-Dense Vector Multiplication Optimization on GPU and CUDA

## 7.1 Introduction

Sparse Matrix and dense Vector multiplication (SpMV) involves the following operation :

$$y = y + A \times x \tag{7.1}$$

Where $A$ is a sparse matrix of size $n \times n$ and $x$ and $y$ are dense column vectors of size $n$. SpMV is an important computational kernel used in many scientific applications which relies on matrix-vector products. In this chapter we consider the optimization of SpMV on Graphics Processing Units (GPU) architecture. The techniques discussed in this chapter also serve as the supportive technology for the approximate inverse based preconditioners in Chapter 8.

### 7.1.1 Krylov Solvers and Sparse Matrix-Vector Products (SpMV)

Krylov subspace solvers such as GMRES, CG , BiCGSTAB, etc, rely on the generation of Krylov subspace bases of the form:

$$\{r_0, Ar_0, A^2 r_0, \ldots, A^{m-1} r_0\} \tag{7.2}$$

Where $r_0$ is the initial residue. Matrix $A$ is usually large and sparse. Total count of non-zero elements in $A$, denoted $nnz$, is low compared with $n^2$, where $n$ is the size of $A$. The generation of Krylov subspace involves the computation of a series of Sparse Matrix-Dense Vector products (SpMV).

SpMV also plays a central role in preconditioners that are based on inverse forms. Preconditioning operations with these preconditioners usually are SpMV operations. Because preconditioners play a crucially important role in the convergence of Krylov solvers, SpMV is an important computational kernel in this sense.

---

The main part of this chapter is published as conference paper [88] and journal paper [89]. Relevant contents are also included in [86] and [91].

### 7.1.2 SpMV– State of the Art

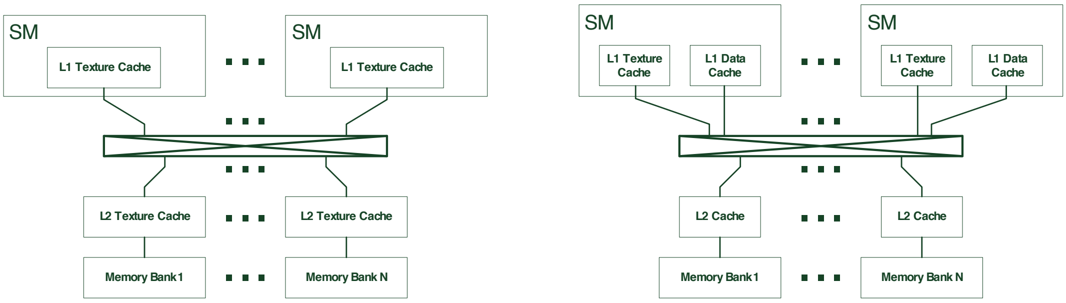
In SpMV, each non-zero element in the sparse matrix $A$ is accessed only once. The values of the elements in $A$, $x$ and $y$ are usually floating point numbers, either single-precision or double-precision. Hence, in total, SpMV involves $2nnz$ floating point operations (a multiplication and an addition are counted as 2 operations). Because are $2nnz+n$ elements accessed and there is no reuse of the information of the matrix, SpMV has a very low computation/data ratio (close to 1). In terms of performance profile, the operation of SpMV is memory intensive and the performance is memory bandwidth bound.

Performance analysis and optimizations of SpMV have been discussed in many works. Most of them focus on the reduction of the amount of data involved in accessing the matrix. These techniques include: (1) Register Blocking and similar ones [79, 83, 37], (2) Sparsity Pattern-based compression [82, 32], and (3) data-based compression [57]. Register Blocking is the most popular technique, in which small dense blocks of the matrix are recorded and accessed, rather than single elements. This reduces the access to the row/column index information of the non-zero elements. Another important optimization with Register Blocking is SIMDization, enabled by unrolling of the inner-block iterations. Also in traditional CPU-oriented era, due to the readily available cache support, the re-use in the dense vector $x$ is taken care of implicitly by the hardware.

Because of the growing adoption and popularity of accelerated platforms, especially GPUs, there have been recent works on porting SpMV to these platforms [83, 33, 43]. In [37] and various other works SpMV is used to construct CG solvers using GPU platforms. In [43] the authors applied Register Blocking to SpMV on GPU. In [33, 43], the authors show that unlike conventional CPUs, effective utilization of the high memory bandwidth of GPUs require carefully choosing the proper matrix formats based on the matrix sparsity pattern and detailed tuning.

### 7.1.3 Outline

This chapter focuses on optimization of SpMV on GPU platforms based on ELLPACK format. The optimization techniques includes: (1) assembly-level utilization of the cache features on new GPUs, and (2) optimization in favor of enhancement to locality in accessing dense vector $x$. The chapter is organized as follows. Firstly a brief introduction to the cache architecture on NVIDIA GPUs in Section 7.2, which is the focus of the performance optimization proposed in this chapter. Most-up-to-date GPU architecture, i.e., GF-100 series GPUs and relevant features are also included. In Section 7.3 we further analyze the state-of-art SpMV implementation on CUDA, focusing on its performance characteristics and ELLPACK formats related SpMV. Section 7.4 focuses on the caching of the dense vector in SpMV and proposes caching strategies on new GPU architecture with better caching support. Section 7.5 proposes matrix profile reduction based SpMV optimization. Reduction in matrix bandwidth enables enhancement of performance in 2 folds: (1) enhanced effect for caching of the dense vector, and (2) reduced matrix data access by column information compression. Performance evaluation of various optimization techniques is included in Section 7.6. Section 7.7 concludes the chapter.



a. Cache on GT-200 b. Cache on GF-100

Figure 7.1: Schematic comparison between cache organization on GT-200 and GF-100

## 7.2 Cache System on GPUs

In Chapter 3 a brief introduction to CUDA platform and GP-GPU was given. Here we relist relevant GPU architecture details in Tab.7.1. We focus on the Cache subsystem. Cache is a small amount of memory on CPU/GPU as temporary buffer for data in the main memory. Compared with main memory, caches have very low access latency and high bandwidth. Caches are designed to tackle the problem of “memory wall”: accessing the off-chip memory is long in latency and the total bandwidth is limited. The cause of the problem is mainly due to: (1) the packaging limitation, i.e., pin-out area of the processor, and (2) thermal limitation. To alleviate this problem, traditional CPU uses hierarchies of caches. Current CPU chips usually dedicate over 50% of the total silicon area to caches. On the contrary, GPUs have always dedicated major part of the on-chip resource to execution units. Before GF-100, there is no writable cache on the NVIDIA GPUs: the cache is small in size and only used for read-only data such as texture and not optimized for access speed. GF-100 introduces writable 2-level caches with large sizes with much lower latencies. Tab.7.1 shows the timing details of Caches on GPU. Fig.7.1 also shows the schematic comparison between GT-200 and GF-100 series of GPUs.

We can see in Tab.7.1, access through texture caches in both GT-200 and GF-100 GPUs are comparable in latency to accesses to the main memory. On the contrary, for GF-100, access through data caches is of an order lower latency than the main memory.

## 7.3 SpMV on CUDA

This section focuses on the existing popular approaches for SpMV on CUDA and GP-GPU platforms. Matrix storage format has a fundamental impact on SpMV implementation. ELLPACK format, as a memory bandwidth friendly format in CUDA, usually shows best performance. In this section a brief introduction to the relevant formats is given. After that, extensive performance analysis of ELLPACK based SpMV on CUDA is given.

### 7.3.1 Sparse matrix formats and SpMV

There are many popular storage formats for Sparse Matrices, among them Diagonal format (DIA) [72], Compressed Sparse Row (CSR), Coordinate format (COO), ELLPACK [53], and Hybrid format (HYB) [33]. To illustrate different formats, a sample matrix of size 4×4 is shown. The red labels are column indicators, and the blue ones are for rows. Note

| GPU | Series | GT-200 | GF-100 |
|---|---|---|---|
| | Year | 2008 | 2010 |
| | Name | Tesla C1060 | GeForce GTX-480 |
| Memory Bandwidth | | ~80 GB/s | ~115 GB/s |
| L1 Data Cache | Size | N/A | 16/48 KB per SM |
| | Hit Latency | | 80 cycles |
| L2 Data Cache | Size | N/A | 768 KB |
| | Hit Latency | | 212 cycles |
| L1 Texture Cache | Size | ~ 5KB per SM | 12 KB per SM |
| | Hit Latency | 258 cycles | 220 cycles |
| L2 Texture Cache | Size | 256 KB | None (unified Data Cache) |
| | Hit Latency | 366 cycles | 427 cycles |
| | Miss Latency | 547 cycles | 632 cycles |
| Global Memory | Size | 4 GB | 1.5 GB |
| | Latency | 506 cycles | 319 cycles |

Table 7.1: Quantitative comparison between GPU architectures

that indices are zero-based. The storage schemes in various matrix formats are shown below.

$$\begin{array}{cc} & \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \end{array} & \begin{bmatrix} 1 & 7 & 0 & 0 \\ 0 & 2 & 8 & 0 \\ 5 & 0 & 3 & 9 \\ 0 & 6 & 0 & 4 \end{bmatrix} \end{array}$$

For DIA format, elements are stored according to their positions in the diagonals. In ELLPACK, suppose that there are at most $m$ non-zero elements per row, then column indices and actual values are to be recorded: each row corresponds to a row in "col_indices" and "data". In each row of "col_indices", the column indices of each non-zero element in the corresponding row of the matrix are recorded. Structure "data" records values in a similar way. Hence "col_indices" and "data" are both of $n \times m$ size, where $n$ is the matrix row count. Note that "data" array in both DIA and ELLPACK are recorded in a column-wise way, i.e., elements in each column are stored in adjacent positions. Also in these two formats, starred elements in the "data" array represent padding. CSR format is the most widely used format. COO format extends the "row_ptr" part of CSR, to contain actual row indices of corresponding non-zero elements. HYB format is a format designed in [33] to combine ELLPACK format and COO format, to avoid too much padding if the matrix is stored in ELLPACK only. It contains: (1) an ELLPACK-based part with size $n \times m'$ where $m' < m$, and minimal padding, and (2) a COO part to contain those elements that cannot be contained ELLPACK, i.e., the $j$-th non-zero element in each row where $j > m'$. In the simple matrix example above, we set $m = 2$, then there is only one element at position (2,3) in the original matrix contained by COO part.

Fig.7.2 shows the storage scheme of various matrix formats for the $4 \times 4$ matrix above. Note that for ELLPACK format $m$ is 3, while for HYB format $m$ is 2. All row and column indices are zero based.

DIA format:

$$\text{diag\_offsets} = \begin{bmatrix} -2 & 0 & 1 \end{bmatrix} \qquad \text{data} = \begin{bmatrix} \ast & 0 & 7 \\ \ast & 2 & 8 \\ 5 & 3 & 9 \\ 6 & 4 & \ast \end{bmatrix}$$

ELLPACK format:

$$\text{col\_indices} = \begin{bmatrix} 0 & 1 & \ast \\ 1 & 2 & \ast \\ 0 & 2 & 3 \\ 1 & 3 & \ast \end{bmatrix} \qquad \text{data} = \begin{bmatrix} 1 & 7 & \ast \\ 2 & 8 & \ast \\ 5 & 3 & 9 \\ 6 & 4 & \ast \end{bmatrix}$$

CSR format:

$$\begin{aligned} \text{row\_ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 9 \end{bmatrix} \\ \text{col\_indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \text{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

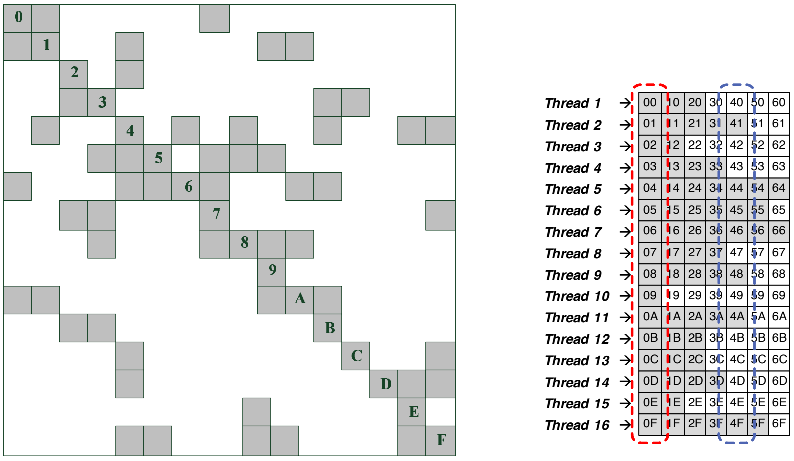
COO format:

$$\begin{aligned} \text{row\_indices} &= \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 3 & 3 \end{bmatrix} \\ \text{col\_indices} &= \begin{bmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{bmatrix} \\ \text{data} &= \begin{bmatrix} 1 & 7 & 2 & 8 & 5 & 3 & 9 & 6 & 4 \end{bmatrix} \end{aligned}$$

HYB format:

$$\text{ELL\_col\_indices} = \begin{bmatrix} 0 & 1 \\ 1 & 2 \\ 0 & 2 \\ 1 & 3 \end{bmatrix} \quad \text{ELL\_data} = \begin{bmatrix} 1 & 7 \\ 2 & 8 \\ 5 & 3 \\ 6 & 4 \end{bmatrix} \quad \begin{aligned} \text{COO\_row\_indices} &= \begin{bmatrix} 2 \end{bmatrix} \\ \text{COO\_col\_indices} &= \begin{bmatrix} 3 \end{bmatrix} \\ \text{COO\_data} &= \begin{bmatrix} 9 \end{bmatrix} \end{aligned}$$

Figure 7.2: Matrix storage formats.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| Thread 1 → | 00 | 10 | 20 | 30 | 40 | 50 | 60 |
| Thread 2 → | 01 | 11 | 21 | 31 | 41 | 51 | 61 |
| Thread 3 → | 02 | 12 | 22 | 32 | 42 | 52 | 62 |
| Thread 4 → | 03 | 13 | 23 | 33 | 43 | 53 | 63 |
| Thread 5 → | 04 | 14 | 24 | 34 | 44 | 54 | 64 |
| Thread 6 → | 05 | 15 | 25 | 35 | 45 | 55 | 65 |
| Thread 7 → | 06 | 16 | 26 | 36 | 46 | 56 | 66 |
| Thread 8 → | 07 | 17 | 27 | 37 | 47 | 57 | 67 |
| Thread 9 → | 08 | 18 | 28 | 38 | 48 | 58 | 68 |
| Thread 10 → | 09 | 19 | 29 | 39 | 49 | 59 | 69 |
| Thread 11 → | 0A | 1A | 2A | 3A | 4A | 5A | 6A |
| Thread 12 → | 0B | 1B | 2B | 3B | 4B | 5B | 6B |
| Thread 13 → | 0C | 1C | 2C | 3C | 4C | 5C | 6C |
| Thread 14 → | 0D | 1D | 2D | 3D | 4D | 5D | 6D |
| Thread 15 → | 0E | 1E | 2E | 3E | 4E | 5E | 6E |
| Thread 16 → | 0F | 1F | 2F | 3F | 4F | 5F | 6F |

* $n = 16$, $m = 7$, $nnz = 67$, Padding= 45, ELLPACK Efficiency: 60%.

Figure 7.3: Sample matrix sparsity pattern and its storage scheme in ELLPACK.

### 7.3.2 Analysis of ELLPACK format

In SpMV with ELLPACK format, each row of the matrix is assigned to a CUDA thread. In each thread, there is an iteration over $m$. The thread first identifies itself and the row it is assigned to, using the thread identity. Then it iterates over the $m$ non-zero elements in that row. The pseudo code for a CUDA thread is shown below. At each iteration, the thread will access the $x$ vector by the column index of the non-zero element, i.e., `col` is the column index and used as an offset to access $x$ vector. The main reason for high performance of ELLPACK format lies in the coalesced memory access enabled by ELLPACK format. We will consider this issue and other performance profile of SpMV in the following sections.

```
FUNCTION spmv_ell_kernel( Aj, Ax, y, STRIDE )
GENERATE row;
sum = y[ row ];
Aj = Aj + row;
Ax = Ax + row;
FOR i = 0 to m - 1
    val = *Ax;
    IF val != 0.0
        col = *Aj;
        sum = sum + val * x[ col ];
    END IF
    Aj = Aj + STRIDE;
    Ax = Ax + STRIDE;
END FOR
y[ row ] = sum;
END
```

#### Effect of Coalesced Memory Accesses

The high performance of ELLPACK-based SpMV relies on the effect of coalesced memory accesses. Fig.7.3 shows a sample matrix of size 16 × 16 and its storage scheme in the main memory. Each row of the matrix corresponds to a row in the storage scheme on the right. The actual relative positions in the memory of each stored element are also shown. Highlighted are the actual visited memory locations for the first and the fifth non-zero element in each row. Each row is assigned to a CUDA thread, it is immediate that adjacent threads in a warp are accessing adjacent memory positions. This results in coalesced accesses where the accesses to these positions can be combined in a single global memory access. Since SpMV is mainly dominated by accesses to the off-chip memory, this access pattern is crucial in improving the overall performance of global memory accesses to values in `Aj` and `Ax` and that of SpMV. If the initial address of the `Aj` and `Ax` are aligned to 128 byte boundaries, one still has to guarantee that accesses to 2nd, 3rd non-zero elements in each row are aligned. An easy solution for this is to use the actual row count as $n'$, where $n' \leq n$ and $n'$ can be divided by 16 exactly. $n'$ corresponds to the STRIDE variable in the pseudo code. In this way, the accesses to the $i$-th non-zero element in each row are coalesced. In the example in Fig.7.3, since the row count is already a multiply of 16, there is no padding row at the bottom of the storage scheme.

| Matrix | $n$ | $nnz$ | ELLPACK Occupancy (%) | ELLPACK Efficiency (%) |
|---|---|---|---|---|
| FEM/Cantiliver | 62451 | 4007383 | 99.7 | 85.3 |
| FEM/Sphere | 83334 | 6010480 | 100.0 | 89.0 |
| FEM/Accelerator | 121192 | 2624331 | 82.8 | 78.0 |
| Economics | 206500 | 1273389 | 81.1 | 71.4 |
| Epidemiology | 525825 | 2100225 | 100.0 | 99.9 |
| Protein | 36417 | 4344765 | 96.0 | 83.0 |
| WindTunnel | 217918 | 11634424 | 99.4 | 98.3 |
| QCD | 49152 | 1916928 | 100.0 | 100.0 |
| FEM/Harbor | 46835 | 2374001 | 81.3 | 74.9 |
| Circuit | 170998 | 958936 | 78.2 | 87.7 |
| Web | 1000005 | 3105536 | 64.2 | 99.7 |

Table 7.2: Test matrices for SpMV– Characteristics.

### Padding and Efficiency

Fig.7.3 also shows the padding in the ELLPACK format. Grey part in the storage scheme contains actual non-zero elements, while white part indicate necessary padding. If the matrix is to be recorded in ELLPACK format, $n' \times m$ memory space should be allocated, where $m$ is the maximum number of non-zeros in each row. One can calculate the actual storage efficiency of the ELLPACK format by dividing $nnz$ by $n' \times m$. The efficiency of the matrix in Fig.7.3 is 60%.

To accommodate situations where non-zero element counts in rows vary significantly, HYB format is proposed in [33]. Based on static performance proportion for ELLPACK format and COO format, HYB hits an trade-off point by finding the value for $m$ where the overall performance is the best. If $m$ is found, then elements that can be contained in ELLPACK, i.e., the first $m$ non-zero elements in each row, while others are contained in another part recorded in COO. In HYB, the original matrix is in effect split into 2 sub-matrices, one recorded by ELLPACK, while the other by COO format.

We use the default model for splitting ELLPACK and COO used in [33]. We also consider the matrix test suite used in [33, 83] for the performance evaluation. The summary of these matrices in the suite is listed in Tab.7.2. The percentage of the actual non-zero element count recorded in ELLPACK in $nnz$, and the storage efficiency of ELLPACK data are also shown.

### Effect of GPU Occupancy

CUDA programs usually have many thousands concurrent threads executing at the same time. For memory intensive applications such as SpMV, it is crucially important that computation latency be hidden in the global memory accesses. One basic yet important criteria for optimal latency hiding is the *occupancy*: the proportion of the actual concurrent threads count on and the hardware limit of total concurrent thread count. For SpMV, 100% occupancy can be easily achieved through setting the thread block size as: 128 (or 256, 512) for GT-200 and 192 (or 256, 384, 512) for GF-100. (Calculated according to Tab.7.1). With other thread block sizes, lower occupancy lower than 100% will occur. Experiments show that lower occupancy always results in lower performance. In this sec-

| | | Single-Precision | Double-Precision |
|---|---|---|---|
| Load column index | | 506 | |
| Load matrix value | | 506 | |
| Load value in $x$ | Hit-1 | 220 | |
| | Miss-1 | 427 | |
| | Miss-2 | 632 | |
| MAD | Warp Latency | 24 cycles | 48 cycles |
| | Issue Rate | 4 cycles | 32 cycles |

**Hit-1** : threads in the same warp all hit in the same line in L1 TC

**Miss-1** : threads in the same warp all exist in an missing L1 TC line, which is found in L2 TC

**Miss-2** : threads in the same warp all exist in an missing L1 TC line, which exists in main memory (i.e., missing in L2 TC too)

Table 7.3: Assembly-level Timing Analysis in GT-200

tion, all thread block sizes are set to 256, to ensure 100% occupancy across GT-200 and GF-100 GPUs.

## 7.4 Caching Vector $x$ in SpMV– Analysis and Optimization

### 7.4.1 Caching on GT-200

The GT-200 architecture is an example of GPUs without full cache support. There are 3 hardware mechanisms that can be used for caching the dense vector $x$ on GT-200 architecture:

1. Use texture cache and treat $x$ as a 1-D texture
2. Use constant cache, by decorating $x$ as a constant
3. Use Shared Memory as a software-managed cache

Approach 1 is chosen by [33]. Approach 2 is virtually limited by the small size of the constant memory space, which for all NVIDIA GPUs is 64KB. This translates to 16384 single-precision numbers or 8192 double-precision ones. Since usually the value of $n$, i.e., the length of $x$ is large, constant cache is not sufficient to contain $x$ hence cannot be used for caching. Approach 3 requires software management of ShMem contents and according to our experiments and according to indication in [43], it introduces too much overhead in codes that dedicated to management of the cache. Hence we do not consider Approach 2 and Approach 3 in this thesis. We analyze the effect of caching of $x$ through Texture Cache to a finer detail, to study the effect of long latency of the Texture Cache on the overall performance of SpMV. The quantitative timing of the latency for accessing the main memory, texture cache and MAD operations are listed in Tab.7.3.

In order to quantitatively investigate the effect of caching of SpMV, we breakdown the computation of SpMV into 2 parts: (1) reading of non-zero elements, including index info

| Matrix | $t_{pseudo\_x}$ | $t_{all}$ | $t_x$ in $t_{all}$ |
|---|---|---|---|
| FEM/Cantiliver | 0.493 | 0.530 | 7.1% |
| FEM/Sphere | 0.604 | 0.658 | 8.2% |
| FEM/Accelerator | 0.470 | 0.624 | 25.7% |
| Economics | 0.421 | 0.508 | 17.1% |
| Epidemiology | 0.275 | 0.324 | 15.1% |
| Protein | 0.652 | 0.703 | 7.2% |
| WindTunnel | 1.203 | 1.246 | 3.4% |
| QCD | 0.190 | 0.207 | 8.4% |
| FEM/Harbor | 0.428 | 0.448 | 4.5% |
| Circuit | 0.231 | 0.310 | 25.4% |
| Web | 0.719 | 0.956 | 24.8% |
| Geo-Mean | | | 13.1% |

* Single precision is used. All time values are in $ms$. Percentage values of the 4 and 6-th column show the proportion of the time spent in accessing $x$ in corresponding SpMV operations. Either ELLPACK or HYB is used for matrix storage; ELLPACK is used when feasible.

Table 7.4: Performance results of SpMV when a pseudo vector of $x$ is used for GT-200.

and value info, reading and writing of $y$ vector, computations (i.e., MAD operations), and (2) reading of elements in $x$. Since SpMV is mainly memory bandwidth bound, we only focus memory operations in both parts. Part (1) includes all deterministic memory accesses: reading to indices and values of non-zero elements are coalesced; reading and writing of $y$ vector are also coalesced (guaranteed when thread block size is the power of 2). Part (2) includes non-deterministic memory accesses: offset into $x$ can be random, and there is no guarantee that they are coalesced or hit in Texture Cache.

To measure the timing of both parts, we use a pseudo $x$ vector with each element as a pre-defined constant value. Hence the access to $x$ now can be hardwired into codes, avoiding access to the actual off-chip dense vector. Then the time required to access dense vectors (denote as $t_x$) can be calculated as:

$$t_x = t_{all} - t_{pseudo\_x} \tag{7.3}$$

Where $t_{all}$ denotes the time required to perform SpMV with a normal, non-pseudo $x$ vector, and $t_{pseudo\_x}$ for that of SpMV with a pseudo vector. Tab.7.4 II shows the results for Berkeley test suite. Single Precision operations are used. $t_{all}$ is the time for an SpMV operation with $x$ vector cached by TC. On average about 13% of the time of SpMV is spent in accessing $x$.

### 7.4.2 Caching Optimization on GF-100

As noted in Tab.7.1, with the evolution of the new architecture, NVIDIA has been improving the architecture of GPUs and programming interface (CUDA). New architecture, called GF-100, has dramatic improvements in the memory subsystem. The overall improvement reflects the trend of merging of conventional CPU architecture and GPU architecture in programmability, i.e., unified and addressable memory space, coherent cache hierarchy, etc. They can be summarized into three categories: (1) cache size improvements, (2) much lower access latency to caches, and (3) cache coherence.

For SpMV, there is no re-use of matrix data or $y$ vector data, and there exists only the re-use of $x$ vector data which is read-only. Hence we do not need cache coherency. Larger caches will benefit hit ratio when accessing $x$, and lower latency for accessing the cache is also beneficiary for latency hiding as mentioned in previous part of this chapter.

While larger caches benefit hit ratio by reducing capacity misses. Unlike in previous generations, all data accessed are filtered through cache by default. This results in conflicts between matrix data and $x$ data when using a popular LRU (Least-Recently-Used) mechanism for cache management, and data in $x$, which are potentially used again, are evicted from the cache by the matrix data which is never used again. While PTX instruction set allows different access patterns for various data types, current CUDA implementations (for up to date version of 3.1) does not expose this feature on the API level.

To fully exploit the capability of the cache in GF-100 GPUs, we use inline PTX assembly to differentiate access to matrix data and vector data, so that these accesses pass through cache in different manners:

1. Let access to matrix data be marked as un-cached, or marked as lowest priority in cache management, so that they are never cached or evicted first when capacity conflict happens.

2. Let access to $x$ be fully cached, so that data in $x$ always have higher priority for staying in cache than matrix data.

Here we give an example about differentiation of memory accesses to various data. These two lines of codes are to be in-lined to the `.cu` file, in order to access data at `addr1` and `addr2`. The syntax is in PTX assembly.

```
__asm("ld.ca.f32  %0, [%1];" : "=f" (a) : "l" (addr1));
__asm("ld.cv.f32  %0, [%1];" : "=f" (b) : "l" (addr2));
```

By the first line of the code, we load a single-precision number to value `a` from address `addr1`, and by the second line, we load a single-precision number to value `b` from address `addr2`. The loading of `a` is cached, by the PTX inline assembly instruction `ld.ca` in which `ca` means “cache all”. The loading of `b` is marked as volatile by `ld.cv`, which means “load-with-cached-as-volatile” and hints that the value at address `addr2` is volatile and access to it will skip the caches and be directly from main memory.

In SpMV, by reading the matrix data and vector data through different inline assembly codes, we differentiate the caching strategy of different data, to attain better utilization of the cache on GF-100 GPUs. Inlining PTX codes corresponds to bypassing the compile phase and inject PTX codes directly to the `ptxas` (i.e., PTX assembler), as shown in Fig.3.5.

We note that future accelerated platforms including GPUs will evolve to feature full cache support which bear more similarity to conventional CPU caches in latency and size, just like GF-100. Differentiation in data access pattern with respect to caches provide potentials for application optimization, provided the programmers have a good knowledge of the data access pattern for the given application.

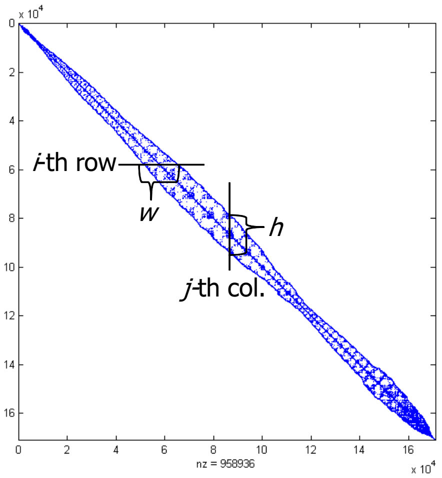


Figure 7.4: SpMV of Matrices in an Reduced Bandwidth form

## 7.5 Optimization based on Matrix Bandwidth Reduction

In this section we consider SpMV optimization based on Matrix Bandwidth/Profile Reduction. There are 2 benefits of matrix bandwidth reduction: (1) improved locality in accessing $x$ vector, and (2) index compression which is enabled by a reduced matrix bandwidth. We use Reverse Cuthill-McKee (RCM) permutation for the bandwidth reduction in the following, mainly for its simplicity and popularity. Other more sophisticated algorithms are possible and included in future research.

### 7.5.1 Locality in Accessing $x$

Fig.7.4 shows the matrix "Circuit" in Tab.7.2 when permuted using RCM. We denote $BW_A$ as the bandwidth of matrix $A$. In terms of accessing $x$, there are two sides of locality:

1. For the $i$-th row, the accessed part of the $i$-th CUDA thread lies within a range of $w$ elements, with $w < BW_A$.

2. For the $j$-th element of $x$, the threads accessing this element will be adjacent, i.e., thread indices would be adjacent to $j$, within a range of $h$ and $h < BW_A$.

The first item implies that with a smaller value of $w$, there will be denser distribution, and better temporal locality for caches, i.e., values in the same cacheline would be more likely re-used. The second implies that for the CUDA threads within the same thread block will access a smaller set of values, improving the hit ratio of the cache.

We evaluate: (1) the memory accessed in $x$ by thread blocks at the granularity of L1 Texture Cache lines, (2) the memory accessed in $x$ by passes at the granularity of L2 Texture Cache lines. We take the sum of the total cache line count used for all thread blocks (or passes). The results for the matrices in both original forms and permuted forms with RCM are shown in Tab.7.5 (Single-Precision is used). Results in Tab.7.5 serve as a

| | Setting-1 | | Setting-2 | |
|---|---|---|---|---|
| | Original | With RCM | Original | With RCM |
| FEM/Cantiliver | 24529 | 24870 | 7947 | 7974 |
| FEM/Sphere | 53880 | 52650 | 12805 | 12253 |
| FEM/Accelerator | 350583 | 83024 | 43333 | 14089 |
| Economics | 124117 | 215952 | 26941 | 27599 |
| Epidemiology | 194104 | 197124 | 67256 | 67569 |
| Protein | 19813 | 34970 | 5443 | 4905 |
| WindTunnel | 92246 | 97338 | 30225 | 28899 |
| QCD | 36352 | 36841 | 9216 | 8070 |
| FEM/Harbor | 18503 | 17806 | 8375 | 5950 |
| Circuit | 126246 | 94675 | 63445 | 28136 |
| Web | 302480 | 226034 | 185659 | 163377 |

**Setting-1** : L1 Texture Cache line usage at thread block level

**Setting-2** : L2 Texture Cache line usage at pass level

Table 7.5: Cache line accesses in $x$ for matrices.

qualitative examination for the cache usage of SpMV. By RCM permutations, the cache-line usage of some matrices have experienced significant drop. The effect on performance is evaluated in Section 7.6.

### 7.5.2 Index Compression

When a matrix has a small matrix bandwidth, there is a strong correlation between the column index $c$ and the row index $r$ for any given non-zero element:

$$BW_L < (c - r) < BW_R \tag{7.4}$$

Where $BW_L$ and $BW_R$ are the left bandwidth and right bandwidth of the matrix, respectively. A smaller number of $BW_L$ and $BW_R$ imply a tighter boundary for the values of $(c - r)$ of all the non-zero elements. A smaller upperbound for the values of $(c - r)$ implies potential of compressing the column index information based on the row index information. In SpMV, since there is an implicit mapping between matrix rows and CUDA threads, the thread identifies itself by calculating its own thread identity hence the row number is retrieved. Normally due to the large size of the matrix, values for $r$ have to be recorded at least in 32-bit integer format. Value range of $c$ usually require same integer format for storage in ELLPACK. But by recording $(c - r)$, we can re-generate values of $c$ by the value of $r$ and $(c - r)$. If it is applicable to record $(c - r)$ in a shorter format, e.g., `short` (2-byte) or `byte` (1-byte), we can reduce the accessed memory amount in the SpMV process, at the overhead of generating values of $c$ on the fly. In this thesis, we consider using short for the purpose of column index compression.

Column index compression may not be applicable to any matrix, due to the potential large values for $(c-r)$. When short is used to record (c..r), index compression is applicable if values of $(c - r)$ falls within the boundary of [-32768, 32767]. We investigate the effect of bandwidth reduction techniques on this issue in Section 7.5.2 and the potential

| Matrix | Applicable to $r$ and $c$ | Applicable to $(c-r)$ | Applicable to $(c-r)$ with RCM |
|---|---|---|---|
| FEM/Cantiliver | Yes | Yes | Yes |
| FEM/Sphere | No | Yes | Yes |
| FEM/Accelerator | No | No | Yes |
| Economics | No | Yes | Yes |
| Epidemiology | No | Yes | Yes |
| Protein | Yes | Yes | Yes |
| WindTunnel | No | No | Yes |
| QCD | Yes | Yes | Yes |
| FEM/Harbor | Yes | Yes | Yes |
| Circuit | No | No | Yes |
| Web | No | No | No |

Table 7.6: Applicability of column index compression for SpMV

| Value Type | Single-Precision | Double-Precision |
|---|---|---|
| Reduction Ratio | 25.0% | 16.7% |

Table 7.7: Reduction ratio in accessed memory amount for SpMV using Index Compression.

performance enhancement enabled by column index compression in Section 7.5.2.

### Applicability of Index Compression

Tab.7.6 shows the applicability of column index compression for matrices in Tab.7.2, and the effect of matrix bandwidth reduction. We can see that for the original form of the matrix, due to the large size of the matrices, a majority of them (7 out of 12) cannot be recorded using a shorter format for indices. When applying the incremental form of recording $(c-r)$ rather than $c$, the reduction of using short formats apply to 3 more matrices. Yet still 4 matrices are out of luck, mainly due to the large, random distribution in the matrix and $(c-r)$ exceeds the boundary for short format. By using RCM, only 1 out of the 11 matrices does not accept reduction in storage format for $(c-r)$. Column index compression now can be applied to all the other matrices.

As shown in Tab.7.6, matrices generated by FEM applications usually accepts the formulation of a small matrix bandwidth. Randomly sparse matrices such as "Circuit" and "Web" have a relatively large bandwidth even after RCM permutation. "Web" does not accept index reduction even with RCM permutation.

### Performance Projection

The ideal case for speeding up SpMV by means of index compression is the reduction ratio of the accessed data during the SpMV by using shorter storage formats. The ratio is shown in Tab.7.7, by recording $(c-r)$ in short format, instead of recording $c$ in 32-bit integer format.

The reduction ratio in Tab.7.7 can be used as the upper bound for the speedup of column index compress on the performance of SpMV. Due to other factors such as runtime overhead, use of HYB format, the actual speedups would be lower than those in Tab.7.7.

#### Potentials for Register Blocking

Index Compression can be combined with conventional SpMV optimization techniques such as Register Blocking in [79]. Actually the reduction of bandwidth usage of index compression is on par with that of register blocking when using blocks of size 1 × 2 or 2 × 1. We would like to mention that column index compression can be used together with Register Blocking. When register blocking of size $m \times k$ is used, bringing in index compression would reduce the amount of column index information by 50%. This translates to 11.1% reduction in the overall accessed matrix data when $2 \times 2$ block size and single-precision is used. Utilizing both register blocking and index compression remains a future work beyond this thesis.

## 7.6 Performance Evaluation

In this section we evaluate the quantitative performance enhancement of various SpMV optimizations proposed in previous sections. Based on GT-200 GPU, we first investigate the effect of reduced matrix bandwidth on performance, including both the effect on accessing $x$ vector and index compression. Second, the cache-oriented optimization based on GF-100 architecture is evaluated. The combined effect of reduced matrix bandwidth and cache-oriented optimization is also evaluated for GF-100 GPU.

### 7.6.1 Effect of Bandwidth Reduction with GT-200

The experiments are carried out on NVIDIA C1060 GPU (belonging to GT-200 series) listed in Tab.7.1. The caching of $x$ is through the texture cache which is the only caching mechanism that can be used for SpMV.

#### Overall Performance and Accesses to $x$

We first evaluate the effect of RCM permutation on $t_{all}$ and $t_x$. We record the values of $t_x$ for matrices before and after RCM permutation. We only test the matrices that have obtained matrix bandwidth reduction using RCM.

Tab.7.8 lists the performance comparison. We see that speedup in $t_{all}$ is not achieved for every matrix with a reduced bandwidth. This is due to the fact that RCM only performs as a heuristics for reducing $t_x$. Especially when: (1) there are substructures in the original matrix which are small, dense blocks, such as “Protein”, or (2) the access pattern into $x$ is already very regular, such as “QCD”, the access to $x$ is already not a performance issue and there is no significant enhancement for reducing the bandwidth.

For the whole set of matrices that the bandwidth is reduced by RCM, the overall speed up for $t_{all}$ is 5% for Single Precision and 7% for Double Precision SpMV operations. The time dedicated to $x$ accesses is also reduced: 17% and 24%, for single-precision and double-precision, respectively. If we only count the matrices that have enhanced $x$ accesses, including: “FEM/Sphere”, “FEM/Accelerator”, “FEM/Harbor”, “Circuit” and “Web”, the average speedups for $t_{all}$ are 8% (single-precision) and 10% (double-precision), while those for $t_x$ are 25% (single-precision) and 34% (double-precision). This set of matrices consists of 5 of the 11 matrices in the test suite in Tab.7.2.

| Matrix | Bandwidth | Bandwidth with RCM | Speedup for Single-Precision | Speedup for Double-Precision |
|---|---|---|---|---|
| FEM/Sphere | 44025 | 5401 | 1% / 2% | 9% / 26% |
| FEM/Accelerator | 121041 | 2931 | 11% / 31% | 23% / 89% |
| Protein | 34065 | 2490 | -2% / -5% | 0% / 0% |
| WindTunnel | 189332 | 2168 | 0% / 0% | 2% / 5% |
| QCD | 43011 | 8466 | 0% / 0% | -4% / -8% |
| FEM/Harbor | 25142 | 671 | 0% / 0% | 1% / 8% |
| Circuit | 170977 | 8643 | 10% / 32% | 14% / 45% |
| Web | 925210 | 473703 | 18% / 76% | 3% / 17% |
| GeoMean | | | 5% / 17% | 7% / 24% |
| GeoMean* | | | 8% / 25% | 10% / 34% |

The first value of each element in Column 4 and Column 5 is the speedup in SpMV performance, i.e., $t_{all}$; the second is the speedup for $t_x$, i.e., the time spent in accesses to $x$.

Table 7.8: Matrices permutated by RCM – Performance Results

| Matrix | Speedup for Single-Precision | Speedup for Double-Precision |
|---|---|---|
| FEM/Cantliver | 18.7% | 4.6% |
| FEM/Sphere | 20.8% | 15.1% |
| FEM/Accelerator | 12.2% | 10.3% |
| Economics | 20.6% | 17.3% |
| Epidemiology | 23.1% | 11.2% |
| Protein | 12.0% | 9.3% |
| WindTunnel | 23.5% | 14.7% |
| QCD | 27.3% | 6.6% |
| FEM/Harbor | 9.9% | 10.1% |
| Circuit | 6.5% | 6.7% |
| GeoMean | 16.2% | 10.5% |

Table 7.9: Evaluation of Index Compression on $t_{pseudo\_x}$

| Matrix | Use RCM? | Use Index Compression? |
|---|---|---|
| FEM/Cantiliver | No | Yes |
| FEM/Sphere | Yes | Yes |
| FEM/Accelerator | Yes | Yes |
| Economics | No | Yes |
| Epidemiology | No | Yes |
| Protein | No | Yes |
| WindTunnel | Yes | Yes |
| QCD | No | Yes |
| FEM/Harbor | Yes | Yes |
| Circuit | Yes | Yes |
| Web | Yes | No |

Table 7.10: Configuration

### Index Compression on $t_{pseudo\_x}$

Tab.7.9 summarizes the speedup on the $t_{pseudo\_x}$ by column index compression. Note that 10 out of 11 matrices now accepts index compression, due to the use of RCM. The speedups for single-precision and double-precision are close to the theoretical speedup projected in Tab.7.7. While other matrices show speedups close to the projection in Tab.7.7, three matrices shows much lower speedups, including "Circuit", "FEM/Harbor" and "Protein". For them, the speedups are amortized by several factors. These matrices use HYB format, rather than pure ELLPACK format. In fact, extra COO part in HYB format is not characterized by the performance modeling/projection in Tab.7.7. Also a separate call to COO part introduces extra overhead. These factors tend to result in lower speedups. Although lower than the ideal speedups in Tab.7.7, the reduced amount of accessed memory still yields solid speedup in $t_{pseudo}$ on average: 16.2% for single-precision and 10.5% for double-precision.

### Overall Speedup

Next we evaluate the overall speedup in $t_{all}$, combining the effect of RCM on $t_x$ and index compression on $t_{pseudo\_x}$. Tab.7.10 gives the configurations for various matrices about whether RCM is applied and index compression.

Tab.7.10 gives the overall speedup on $t_{all}$ for all the matrices in Tab.7.2. Note that we have obtained speedups for all the matrices in the test suite. The geometric mean of the speedup for single-precision and double-precision is 16% and 12.6%, respectively.

## 7.6.2 GF-100 based Optimization

In this subsection we consider the quantitative evaluation of Cache-based optimization effect on SpMV based on GF-100 based GPU. We use NVIDIA GeForce GTX480 for the tests. First we describe the three strategies for using cache in SpMV on GF-100 architecture.

**Strategy-1** : Use texture fetching mechanism for accessing vector data, as used in [33]

**Strategy-2** : Use texture fetching mechanism for accessing vector data (as in [33]), and mark access to matrix data as volatile and avoid contamination to cache

| Matrix | Speedup for Single-Precision | Speedup for Double-Precision |
|---|---|---|
| FEM/Cantiliver | 11.1% | 5.0% |
| FEM/Sphere | 23.0% | 10.9% |
| FEM/Accelerator | 17.1% | 32.5% |
| Economics | 13.0% | 10.6% |
| Epidemiology | 23.1% | 9.7% |
| Protein | 9.3% | 9.3% |
| WindTunnel | 22.0% | 14.4% |
| QCD | 19.0% | 10.1% |
| FEM/Harbor | 7.7% | 9.7% |
| Circuit | 14.3% | 17.8% |
| Web | 18.0% | 3.0% |
| Geo-Mean | 16.0% | 12.6% |

Table 7.11: Performance results on SpMV speedups

**Strategy-3** : Filter data in the vector through data cache, and mark access to matrix data as volatile and avoid contamination to cache

Strategy-1 is the caching strategy for achieving high performance in [33]. We consider it as the baseline for comparison. When used, Strategy-1 will cause both matrix data and vector data occupying L2 Cache, while L1 Texture Cache is still dedicated to data in the dense vector $x$. Strategy-2 avoids contamination of L2 Cache caused by matrix data, hence vector data will consume both L1 Texture Cache and L2 Cache. However because the texture fetching mechanism is used, the latency is high as in GT-200 GPUs. Strategy-3 is the caching strategy we proposed in Section 7.4. It uses fast L1 Data Cache (configured to be 48KB) and L2 Cache for accesses to $x$. Strategy-3 has 2 sides of enhancements: (1) fast access to $x$ data, and (2) avoidance of contamination of matrix data in the cache.

Tab.7.12 lists the speedup of SpMV execution time between: (1) Strategy-3 and Strategy-1, and (2) Strategy-3 and Strategy-2. The use of RCM permutation is also listed. Note that Strategy-2 usually performs better than Strategy-1 (in 20 out of 22 cases, for both Single-Precision and Double-Precision). The caching strategy proposed in previous section outperforms the original strategy used in [33] by 17% and 14% for Single and Double-Precision respectively. Comparison between Strategy-2 and Strategy-3 is the comparison between the effect of caching through texture fetching mechanism and through ordinary data loads. The results show 13.6% and 10.3% speedup when ordinary data fetches and data caches are used. This is because of the following 2 reasons. First, L1 Data Cache is configured to 48 KB (4 times the size of L1 Texture Cache), this reduces capacity misses and enhances hit ratio at the SM level. Second, a hit/miss in data cache incurs much lower latency than a hit/miss caused by Texture fetches on both L1 and L2, and this reduces the chance that the data fetch latency is not well hidden by accesses to the matrix data.

In Tab.7.13 we show the SpMV performance comparison between GF-100 and GT-200 series by taking GeForce GTX-480 and Tesla C1060 as the representative GPU. Overall, GeForce GTX-480 is faster than Tesla C1060 in SpMV performance by 50% and 75% for single-precision and double-precision, respectively. Note that Tesla C1060 GPU is designed for GP-GPU, while GeForce GTX-480 is designed for gaming purposes. These

| Matrix | Use RCM? | Speedup for Single-Precision | | Speedup for Double-Precision | |
|---|---|---|---|---|---|
| | | Versus Strategy-1 | Versus Strategy-2 | Versus Strategy-1 | Versus Strategy-2 |
| FEM/Sphere | No | 14.1% | 8.4% | 14.7% | 10.6% |
| FEM/Accelerator | Yes | 23.6% | 21.2% | 9.6% | 9.8% |
| Economics | No | 23.4% | 22.3% | 10.2% | 9.1% |
| Epidemiology | No | 8.4% | 10.2% | 5.3% | 4.7% |
| Protein | No | 13.7% | 7.8% | 14.0% | 9.8% |
| WindTunnel | No | 18.7% | 12.1% | 21.3% | 9.9% |
| QCD | No | 21.9% | 16.1% | 21.9% | 15.7% |
| FEM/Harbor | No | 13.9% | 7.7% | 11.4% | 7.3% |
| Circuit | Yes | 30.6% | 28.8% | 21.0% | 20.4% |
| Web | Yes | 10.7% | 10.6% | 8.8% | 5.9% |
| Geo-Mean | | 17.7% | 14.3% | 13.7% | 10.2% |

a. Comparison between different caching strategies

| Matrix | Speedup for Single-Precision | | | Speedup for Double-Precision | | |
|---|---|---|---|---|---|---|
| | By RCM | By Strategy-3 | Overall Speedup | By RCM | By Strategy-3 | Overall Speedup |
| FEM/Sphere | - | 14.1% | | - | 14.7% | |
| FEM/Accelerator | 5.0% | 23.6% | 29.8% | 4.9% | 9.6% | 15.0% |
| Economics | - | 23.4% | | - | 10.2% | |
| Epidemiology | - | 8.4% | | - | 5.3% | |
| Protein | - | 13.7% | | - | 14.0% | |
| WindTunnel | - | 18.7% | | - | 21.3% | |
| QCD | - | 21.9% | | - | 21.9% | |
| FEM/Harbor | - | 13.9% | | - | 11.4% | |
| Circuit | 0.9% | 30.6% | 31.8% | 1.6% | 21.0% | 22.9% |
| Web | 4.8% | 10.7% | 16.0% | 5.6% | 8.8% | 14.9% |
| Geo-Mean | | | 19.0% | | | 15.0% |

b. Overall Speedup

Table 7.12: Performance Enhancement for GF-100 (GTX-480) GPU

| Matrix | Single-Precision | | | Double-Precision | | |
|---|---|---|---|---|---|---|
| | C1060 | GTX-480 | Speedup | C1060 | GTX-480 | Speedup |
| FEM/Sphere | 0.6661 | 0.4203 | 58.5% | 1.0835 | 0.5579 | 94.2% |
| FEM/Accelerator | 0.5665 | 0.3982 | 42.3% | 0.8141 | 0.5270 | 54.5% |
| Economics | 0.5133 | 0.2972 | 72.7% | 0.6658 | 0.4035 | 65.0% |
| Epidemiology | 0.2959 | 0.1733 | 70.8% | 0.4724 | 0.2561 | 84.5% |
| Protein | 0.7753 | 0.4545 | 70.6% | 1.1226 | 0.5844 | 92.1% |
| WindTunnel | 1.2649 | 0.8217 | 54.0% | 2.0217 | 1.0825 | 86.8% |
| QCD | 0.2105 | 0.1429 | 47.3% | 0.3182 | 0.1863 | 70.9% |
| FEM/Harbor | 0.5466 | 0.3802 | 43.8% | 0.8896 | 0.4663 | 90.8% |
| Circuit | 0.2918 | 0.2381 | 22.6% | 0.4183 | 0.2912 | 43.7% |
| Web | 0.9564 | 0.6203 | 54.2% | 1.3850 | 0.7630 | 81.5% |
| Geo-Mean | | | 52.9% | | | 75.6% |

* All timing are in milliseconds.

Table 7.13: Performance Comparison – GT-200 and GF-100 GPUs

two architectures differ in the peak memory bandwidth: Tesla C1060 is lower than GeForce GTX-480 in memory bandwidth by about 30% and larger in global memory latency, as shown in Tab.7.1. These hardware differences, together with the firmware differences, contribute partially to the performance gain in the table. Also the cache subsystem difference contributes to the difference in that accesses to the dense vector are now faster and incurs fewer cache misses.

## 7.7 Summary

GPU accelerated HPC systems have become popular and widely adopted. How to exploit the performance potentials of GPUs for Krylov subspace solvers depends heavily on effective optimization of SpMV operations on GPU architecture. Due to the memory bandwidth bound characteristics of SpMV operations, how to reduce the overall memory accesses and enhance cache usage is of crucial importance of high performance SpMV on GPU. With a proper matrix storage scheme and the use of matrix permutation schemes such as RCM, SpMV operations are able to fully exploit the potential of the high peak memory bandwidth available on GPUs. Proper matrix format such as ELLPACK for general sparse matrices is necessary to ensure the coalesced memory accesses and high performance of SpMV operations. The techniques developed in this chapter serve as the foundation for Krylov subspace iterative solvers, which uses SpMV for the generation of subspace bases.

The optimizations proposed in this chapter includes: (1) matrix bandwidth reduction based optimization, and (2) differentiated cache access for Fermi architecture. Matrix bandwidth reduction is an effective optimization to SpMV performance on GPUs. It achieves two aspects of performance enhancements: (1) better locality for accessing the dense vector $x$, and (2) column index information compression. The overall speedup is 16% and 13%, for single-precision and double precision, respectively. On GT-100 GPU with better cache support, we propose the differentiated strategy for accessing data in $A$ and $x$ for better utilization and avoidance of cache contamination. By inlining PTX assembly to CUDA codes to differentiate the accesses to cache, on average we further achieve 19%

and 15% speedup for Single Precision and Double Precision operations respectively.

Column index compression can be combined with conventional Register Blocking techniques, to further reduce the accessed memory amount for index information in ELLPACK format. Also, cache-based performance modeling can be introduced to the analysis of GPUs, especially new GPU with more complete cache support such as GF-100. These two aspects serve as potential gdirections for future researches.

# Chapter 8

# GPU-based Iterative Solvers and Preconditioners

## 8.1 Introduction

In this chapter we discuss the acceleration of preconditioned Krylov subspace iterative solvers on GPU platforms. There are two major components of Krylov solvers: (1) the iterative solver algorithm, and (2) the preconditioners. Iterative solvers depend on matrix-vector products for the generation of Krylov subspace basis. Long recurrence based solvers such as GMRES include an orthgonalization process. Since these operations contain much inherent parallelism and very low computation-data ratio, the performance of iterative solvers is mainly bounded by the available memory bandwidth. The iterative solvers can be ported to GPU platform without much performance problems. However, preconditioners, especially general-purpose ones based on incomplete factorizations such as Incomplete Cholesky (IC) or Incomplete LU (ILU) factorization, have not been successfully ported to the GPU platform so far. The main reason for this is the preconditioning operation with IC or ILU are virtually substitution process which is inherently sequential and contains limited parallelism. At the same time, preconditioners based on inverse form usually have low memory efficiency and have not gained much popularity, despite the fact that the preconditioning operations with them are based on matrix-vector products and can be efficiently executed on GPU. To utilize GPU for the high performance iterative solvers, it is crucially important to design general purpose preconditioners which can exploit the parallelism on GPUs.

In this chapter we propose a new preconditioning framework based on multilevel framework and approximate inverse preconditioners. The proposed framework, denoted multilevel approximate inverse preconditioner (ML-AINV), bears close relationship with incomplete factorizations. We construct multilevel structure based on Independent Sets (INDSET) and symbolic analysis of the factorization process with elimination tree (ETREE). With ML-AINV, the preconditioning process includes a series of matrix vector products, which can be executed on GPU efficiently. Unlike the approximate inverse preconditioner (AINV), ML-AINV achieves good memory efficiency with the multilevel framework. Contrary to the traditional incomplete factorization preconditioners which rely on substitutions, the preconditioning of ML-AINV is based on (sparse) matrix-vector products (SpMV), which can fully exploit the parallelism available on GPUs.

---

Part of this chapter is published in conference paper [86] and journal paper [91]

The rest of the chapter is organized as follows. We analyze the various aspects of the performance of iterative solvers on GPU platforms in Section 8.2, including the generation of Krylov subspace basis and the orthogonalization process of long-recurrence based iterative solvers. On the preconditioner side, before introducing ML-AINV, symbolic analysis with elimination tree (ETREE) is introduced in Section 8.3. Also as a reference, a short introduction to approximate inverse (AINV) preconditioner is included in Section 8.4, with the analysis of its shortcoming in terms of memory efficiency. Memory-efficient inverse-based preconditioning is covered in Section 8.5. Specifically, we propose the design of ML-AINV in Section 8.5.2. Section 8.5.3 covers the experiments of ML-AINV with matrices from various application areas, including Power Grid simulation. Comparison of convergence properties and performance with ILU preconditioners is also included. Section 8.7 concludes this chapter.

## 8.2 Iterative Solvers on GPU-based Accelerated Platforms

Krylov solvers seek the minimization of error norm for a given linear system of $Ax = b$ within the Krylov subspace :

$$\{r_0, Ar_0, \dots A^m r_0\} \tag{8.1}$$

$A$ is a matrix of size $n \times n$. It is usually a sparse matrix. The sparsity pattern of $A$ depends on the application area and specific treatment of the numerical discretization schemes. For example, matrices generated from finite difference applications usually have regular structures, with non-zero elements only occuring on a few diagonals. Matrices generated from finite element applications usually have a slightly more randomized structure, and non-zero element count per row is usually not constant. Power network simulation usually generates matrices with randomly sparse structure, reflecting the nature of the actual network topology.

Depending on the properties of $A$, different iterative solvers are applied for the linear systems based on $A$. Here we list two popular iterative solvers: GMRES and CG . GMRES is a long-recurrence based iterative solver for unsymmetric matrix $A$. It minimizes error norm within the Krylov subspace. CG is designed for symmetric positive-definite matrices, which minimizes the $A$-norm of the error.

```
   Input: Sparse matrix $A$
          Right hand side $\vec{b}$
          Initial guess $\vec{x}_0$
   Output: Solution $\vec{x}$
 1 $\vec{r}_0 \leftarrow \vec{b} - A\vec{x}_0$ ;
 2 $\vec{v}_1 \leftarrow$ normalized $r_0$ ;
 3 for $i \leftarrow 1$ to $m$ do
 4     $\vec{w}_i \leftarrow A\vec{v}_i$ ;
 5     Orthogonalize $\vec{w}_i$ against $\vec{v}_j$, for $1 \leq j \leq i$ ;
 6     if $\| \vec{w}_i \|_2 < \epsilon$ then
 7         Break ;
 8     end
 9     Update Hessenberg matrix $\bar{H}$ ;
10     $\vec{v}_{i+1} \leftarrow$ normalized $\vec{w}_i$ ;
11 end
12 $\vec{y}_m \leftarrow \arg\min_{\vec{y}} \|\beta\vec{e}_1 - \bar{H}\vec{y}\|_2$ ;
13 $\vec{x} \leftarrow \vec{x}_0 + \sum_{i=1}^{m} y_i \vec{v}_i$ ;
14 return $\vec{x}$ ;
```

**Algorithm 10**: GMRES

```
   Input: Sparse matrix $A$
          Right hand side $\vec{b}$
          Initial guess $\vec{x}_0$
   Output: Solution $\vec{x}$
 1 $\vec{r}_0 \leftarrow \vec{b} - A\vec{x}_0$ ;
 2 $\vec{p}_0 \leftarrow$ normalized $r_0$ ;
 3 for $i \leftarrow 0$ to $n$ do
 4     $\alpha_i \leftarrow \frac{\langle \vec{r}_i, \vec{r}_i \rangle}{\langle A\vec{p}_i, \vec{p}_i \rangle}$ ;
 5     $\vec{x}_{i+1} \leftarrow \vec{x}_i + \alpha_i \vec{p}_i$ ;
 6     $\vec{r}_{i+1} \leftarrow \vec{r}_i - \alpha_i A \vec{p}_i$ ;
 7     $\beta_i \leftarrow \frac{\langle \vec{r}_{i+1}, \vec{r}_{i+1} \rangle}{\langle \vec{r}_i, \vec{r}_i \rangle}$ ;
 8     $\vec{p}_{i+1} \leftarrow \vec{r}_{i+1} + \beta_i \vec{p}_i$ ;
 9 end
10 return $\vec{x}_i$ ;
```

**Algorithm 11**: CG

One major computational task in GMRES, CG , and various other solvers is the generation of the Krylov subspace. For CG it is on line 1, 4 and 6, and for GMRES on line 4. This involves the product of $A$ and the current residue $r$. With a sparse $A$, the generation of the next subspace basis vector actually involves a sparse matrix-dense vector multiplication, i.e., SpMV. In GMRES, due to the use of Arnoldi process and the unsymmetry of the Hessenberg matrix, restart is used in practice. This is mandatory for any long recurrence based algorithms used in practice, such as GCR . The long recurrence requires orthogonalization of newly generated basis vector against the basis generated previously, as on line 5. Besides generation of bases and orthogonalization, other components in GMRES

| Name | $n$ | $nnz$ | Application Area |
|---|---|---|---|
| Protein | 36417 | 4344765 | Protein data bank 1HYS |
| FEM/Cantiliver | 62451 | 4007383 | FEM cantiliver |
| WindTunnel | 217918 | 11634424 | Pressurized wind tunnel |
| Epidemiology | 525825 | 2100225 | 2D Markov model for epedemiology |
| Circuit | 170998 | 958936 | Motorola circuit simulation |
| Petro | 50400 | 1585944 | Petroleum reservoir simulation |
| OPF | 2063494 | 8130343 | Optimal power-flow (optimization) |
| Cube | 101492 | 874378 | FEM, electromagnetics |
| TDS-10188 | 25308 | 160159 | Time-domain simulation of power network |
| Parabolic | 525825 | 2100225 | Parabolic FEM, diffusion-convection reaction |

Table 8.1: Test matrices for SpMV and Krylov subspace bases generation

and CG are level-1 BLAS operations such as linear combinations of vectors and inner products, which are all basic and easy to compute and will not be subjected to further analysis. Hence we only focus on the 2 main part of the Krylov solvers: (1) generation of Krylov subspace basis, and (2) long-recurrence related orthogonalization. Section 8.2.1 reports the performance comparison for the kernel of generation of bases, i.e., SpMV for various matrices. Section 8.2.2 describes the performance test of the orthogonalization operation which is required by long-recurrence algorithms such as GMRES.

### 8.2.1 Generation of Krylov Subspace – SpMV Operations

We test a suite of matrices that are candidates for Krylov solvers. The description of the matrices is listed in Tab.8.1. They are from various application areas. Some are from the matrix collection from University of Florida [45] and Matrix Market [22], while others are from our own simulations, including the TDS simulation Jacobian matrix from the 10188 case.

Fig.8.1 shows the speedup of SpMV operations on GPU compared to on CPU. On CPU we use traditional CSR format for the performance test. On GPU we use the optimized SpMV CUDA kernel as described in Chapter 7. Note that from Chapter 7 we know that the main performance bounding factor for SpMV is the memory bandwidth. Potentially GPU has a clear lead in the performance for SpMV due to the much higher theoretical memory bandwidth as compared with CPU. The speedups for each matrix varies for several reasons: (1) with smaller matrix sizes, the overhead in GPU operations, especially in kernel invocation, plays a more important factor, (2) various matrix formats such as ELLPACK, introduce paddings that have negative effects on the SpMV performance for GPU. The geometric mean of the actual speedup's reflects the theoretical performance which is the ratio between the memory bandwidth of GPU and that of CPU.

### 8.2.2 Long-Recurrence Krylov Solvers – Orthogonalization

GMRES and GCR depend on the long recurrence and require restarts or truncation in practice. The orthogonalization process involves orthogonalizing a new vector against the vectors in the orthogonal basis computed in previous iterations. Here we use modified Gram-Schimidt (MGS) for the orthogonalization because it poses more parallelism. The adoption of MGS is mainly based on these considerations: (1) it has better numerical

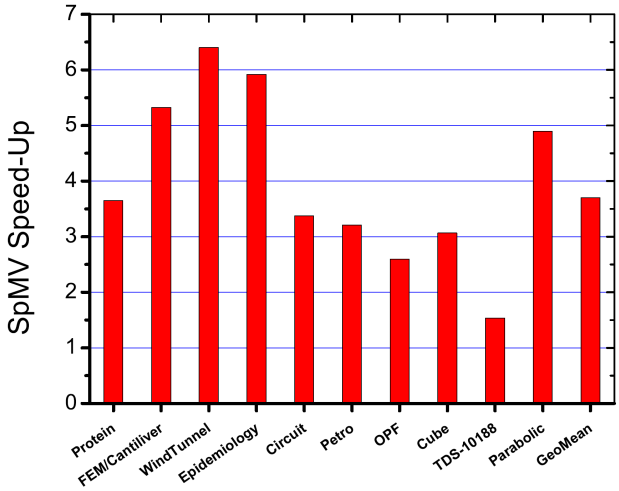


Figure 8.1: SpMV Speedup – GPU v.s. CPU

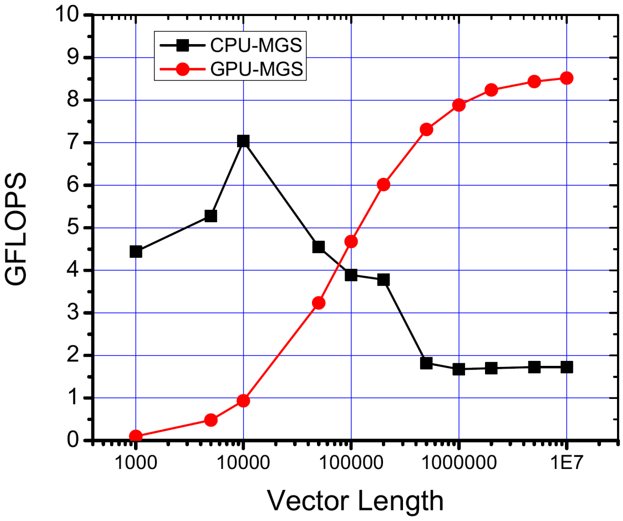


Figure 8.2: Performance comparison between CPU and GPU based MGS

stability than the standard Gram-Schimidt process, (2) it has high parallelism in BLAS-1 operations and a lower overall computation amount compared with Householder reflector based algorithms (see p.g. 158 in [72]). Fig.8.2 shows the performance of MGS orthogonalization of a vector against a basis of 32 orthogonal vectors on CPU and GPU using CUDA. This micro-benchmark mimics the scenario of the orthogonalization process in GMRES or GCR . We vary the length of the vectors $n$ from 1000 to $10^7$, which corresponds to matrix size ranging from $10^3$ to $10^7$.

When $n$ is small, the vectors of the entire orthogonal basis can be loaded in the CPU cache, the performance on CPU is higher than on GPU. The performance on CPU now translates to the cache bandwidth rather than the memory bandwidth. Because on GPU there is no cache used for MGS, the CUBLAS performance only saturates when $n$ is large, with lower performance for short vectors. For long vectors, especially when $n$ exceeds $10^5$, GPU shows better performance, and the FLOPS ratio between CPU and GPU is close to the memory bandwidth ratio of 5. When $n$ is small, CUBLAS based MGS also suffers from other factors such as high startup overhead, which may be reduced in future CUBLAS versions.

Note that the performance test for MGS in Fig.8.2 is a micro-benchmark. In Krylov solvers, due to that SpMV, inner products and other operations are also calculated in addition to orthogonalization operations. Performance advantages of CPUs by caches in orthogonalization would be compromised due to the contamination of the data accessed by other operations, especially SpMV which involves access to matrix data. Hence the actual performance of CPU-based MGS orthogonalization as shown in Fig.8.2 is the best case for CPU. Actual performance with CPU may be compromised due to the aforementioned factors. On the contrary, since no caching is used for the GPU-based version and MGS is free from cache contamination problems, it closely reflects the actual performance of orthogonalization operation on GPU.

From Section 8.2.1 and Section 8.2.2 it can be concluded that the major part of the Krylov solvers can be efficiently implemented on GPUs with CUDA. The main performance profile for Krylov subspace generation and orthogonalization with MGS is memory bandwidth bound. GPU has very high theoretical memory bandwidth compared with state-of-the art CPU. Especially when the matrix is large, the overhead in GPU operation can be amortized and the benefit of much higher memory bandwidth translates to 4 to 5 times speedup in various parts of Krylov solvers.

## 8.3 Symbolic Analysis for Incomplete Factorizations based Preconditioners

This section and the following two section describes the preconditioner design which is optimized for GPU platform by efficient utilization of the massively parallelism available on GPUs. We consider general purposed preconditioners which are based on incomplete factorization such as Incomplete Cholesky (IC) and Incomplete LU (ILU). Before diving into any specific preconditioners, in this section we first layout the theoretical framework for the analysis of the incomplete factorization based preconditioners. We use elimination tree (ETree) [46] for the symbolic analysis of the (incomplete) factorization of sparse matrices. ETree formally determines: (1) the elimination partial order in the factorization, and (2) transitive reduction for the DAG graph defined by the factorization.

Suppose that $A$ is either a symmetric positive definite matrix, or a structurally symmetric matrix which accepts LU(or ILU) factorization without the need for pivoting. Then ETree can be used to analyze (incomplete) Cholesky factorization or the (incomplete) LU factorizations for $A$. Note that this also applies to un-symmetric matrices which are nearly symmetric in structure: for these matrices we base the ETree analysis on the sparsity pattern of $A^* = A + A^T$.

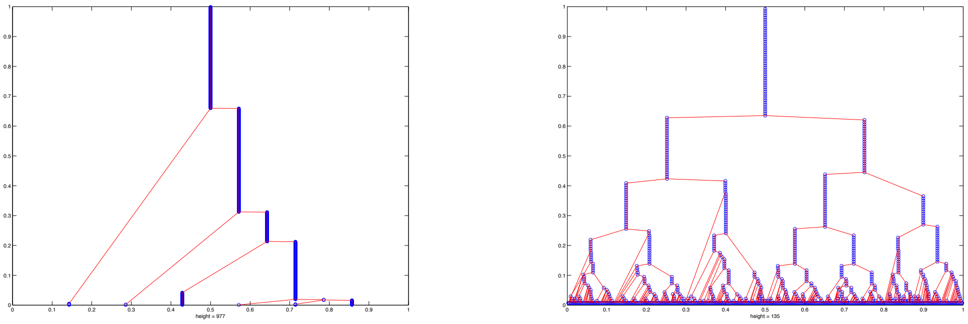
In the following we analyze the non-zero structure of $L$ matrix in IC(as $A \simeq LL^T$ ) or in ILU(as $A \simeq LU$). For ILU, the sparsity pattern of $U$ matrix can be analyzed in a similar way as $L$, given the assumptions above.

Let $G_A$ be the graph defined by $A$, and similarly $G_L$ the directed acyclic graph (DAG) defined by $L$, $G_T$ the DAG defined by ETree of $A$, then following theorems yield:

**Theorem 3.** *$G_T$ is a transitive reduction for $G_L$.*

**Theorem 4.** *The leaves of the ETree of matrix $A$ form an INDSET of the $G_A$.*

**Theorem 5.** *$G_L$ is the transitive closure of $G_T$ and ETree. $G_T \subset G_L \subset G_{L^{-1}}$.*



a. With RCM permutation. b. With ND permutation.

Figure 8.3: Elimination tree of “orsirr_1” treated with RCM and ND permutations

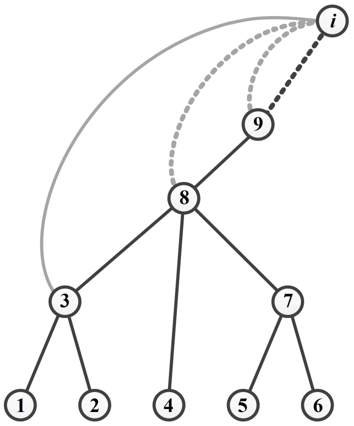


Figure 8.4: ETREE– Fill-in Characterization

In Fig.8.3 we illustrate the ETREE of “orsirr_1” matrix of size 1030 [20], with Reverse Cuthill-Mckee (RCM) and Nested-Dissection (ND) permutations. RCM produces an ETREE which is tall and sequential, while ND produces a much shorter, wider and more balanced ETREE. This improves the inherent parallelism in factorization and perconditioning operations.

ETREE is the ideal tool for the analysis of fill-in’s during the elimination/factorization process. Fig.8.4 shows a sample scenario of fill-in’s being introduced during the elimination process. Suppose that in $G_A$ there exists an edge between vertex 3 and vertex $i$ (curved line in solid grey), and no edge exists between vertex 8 (or 9) and vertex $i$. Then through the elimination process, fill-in would occur between vertex 8 and vertex $i$, and afterwards between vertex 9 and vertex $i$. Fill-in’s are shown in curved, dotted lines in grey in Fig.8.4. The linkage between nodes are carried up in the elimination tree and through the elimination process.

## 8.4 Preconditioners on GPUs – **AINV** Preconditioners

In this section we analyze the $A$-biconjugate based approximate inverse preconditioner, AINV [35]. The $A$-biconjugation process generates matrices: $W$, $Z$ and $D$. $W$ and $Z$ are upper triangular matrices, while $D$ is a diagonal matrix. In the exact factorization, we have:

$$W^T AZ = D \tag{8.2}$$

In fact, $W^T$ , $Z$ and $D$ correspond to the $L$, $U$ and $D'$ in the LDU factorization of $A$ ($A = LD'U$) in the following way:

$$\begin{cases} W^T & = L^{-1} \\ Z & = U^{-1} \\ D & = D' \end{cases} \tag{8.3}$$

Dropping values during the $A$-biconjugation results in incomplete factorization of $A$ in the inverse form. With the matrices from incomplete factorization, we have: $A \simeq W^{-T} D Z^{-1}$. When $W^T$, $D$ and $Z$ are used for preconditioning, the effective linear operator used for iterations is: $(D^{-1}W^T)A(Z)$. Here we use left-right preconditioning, with $D^{-1}W^T$ as the left preconditioner and $Z$ as the right preconditioner.

The preconditioning process using AINV involves matrix-vector products with $Z$ and $W^T$. Compared with the substitution process which is the preconditioning with IC or ILU, matrix-vector products reveal more fine-grain level parallelism and are suitable for GPU platforms.

```
Input: Square matrix $A$ ($a_i$ and $c_i$ is the $i$-th column and row of $A$, respectively
Output: Factorized matrices $W$, $Z$ and $D$
$W \leftarrow I$ (i.e., $\vec{w}_i^{(0)} = \vec{e}_i$ for $1 \leq i \leq n$);
$Z \leftarrow I$ (i.e., $\vec{z}_i^{(0)} = \vec{e}_i$ for $1 \leq i \leq n$);
for $i = 1, 2, ..., n$ do
    for $j = i, ..., n$ do
        $p_j^{(i-1)} = \langle \vec{a}_i, \vec{z}_j^{(i-1)} \rangle$;
        $q_j^{(i-1)} = \langle \vec{c}_i, \vec{w}_j^{(i-1)} \rangle$;
    end
    for $j = i+1, ..., n$ do
        $\vec{z}_j^{(i)} = \vec{z}_j^{(i-1)} - \frac{p_j^{(i-1)}}{p_i^{(i-1)}} \vec{z}_i^{(i-1)}$;
        $\vec{w}_j^{(i)} = \vec{w}_j^{(i-1)} - \frac{q_j^{(i-1)}}{q_i^{(i-1)}} \vec{w}_i^{(i-1)}$;
    end
end
return $W$, $Z$ and $D = \begin{bmatrix} p_1^{(0)} & & & \\ & p_2^{(1)} & & \\ & & \ddots & \\ & & & p_n^{(n-1)} \end{bmatrix}$;
```

**Algorithm 12**: $A$-biconjugate

For reference, the algorithm for right-looking version of $A$-biconjugate (original form as in [35]) is shown in Algorithm 12. At the beginning, $W$ and $Z$ are initialized as identity matrices. Denote $\vec{w}_i^{(j)}$ and $\vec{z}_i^{(j)}$ the value of the $i$-th column of $W$ and $Z$ at the $j$-th iteration respectively. Then we have $\vec{w}_i^{(0)} = \vec{z}_i^{(0)} = \vec{e}_i$. The $i$-th iteration updates the $i + 1$

to the $n$-th column of $W$ and $Z$ with the $i$-th column of $W$ and $Z$, respectively. Hence after the $(i-1)$-th and before the $i$-th iteration, the values for $\overrightarrow{w_k}$ (with $1 \leq k \leq i$) have already been computed in its final form. We can consider the iteration number as a version number for each column:

1. The final version number for the $i$-th column of $W$ or $Z$ is $(i-1)$
2. At the $i$-th iteration, we update the columns of number $(i+1)$ to $n$ from version $(i-1)$ to version $i$, resulting in the final version of the $i$-th column.

The updating process involves computing linear combinations of vectors. The value for combinations are also versioned, namely, $p_i^{(j)}$. We formulate the values for $p_i^{(j)}$'s into a matrix, and denote as $P_{hist}$. Note that the main diagonal of $P_{hist}$ forms $D$.

$$P_{hist} = \begin{bmatrix} p_1^{(0)} & & & & & \\ p_2^{(0)} & p_2^{(1)} & & & & \\ \vdots & \vdots & \ddots & & & \\ p_i^{(0)} & p_i^{(1)} & \dots & p_i^{(i-1)} & & \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ p_n^{(0)} & p_n^{(1)} & \dots & p_n^{(i-1)} & \dots & p_n^{(n-1)} \end{bmatrix} \tag{8.4}$$

For a given sparse matrix $A$, the following theorem provides the sparsity pattern of $P_{hist}$ based on $A$:

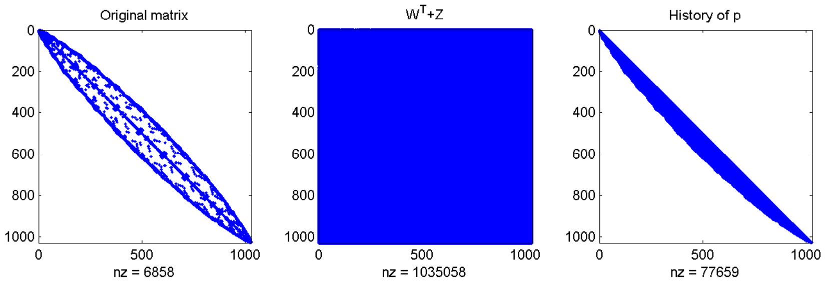
**Theorem 6.** *The sparsity pattern of $P_{hist}$ matrix in $A$-biconjugate algorithm is the same as that of $L$, where $L$ is from the LU (or Cholesky) factorization of $A$.*

*Proof.* See [4]. □

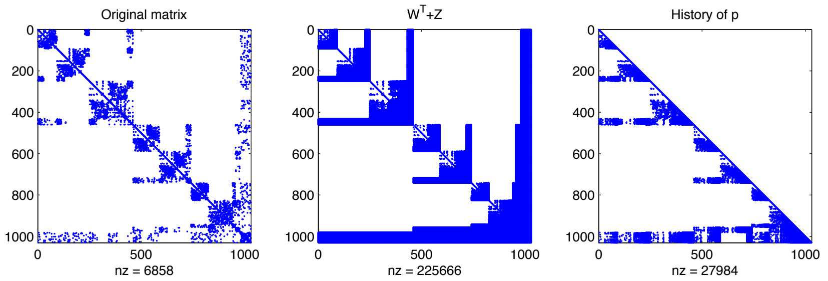
Fig.8.5 illustrates the sparsity patterns of $W^T+Z$ and $P_{hist}$ for "orsirr_1", when treated with RCM and Nested Dissection. From the theorem above it is immediate that the non-zero elements in $P_{hist}$ indicates an upperbound of the memory usage of the incomplete factorization.

With RCM permutations, the matrix has a low bandwidth/profile, the non-zero element in $P_{hist}$ is kept low due to the reduced profile, although it results in more non-zero's compared with Nested Dissection. However, what is important is not the non-zero element in $P_{hist}$ but in $W$ and $Z$. With RCM, the ETREE of "orsirr_1" is almost sequential. According to Theorem 2, the transitive closure of a "sequential" DAG is a "dense" DAG, in the sense that the corresponding matrix is dense. This is reflected by the sparsity pattern of $W^T+Z$. The non-zero element count in $W^T+Z$ is almost 14 times that in $P_{hist}$ (or $L$). When permuted with Nested Dissection, the height of the ETREE is reduced by 86% as compared with RCM permutation. The resulting $nnz$ in $W^T+Z$ is reduced by 75%. But still the $nnz$ in $W^T+Z$ is about 8 times that of $P_{hist}$. Hence generally speaking, the edge count in the transitive closure of $G_T$ (i.e., the DAG defined by ETREE) is on the order of $O(h)$, where $h$ is the height of ETREE. The edge count serves as an upperbound for the non-zero element count in $W$ and $Z$. This has 2 implications:

- Reducing the tree height serves as an effective heuristics to reduce the number of potential non-zero elements in $W$ and $Z$.



a. With RCM permutation



b. With Nested Dissection permutation

Figure 8.5: Sparsity of resulting matrices of $A$-biconjugate algorithm on “orsirr_1”.

- When using AINV instead of ILU or IC, potentially much more severe dropping would ensue: many more non-zero elements have to be dropped under a certain memory usage budget. This analysis is also reflected in [34] that Nested Dissection usually yields better convergence with AINV due to less severe droppings.

From the theoretical analysis above we conclude that the high memory usage of AINV offsets the advantage of the easy, parallelizable preconditioning operation with AINV. Even if high parallelism and performance is achieved by matrix-vector products for preconditioning, the total amount of the memory accessed is far more than that with IC and ILU. Preconditioners is needed for GPU-based platforms that can both utilize the massive parallelism of GPUs and at the same time have feature good memory efficiency as their traditional IC or ILU counterparts.

## 8.5 Preconditioners on GPUs – Memory Efficient Variants

We focus on inverse-based preconditioners with incomplete factorization and good memory efficiency. Traditional ILU or IC preconditioners cannot fully exploit the parallelization due to that preconditioning operation is based on substitution, which is sequential and contains limited parallelism. The scalability of these operations on GPU is very limited. We target inverse-based preconditioner because inverse-based formulation enables sparse matrix-vector products (SpMV) based preconditioning operations. However, our design of the preconditioner for GPU should overcome the excessive fill-in's as in approximate inverse preconditioners such as AINV. Besides, the preconditioners should have good convergence properties similar to incomplete factorizations, such as ILU or IC.

In this section we propose the design of preconditioner which satisfies the requirements above: Multi-Level Approximate Inverse preconditioner (ML-AINV). We assume that with proper preprocessing, pivoting is not needed for the preconditioner construction. Also without loss of generality, we assume a structurally symmetric matrix $A$ as the matrix to be preconditioned. Then all the symbolic analysis can be applied to symmetric matrices in which IC preconditioners are applied. Note that a non-symmetric matrix $A$ can also be considered as structurally symmetric by using the sparsity pattern of $A + A^T$. Before introducing the details of ML-AINV, in Section 8.5.1 we shortly introduce an SpMV based preconditioning scheme, denoted as "SpMV based Substitution". It is closely related to ML-AINV in terms of formulation. Afterwards, Section 8.5.2 includes the design details of ML-AINV preconditioner. Section 8.5.3 compares ML-AINV and "SpMV based substitution" and points out the innate relationship and differences between them.

### 8.5.1 SpMV based Substitution for Incomplete Factorizations

The first solution to the limited parallelism in substitution process is to transform the substitution process based on $L$ (or $U$) into a series of matrix-vector products without introducing any extra non-zero elements beyond that of $L$ (or $U$). The algorithm is proposed in [23]. Given an existing (incomplete) factorization $L$, it constructs a series of lower triangular matrices, each of which can be inverted in place. For lower-triangular matrix $L$ of size $n \times n$, it can be written as the multiplication of a series of elementary triangular matrices: $L_i$ for $1 \leq i \leq n$. Except the main diagonal, $L_i$ only have non-zero elements in its $i$-th column, and the $i$-th column equals that of $L$.

$$L = L_1 L_2 \dots L_n \tag{8.5}$$

Based on the transitive closure structure of $G_L$ and topological sorting, we can arrange $L$ into:

$$\begin{aligned} L &= L_1 L_2 \dots L_n & (8.6)\\ &= (L_{m_0+1} \dots L_{m_1})(L_{m_1+1} \dots L_{m_2}) \dots (L_{m_{k-1}+1} \dots L_{m_k}) & (8.7) \end{aligned}$$

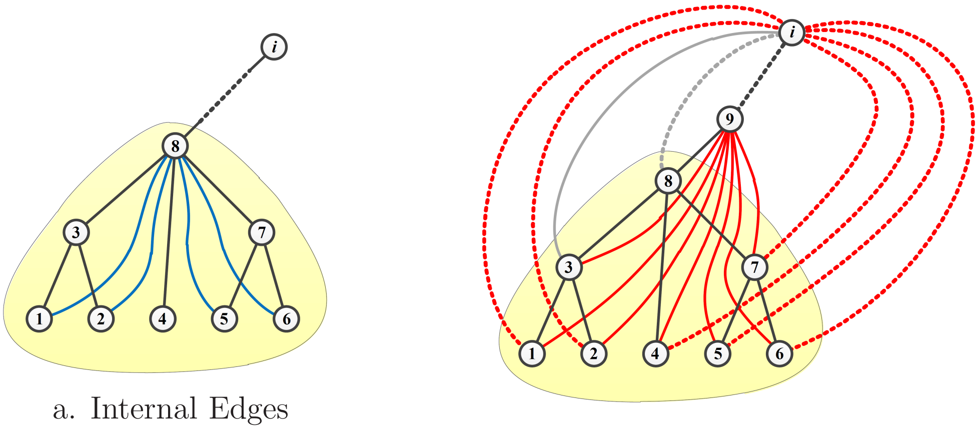
With $0 = m_0 \leq m_1 \leq \cdots \leq m_{k-1} \leq m_k = n$. Each segment, i.e., $L^*_{i+1} = (L_{m_i+1} \dots L_{m_i+1})$ for $1 \leq i \leq k$ can be inverted in place: $(L^*_i)^{-1}$ has the same sparse structure as $L^*_i$. This is equivalent to that $G_{L^*_i}$ is transitively closed.

The substitution process with $L$ can then be transformed into:

$$\begin{aligned} L \backslash b &= (L^*_1 L^*_2 \dots L^*_k) \backslash b & (8.8)\\ &= (L^*_k)^{-1} (L^*_{k-1})^{-1} \dots (L^*_2)^{-1} (L^*_1)^{-1} b & (8.9) \end{aligned}$$

Note that each $(L^*_i)^{-1}$ (for $1 \leq i \leq k$) is explicitly computed with no extra storage overhead, and the substitution process is transformed into a series of $k$ (sparse) matrix-vector pruducts (SpMV) operations.

We analyze the implication of property of $L^*_i$ which are transitively closed using ETREE. For the ETREE shown in Fig.8.6, whether we can merge the vertices under vertex 8 into



a. Internal Edges

b. External Edges

Figure 8.6: ETREE– Example for Transitive Reduction

a single $L_i^*$ depends whether these nodes form a transitively closed subgraph. This is equivalent to:

- The subgraph with vertices 1 to 8 are transitively closed by itself. This is the requirement on the internal structure of the subgraph, i.e., the diagonal block corresponding to the subgraph can be inverted in place.
- The external links from the vertices from 1 to 8 also should satisfy the requirement of transitive closure.

In Fig.8.6, we assess the combination of vertex 1 to 8 to be included in $L_1^*$. If it can be all included in $L_i^*$, then: (1) in Fig.8.6.a, all the links of blue-color should be present in $G_L$, and (2) in Fig.8.6.b, all the solid red edges should be present in $G_L$, and (3) in Fig.8.6.b if there is an uplink such as between vertex-3 and vertex-$i$, then all the dotted red edges should be present in $G_L$. Following the analysis above, we can deduce the theorem below:

**Theorem 7.** *Suppose $L$ is the Cholesky factorization of some matrix $A$. A subtree under the vertex $i$ in $G_L$ can be included in $L_1^*$ which can be inverted in place i.f.f. (1) the subgraph formed by of all the vertices in the subtree forms a transitive closure, and (2) for the destination vertex of any uplink, the uplinks are present between the leaves of the subtree and that destination vertex.*

*Proof.* The subgraph of the vertices in the subtree should satisfy the requirement of transitive closure. This part is trivial due to the fact that we require that the corresponding diagonal block in the matrix can be inverted in place. For the second part, suppose that $L_1^*$ can be inverted in place, and we have an uplink linking to node $i$ starting from a non-leaf node $j$ in the tree but not from a leaf $k$ in the subtree under $j$, then from the definition of transitive closure: $k \rightarrow j \rightarrow i$ implies $k \rightarrow i$. This implies that the edge between $i$ and $k$ is present in $G_{(L_1^*)^{-1}}$. This contradicts with our assumptions above. Hence for any of the destinations of the uplinks from the subtree, there should be edges between all the leaves and this destination vertex in $G_L$.

On the reverse part, suppose the tree root is $r$. If for any destination node $i$ that has edges from the subtree, there are edges between $i$ and all the leaves of the subtree, then according to the property of the elimination tree, there are edges between any vertex in the subtree and $i$. Thus the property of transitive closure is guaranteed for the external edges. Combined with the fact that the subgraph defined by the subtree vertices is transitively closed. Therefore inverting the subtree related parts in $L$ is in place, hence can be included in $L_1^*$. This completes the proof. ☐

**Theorem 8.** *If a subtree in the elimination tree can be included into $L_1^*$, then the row structure of the sub-matrix representing uplinks must be either dense or all zero.*

*Proof.* This follows immediate from Theorem 7 that if there is an edge between some vertex of this subtree and a vertex above the tree, then every vertex in the subtree is connected to that vertex. Thus the rows that corresponds to these vertices that are connected to this tree are dense. For those vertices that are not connected to this subtree, the corresponding rows are empty, with only zero elements. This completes the proof. ☐

For solving/preconditioning of non-symmetric matrices with this scheme, we have $L$ and $U$ matrix which are organized into the the formulation:

$$L = L_1^* L_2^* \dots L_k^* \tag{8.10}$$

$$U = U_k^* U_{k-1}^* \dots U_1^* \tag{8.11}$$

Each sub-matrix, i.e., $L_s^*$ or $U_s^*$ (for $1 \leq s \leq k$) is invertible in place. The substitution process with $L$ and $U$ is transformed into $2k$ SpMV operations.

We denote the algorithm mentioned above as "SpMV based Substitution", because that it combines parts of the $L$ matrix which is used for substitution and achieves equivalent effect by using a series of SpMV operations. Each subpart of $L$ can be inverted in place, so the substitution with each of them is converted into an SpMV operation. Each of the SpMV operation contains much inherent parallelism hence is suitable for platforms similar to GPU.

One problem of "SpMV based Substitution" is that it computes inverses of each part based on an existing factorization. Besides, for incomplete factorizations, the value for $k$ is usually large. Potentially there will be performance degradation with a large value of $k$ since too many sequential SpMV operations would be involved, with each operation including too few non-zero elements. The fact that "SpMV based Substitution" only operates on incomplete factorizations does not allow flexible trade-off between the value of $k$ and non-zero element count.

### 8.5.2 ML-AINV– Multilevel Preconditioner with AINV

We aim to design a preconditioners with fine-grain parallelism and good memory efficiency for GPU platforms. The proposed preconditioner is based on incomplete factorization and inverse forms. With inverse-based forms, the preconditioning operation translates into (sparse) matrix-vector products. For large sized matrices, sparse matrix-vector products contain inherent parallelism and is potentially high performance on GPUs. To enhance the memory efficiency and avoid the disadvantages of AINV, we adopt the multilevel framework used in Chapter 4 (there multilevel scheme was used for efficient preconditioning for TDS). The proposed algorithm is denoted Multi-Level Approximate Inverse preconditioner (ML-AINV). By using symbolic analysis, ML-AINV construct multilevel structure based on INDSET's. The preconditioning of the last-level in the multilevel structure is based on AINV. Trade-off is achieved by relaxing the memory usage for INDSET and last-level system by threshold-based strategies. This section discusses all the design aspects for ML-AINV. In the following part ofthe section, firstly the recursive multilevel framework based on INDSET's is introduced. Then the choice for INDSET selection based on ETREE in ML-AINV is described. The termination of the recursive multilevel structure is introduced afterwards. Finally the design of ML-AINV is summarized.

### Recursive Multilevel Formulation

In ML-AINV, a recursive multilevel structure is constructed based on independent sets (INDSET). The matrix $A$ is treated as the system on the first level, and it is mapped onto lower levels in a recursive way. On level $i$, we formulate the system into the following blocked form:

$$P_i A_i P_i^T = \begin{bmatrix} D_i & F_i \\ E_i & C_i \end{bmatrix} \tag{8.12}$$

Number $i$ is the level index, with $A_0$ being the original matrix. $P_i$ is the permutation matrix to transform $A_i$ into the blocked form, with $D_i$ composed of small dense diagonal blocks. The sub-blocks of $D_i$ form a maximal block INDSET of $G_{A_i}$. For the dense structure of $D_i$, we invert it explicitly. Similar to the formulation in Chapter 4, the formulation based on Schur complements is:

$$\begin{bmatrix} D_i & F_i \\ E_i & C_i \end{bmatrix} = \begin{bmatrix} D_i & \\ E_i & I \end{bmatrix} \times \begin{bmatrix} I & D_i^{-1} F_i \\ & C_i - E_i D_i^{-1} F_i \end{bmatrix} \tag{8.13}$$

We denote $S_i = C_i - E_i D_i^{-1} F_i$, and it will serve as the matrix for the next level, i.e., $A_{i+1}$. The original matrix $A$ undergoes a recursive process of mapping from $A_i$ to $A_{i+1}$, until a certain level $m$, when this process is terminated and we obtain a system on the inner most level as $A_m$. Now we treat $A_m$ with a SpMV-based preconditioning technique: AINV preconditioner. We require that the size of $A_m$ is relatively small to avoid too much memory usage of AINV as described in Section 8.4.

### Selection of INDSET's – Relaxed Supernodes

The construction of multilevel structure is based on symbolic analysis of the matrix $A$. This includes the selection of the INDSET's at each level, and the criterion for the level count. ETREE is used for the symbolic analysis. Before the symbolic analysis, we use the heuristics by METIS [7] to reduce fill-in, shorten ETREE, and enhance parallelism.

The selection of INDSET's is based on the analysis over ETREE. We use a single vertex or a dense clique of vertices (or called a supernode, a set of vertices which form a nearly complete connected subgraph) to construct the INDSET. We relax the requirement of full connectivity in cliques by including a set of vertices that are almost fully connected. Define a sparsity of a graph as the proportion of the non-zero element count of the total element count in the matrix the graph represents. Hence we can define quantitatively the relaxation over the cliques in terms of sparsity: cliques with sparsity over certain threshold $\delta_1$ is considered as a relaxed supernode. The value of $\delta_1$ should satisfy: $0 < \delta_1 \leq 1$. If $\delta_1$ is 1, then we require complete connected cliques. The search for cliques is integrated within the framework of ETREE: we evaluate the sparsity of the sub-tree from bottom up. This process is described in the following algorithm.

```
Input: matrix $A$
       elimination Tree $T$
       threshold $\delta_1$
Output: IndSet in $G_A$
$S \leftarrow \emptyset$ ;
for $l$ in leaves of $T$ do
    $p \leftarrow$ parent of $l$ in $T$ ;
    while $l$ is not a root of $T$ do
        evaluate subtree in $T$ rooted at $p$ ;
        if sparsity higher than $\delta_1$ then
            $l \leftarrow p$ ;
            $p \leftarrow$ parent of $p$ in $T$ ;
        end
        else
            remove all the other leaves in the subtree under $l$ to avoid unnecessary
            evaluation ;
            $S \leftarrow \{$ vertices in the subtree under $l$ $\}$ ;
            break ;
        end
    end
end
return $S$ ;
```

**Algorithm 13**: INDSET searching based on ETREE with relaxed supernodes

Relaxing over the sparsity of supernodes has two side effects: (1) the size of supernodes can be larger and the resulting $D_i$ composes a larger proportion in $A_i$, hence the system size is reduced to a larger extent to the next level, (2) the overhead of inverting $D_i$ would be higher due to the lowered sparsity of $D_i$.

### Termination of Recursive Structure Construction

Suppose that the multilevel structure for $A$ consists of $m$ levels (excluding the out-most level), the system on the $m$-th level is denoted as $A_m$. We treat $A_m$ using AINV, so that the preconditioning on the innermost level consists of SpMV operations based on the approximate inverse factorizations of $A_m$. The decision on the value of $m$ also depends on the symbolic analysis. The criterion used in Ml-Ainv is:

- The memory usage of $W$ and $Z$ be within a certain bound based on the memory usage of $L$ and $U$. $W$ and $Z$ are the matrices generated from $A$-biconjugation process, and $L$ and $U$ are matrices from $LU$ decomposition based on $A_m$.

To achieve this, we use a controlling threshold: $\delta_2$ for the evaluation, with $0 < \delta_2 < 1$. If the non-zero element count in $L + U$ is higher than the non-zero element count in $W^T + Z$ multiplied by $\delta_2$, the recursive construction is terminated and $A_m$ is treated as the last-level system. Otherwise, the recursive construction continues to the $(m + 1)$-th level. Note that the computation of the sparsity of $L + U$ and $W^T + Z$ is immediate from the ETree of $A_m$ and $G_{A_m}$.

### Overview of ML-AINV– Construction and Preconditioning

We summarize the construction and preconditioning operation of ML-AINV. The construction of the preconditioner takes 3 phases:

1. Preprocessing: treat $A$ with METIS and apply (diagonal) scaling
2. Symbolic Analysis: construct the multilevel structure
3. Numerical Factorization: construct the multilevel preconditioner numerically

The symbolic analysis process is described in the following algorithm:

```
Input: matrix $A$
       threshold $\delta_1$
       threshold $\delta_2$
Output: multilevel structure for $A$
retrieve elimination tree $T$ of $A$ ;
while $A$ not suitable for AINV with $\delta_2$ do
    retrieve INDSET in $A$ based on $T$ and $\delta_1$ ;
    record vertices in INDSET for matrix $D$ ;
    remove vertices and corresponding edges from $A$ ;
    remove vertices in $D$ from $T$ ;
end
record vertices left over in the last level ;
return Vertex list in each level ;
```

**Algorithm 14**: Symbolic analysis in ML-AINV

Numerical factorization takes another parameter, $\delta_3$, for dropping values to maintain sparsity. On level $i$ (with system $A_i$), element values in $E$, $F$ and $C$ are dropped if they satisfy the following criteria:

$$a_{j,k} < \delta_3 \times a_{j,j} \tag{8.14}$$

Note that values in $D_i$ are never dropped. The numerical factorization process is described in the following algorithm:

```
Input: matrix $A$
       level count $m$
       multilevel structure of $A$
       threshold $\delta_3$
Output: $D_i^{-1}$, $E_i$ and $F_i$ for each level
        $W^T$ and $Z$ for the last level
$i \leftarrow 1$ ;
while $i \leq m$ do
    retrieve matrix $D_i$, $E_i$ and $F_i$ ;
    compute $D_i^{-1}$ ;
    compute $A_{i+1} = C_i - E_i D_i^{-1} F_i$ ;
    drop values in $A_{i+1}$ by $\delta_3$ ;
end
construct AINV preconditioner based on $A_m$ and $\delta_3$, to retrieve $W^T$ and $Z$ ;
return $D_i^{-1}$ $E_i$ and $F_i$ for $1 \leq i \leq m$, $W^T$ and $Z$ ;
```

**Algorithm 15**: Numerical factorization for ML-AINV

The preconditioning process using ML-AINV is similar to that proposed in Chapter 4. First $m$ recursive mapping from the outer-most level to the inner-most level by a series of SpMV operations using $D_i^{-1}$'s and $E_i$'s. Then on the innermost level, preconditioning with AINV preconditioner is carried out by 2 SpMV operations with $W^T$ and $Z$. Finally the preconditioned vector is mapped back to the outer-most level, by a series of SpMV operations using $D_i^{-1}$'s and $F_i$'s. Specifically the preconditioning on the $i$-th level is as follows:

$$\begin{bmatrix} D_i & F_i \\ E_i & C_i \end{bmatrix} \times \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \tag{8.15}$$

$$\begin{bmatrix} D_i & \\ E_i & I \end{bmatrix} \times \begin{bmatrix} I & D_i^{-1}F_i \\ & S_i \end{bmatrix} \times \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} \tag{8.16}$$

$S_i = C_i - E_i D_i^{-1} F_i$. Let $w_1 = D_i^{-1} v_1$ and $w_2 = v_2 - E_i w_1$. We have:

$$\begin{bmatrix} I & D_i^{-1}F_i \\ & S_i \end{bmatrix} \times \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \tag{8.17}$$

Then $u_2$ is computed, i.e., preconditioned by solving $S_i u_2 = w_2$, which incurs recursive process. When solving/preconditioning is finished from the lower level, the value of $u_2$ is available. Then $u_1$ can be calculated as:

$$u_1 = w_1 - D_i^{-1} F u_2 \tag{8.18}$$

We denote $nnz_A$ the number of non-zero elements in matrix $A$. Then the total memory usage of ML-AINV preconditioner can be expressed in Eqs.8.19 and the total amount of memory accessed per preconditioning operation in Eqs.8.20.

$$\sum_{i=1}^{m} \left( nnz_{D_i} + nnz_{E_i} + nnz_{F_i} \right) + (nnz_{W^T} + nnz_Z) \tag{8.19}$$

$$\sum_{i=1}^{m} \left( 2nnz_{D_i} + nnz_{E_i} + nnz_{F_i} \right) + (nnz_{W^T} + nnz_Z) \tag{8.20}$$

### 8.5.3 Comparison of **ML-AINV** and **SpMV** based Substitution

We take a Schur-complement formulation of the problem. Suppose that we have a structurally symmetric matrix $A$ divided into the blocks as follows:

$$PAP^T = \begin{bmatrix} D & F \\ E & C \end{bmatrix} \tag{8.21}$$

$$= \begin{bmatrix} L_D & \\ EU_D^{-1} & I \end{bmatrix} \times \begin{bmatrix} U_D & L_D^{-1}F \\ & C - EU_D^{-1}L_D^{-1}F \end{bmatrix} \tag{8.22}$$

If the vertices in $D$ in Eqs.8.21 satisfy the requirement of transitive reduction in Section 8.5.1, the following criteria are satisfied: (1) $G_{L_D}$ is transitively closed, i.e., $L_D^{-1}$ has the same sparsity structure as $L_D$, and (2) $EU_D^{-1}$ has the same sparsity pattern as $E$. Hence we compute the values in $EU_D^{-1}L_D^{-1}$ and $L_D^{-1}F$ explicitly, so that the substitution with $\begin{bmatrix} L_D & \\ EU_D^{-1} & I \end{bmatrix}$ can be transformed as an SpMV operation with $\begin{bmatrix} L_D^{-1} & \\ EU_D^{-1}L_D^{-1} & I \end{bmatrix}$.

Due to the row structure of $E$ as described in Theorem 6, we can easily deduce that $E$, $EU_D^{-1}$, and $EU_D^{-1}L_D^{-1}$ all have the same structure as $E$: some rows are completely dense while other are completely empty. Such requirement may be tight, resulting in an $D$ matrix with a small size.

Compared with SpMV based substitution, the proposed ML-AINV relaxes the requirement on $D$ and $E$, and the last-level system:

- Relaxed sparsity pattern for $D$: each diagonal block of $D$ should be dense enough, and the inverse of $D$ is computed rather than the $LU$ decomposition.
- Relaxed sparsity pattern for $E$: $EU_D^{-1}$ or $EU_D^{-1}L_D^{-1}$ is not recorded, hence no requirement of the transitive closeness of the external links, as posed by Theorem 6.
- Relaxation in last-level system: In ML-AINV, when the system on current level accepts an $A$-biconjugate factorization which does not contain too many non-zero elements compared with an $LU$ decomposition (controlled by a threshold), we treat it as the last-level system and use AINV for its preconditioning.

From the algebraic representations, ML-AINV can be viewed as a relaxed version of SpMV based substitution. The main relaxation is in the sparsity pattern on each level in ML-AINV by allowing potentially more non-zero elements in order to achieve a the multilevel structure with smaller number of levels.

## 8.6 Experiments with ML-AINV Preconditioners

In this section we evaluate the ML-AINV preconditioners. We focus on two aspects: (1) convergence property, and (2) performance. For the convergence property, we compare the ML-AINV preconditioners with Crout version of ILU (ILU-C) and SuperILU [12]. For performance tests, we compare ML-AINV performance on GPU platform with the performance of SuperILU on CPU. SuperILU is chosen as the representative on CPU platform. It exploits multi-core parallelism on CPU platform with supernodal support to achieve high performance.

Tab.8.2 lists the platform for the performance tests and comparison used throughout this chapter. The CPU and GPU configurations are up to the state-of-the-art platforms for computation as of the year 2010.

SuperILU is compiled with GotoBLAS (ver. 2.0), which is the state-of-the-art BLAS implementation on GPU. ML-AINV relies on inverse-forms for preconditioning, so it uses sparse matrix-vector multiplication operations (SpMV) on GPU.

In Section 8.6.1 we first test the properties of preconditioners with Jacobian matrix-based linear systems from TDS. We also test ML-AINV on several sparse matrices from various scientific applications. Results are included in Section 8.6.2.

The preconditioning overhead of ILU-like preconditioners and ML-AINV are computed differently. The preconditioning in ML-AINV involves access to $D_i$ matrices twice, while in ILU preconditioners the $L$ and $U$ matrices are accessed only once. Hence the time

| Platform | CPU | GPU |
|---|---|---|
| Name | Intel i7-920 | NVIDIA Tesla C1060 |
| Core Configuration | 4 cores | 30 Stream Multiprocessors |
| Cache Organization | 32 KB per core for L1<br>8 MB for L2 | Texture Cache |
| Peak Memory Bandwidth | ∼ 16 GB/s | ∼ 80 GB/s |

Table 8.2: Test Platforms - CPU and GPU

| System | $\delta_1$ | $\delta_2$ | Level Count |
|---|---|---|---|
| 2K | 1.0 | 1.0 | 7 |
| | 0.75 | 0.75 | 5 |
| | 0.75 | 0.5 | 3 |
| 10188 | 1.0 | 1.0 | 10 |
| | 0.75 | 0.75 | 7 |
| | 0.75 | 0.5 | 4 |

Table 8.3: Symbolic analysis of TDS matrices

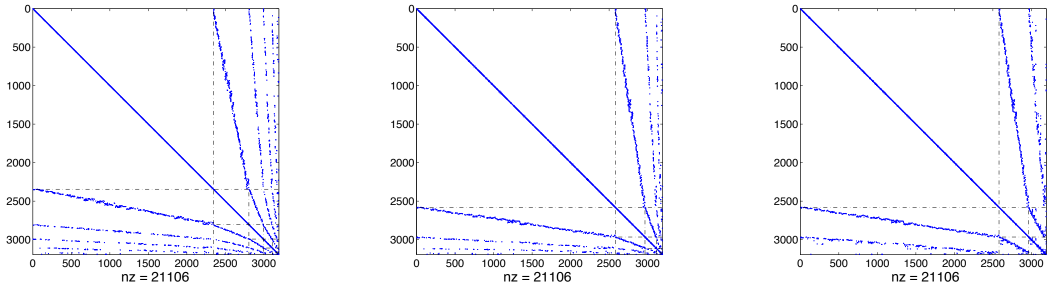
for the preconditioning of ML-AINV should be calculated as (with $T_S$ denoted the SpMV execution time for matrix $S$):

$$T = \sum_{i=1}^{l}\left(2T_{D_i^{-1}} + T_{E_i} + T_{F_i}\right) + T_Z + T_{W^T} \tag{8.23}$$

### 8.6.1 Test on Jacobian Matrices from TDS

We test ML-AINV on two matrices from the 2K and 10188 simulation. We vary the value of $\delta_1$ and $\delta_2$ to evaluate the effects of these parameters on the symbolic analysis. Tab.8.3 shows the level count in ML-AINV w.r.t different combinations of values of $\delta_1$ and $\delta_2$.

When the values of $\delta_1$ and $\delta_2$ decrease, the level count decreases. With a smaller value of $\delta_1$, the size of each level increases due to the relaxation of the sparsity density of each supernode, resulting a smaller size for the next level. For a smaller value of $\delta_2$, the multilevel structure may terminate earlier. Hence the two parameters have a combined effect on the level count. Fig.8.7 shows the multilevel structure for the Jacobian matrix for the 2K system. Boundaries between levels are shown.



a. $\delta_1 = 1.0$, $\delta_2 = 1.0$ b. $\delta_1 = 0.75$, $\delta_2 = 0.75$ c. $\delta_1 = 0.75$, $\delta_2 = 0.5$

Figure 8.7: Sparsity pattern of 2K with ML-AINV permutation

| Case | Preconditioner | | | | $nnz$ | Preconditioning Overhead | Iteration Count |
|---|---|---|---|---|---|---|---|
| | Type | $\delta_1$ | $\delta_2$ | $\delta_3$ | | | |
| 2K | ILU-C | | | $10^{-2}$ | 20074 | | 24 |
| | ILU-C | | | $10^{-3}$ | 25349 | | 8 |
| | ML-AINV | 1.0 | 1.0 | $10^{-2}$ | 28125 | 43487 | 8 |
| | ML-AINV | 0.75 | 0.5 | $10^{-2}$ | 27282 | 51996 | 5 |
| | ML-AINV | 1.0 | 1.0 | $10^{-3}$ | 28584 | 44030 | 8 |
| | ML-AINV | 0.75 | 0.5 | $10^{-3}$ | 27361 | 54532 | 5 |
| 10188 | ILU-C | | | $10^{-4}$ | 172751 | | 78 |
| | ILU-C | | | $10^{-5}$ | 186686 | | 27 |
| | ML-AINV | 1.0 | 1.0 | $10^{-4}$ | 206074 | 296848 | 16 |
| | ML-AINV | 0.75 | 0.5 | $10^{-4}$ | 228770 | 390643 | 16 |
| | ML-AINV | 1.0 | 1.0 | $10^{-5}$ | 207945 | 298993 | 8 |
| | ML-AINV | 0.75 | 0.5 | $10^{-5}$ | 229805 | 393128 | 8 |

* Parameter $\delta_3$ in the table is used as the dropping threshold for ILU-C and SuperILU.

Table 8.4: ML-AINV characteristics on TDS Jacobian matrices

| Matrix | Size | $nnz$ | Application Area |
|---|---|---|---|
| TDS-10188 | 25308 | 160159 | Time Domain Simulation of Power Grids |
| epb3 | 84617 | 463625 | DAE of Heat Transfer |
| Petro | 50400 | 1585944 | Petroleum Resevior Simulation |
| FEM/Cantiliver | 62451 | 4007383 | Cantiliver model (FEM) |
| FEM/Consph | 83334 | 6010480 | Con-Sphere (FEM) |
| filter3D | 106437 | 1406808 | 3D model for optical filter |
| poisson3Db | 85623 | 2374949 | 3D Poisson problem |

Table 8.5: Matrix test suite for ML-AINV

Tab.8.4 shows the timing and preconditioner overhead for ML-AINV with TDS matrices. The value of $\delta_3$ ranges from $10^{-2}$ to $10^{-5}$, depending on the TDS system. Note that the same value of $\delta_3$ is used as the dropping threshold in ILU-C preconditioners. The effect of lowering the value of $\delta_1$ and $\delta_2$ results in more fill-in's within each level in $D_i$'s and the preconditioner for the last level, i.e., in $W^T$ and $Z$. This is reflected in the "$nnz$" and "Preconditioning Overhead" columns in Tab.8.4.

Note that the iteration count for ML-AINV and ILU-C with the same dropping threshold are comparable in general for 2K system. For 10188, ML-AINV generally performs better than ILU-C, with much lower iteration count and comparable memory usage. Note that the preconditioning overhead is indeed higher for ML-AINV than for ILU-C. This translates into the performance comparison in the next subsection.

### 8.6.2 Test of ML-AINV on Matrix Test Suite

Tab.8.5 shows the characteristics of matrix test suite for the comparison between ML-AINV and ILU preconditioners. The convergence properties of ILU-C, SuperILU, and ML-AINV is listed in Tab.8.5. GMRES(50) is used for all the matrices.

Tab.8.6 shows the iteration count required for convergence with ILU-C, SuperILU and ML-AINV. The same dropping threshold is used for ILU-C, SuperILU, and $\delta_3$ in ML-AINV.

| Matrix | $\delta$ | ILU-C | | SuperILU | | ML-AINV | |
|---|---|---|---|---|---|---|---|
| | | *nnz* | Iter. Count | *nnz* | Iter. Count | *nnz* | Iter. Count |
| TDS-10188 | $10^{-4}$ | 172751 | 78 | 232598 | 22 | 206074 | 16 |
| epb3 | $10^{-3}$ | 1805724 | 38 | 3275808 | 17 | 2563610 | 14 |
| Petro | $10^{-4}$ | 445079 | 20 | 578260 | 7 | 833325 | 17 |
| FEM/Cantiliver | $10^{-4}$ | 24897255 | 33 | 3542244 | 8 | 29166570 | 21 |
| FEM/Consph | $10^{-5}$ | 59927690 | 10 | 7112337 | 5 | 81777830 | 6 |
| filter3D | $10^{-4}$ | 15225539 | 45 | 2018064 | 8 | 17293951 | 25 |
| poisson3Db | $10^{-5}$ | 32180520 | 29 | 22530808 | 125 | 39095440 | 35 |

* Parameter $\delta$ in the second column is used as the dropping threshold for ILU-C and SuperILU, as well as the value of $\delta_3$ for ML-AINV.

Table 8.6: Convergence comparison between ML-AINV, ILU-C and SuperILU

| Matrix | SuperILU | | ML-AINV | | |
|---|---|---|---|---|---|
| | Performance (MFLOPS) | Time for Preconditioning | Level | Performance (GFLOPS) | Time for Preconditioning |
| TDS-10188 | 11.93 | 858.0 | 5 | 1.001 | 12.56 |
| epb3 | 49.75 | 2239 | 10 | 2.910 | 29.97 |
| Petro | 25.82 | 313.6 | 8 | 2.207 | 32.67 |
| FEM/Cantiliver | 80.51 | 704.0 | 7 | 7.124 | 202.8 |
| FEM/Consph | 76.19 | 933.5 | 12 | 8.141 | 133.4 |
| filter3D | 53.32 | 605.6 | 9 | 6.135 | 9.827 |
| poisson3Db | 140.2 | 40160 | 14 | 8.565 | 301.3 |

* All timing are in milliseconds

Table 8.7: Performance comparison between ML-AINV and SuperILU

The values of $\delta_1$ and $\delta_2$ are chosen as 0.75 and 0.5, respectively. These values result in as a good balance between performance and overall memory usage. From Tab.8.6 we can see that the convergence of ML-AINV is on the average better than ILU-C. Also ML-AINV consumes more memory than ILU-C on the average. This is due to ML-AINV keeps more elements in $D_i^{-1}$ and in the last level system. Compared with ML-AINV, SuperILU achieves lower iteration counts for some matrices. This is mainly because the more sophisticated preprocessing and the enabling of pivoting available in SuperILU. For "FEM/Cantiliver", "FEM/Consph" and "filter3D", SuperILU uses much less memory than ML-AINV. The main cause for this is the relaxed supernode structure and no control of per-row fill-in amount control. On the average, ML-AINV and SuperILU have comparable convergence, and they have a clear lead in this aspect compared with ILU-C.

Tab.8.7 shows the performance of ML-AINV as compared with SuperILU. Note that SuperILU uses supernodal approach to exploit BLAS based operations which can be vectorized for performance enhancement on CPUs. The performance of SuperILU for preconditioning is mainly on the scale of 10 to 100 MFLOPs. The usual performance of ML-AINV is between the range of 1 to 10 GFLOPs. The performance enhancement is thus on the average 60 to 100 times. The total time during the iterations which is used for preconditioning is also shown in Tab.8.7. Since the iteration count with SuperILU and ML-AINV is different, the total time used for preconditioning shows a speedup range above 3.5 times for "FEM/Cantiliver", up to about 130 times for "poisson3Db". For "FEM/Consph" and "Petro", speedup is between 7 and 10 times. For the other two matrices except for "poisson3Db", the speedup is around 60 times. On the performance side, SpMV based preconditioning shows its good performance effects on GPUs. On the convergence property side, ML-AINV achieves better or comparable iteration count as compared with ILU-C. When compared with SuperILU, ML-AINV achieves over 10 times higher preconditioning efficiency. One thing to note is that SuperILU generally achieves smaller iteration count. To further reduce iteration count, ML-AINV can be aided with certain techniques such as preprocessing to enhance diagonal dominance to alleviate this problem.

## 8.7 Summary

In this chapter we design and implement Krylov subspace iterative solvers and preconditioners on GPU based accelerated platforms. To effectively accelerate iterative solvers and preconditioners on GPU, the algorithm is adapted to match the fine-grain massive parallelism on GPUs. Basic components in iterative solvers such as CG and GMRES can be easily implemented on GPU platforms. The most essential operation in these solvers is the matrix-vector products. Long-recurrence based algorithms also depend on orthogonalization operations. Both these operations are suitable for GPU platforms because of to their memory bandwidth bound characteristics and much inherent parallelism.

On the contrary, preconditioners on GPU, especially non-trivial, incomplete factorization based preconditioners have not been successfully implemented on GPU platforms so far. The fundamental reason is that the preconditioning operations are based on substitutions which contain limited parallelism and are not suitable for GPUs. We propose the use of matrix inverse based preconditioners to make full utilization of the parallelism on GPUs. The proposed preconditioner, ML-AINV combines multilevel framework based on INDSET and $A$-biconjugate approximate inverse preconditioner. Symbolic analysis with elimination tree is used to form the multilevel structure. ML-AINV holds tight relationship with ILU preconditioners and contain inherent parallelism by SpMV-based preconditioning operations. Experiments with various matrices show that it achieves comparable iteration counts with ILU-C, while the preconditioning performance is of 60 times or higher than SuperILU with the state-of-the-art CPU. The overall speedup in the time spent in preconditioning is in the range starting at 3.5 times to as high as 130 times compared with SuperILU.

# Chapter 9

# Conclusions and Outlook

## 9.1 Conclusions

This thesis mainly dealt with two problems: (1) numerical solution of Jacobian matrix-based linear systems in Time Domain Simulation (TDS) of Power Grids, and (2) the design of GPU-efficient Krylov solver and preconditioner. Mathematical model of Power Grids shows that the dynamic behavior of Power Grids can be characterized by a set of nonlinear differential algebraic equations (DAE's). The dynamic part of the DAE characterizes time differentials of dynamic components such as Power Generators, Motors, etc, while the algebraic part represents the constrains of the status in the Power Grid, usually Kirchhoff's Law.

The numerical integration of TDS results in solving a sequence of Jacobian matrix-based linear systems, which is the most important numerical task in TDS. Iterative solvers and preconditioners are applied to the solution of these system. But with a changing Jacobian matrix, the preconditioner is reconstructed, otherwise a dishonest preconditioner is used. From the analysis in Chapter 2 we have derived that the sparsity structure of Jacobian matrices has close relationship with the topology of the Power Network. In particular, we have two conclusions:

- In the Jacobian matrix, the Schur complement of the dynamic part in the algebraic part of the matrix is actually nil;
- The sparsity pattern of the algebraic part of the Jacobian matrix is fully characterized by the topology of the Power Network.

The results above enable us constructing a multilevel structure for the Jacobian matrix, by static analysis of the Power Network topology. A multilevel structure based on Independent Sets (IndSet) is introduced and constructed based on the Power Network topology. The IndSet based multilevel structure is then mapped to the structure of the Jacobian matrix to form a multilevel preconditioner. Compared with the ILU preconditioners, multilevel preconditioner based on IndSet on Power Network shows both good convergence properties and very low memory usage. By using IndSet with dense vertex sets (i.e., supernodes) instead of single vertices, multilevel preconditioner achieves more effective system size reduction in the multilevel framework, and with lower memory usage.

TDS involves solving a sequence of Jacobian matrices based linear systems. With the Jacobian changing at each step, potentially much computation is wasted due to preconditioner reconstructions and restarted Krylov iterations. Through the analysis of the

Jacobian matrices in Chapter 5, we reveal important properties of the Jacobian matrix in relation to the admittance matrix. The linear system based on Jacobian matrix can be transformed into an equivalent linear system with the linear operator as $Y + \Delta_Y$, where $Y$ is the (transformed) admittance matrix and $\Delta_Y$ is a rank-deficient sparse matrix with only non-zero elements at certain diagonals. $\Delta_Y$ changes with each Newton step and time step, while $Y$ remains unchanged across the simulation. This formulation enables the design of multi-step techniques for TDS linear systems: (1) preconditioner reuse and updates, (2) spectra deflation with GCRO-DR.

Since dishonest preconditioners may lose its quality during the simulation process, in Chapter 5 we explored the effectiveness of preconditioner reuse and updates. An ILU preconditioner is constructed based on $Y$, and further applied to actual Jacobian matrices of form $Y + \Delta_Y$. One benefit is that the preconditioner can be retrieved in an off-line manner. The sparsity pattern of $\Delta_Y$ matrix enables more precise preconditioner updates with a full decomposition of the augmented upper triangular part. In Chapter 6 we discussed the use of matrix spectra deflation of Jacobian matrices with GCRO-DR. The formulation of $Y + \Delta_Y$ enables computationally feasible deflation with the easy update of $U$ and $C$ matrices in GCRO-DR. During simulation, large eigenvalues appear for the spectra of the Jacobian matrices. To accommodate this, a heuristics based on the norm of the eigenvalues/Rits values is developed to dynamically choose the eigenvalue for deflation. We show that combining Deflation and Preconditioner Updates achieves 50% to 70% reduction in iteration count compared with standard GMRES method.

GPU-based acceleration is an on-going trend in high performance computing. This brings new frontiers to traditional numerical computing and linear algebra. Due to the lack of full cache support and throughput-oriented optimization, they pose specific problem for traditional numerical computation kernels. In this thesis we focus on the solution of linear systems using GPU. Linear system solution and accompanying preconditioning techniques plays a central role in the solution of Jacobian-based TDS and many other scientific applications. In Chapter 7 and Chapter 8 we apply GPU based accelerated computing to the solution of linear systems by Krylov subspace solvers and preconditioners. Computationally, there are 3 major parts in Krylov subspace solvers: (1) generation of Krylov subspace bases, (2) orthogonalization for long-recurrence based algorithms, and (3) preconditioning. Chapter 7 focuses on the first part: optimization of the basic operation used in Krylov subspace solvers – generation of Krylov subspace bases by Sparse Matrix-Vector multiplication (SpMV). We propose the optimization based on matrix profile reduction to enable 2 optimizations: (1) enhancement over the locality in accessing the dense vector $x$, and (2) column index compression. On NVIDIA GT-200 GPU, The combined speedup by these optimizations is 16% and 13% for Single-Precision and Double-Precision, respectively. On new GPU architecture (GF-100) with better cache support, we use inline PTX assembly to differentiate the access to matrix data and vector data, to exploit the architecture potential and achieve better cache utilization. The speedup due to better cache utilization for GTX-480 GPU is 19% and 15% for single-precision and double-precision, respectively.

Long-recurrence Krylov subspace solvers such as GMRES and GCR depend on orthogonalization of the newly generated vector against existing orthogonal basis of the Krylov subspace. Modified Gram-Schmidt is used for the implementation on GPU. Due to the memory bandwidth-bound nature of this algorithm, compared with state-of-the-art CPU, GPU shows $5 \sim 7$ times speedup in terms of orthogonalization performance for large sized matrices.

Preconditioner is of crucial importance for the convergence of Krylov solvers. General-purpose preconditioners such as IC and ILU face practical limitation on GPU due to that IC and ILU have limited parallelism, while GPU relies on massive, fine-level parallelism for high performance. We design preconditioner with good memory efficiency and inverse-based preconditioning operations which suits the GPU platform. In particular, in Chapter 8 we propose Multi-Level Approximate Inverse preconditioner (ML-AINV), by combining multilevel framework with Independent Sets (INDSET) and $A$-biconjugate approximate inverse preconditioner (AINV). ML-AINV relies on symbolic analysis with Elimination Tree for the choice of INDSET's and the last-level system. In order to ensure effectiveness of symbolic analysis, Nested Dissection (e.g., using METIS) is used in ML-AINV for a reduced height of the elimination tree. Algebraically, ML-AINV is closely related to IC or ILU in that its construction is based on the elimination process. Multilevel structure ensures good memory efficiency of ML-AINV compared with AINV. The preconditioning with ML-AINV involves a series of SpMV operations. Since SpMV operations contain inherent parallelism, the preconditioning with ML-AINV can be efficiently carried out on GPU platform. Experiments with various matrices show that the preconditioning performance of ML-AINV ranges from 1 to 10 GFLOPs with NVIDIA GT-200 GPU, and achieving 60 to 100 times speedup as compared with SuperILU on state-of-the-art CPUs. The overall speedup in the preconditioning operations ranges from 3.5 times to 130 times compared with SuperILU. In terms of convergence properties, because of the close relationship of ML-AINV with ILU, the convergence property of ML-AINV is also comparable to ILU-C and ML-AINV preconditioners.

Through Chapter 7 and Chapter 8 we show that iterative solvers and preconditioners can fully exploit the massive parallelism on the GPU platforms. Overall speaking, the computation-data ratio of iterative solvers and preconditioners is low. The bandwidth bound nature of these operations imply a great potential with GPUs. However the effective preconditioning on GPU requires careful design, in order to achieve: (1) good memory efficiency with matrix inverse forms, and (2) massive parallelism of preconditioning operations. ML-AINV bridges these two contradicting design goals and hit a balance between memory usage and high performance. With ML-AINV, Krylov subspace solvers can fully utilize the high performance on GPU platforms.

## 9.2 Outlook for Future Research

The solution of Jacobian-based linear systems is the most important numerical task in Time Domain Simulation of Power Grids. But online analysis of running Power Grids faces several challenging problems.

- Online analysis of Power Grids requires fast, sometimes real-time simulation speed, which is needed for practical analysis and precaution operations. This poses a tight upper-bound for the simulation speed due to the real-time requirement. Fully accelerated simulation with GPUs serves as a potential solution to this problem.

- Usually the online analysis involves several geographically distributed sites, such as administrative agencies spread across the country. These sites are responsible for the collection of Power Grid specific information. The simulation could be aided with these information for better accuracy and carried out concurrently through several sites. This poses additional communication overhead for the simulation.

Communication needs to be hidden in computation to alleviate the overhead of multi-site simulation.

Multi-case simulation of Power Grids yields special interest. This arises from the practical need for the analysis of a Power Grid. Usually independent simulation cases (to characterize various situations the Power Grid may undergo) is simulated concurrently. The naive way for this is to simulate each case independently. But with the analysis of the inter-relationship among these cases would enable faster simulation or save overall computation amount. The reuse and updates of preconditioner can be applied to multi-case simulation, to avoid unnecessary construction of preconditioners. Also the Jacobian matrices of various cases may bear similarities in spectra structure, hence the application of spectrum deflation is a potential research direction.

Sparse matrix technology was firstly introduced in the Power Network analysis. Through the years of study it is now widely used in various other scientific applications. Note that in Chapter 8 we show that compared with more enhanced preprocessing and pivoting enabled preconditioner such as SuperILU, the proposed ML-AINV preconditioner sometimes suffer from worse convergence properties. Preprocessing to improve diagonal dominance can be utilized for better robustness and convergence properties, like those in [62].

In practice, the iterative solvers and preconditioners are widely used at the scale from single machine to multiple machine based clusters with thousands of processor cores. GPU-accelerated clusters usually involve several, to many GPU-accelerated computing nodes. The design of preconditioner on a multi-machine, multi-GPU platform can be achieved by a 2-level structure through domain decomposition. The first level is dedicated to inter-domain regions (or edge splitters of the graph), while sub-domains resides on the second level. A global Schur complement $S$ for the first level is formed and a preconditioner is applied to it. Each sub-domain should be non-trivial and ML-AINV preconditioner is applied. Two-level framework based on domain decomposition and GPU-enabled local preconditioner is potentially the fundamental framework for the future research on iterative solver and preconditioning on GPU-accelerated computer clusters.

# Bibliography

[1] BILUM: Multi-Level Block ILU Preconditioning Techniques for Solving General Sparse Linear Systems. http://www.cs.uky.edu/ jzhang/bilum.html.

[2] CBC News Indepth: Power outage. http://www.cbc.ca/news/background/poweroutage/.

[3] CUDA Zone. http://www.nvidia.com/cuda.

[4] Electrical Bus. http://en.wikipedia.org/wiki/Electrical_bus.

[5] GPGPU.org. http://www.gpgpu.org.

[6] Magma. http://icl.cs.utk.edu/magma/.

[7] METIS. http://glaros.dtc.umn.edu/gkhome/metis/metis/overview.

[8] Moore's Law. http://en.wikipedia.org/wiki/Moore's_law.

[9] OpenCL. http://www.khronos.org/opencl.

[10] Shading Language. http://en.wikipedia.org/wiki/Shading_language.

[11] SPARSKIT. http://www-users.cs.umn.edu/ saad/software/SPARSKIT/sparskit.html.

[12] SuperLU. http://crd.lbl.gov/ xiaoye/SuperLU/.

[13] The Green 500 List - November 2010. http://www.green500.org/lists/2010/11/top/list.php.

[14] The Green 500 List :: Environmentally Responsible Supercomputing. http://www.green500.org.

[15] The Little Green Machine. http://www.littlegreenmachine.org.

[16] Tianhe-I. http://en.wikipedia.org/wiki/Tianhe-I.

[17] TOP500 Supercomputing Sites. http://www.top500.org/.

[18] TOP500 Supercomputing Sites — November 2010. http://www.top500.org/lists/2010/11.

[19] Transcript - Obama's Speech on the Economy. http://www.nytimes.com/2009/01/08/us/politics/08text-obama.html?_r=1&pagewanted=4.

[20] Matrix ORSIRR 1. http://math.nist.gov/MatrixMarket/data/Harwell-Boeing/oilgen/orsirr_1.html, 2004.

[21] Statistics Brief in China, 2009. National Bureau of Statistics, 2010.

[22] Matrix market. http://http://math.nist.gov/MatrixMarket/, retrieved on 15-Jun-2010.

[23] Fernando L. Alvarado and Robert Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM Journal on Scientific Computing*, 14(2):446–460, 1993.

[24] A.B. Alves, E.N. Asada, and A. Monticelli. Critical Evaluation of Direct and Iterative Methods for Solving Ax = b Systems in Power Flow Calculations and Contigency Analysis. *IEEE Trans. on Power Systems*, pages 702–708, May 1999.

[25] M. Ament, G. Knittel, D. Weiskopf, and W. Strasser. A Parallel Preconditioned Conjugate Gradient Solver for the Poisson Problem on a Multi-GPU Platform. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 583 –592, 2010.

[26] Diogo V. Andrade, Mauricio G. C. Resende, and Renato F. Werneck. Fast local search for the maximum independent set problem. In *Proceedings of the 7th international conference on Experimental algorithms*, WEA'08, pages 220–234, Berlin, Heidelberg, 2008. Springer-Verlag.

[27] O. Axelsson and S. Margenov. On Multilevel Preconditioners which are Optimal with respect to Both Problem and Discretization Parameters. *Computational Methods in Applied Mathematics*, 3(1):6–22, 2003.

[28] J. Baglama, D. Calvetti, G. H. Golub, and L. Reichel. Adaptively Preconditioned GMRES Algorithms. *SIAM Journal on Scientific Computing*, 20(1):243–269, 1998.

[29] A. H. Baker, E. R. Jessup, and T. Manteuffel. A Technique for Accelerating the Convergence of Restarted GMRES. *SIAM J. Matrix Anal. Appl.*, 26:962–984, April 2005.

[30] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* SIAM, Philadelphia, PA, 1994.

[31] R. Battiti and M. Protasi. Reactive local search for the maximum clique problem. *Algorithmica*, 29(4):610–637, 2001.

[32] Mehmet Belgin, Godmar Back, and Calvin Ribbens. A Library for Pattern-based Sparse Matrix Vector Multiply. *International Journal of Parallel Programming*, pages 1–26, 2010. 10.1007/s10766-010-0145-2.

[33] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[34] Michele Benzi. Preconditioning Techniques for Large Linear Systems: A Survey. *Journal of Computational Physics*, (182):418–477, 2002.

[35] Michele Benzi and Miroslav Tuma. A sparse approximate inverse preconditioner for nonsymmetric linear systems. *SIAM J. Sci. Comput.*, 19:968–994, May 1998.

[36] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[37] Luc Buatois, Guillaume Caumon, and Bruno Levy. Concurrent number cruncher: a gpu implementation of a general sparse linear solver. *Int. J. Parallel Emerg. Distrib. Syst.*, 24:205–223, June 2009.

[38] Kevin Burrage and Jocelyne Erhel. On the performance of various adaptive preconditioned GMRES strategies. *Numerical Linear Algebra with Applications*, 5(2):101–121, 1998.

[39] Zhichao Cao, Shiming Xu, Wei Xue, and Wenguang Chen. Improving Dense Linear Equation Solver on Hybrid CPU-GPU System. In *Pervasive Systems, Algorithms, and Networks (ISPAN), 2009, 10th International Symposium on*, pages 556–562. IEEE Computer Society 2009, 2009.

[40] K.W. Chan. Parallel algorithms for direct solution of large sparse power system matrix equations. *Generation, Transmission and Distribution, IEE Proceedings-*, 148(6):615 –622, November 2001.

[41] D. Chaniotis and M.A. Pai. Iterative solver techniques in the dynamic simulation of power systems. In *Power Engineering Society Summer Meeting, 2000. IEEE*, volume 1, pages 609 –613 vol. 1, 2000.

[42] Andrew Chapman and Yousef Saad. Deflated and Augmented Krylov Subspace Techniques. *Numerical Linear Algebra with Applications*, 4(1):43–66, 1997.

[43] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. *SIGPLAN Not.*, 45(5):115–126, 2010.

[44] H. Dag and A. Semlyen. A New Preconditioned Conjugate Gradient Power Flow. *IEEE Trans. on Power Systems*, 18:1248–1255, Nov. 2003.

[45] Timonthy Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transaction on Mathematical Softwware*, 2010. to appear.

[46] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2).* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2006.

[47] Stanley C. Eisenstat, Howard C. Elman, and Martin H. Schultz. Variational Iterative Methods for Nonsymmetric Systems of Linear Equations. *SIAM Journal on Numerical Analysis*, 20(2):345–357, 1983.

[48] A.J. Flueck and H. Chiang. Solving the Nonlinear Power Flow Equations with an Inexact Newton Method Using GMRES. *IEEE Trans. on Power Systems*, 13:267–273, May 1998.

[49] F.D. Galiana, H. Javidi, and S. McFee. On the Application of a Pre-Conditioned Conjugate Gradient Algorithm to Power Network Analysis. *Power Systems, IEEE Transactions on*, 9(2):629 –636, May 1994.

[50] L. Giraud, S. Gratton, and E. Martin. Incremental Spectral Preconditioners for Sequences of Linear Systems. *Applied Numerical Mathematics*, 57(11-12):1164 – 1180, 2007. Numerical Algorithms, Parallelism and Applications (2).

[51] Gene H. Golub and Charles F. van Loan. *Matrix Computations*. The Johns Hopkins University Press, 3rd edition, 1996.

[52] Anne Greenbaum. *Iterative Methods for Solving Linear Systems*, volume 17 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), 1997.

[53] Roger G. Grimes, David R. Kincaid, and David M. Young. ITPACK 2.0 Users Guide. Technical Report CNA-150. 1979.

[54] A. Grosso, M. Locatelli, and W. Pullan. Simple ingredients leading to very efficient heuristics for the maximum clique problem. *J. Heuristics*, 14:587–612, 2008.

[55] Murat Efe Guney. *High-Performance Direct Solution of Finite Element Problems on Multi-Core Processors*. PhD thesis, Georgia Institute of Technology, May 2010.

[56] Magnus R. Hestenes and Eduard Stiefel. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards*, 49(6), December 1952.

[57] Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. Optimizing Sparse Matrix-Vector Multiplication using Index and Value Compression. pages 87–96, May 2008.

[58] A.Y. Kulkarni, M.A. Pai, and P.W. Sauer. Iterative solver techniques in fast dynamic calculations of power systems. *Intl. J. of Electrical Power and Energy Systems*, 23(3):237–244, 2001.

[59] P. Kundur. *Power System Stability and Control*. McGraw Hill, New York, 1994.

[60] Na Li, Yousef Saad, and Edmond Chow. Crout Versions of ILU for General Sparse Matrices. *SIAM Journal on Scientific Computing*, 25(2):716–728, 2003.

[61] H. X. Lin. A Unifying Graphs Model for Designing Parallel Algorithms for Tridiagonal Systems. *Parallel Computing*, 27:925–939, 2001.

[62] Jan Mayer. A numerical evaluation of preprocessing and ilu-type preconditioners for the solution of unsymmetric sparse linear systems using iterative methods. *ACM Trans. Math. Softw.*, 36:1:1–1:26, March 2009.

[63] F. Milano, L. Vanfretti, and J. C. Morataya. An Open Source Power System Virtual Laboratory: The PSAT Case and Experience. *IEEE Trans. on Education*, 51(1):17–23, 2008.

[64] Ronald B. Morgan. A Restarted GMRES Method Augmented with Eigenvectors. *SIAM Journal on Matrix Analysis and Applications*, 16(4):1154–1171, 1995.

[65] Ronald B. Morgan. GMRES with Deflated Restarting. *SIAM Journal on Scientific Computing*, 24(1):20–37, 2002.

[66] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krger, Aaron Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[67] M. A. Pai, P. W. Sauer, and A. Y. Kulkarni. A Preconditioned Iterative Solver for Dynamic Simulation of Power Systems. *Proc. ISCAS'95*, 2:1279–1282, 1995.

[68] M.A. Pai and H. Dag. Iterative Solver Techniques in Large Scale Power System Computation. *Proc. of IEEE Conf. on Decision and Control*, 4:3861–3866, 1997.

[69] Michael L. Parks, Eric de Sturler, Greg Mackey, Duane D. Johnson, and Spandan Maiti. Recycling Krylov Subspaces for Sequences of Linear Systems. *SIAM Journal on Scientific Computing*, 28(5):1651–1674, 2006.

[70] S.V. Partner. The Use of Linear Graphs in Gaussian Elimination. *SIAM Review*, 3:119–130, 1961.

[71] Y. Saad. ILUM: A Multi-Elimination ILU Preconditioner For General Sparse Matrices. *SIAM Journal of Scientific Computing*, 17:830–847, 1996.

[72] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* SIAM, 2nd edition, 2003.

[73] Yousef Saad and Martin H. Schultz. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3):856–869, 1986.

[74] M. La Scala, G. Sblendorio, and R. Sbrizzai. Parallel-in-time implementation of transient stability simulations on a transputer network. *Power Systems, IEEE Transactions on*, 9(2):1117 –1125, May 1994.

[75] A. Semlyen. Fundamental Concepts of a Krylov Subspace Powerflow Methdology. *IEEE Trans. on Power Systems*, 11:1528–1537, Aug. 1996.

[76] J.W. Shu, W. Xue, and W.M. Zheng. A Parallel Transient Stability Simulation for Power Systems. *IEEE Trans. on Power Systems*, 20:1709–1717, 2005.

[77] Jurjen Duintjer Tebbens and Miroslav Tuma. Efficient Preconditioning of Sequences of Nonsymmetric Linear Systems. *SIAM Journal on Scientific Computing*, 29(5):1918–1941, 2007.

[78] Vasily Volkov and James W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[79] Richard Wilson Vuduc. *Automatic Performance Tuning of Sparse Matrix Kernels.* PhD thesis, University of California, Berkeley, 2002.

[80] Ke Wang, Wei Xue, HaiXiang Lin, ShiMing Xu, and WeiMin Zheng. Updating preconditioner for iterative method in time domain simulation of power systems. *SCIENCE CHINA Technological Sciences*, 54(4):1024–1034, 2011.

[81] Mingliang Wang, Hector Klie, Manish Parashar, and Hari Sudan. Solving Sparse Linear Systems on NVIDIA Tesla GPUs. *In proc. ICCS'09*, pages 864–873, 2009.

[82] Jeremiah Willcock and Andrew Lumsdaine. Accelerating sparse matrix computations via data compression. In *Proceedings of the 20th annual international conference on Supercomputing*, ICS '06, pages 307–316, New York, NY, USA, 2006. ACM.

[83] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 38:1–38:12, New York, NY, USA, 2007. ACM.

[84] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying gpu microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235 –246, 2010.

[85] Shiming Xu, Hai Xiang Lin, Ke Wang, and Wei Xue. Fill-in Guided Multilevel Preconditioner for Time-Domain Simulation of Large-Scale Power Systems. *Proc. of Intl. Symp. on Appl. Computing and Computational Sciences (ACCS'08)*, pages 114–122, Aug 2008.

[86] Shiming Xu, Hai Xiang Lin, Wei Xue, and Ke Wang. Utilizing CUDA for Preconditioned GMRES Solvers. *In proc. DCABES'09*, 2009.

[87] Shiming Xu, Wei Xue, and Hai Xiang Lin. Fast Power Grid Simulation with Jacobian Matrix based Spectra Deflation. *submitted to IEEE Trans. on Power Systems.*

[88] Shiming Xu, Wei Xue, and Hai Xiang Lin. Sparse Matrix-Vector Multiplication Optimizations based on Matrix Bandwidth Reduction using NVIDIA CUDA. In *Distributed Computing and Applications to Business, Engineering and Science (DCABES), 2010 9th International Symposium on*, pages 609–614, 2010.

[89] Shiming Xu, Wei Xue, and Hai Xiang Lin. Performance modeling and optimization of sparse matrix-vector multiplication on NVIDIA CUDA platform. *The Journal of Supercomputing*, pages 1–12, 2011. 10.1007/s11227-011-0626-0.

[90] Shiming Xu, Wei Xue, Ke Wang, and Hai Xiang Lin. Fast Time Domain Simulation of Power Systems using Multilevel Preconditioners with Adaptive Reconstruction Strategies. *accepted by Simulation Modelling Practice and Theory (SIMPRA).*

[91] Shiming Xu, Wei Xue, Ke Wang, and Hai Xiang Lin. Generating Approximate Inverse Preconditioners for Sparse Matrices using CUDA and GPGPU. *Journal of Algorithms and Computational Technology*, 5(3):475–501, 2011.

[92] Boming Zhang, Gang Wang, and Hongbin Sun. Decomposition and Coordination Modes for Transient Stability Simulation. *Automation of Electric Power Systems*, (20):24–28, 2005.

[93] Yi-Shan Zhang and Hsiao-Dong Chiang. Fast Newton-FGMRES Solver for Large-Scale Power Flow Study. *Power Systems, IEEE Transactions on*, 25(2):769 –776, May 2010.

# Acknowledgements

This thesis would not have been possible without the supports and contributions from many people, both in Holland and in China. Here I would like to express my gratitude to all these people who helped to make this work a reality.

First of all, I would like to thank my supervisor, Hai Xiang Lin, for offering me the opportunity to carry out Ph.D. study in TU Delft, for his guidance in academic research, for the freedom and openness he provided me for choosing research topics, and his patience in allowing me to wandering around and taking detours academically in the long Ph.D. years of mine. I would also like to express my gratitude to my promotor, professor Arnold Heemink for his kindness, careful reading of the manuscript and support during the long years of my Ph.D. study.

I would also like to thank my advisors and colleagues at Tsinghua University, whom I have been working with since 2003. I would like to thank professor Weimin Zheng for his long-term support and guidance in my academic career. I would like to thank professor Wenguang Chen for his guidance in my master's years and for setting me on the track of Ph.D. study at TU Delft in the first place. Many thanks go to professor Wei Xue for his help from all angles, including but not limited to: guidance in power grid modeling, providing me with support, both academically and mentally. To Dr. Ke Wang for sharing the rainy days in Delft with me and lots of discussion about mathematics. To Ruini Xue, for his unique taste in Chinese poems and discussion about life. I would also like to express my gratitude towards many friends since the early days I was in Tsinghua, including Eddie Cao, Jidong Zhai, Yongjun Xu, Duo Li, Jianian Yan, Hongshan Jiang, Xi Deng, and many many others, for their support and those unforgettable years we spent together.

I've met many great people in Delft, without whom the life in Delft would not be complete. I would like to thank my officemates in HB.05.270, Sajad and Morteza, for constructing and sharing such a harmonious working place with me. I would also like to express my gratitude to all the colleagues in the DIAM who have offered help and invaluable discussions with me. A special word of thanks goes to Rohit Gupta for sharing the discussion on GPU and CUDA, which is quite rare and invaluable for me. To Evelyn Sharabi and Dorothee Engering, many thanks for their administrative assistance. Special thanks go to those many Pakistani friends in Delft: Chinese and Pakistani have always been friends, yet we've achieved even better friendship in Delft. Last but not the least, I would like to thank Theda Olsder, for her support and professionalism in arranging all the tedious yet important administrative issues.

I have made many great Chinese friends at Delft. Some have left Delft while a few remains. Many thanks to Yizhi Zhao, Huiling Yang, Zuopeng Qu for the unforgettable moments and meals of my first year in Delft and the ongoing support since then. To Xiaoyu, thank you very much for all the mind-soothing hours with Chopin and traditional Chinese piano songs. Many thanks go to the great Chinese guys I've met during my stay

in Delft, especially Jia Wei, Xin Liu, Hao Liu, Xiaohui Cheng and Yufei Dong. For friends with whom I shared so many wonderful trips and gatherings together, especially Xin Ge, Zheng Li, Xiaogang Yang and Dan Yang, I thank you and wish you all the best.

Lastly, I would like to thank my family. I'm deeply grateful to my parents, for their endless love, unwavering support, and encouragements in my life. Special thanks go to my wife Adele, for her kindness, patience, understanding, and sharing the life with me.

# Curriculum Vitae

Shiming Xu was born on December 14, 1981 in Xinxiang of Henan Province, China. He received his secondary education between 1993 and 1999 at No. 10 Middle School and No. 1 Middle School of Xinxiang. From 1999 to 2003 he studied at Department of Electronic Engineering, Tsinghua University in Beijing, China. He received Bachelor's Degree in 2003, and continued to study at Department of Computers Science and Technology. He received his Master's Degree in Computer Science from Tsinghua University in January, 2006.

In February, 2006, he joined Delft University of Technology to pursuit a Ph.D. degree, working in the group of Mathematical Physics, Delft Institute of Applied Mathematics, Faculty of Electrical Engineering, Mathematics and Computer Science. His research work also includes close collaboration with High Performance Computing Institute of Tsinghua University, where he spent one and a half years in total during his Ph.D. study. From February, 2011 he has been working at Center of Earth System Science, Tsinghua University.