GPU-BASED ACCELERATED ALGORITHMS FOR THE POWER FLOW CALCULATION

by

LEI ZENG

A dissertation submitted in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY IN ELECTRICAL AND COMPUTER ENGINEERING

2023

Oakland University
Rochester, Michigan

Thesis Advisory Committee:

Shadi G. Alawneh, Ph.D., Chair
Aycil Cesmelioglu, Ph.D.
LianXiang Yang, Ph.D.
Ping He, Ph.D.
Seyed Ali Arefifar, Ph.D.

ACKNOWLEDGMENTS

The research presented in this dissertation is the product of a wide-ranging collaboration. I am grateful for my committees; not only for their contributions to scientific knowledge, but also for my personal expertise skills learning from them.

I would like to share my gratefulness with those who helped me to achieve my academic and research goals. Foremost, I would like to express my deepest appreciation to my advisor, Shadi G. Alawneh, who introduced me to the field of GPU accelerated computing. Dr. Alawneh instructed me how to conduct academic research, how to effectively utilize GPUs to improve performance of computing and how to present my work to others in papers. I acquired a lot of experience, skills, and techniques from his high standard of academic requirement. The journey of study was so joyful with his endless help during the past four years.

I am deeply grateful to Dr. Arefifar. I benefited from his guidance on the paper review and writing. I appreciated his precious advice on the overall research. I would like to acknowledge my thesis committee members, Dr. Yang, Dr. Cesmelioglu and Dr. He for their advice during my Ph.D. study and research. Their encouragements motivated me when I was frustrated by the difficulties in the research and language barriers.

Finally, I am grateful to my family. They always encourage me to pursue my dream. Without their support and encouragement, I might not have been able to pursue and complete my Ph. D. degree.

Lei Zeng

ABSTRACT

GPU-BASED ACCELERATED ALGORITHMS FOR THE POWER FLOW CALCULATION

by

LEI ZENG

Adviser: Shadi G. Alawneh, Ph.D.

Power flow (PF) calculation is critical for power systems, as the development of multiple energy supplies. Power system modeling and analysis have been challenging on power engineers and leading to great pressure for the PF calculation. For the safety, stability and real-time response in grid operation, grid planning and analysis of the power system, it is urged to require designing high-performance computing methods, accelerating PF calculation, obtaining the voltage magnitude and phase angle of buses inside the power system, and coping with the increasingly complex large-scale power system. The PF algorithm is, generally, classified into the iterative and direct methods in the perspective of numerical methods. As for iterative method, a pre-conditioner is required to be designed to reduce the condition number of sparse Jacobian matrix to improve convergency of the power system. Although the iterative method can save much memory and solve some large-scale sparse linear equations, the PF solver severely depends on the complicated pre-conditioner of the Jacobian matrix. Usually, the PF calculation cannot get a convergent solution without validating the pre-conditioner, repeatedly. For direct method, the traditional sequential, the Newton-Raphson (NR)

algorithm will consume much of the computing resource and take a long time to converge on solving large-scale sparse linear equations of the power system.

To address these issues, the GPU-based parallel computing architecture, single-GPU, and multi-GPUs, was proposed to take advantage of multi-thread, task parallelism and data parallelism, accelerating the PF calculation. Also, the utilization of GPUDirect technology enhances communication efficiency and significantly reduces data transmission overhead, leading to superior performance improvements compared to the traditional sequential methods.

TABLE OF CONTENTS

TABLE OF CONTENTS—Continued

TABLE OF CONTENTS—Continued

TABLE OF CONTENTS—Continued

LIST OF TABLES

LIST OF FIGURES

LIST OF FIGURES – Continued

LIST OF ABBREVIATIONS

PF	Power flow
NR	Newton-Raphson
FD	Fast decoupled method
FDPF	Fast decoupled power flow
GPU	Graphic processing unit
CUDA	Computer unified device architecture
HPC	High-performance computing
ILU	Incomplete LU factorization
CRS	Compressed row storage
CSC	Compressed column Storage
COO	Coordinate format
SM	Streaming multiprocessors
IC	Incomplete Cholesky factorization
CG	Conjugate gradient
P2P	Peer to peer
SPD	Symmetric positive definite

CHAPTER ONE

INTRODUCTION

1.1 PF Calculation Concerns in the Power System

PF calculation is widely utilized in the analysis of the electrical power systems, aiming to estimate the voltage magnitude and phase angle of buses within the power system network, and the PF calculation algorithm involves in repeatedly applying various numerical methods to solve nonlinear equations of such electric power systems. PF calculation plays a crucial role in power system analysis, planning and operation [1], as the development of multiple renewable energy supplies. The power system analysis and modeling have been challenging on power engineers, due to increasing scale and heavier loading of power system, which not only brings great pressure for power flow calculation [2], [3] but motivates engineers to design high-performance methods to fix the issues.

Typically, PF calculation is repeatedly to utilize several numerical methods linearizing non-linear equations, solving these linearized equations of such electrical power systems; The execution time of the PF calculation is usually substantial, due to updating the voltage magnitude and angle at each iterative process [4]. In practice, the classic algorithm usually employs the Newton-Raphson (NR) method in the process of the PF calculation, but the acceleration efficacy is constrained by the sequential execution nature of the program. Furthermore, the classic sequential NR method tends to be time-consuming and demands substantial computational resources to solve the PF equations. This occasionally leads to program crashes and convergence failures, particularly in the presence of the ill-conditioned sparse large-scale power systems.

1.2 Parallel Methods on the PF Calculation

The development of a parallel computation hardware platform has brought about revolutionary changes. Recently, the graphic processing unit (GPU) has been enabled to carry general purpose computations, although it was originally designed for a graphic processing purpose. The GPU has massive parallel computational units on board, and it is now being used as a co-processor, accelerating specific computation types. As a peripheral, the GPU communicates with the CPU through PCI-Express bus. However, the efficacy of acceleration is severely limited by bandwidth of PCI-Express when numerous data transmissions frequently occur between the CPU and the GPU. Sometimes, the transmission overhead becomes the bottleneck in classic PF calculation, necessitating the exploration of alternative algorithms with varying levels of parallelism granularity. Consequently, achieving efficient performance in GPU parallel algorithms specific considerations.

With the evolution of GPU software support, NVIDIA introduces Computer Unified Device Architecture (CUDA) as a versatile platform, and this platform accommodates various programming languages including C++, Python, Fortran, OpenCL, Open Multi-Processing (OpenMP), and more [5]. This significantly mitigates the challenges associated with developing parallel paradigm on the heterogeneous computing architecture, providing a cost-effective solution without relearning new programming language and achieving dramatic speedups in the process of computation. Furthermore, NVIDIA CUDA provides us an opportunity to address the PF problem through vectorization, due to its compatibility with numerous high-level math libraries. NVIDIA GPUs have successfully accelerated example cases in cutting-edge fields such

as artificial intelligence [6], scientific simulations [7-10], cryptography [11], [12], integrated circuit analysis [13-15], medical imaging [16] and optimal PF [17].

Researchers have proposed various strategies to enhance parallelization of the large-scale PF calculation, adapting to various hardware platforms and computing architectures. In particular, the extensive exploration of high-performance computing (HPC) method has been a subject of active studies over many years, aiming at reducing the execution time in PF calculation [18-22]. From the perspective of numerical computing methods, these approaches can be classified into two categories of methods, i.e., iteration methods [23-26] and direct methods [27-37]. In the case of iterative solvers, it has no need to permutate sparse Jacobian matrix in contrast to the direct solver. However, a preconditioner is typically designed to improve the convergence rate during the PF calculation. Essentially, the preconditioner is a matrix that transforms the original linear system into another one with clustered eigenvalue spectrum, or smaller condition number-—a process referred to as preconditioning. Specifically, several examples of preconditioners include the approximated inverse preconditioner [38], incomplete LU (ILU) factorization [39], and polynomial-based preconditioner such as the Chebyshev preconditioner [40]. In the process of solving linear equations, all iteration methods have a similar iterative formula, expressed as $Mx^{k+1} = Nx^{k} + b$, where $A = M - N$ is, technically, referred to as the matrix A splitting. The property of Matrix M has association with the convergence in the PF calculation. Generally, the spectrum radius of $M^{-1}N$ should be less than one, achieving enhanced convergence when solving linear systems. X. Li et al. [41-43] designed a preconditioner to reduce condition number of Jacobian matrix on the parallel platform, improving the stability of solving linear equations when applying the NR

method. Although these iteration methods facilitate acceleration of the computation for the large-scale power systems, saving much of memory and addressing large-scale power equations with the assistance of parallel devices, the execution time is significantly influenced by the design of the preconditioner. Typically, the PF calculation easily fails to converge, due to the sparsity and singularity inherent in power system. To address this issue, researchers need to validate the preconditioner repeatedly, until they obtain a stable and robust preconditioner for solving PF equations. Thus, the preparation process for the iteration method becomes a time-consuming and tedious task.

Typically, the direct solvers exhibit better robustness when addressing the ill-conditioned problems in the process of PF calculation [44]. Most direct approaches employ either the NR or fast decoupled PF (FDPF) method, solving the linearized PF equations on the parallel devices. These approaches operate the coefficient matrices in a vectorization parallelization manner, thereby improving the program performance. Reference [2, 27] implemented the dense NR method on the single GPU platform, indeed achieved a speedup of almost $8x$, compared to its CPU counterpart. Similarly, D.J. Sooknanan et al. [45] concentrated on enhancing the computation of the Jacobian and bus admittance matrix, demonstrating notable acceleration on the single GPU. Also, Guo, C. et al. [29] conducted a performance analysis in parallel PF solvers, proving outperformance of the NR method. Reference [28] proposed that the GPU-based parallel NR method can achieve a speedup ratio of 3.27x for a system comprising around 300 buses. These dense parallel methods leveraged the high floating-point operation capability and throughput of the GPUs to achieve a significant speedup. However, while applied to power systems with large-scale dense matrix formats, these methods may fail

to converge, leading to out-of-memory on the single GPU platform. To address the challenges, in [32, 36, 46-51], researchers adopted a sparse matrix format and exploited the inherent parallelism of the power system represented in sparse format. Although the GPU-based sparse method saved much amount of global memory and minimize the computations involved in zero entries in sparse matrices, the scalability of computing units was restricted by the single-GPU architecture, due to the strong data dependence and irregular memory access pattern in sparse matrix format. Furthermore, the GPU-based sparse linear solver has been explored in connection with linear equation solvers in PF engineering. In [35, 52-55], most of them attempted either the left-looking or the hybrid column-based right-looking algorithm to perform a parallel LU decomposition, accelerating the sparse linear solver. Besides, multi-GPU methods have also been proposed to improve the performance of sparse linear solver in [56-60], as the development of parallel devices. The communication overhead, however, has become the primary bottleneck, rather than computing capability, limiting the performance of PF calculation.

To address the mentioned challenges, research has focused on evaluating a fixed Jacobian matrix through the vectorization and parallelization methods to solve the large-scale sparse linear system, instead of performing matrix inversion on the single-GPU architecture. Furthermore, the single-GPU method adopts the CRS format to save much memory, alleviating the data transmission burden between the CPU and GPU. For the multi-GPU method, the performance and speedup are severely constrained by communication across the different GPUs, rather than computing capability. The two-hierarchy architecture, i.e., task and data parallelism, is proposed to improve the

scalability and efficiency in PF calculation. Additionally, the introduction of GPUDirect technology facilitates direct operation on the GPU memory, eliminating unnecessary memory copies, decreasing the data transmission overheads, reducing latency and the time-consuming communication overheads between host and devices.

1.3 Contributions

The main contribution of this dissertation is a set of parallel methodologies proposed at the architecture level to optimize the PF calculation on both the single-GPU and multi-GPU platform. A brief description of the contributions is listed below, as follows:

- The single GPU-based method utilizes Compressed Row Storage (CRS) format to store the nodal admittance and the Jacobian matrix elements. This significantly reduces memory-consumption by excluding storage and computation of zero entries in the process of PF calculation. As a result, the computational complexity is reduced from $O(n^3)$ to $O(n^2)$.
- A fixed Jacobian matrix is evaluated in a vectorization parallelization manner, alleviating the computational burden on the single-GPU platform, compared to updating the Jacobian matrix in each iteration. Additionally, the modified high-order Runge-Kutta method is introduced to prevent the divergence in PF calculation. Besides, the approach for updating the power mismatch involves solving large-scale sparse linear systems, avoiding directly to calculate sparse Jacobian matrix inversion, for inverting a large-scale sparse matrix not only requires allocating extra

memory to store submatrix, but also generates potential fill-ins during the inversion process.

- The single GPU-based method performs as precisely as the original CPU-based NR with MATPOWER, a Matlab-based open-source tool, resulting in almost 3x better performance for a power system over 13,000 buses.
- The multi-GPU method consists of two hierarchy architectures, i.e., task and data parallelism, rather than designing complex graph structures. At one level, each task, comprising voltage magnitude and voltage angle, is simultaneously managed by its own process unit, and the second level is processing the computationally intensive data of each task across multiple GPUs. The storage spaces on different process units are isolated, avoiding the data race, effectively, accelerating the PF calculation.
- GPUDirect technology is incorporated into the multi-GPU based method to minimize the communication overhead between the host and devices. Data migrations are almost transparent for all devices on the same process unit, eliminating the time-consuming data transmission between the host and devices. Besides, one master and multi-salve communication model of GPUs is proposed, facilitating data broadcast and collection in a coalesce manner. This model enhances the efficiency of data transmission, resulting in a speedup of 9x~33x, when compared to the naïve single GPU-based method on a large-scale power system.

1.4 Dissertation Outline

This work focuses on exploring parallel methods with GPU-based implementation to accelerate the linear computations in PF analysis and provides guidance to handle the extremely massive parallel numerical computations for the future large-scale and complex power systems.

Chapter 2 provides the background and discusses the research that is relevant to the topic in the dissertation. It comprehensively reviews the literature related to parallel computing in PF calculation, including classic parallel hardware platforms, direct solvers and iterative solvers on parallel platforms.

Chapter 3 introduces the fundamentals of power system modeling and discusses the classic linear system solver in PF calculation. Furthermore, A concise introduction to the storage of sparse matrix format follows.

Chapter 4 presents the GPU-based sparse PF method, which modifies the classic NR method and accelerates the Jacobian matrix in a vectorization manner, improving efficiency and stability of PF calculation. Subsequently, various GPU-based direct linear solvers are compared on the large-scale power system over 10, 000 buses. The achieved speedups, in comparison with the different GPU-based solvers, range from almost 3x up to 200x, respectively.

Chapter 5 proposes the multi-process and multi-GPU based FD method and it designs two-hierarchy architecture, task parallelism and data parallelism, to optimize the FD solver parallelization. Moreover, GPUDirect technology is employed to enhance data transmission efficiency and significantly reduce copy overhead, achieving a notable

speedup of 9x ~ 33x, when compared to the naïve CUDA kernel on the single-GPU with a power system around 13, 000 buses.

Chapter 6 concludes this work and provides suggestions for future research, focusing on sparse PF calculation on the multi-GPU node. The exploration will involve the utilization of a multi-node partitioned global address space programming model to mitigate the major limiting factor — the communication overhead — in the multi-GPU systems. This approach is anticipated to the outperformance in PF calculation for the future power systems.

CHAPTER TWO

LITERTURE REVIEW

2.1 PF Calculation in Power System

PF calculation is critical for power systems, as the development of multiple energy supplies. To ensure the safety, stability, and real-time response of grid operation, planning and analysis in power system, power engineers are required to design high-performance computing methods, aiming to accelerate PF calculation, obtaining the voltage magnitude and phase angle of buses within the power system, and addressing the increasingly complex large-scale power networks. Simultaneously, obtaining more information about real-time system states and predicting the power system activities are critical to the power industry. Consequently, the pursuit of enhancing computing efficiency will always remain a prominent area of research in power system engineering.

2.1.1 PF Survey

PF calculation is the cornerstone in power system application, driving extensive research to optimize its efficiency with large datasets through parallel computing. J. Singh et al. [28] leveraged the GPU-based parallel NR method, achieving an impressive 3.27x on a 300-bus power system. Amano et al. [61] introduced an epsilon decomposition method, uncoupling elements of the coefficient matrix and employing block-parallel Newton method for the concurrent PF equation solutions. The convergence rate, however, tended to be occasionally slower due to a fixed Jacobian matrix in the method. S.-D. Chen et al [62] proposed an innovative factorization tree, optimizing parallelism on multi-core devices by calculating maximum fill-ins and node degrees. Wang et al. [63]

devised a novel partitioning scheme for the nonsymmetric Jacobian matrices in the NR method, leading to the efficient parallelization of LU factorization and the subsequent PF equation solutions. Their experiments achieved an average 7x speedup compared to a single processor over 7000 power buses. D. Koester et al. [64] applied a reordering scheme to generate block diagonal-bordered form matrices that reduce the fill-ins during the factorization process in PF calculation. This decoupling scheme allowed for the simultaneous factorization of block-diagonal sub-matrices. The work of C. Guo et al. [29] showcased the remarkable speedups through the utilization of GPU-based methods such as Gauss-Seidel (GS), NR and FD methods, achieving speedups of 0.05x, 1.74x and 2.1x, respectively, when compared to CPU counterparts. Introducing a GPU-based parallel Forward-Backward method for power flow calculations, D. Ablakovic et al [65] achieved a speedup ratio, approximately, 1.58x faster than the CPU execution. S. Huang et al. [33] implemented the sparse FD method with CUSPARSE, resulting in almost 4.16x speedup compared to the Matlab counterpart. Similarly, X. Su et al. [32] proposed a sophisticated GPU-based approach, seamlessly integrating novel vectorization parallelization and sparse techniques into ArrayFire libraries, improving performance of PF calculations.

The iterative method provides an alternative view on the linear system solvers, where the Conjugate Gradient (CG) method can be utilized as an optimal approach, designed for finding the numerical solutions in the symmetric positive definite linear system. Y. Wallac [66] redefined the load flow problem as an optimization problem and applied the steepest-descent and conjugate gradient methods to address it, eliminating the need for matrix inversion and conserving storage space. Galiana et al. [67] combined a conjugate gradient solver with incomplete Cholesky preconditioner to tackle power load

flow. The method had been tested on power system with up to 5000 buses, but it primarily addressed the symmetric positive systems and probably failed to converge when the Jacobian matrix of the power system has a large numerical condition number. To address asymmetric power system such as AC PF or power system dynamics, researchers proposed modifications of CG variants, such as bi-conjugate (BiCG), conjugate gradient square (CGS), general minimized residual (GMRES) and quasi-minimal residual (QMR), to enhance the scalability and portability of the CG method in PF calculation. Additionally, N. Garcia. [68] implemented a preconditioned BiCG linear system solver on GPU to accelerate PF calculation, while this method involved updating the Jacobian matrix on the CPU side. Thus, the communication overhead between CPU and GPU limited the acceleration efficacy, achieving 2.47x on the small-scale IEEE-118 buses. Z. Li et al. [69] applied a GPU-based CG normal residual (CGNR) with the Jacobi preconditioner for power system state estimation and PF calculation, resulting in a remarkable speedup of 49x on the synthetic large-scale power system examples.

While iterative methods can enhance performance, their convergence rate heavily depends on an effective pre-conditioner to reduce the condition number of large-scale sparse matrix. Li. X. et al [41-43] proposed a GPU-based two-step preconditioning method for the GPU-based iterative PF calculation. The authors conducted a thorough analysis of the Jacobian and Chebyshev preconditioners, merging advantages of both into a two-step preconditioner to improve the convergence of PF calculation on the GPU platform. While it indeed proved effective for the large-scale sparse matrix with a large condition number, it also introduced challenges related to portability and compatibility for the future complex power system. Additionally, the preparation of the preconditioner

will take a considerable amount of time for thorough validation to ensure its stability and robustness.

2.2 PF Calculation on Different Parallel Devices

This section reviews several classic parallel architectures, including multi-core CPU and FPGA, with a particular emphasis on the GPU computing architecture. Then the discussion about the review of GPU-based linear solver is presented, according to the iteration and direct method, respectively.

2.2.1 The GPU Architecture

GPU parallel computing architecture is a heterogeneous architecture composed of a host and a device, respectively. GPU devices are, typically, not a standalone platform but a co-processor for the host (CPU), operating in collaboration with the host through PCI-Express buses [70], as depicted in Figure 1.

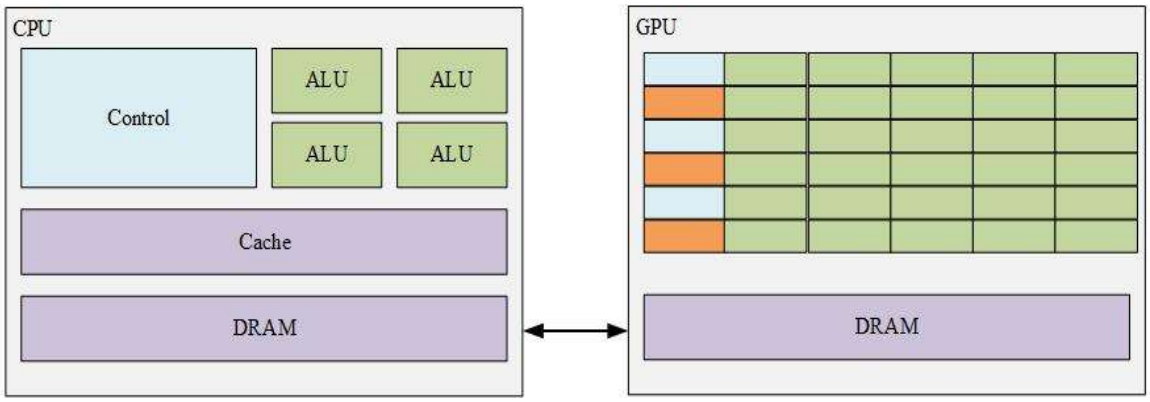

Figure 1. Heterogeneous architecture.

From the hardware perspective on the device side, streaming multiprocessors (SM) establish the foundation for parallelism and scheduled threads on the GPUs. Also, L1 cache and shared memory contribute to minimizing the communication overhead within the same thread blocks.

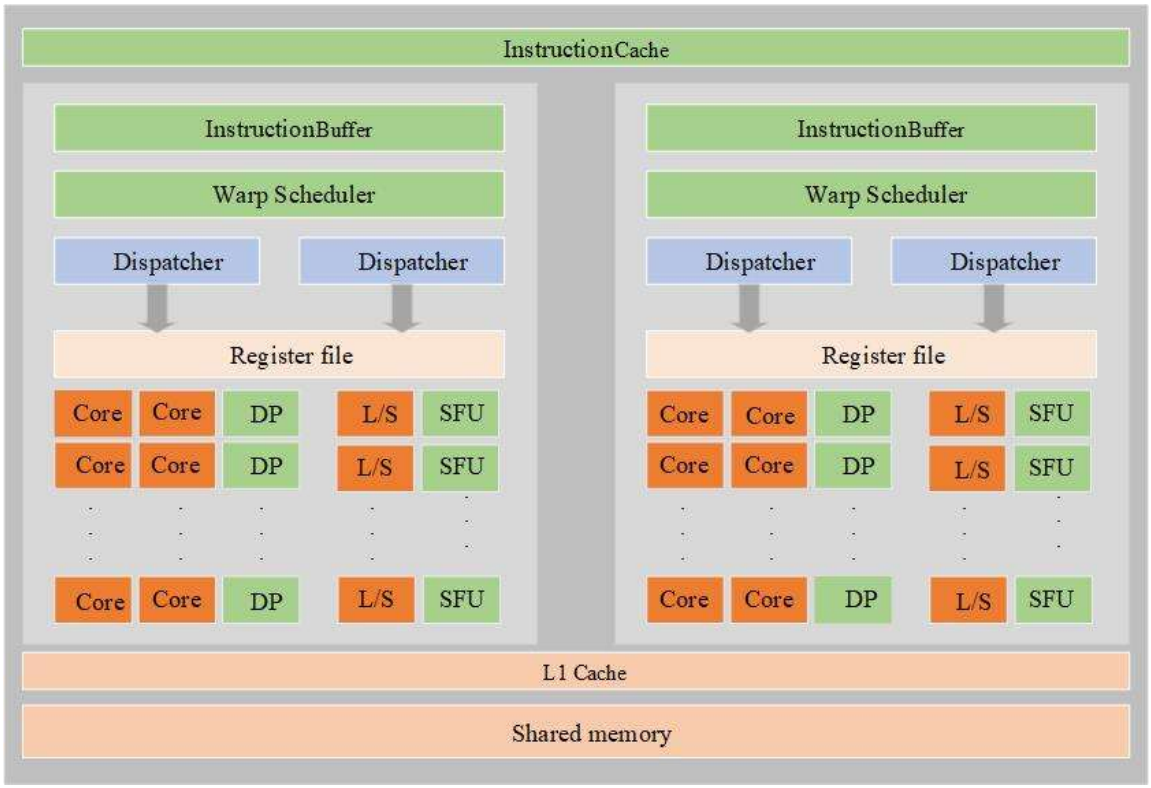

Figure 2. Streaming multiprocessor on the GPU.

Figure 2 illustrates a typical SM schematic. Specifically, the GPUs are intricately designed as parallel, throughput-oriented computing engines, efficiently handling computing-intensive tasks. The highly capable floating-operation capabilities of the

GPUs make them ideal for executing such tasks. Figure 3 depicts this heterogeneous programming mode in a 2D fashion from a programming viewpoint. The foundation of Single Instruction Multiple Threads (SIMT) lies in the massive threads, which can be simultaneously invoked and organized in a two-level hierarchy, with each level supporting up to three dimensions [36].

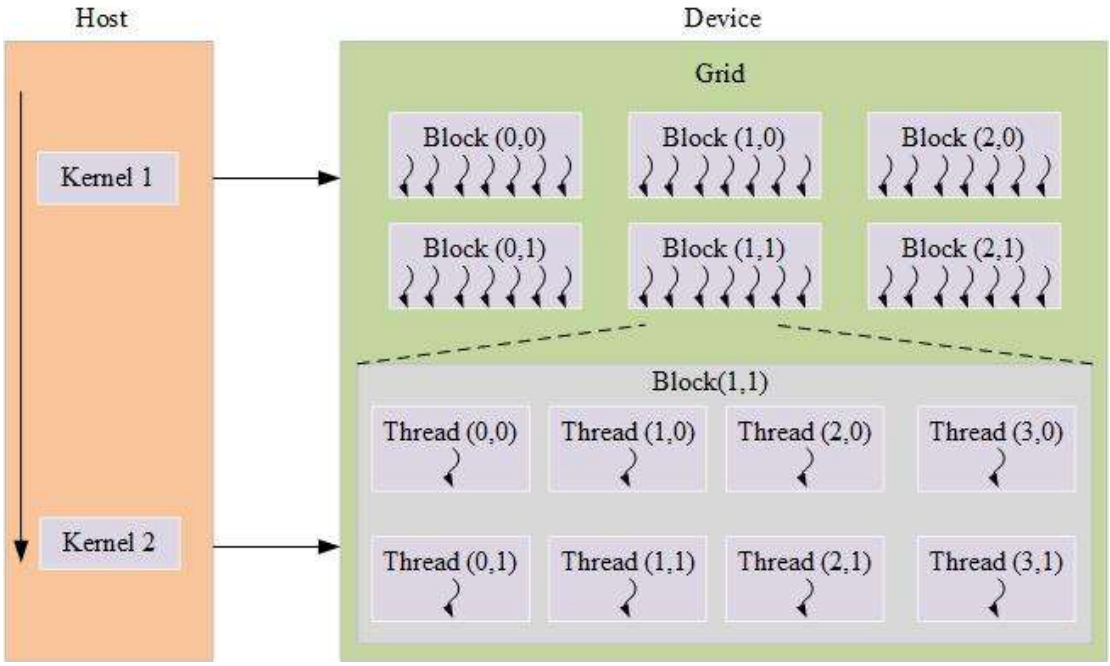

Figure 3. The heterogeneous programing on the GPUs.

Upon kernel launch, the massive threads in the first level are grouped into a block, assigned to a Streaming Multiprocessor (SM) for execution [35]. Synchronization within in the same thread is achieved through an explicitly invoked synchronization barrier [71]. Similarly, the blocks are also organized as the grid in the second level. During the

execution of a CUDA kernel, every set of 32 consecutive threads within a block is integrated into the minimum execution unit, known as a warp [72].

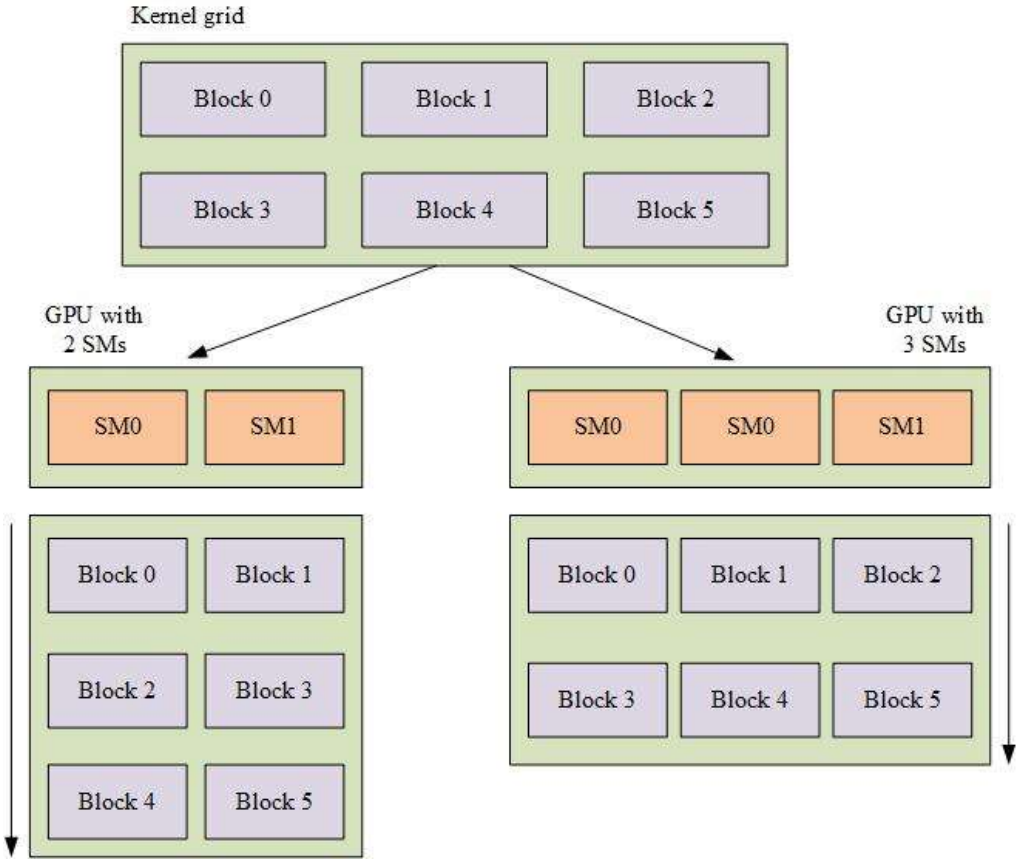

Figure 4. The transparent scalability on the NVIDA GPUs.

All threads in a warp execute instructions in a lock-step manner in accordance with the Single Program Multiple Data (SPMD) concept [73]. While some threads in the same warp encounter an if-statement, it might lead to the thread divergence, causing these threads in a blind-wait state and compromising the advantages of the parallel

devices [71]. Thus, it is crucial for threads to access data from memory in a coalesce manner, leveraging the feature of the GPUs. Additionally, the program should reduce multiple kernel function launches to eliminate the time-consuming communication between the host and the device. Furthermore, NVIDIA CUDA provides a flexible method for automatically adapting the computing cores, allowing the execution of the same application on a varying number of computing cores depending on hardware features, as depicted in Figure 4. This transparently scalable platform broadens the use-cases for existing applications, and reduces the burden on researchers, due to eliminating the need for extensive changes for new or different hardware.

2.2.2 The Multi-core CPU Architecture

The design philosophy of a multi-core platform is centered on minimization of the execution latency for individual threads.

Typically, a multi-core processor consists of a shared memory with multi-CPU on a single chip, offering robust support for multi-threaded applications, improving overall performance by concurrently managing multiple tasks in a parallel fashion [74]. Figure 5 depicts the typical framework of the multi-core CPU architecture. Large last-level on-chip caches are designed to capture frequently accessed data, seamlessly transforming prolonged-memory access into short-latency cache accesses, and the design principle of the arithmetic units and operand data delivery in this architecture focused on the minimizing the latency of operations, even if it led to the increase in both chip area and power consumption [75].

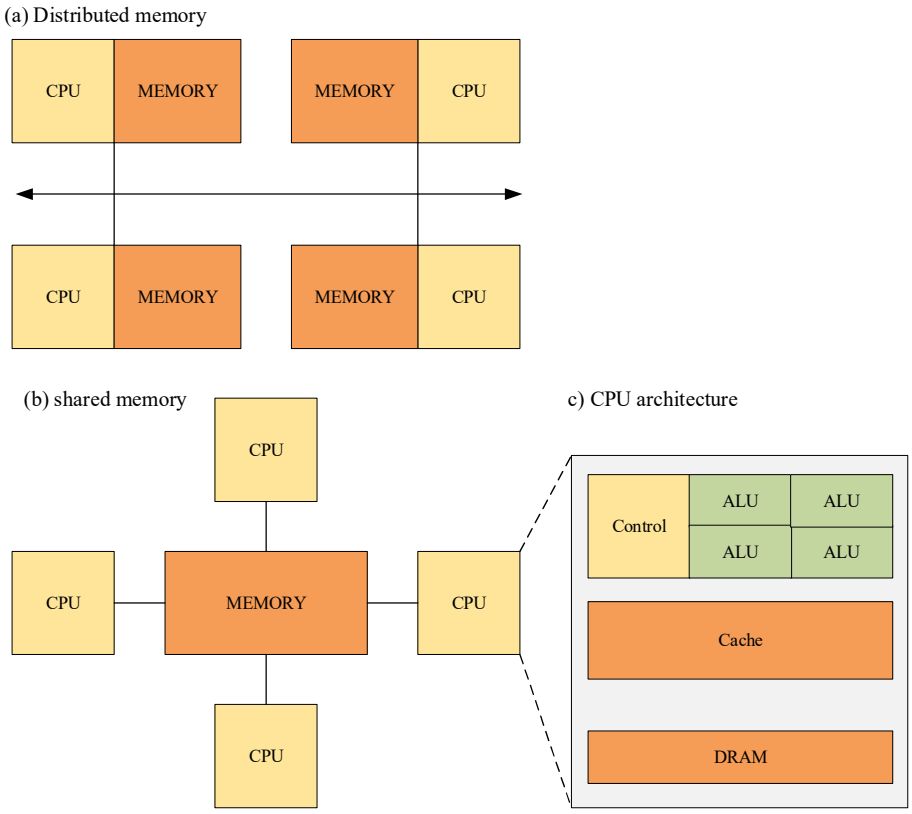

Figure 5. The classic multi-core architecture.

2.2.3 The FPGA Architecture

The Field Programmable Gate Array (FPGA) is an array of configurable logic blocks (CLBs) linked through a programmable interconnection network, along with configurable inputs and outputs [76], as illustrated in Figure 6. Typically, a programmer designs the desired functionality using VHDL or Verilog, and this hardware description is subsequently mapped to logic blocks and memory in the FPGA through a compiler [77]. Furthermore, the FPGA technology facilitates the optimal utilization of parallel computations including features such as pipelining [78], [79], and for-loop optimization

[80], delivering a performance level surpassing that of other fixed architectures, such as microcontrollers [81].

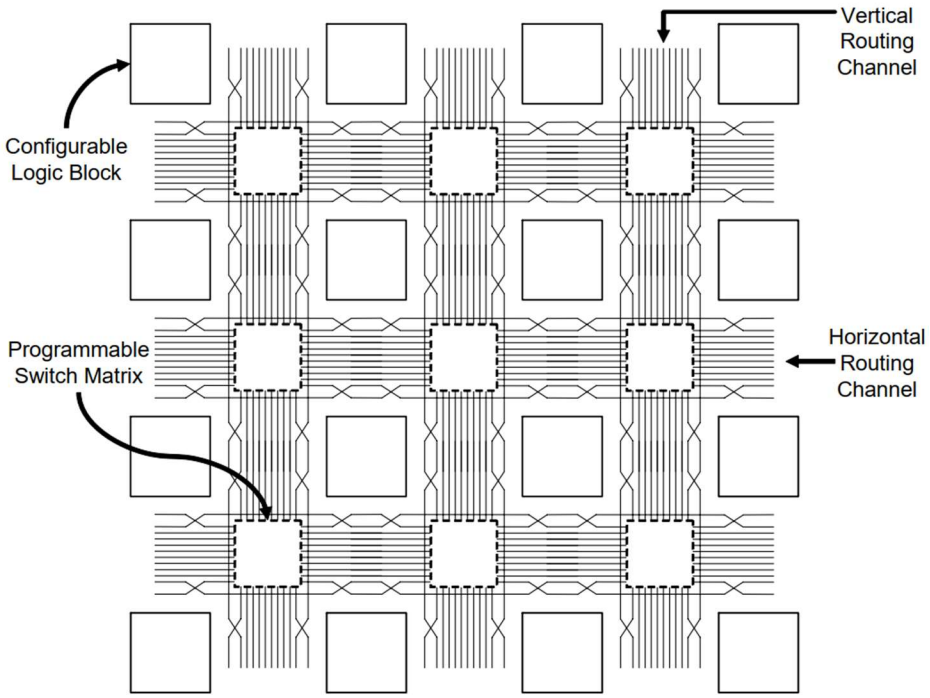

Figure 6. The FPGA architecture [82].

2.2.4 The Review of Parallel Linear System Solver

The linear system solver is critical for PF calculation and consumes much time in the process of updating power mismatches. Specifically, the parallel linear system solver can be summarized as shared and the distributed memory fashion, based on memory utilization. This section will comprehensively review the multi-cores, hybrid CPU-GPU and FPGA solvers in the following discussion.

2.2.4.1 The Multi-core Solver in Power System

In the process of PF calculation, the multi-core method is designed to achieve superior performance improvement through task parallelism [83]. In the work presented in [83], OpenMP leverages coarse-grained parallelism to accelerate the linear system solver in a shared memory fashion. To find the solutions of PF equations in each iteration, Jacobian matrix is initially factorized into two parts, matrix L and U, using the Gaussian elimination. Then the forward-back substitution method is applied to find the linear system's solutions [83]. Zimmerman et al. conducted the implementation on an Intel Core i5-2400 Quad CPU operating at 3.1 GHz. The proposed approach achieves, on average, 2.0x speedup in comparison with the sequential MATPOWER [83], [84]. In the process of solving equations, only the outer loop has been parallelized, due to the sparse data dependencies. Although it improved the performance of linear system solver, the program's efficiency might degrade as the power system scales increase, due to the potential data race issue. To mitigate the data race concerns, Li, F. et al. [85] proposed parallel PF calculation based on the distributed memory, employing the MPI (message passing interface) library. The method utilized triangular decomposition in the parallel algorithm and distributed the data across the multiple computing units. Furthermore, to prevent the deadlock from the asynchrony, this approach incorporates data independence and the process number independence strategy [85]. Specifically, it adopts a master and multi-slave model, where each computing unit calculates its own independent task. The results from the child computing unit are then sent back to the main computing unit to form the complete result. In [85], the implementation achieves a speedup of almost 3x on, a self-built pc-cluster system, depending on the IEEE datasets. However, the authors

limited their test and analysis to the small-scale datasets such as IEEE 30, IEEE 118, and IEEE 300 buses. Consequently, this method may not ensure high-efficiency performance with an increase in power system scale, due to the high communication overhead between different computing units in the distributed memory fashion.

2.2.4.2 The Hybrid CPU-GPU Solver in Power System

The exploration on heterogenous GPU-based parallel PF calculation has been an active topic for several years, and these methods can be broadly categorized into iteration and direct method. Primarily, most iteration methods in PF calculation are considered as optimal algorithms, based on the CG method. Assuming the initial value of voltage magnitude and phase angle, then the program is allowed to find the optimal solution depending on the optimal path. These methods, however, have a limited application and tend to focus on the symmetric and positive definite matrices, due to the large condition number of sparse matrices. To ensure the convergence of PF calculation, preconditioners are designed to reduce the condition number of the matrix, enhancing the program stability and overall performance. In [41], Li. X. et al. proposed the GPU-based FD method and transmitted the Jacobian matrix to the GPU side at the beginning of program execution. The authors further designed the inner iteration with the inexact Newton method, efficiently approaching the solution of linear equations. Although the inexact Newton method may not yield a precise solution for the inner CG iteration, it significantly accelerates the outer iteration, facilitating convergence in the linear equation-solving process. During the whole PF calculation, multiple communications between host and device frequently occurred through the PCI Express (PCIE) bus, causing the program degradation, as shown in Figure 7.

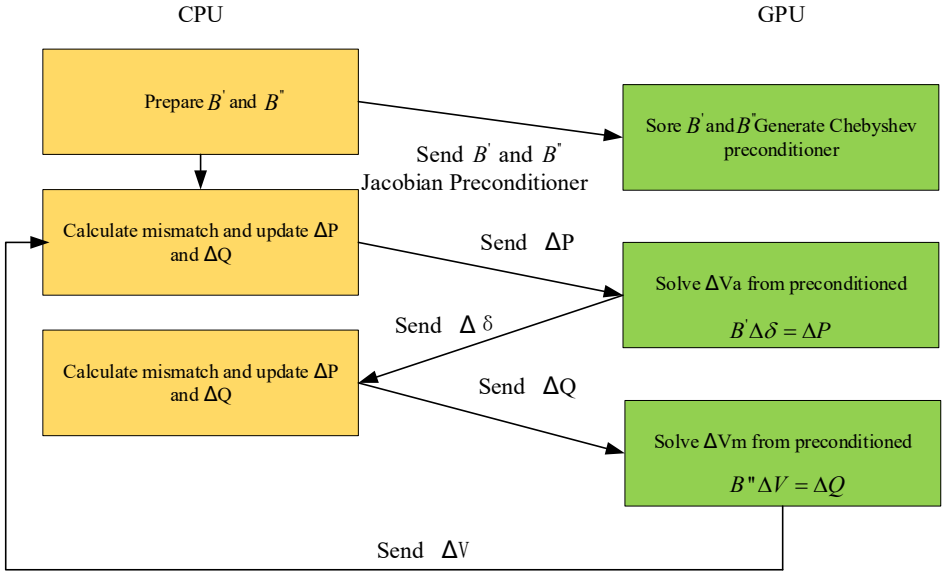

Figure 7. The flowchart of FDPF iteration method [41].

In general, the direct method is acknowledged for its robustness in addressing ill-conditioned problems in PF calculation [33]. However, this method consumes much memory, compared to the iterative methods. Thus, researchers usually adopt a sparse format to distribute parallel tasks on a single GPU platform when dealing with the large-scale power systems. In studies such as [28] and [45], researchers employed a dense matrix format and distributed the computationally intensive tasks on the single GPU platform. For instance, in [28], several power systems with bus sizes ranging from 14 to 2383 were tested on the Intel i7-4500U CPU equipped with an NVIDIA GeForce GT745, yielding significant speedups of 10x ~ 40x compared to the CPU counterpart. Although

the result demonstrated the efficacy of the parallel acceleration, the kernel faced potential crashes due to out-of-bounds memory access, when power system size continues to increase [28]. Similarly, Sooknanan, D.J et al. [45] tested the same datasets on the Intel Xeon Quad Core with the NVIDIA TESLA C1006 GPU, utilizing the same matrix format and achieving almost 7x faster than the CPU for the power system with 2383 buses due to high parallelism of the dense matrix format. The application of the dense format is expected to be suitable for the small-scale power system on the single GPU platform, avoiding memory overflow issues. Thus, the majority of researchers exploited parallelization of PF calculation in the sparse format, avoiding the memory out-of-boundary, and improving the scalability of power system. In [32], researchers explored a CPU-GPU based parallel PF approach, combing sparse matrix techniques. The dataset is organized in CRS and Compressed Column Storage (CSC) formats, reducing the computational complexity from O(N^3) to O ($N^{1.4}$) for an $N \times N$ Jacobian matrix [86]. X., Su. et al. [32] not only focused on solving linear equations in parallel but optimized the creation of Jacobian matrix in a vectorization parallel fashion through the ArrayFire library. Vectorization parallelization involves transforming a scalar program into a vectorized one. In vectorized programs, multiple operations concurrently can be executed concurrently via a single instruction, offering a significant advantage over scalar program, which can only operate on pairs of operands [32]. Similarly, researchers employed the same method to accelerate the generation of Jacobian matrix in [34]. Wang, Z.-Q et al [36] leveraged the strengths of both GPU and CPU to enhance the parallelism of PF calculation. Researchers employed the multi-threaded task-level parallelization with OpenMP [87]. The CPU components are responsible for the tasks such as

initialization of sparse matrix, LU refactorization and scheduling data transmission between the CPU and GPUs, hiding the program execution time through the task scheduling. Furthermore, the computationally intensive tasks were distributed across the multi-GPUs, and the results demonstrated a speedup of over 100x, compared to the open-source tool PandaPower, achieved through the hybrid CPU-GPU based parallel program with a sparsity pattern on both CPU and GPU [36]. However, the scalability of multi-GPUs was limited due to the high dependency on sparse matrices. Despite the significant speedups, the execution time remained slower on the Intel i7-8700k with two NVIDIA GTX 1080 GPUs. The primary constraint appeared to be the communication overhead rather than the computing capabilities of the GPUs.

2.2.4.3 The FPGA Solver in Power System

Although the FPGA parallel methods have been applied to the power industry for a long time, most of them centered on accelerating LU factorization for PF calculation and analysis. Murach, M. et al. [77] utilized the FPGA to optimize the PF analysis through the primal-dual interior point method (PDIPM). Essentially, the authors treated it as a general-purpose optimization problem and leveraged the pipelined and parallel capabilities of the FPGA to improve the performance. Thus, the method in [77] can be classified into iterative method as the CG method. In the process of LU decomposition, lp_solve [88], an open-source package, was utilized to accelerate LU factorization]. This proposed approach indicated that LU performance can be enhanced by 6x, and the overall large-scale PF computation could achieve at least 3x speedup, using FPGA technology over general-purpose workstations [77]. Wang, X.F. et al. [63] introduced a novel partitioning scheme for the nonsymmetric Jacobian matrices in the Newton's method,

resulting in highly efficient parallelization of LU factorization and the subsequent solution of the PF equations. Specifically, the authors reordered the sparse Jacobian matrices into doubly bordered block diagonal (DBBD) form, facilitating the numerous simultaneous operations in LU decomposition. Thus, PF equations can be solved using the parallel DBBD LU factorization algorithm described in [89], after reordering the Jacobian matrix into the DBBD form. The lower and upper triangular matrices were calculated through the parallel forward reduction and backward substitution [90]. The implementation was tested on the ES2S180 platform and achieved almost the speedup of 7x, compared to the EP20KE1500 counterpart.

CHAPTER THREE

POWER SYSTEM MODELING AND FORMAT OF SPARSE MATRIX

PF calculation is a fundamental computation in power system analysis, planning and operation. Many power system applications, such as optimal PF and contingency analysis, require the creation of a power system model and acceleration in PF calculation. To fix PF issues, numerical methods are applied to the power system model, linearizing the non-linear system, and obtaining the numerical solution for voltage magnitude and angle equations. The usage of sparse matrix format can alleviate the computation burden on single-GPU parallel platform. However, the different storage formats, such as CRS or CCS, necessitate the different optimal strategies on the hardware platforms based on the feature of the storage format. This chapter commences with an introduction to power system modeling, followed by the exploration of solving linear systems. The discussion concludes with features of sparse matrix storage formats.

3.1 Power System Modeling and Analysis

A PF model and analysis serve as a comprehensive representation of engineering modeling and numerical method employed to estimate the electrical PF within an interconnected power system. They are designed to repeatedly apply specific numerical methods to obtain the voltage magnitudes and angles for all the buses in a power system and to calculate the real and reactive powers of peripheral equipment connected to the buses [1]. Specifically, Power system is modeled as a set of non-linear equations, which are linearized into a set of linear equations. This transformation allows for the approach to the numerical solution of the original non-linear system, gradually, avoiding solving

non-linear equations, directly. The NR method is one of classic methods used to transform the non-linear systems into linear ones, laying the foundation for other direct solvers such as the FD method. This section will introduce the computations involved in power system analysis and corresponding linear system solver.

3.1.1 The Model of AC PF

PF problem involves calculating bus voltage magnitude and angle at system buses, as well as line flows, based on loads, generation, and transmission network. Kirchoff's law is applied to each bus in the system, where the sum of the complex power injected at a bus should be equal to output of the complex power on same bus, under the stable conditions. Thus, a mathematical power flow model can be created, based on the relationship between the injection and the consumption of complex power. To solve nonlinear PF problem, the nodal equations for a power system network are expressed as follows:

$$I = Y_{bus}V \tag{3.1}$$

Where I is the N vector of source currents injected into each bus, and V is the N vector of bus voltages. Y_{bus} is the admittance matrix of the power system. For bus k , the k_{th} equation in (3.1) is written as:

$$I_k = \sum_{n=1}^{N} Y_{kn}V_n \tag{3.2}$$

The complex power delivered to bus k is expressed as:

$$S_k = P_k + jQ_k = V_k I_k^* \tag{3.3}$$

Where S_k is the complex on bus k . P_k and Q_k are the active and reactive power on bus k , respectively. Using (2) and (3),

$$P_k + jQ_k = V_k[\sum_{n=1}^{N} Y_{kn} V_n]^* \tag{3.4}$$

With the following notation,

$$V_n = V_n e^{j\delta_n} \tag{3.5}$$

$$Y_{kn} = Y_{kn} e^{j\theta_{kn}} = G_{kn} + jB_{kn} \tag{3.6}$$

Therefore, equation (3.4) can be written as:

$$P_k + jQ_k = V_k \sum_{n=1}^{N} Y_{kn} V_n e^{j(\delta_k - \delta_n - \theta_{kn})} \tag{3.7}$$

Where δ_k and δ_n are the phase angle on the bus k and n , respectively. Y_{kn} comes from the admittance matrix, Y_{bus} . θ_{kn} is the angle of admittance between bus k and n . Taking the real and imaginary parts of (3.7), the power balance equations are written as follows:

$$P_k = V_k \sum_{n=1}^{N} Y_{kn} V_n \cos(\delta_k - \delta_n - \theta_{kn}) \tag{3.8}$$

$$Q_k = V_k \sum_{n=1}^{N} Y_{kn} V_n \sin(\delta_k - \delta_n - \theta_{kn}) \tag{3.9}$$

When the Y_{kn} is expressed in rectangular coordinates as:

$$P_k = V_k \sum_{n=1}^{N} V_n \left[G_{kn} \cos(\delta_k - \delta_n) + B_{kn} \sin(\delta_k - \delta_n) \right] \tag{3.10}$$

$$Q_k = V_k \sum_{n=1}^{N} V_n \left[G_{kn} \sin(\delta_k - \delta_n) - B_{kn} \cos(\delta_k - \delta_n) \right] \tag{3.11}$$

Where the G_{kn} and B_{kn} are conductance and susceptance, respectively. From the PF model, solving the power balance equations facilitates power system engineers to maneuver the voltage magnitude and phase angle at each bus, allowing them to analyze whether the power system is operating under the balanced three-phase steady-state conditions [91].

3.1.2 Power Flow Methods

PF problem involves calculating the voltage magnitude and phase angle at each bus in a power system under balanced three-phase steady-state conditions [91]. To address this issue, various methods have been developed, including Gauss-Seidel (GS), NR [92], and FDPF [93]. Among these, the NR method is widely used for PF analysis and calculation, due to its rapid convergency rate. Equation (10) and (11) employe the Taylor series techniques on the non-linear equations for linearization, and the set of non-linear equations can be formulated as follows:

$$\Delta f = J \Delta x \tag{3.12}$$

$$\Delta x = \begin{bmatrix} \Delta \delta \\ \Delta V \end{bmatrix} \tag{3.13}$$

Where J is the Jacobian matrix. Specifically, equation (3.12) and (3.13) can further be expressed in the form of matrix:

$$\begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} = \begin{bmatrix} J_1 & J_2 \\ J_3 & J_4 \end{bmatrix} \begin{bmatrix} \Delta \delta \\ \Delta V \end{bmatrix} \tag{3.14}$$

Where ΔP and ΔQ are the power mismatches, respectively. J_1, J_2, J_3 and J_4 are listed below:

$$J_1 = \begin{cases} \dfrac{\partial P_k}{\partial \delta_n} = V_k Y_{kn} V_n \sin(\delta_k - \delta_n - \theta_{kn}) & (n \neq k) \\ \dfrac{\partial P_k}{\partial \delta_k} = -V_k \sum\limits_{\substack{n=1 \\ n \neq k}}^{N} Y_{kn} V_n \sin(\delta_k - \delta_n - \theta_{kn}) & (n = k) \end{cases} \tag{3.15}$$

$$J_2 = \begin{cases} \dfrac{\partial P_k}{\partial V_n} = V_k Y_{kn} \cos(\delta_k - \delta_n - \theta_{kn}) & (n \neq k) \\ \dfrac{\partial P_k}{\partial V_k} = V_k Y_{kk} \cos\theta_{kk} + \sum\limits_{n=1}^{N} Y_{kn} V_n \cos(\delta_k - \delta_n - \theta_{kn}) & (n = k) \end{cases} \tag{3.16}$$

$$J_3 = \begin{cases} \dfrac{\partial Q_k}{\partial \delta_n} = -V_k Y_{kn} V_n \cos(\delta_k - \delta_n - \theta_{kn}) & (n \neq k) \\ \dfrac{\partial Q_k}{\partial \delta_k} = V_k \sum\limits_{\substack{n=1 \\ n \neq k}}^{N} Y_{kn} V_n \cos(\delta_k - \delta_n - \theta_{kn}) & (n = k) \end{cases} \tag{3.17}$$

$$J_4 = \begin{cases} \dfrac{\partial Q_k}{\partial V_n} = V_k Y_{kn} \sin(\delta_k - \delta_n - \theta_{kn}) & (n \neq k) \\ \dfrac{\partial Q_k}{\partial V_k} = -V_k Y_{kk} \sin\theta_{kk} + \sum\limits_{n=1}^{N} Y_{kn} V_n \cos(\delta_k - \delta_n - \theta_{kn}) & (n = k) \end{cases} \tag{3.18}$$

Initially, the arbitrary values of V and θ are assumed and introduced into the NR iteration of (3.14). The voltage magnitude, phase angle and power mismatch are then updated in the process of each iteration, repeatedly, until the power mismatches satisfied the convergent condition, for all $k = 1, 2, \cdots, N$ as follows:

$$\| \frac{x_k(i+1) - x_i(i)}{x_i(i)} \|_\infty < \varepsilon \tag{3.19}$$

For the sake of clarity, Figure 1 illustrates the iterative process of PF calculation based on the NR method.

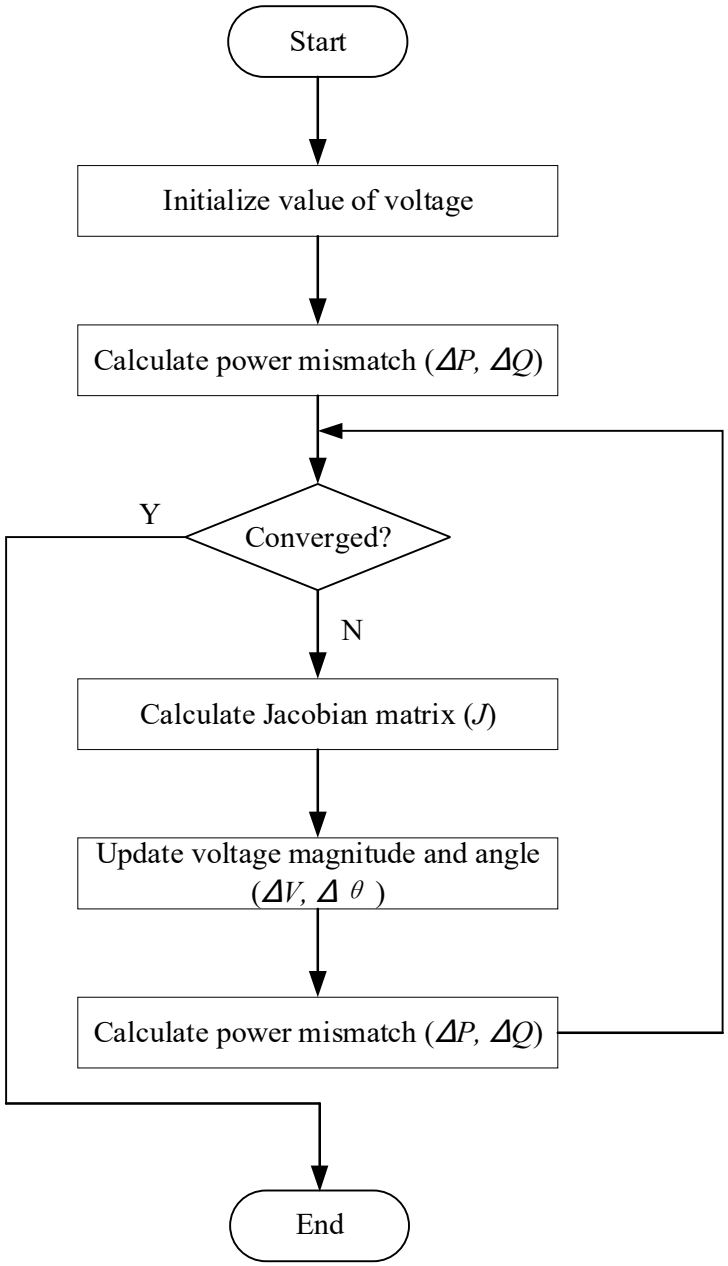

Figure 8. The flow chart of NR method.

The computationally expensive calculation involves updating both Jacobian coefficient matrices and the power mismatches. Nevertheless, the NR method outperforms the GS method during the PF calculation, yielding the results in fewer iterations [1], [29]. Consequently, the NR method is also widely adopted by some researchers, particularly for the large-scale systems [1], [91]. Furthermore, the Jacobian matrix in the NR method can serve as a valuable tool for sensitivity analysis and other control-related issues. This significance has led to the focus of several studies on parallel algorithms to accelerate the GPU-based NR method. The FDPF method, an extension of the NR method proposed by Stott and Alsac [93], simplifies the complicated calculation of Jacobian coefficient matrices. Consequently, the formula can be modified from (3.14):

$$\begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} = \begin{bmatrix} J_1 & 0 \\ 0 & J_4 \end{bmatrix} \begin{bmatrix} \Delta \delta \\ \Delta V \end{bmatrix} \tag{3.20}$$

Transmission lines in power system have a very high X/R ratio. For such system, $\frac{\partial P_i}{\partial \delta_j}$ is much larger than $\frac{\partial Q_i}{\partial \delta_j}$, and $\frac{\partial Q_i}{\partial V_j}$ is much larger than $\frac{\partial P_i}{\partial V_j}$ for all $i, j = 1, 2, \cdots, N$. Also, the angular differences $(\delta_k - \delta_n)$ between typical buses in the power system are usually so small that $\cos(\delta_k - \delta_n) \approx 1$ and $\sin(\delta_k - \delta_n) \approx (\delta_k - \delta_n)$. With such approximation, the elements of the submatrices, J_2 and J_3, in the Jacobian matrix, are approximately zero. As a result, equation (3.20) can further be separated into two independent and decoupled tasks as follows:

$$\Delta P = J_1 \Delta \delta \tag{3.21}$$

$$\Delta Q = J_4 \Delta V \tag{3.22}$$

On the other hand, the diagonal ($n = k$) elements of J_1 in equation (3.15) can be written as:

$$\frac{\partial P_k}{\partial \delta_k} = -V_k \sum_{n=1}^{N} Y_{kn} V_n \sin(\delta_k - \delta_n - \theta_{kn}) - \left|V_k\right|^2 Y_{kk} \sin\theta_{kk} \tag{3.23}$$

Substituting the first term of above equation with $-Q_k$, as defined in equation (3.9), yields:

$$\frac{\partial P_k}{\partial \delta_k} = -Q_k - \left|V_k\right|^2 Y_{kk} \sin\theta_{kk} = -Q_k - \left|V_k\right|^2 B_{kk} \tag{3.24}$$

Where $B_{kk} = Y_{kk} \sin\theta_{kk}$ represents the imaginary part of the diagonal elements of the bus admittance matrix. Specifically, B_{kk} is the sum of susceptance of all the elements incident to bus k . Typically, the self-susceptance B_{kk} □ Q_k in the power system. Consequently, the Q_k can be approximated as zero, which yields equation (3.25), when assuming $\left|V_k\right|^2 \approx \left|V_k\right|$.

$$\frac{\partial P_k}{\partial \delta_k} = -V_k B_{kk} \tag{3.25}$$

Under the stable conditions, $(\delta_k - \delta_n)$ is quite small. Thus, in equation (3.15) supposing $(\delta_k - \delta_n - \theta_{kn}) \approx -\theta_{kn}$, the off-diagonal ($n \neq k$) elements of J_1 are expressed as:

$$\frac{\partial P_k}{\partial \delta_n} = -V_k V_n B_{kn} \tag{3.26}$$

Where $B_{kn} = Y_{kn} \sin\theta_{kn}$. Further simplification is obtained by assuming $V_n \approx 1$.

$$\frac{\partial P_k}{\partial \delta_n} = -V_k B_{kn} \tag{3.27}$$

Similarly, the diagonal and off-diagonal elements of J_4 are written as:

$$\frac{\partial Q_k}{\partial V_k} = -V_k B_{kk} \tag{3.28}$$

$$\frac{\partial Q_k}{\partial V_n} = -V_k B_{kn} \tag{3.29}$$

With these assumptions, equations (3.21) and (3.22) can be expressed as follows:

$$\frac{\Delta P}{V} = -B' \Delta \delta \tag{3.30}$$

$$\frac{\Delta Q}{V} = -B'' \Delta V \tag{3.31}$$

Where B' and B'' are the imaginary parts of the bus admittance matrix Y_{bus}. Since the components of B' and B'' are constant, they only need to be calculated at the beginning of the iteration. Although the FDPF method is derived from the NR method, it is algorithmically much simpler and more efficient [4]. One critical reason is that the non-zero Jacobian coefficient matrices are fixed and reusable throughout the computation, unlike the NR method that needs to frequently update the Jacobian coefficient matrices during each decomposition process [4].

In summary, the aforementioned simplifications can lead to rapid solutions in PF calculation for most power systems. While the FDPF usually takes more iterations to converge, it is significantly much faster than the NR algorithm due to the absence of Jacobian matrix repeated computation in each iteration. Importantly, since the power mismatch equations remain unchanged, the solution obtained through the FD algorithm is the same compared to the NR method. Additionally, according to equations (3.21) and

(3.22), two tasks, solving voltage magnitude and phase angle equations, are well-suited for allocation to the different parallel devices, such as multi-GPUs and multi-cores, enhancing the granularity of parallelization.

3.2 Linear System Solver in PF Calculation

In the process of the PF calculation through either the NR or FDPF method, solving the linear system of matrix equations is critical [1]. The discussions in this section will be built on a general linear system formulated as equation (3.23).

$$Ax = b \tag{3.32}$$

Where the size of the linear system A is $n \times n$, and the size of right-hand side vector b is $n \times 1$. The vector x is the unknowns to be solved. Generally, the linear solvers can be classified into two categories, i.e., direct and iteration methods, respectively. Specifically, the direct method, based on Gaussian elimination, achieves the precision solutions through a finite set of arithmetic operations, including lower-upper (LU) factorization, QR decomposition or inverse matrix A^{-1} calculation. On the other hand, the iteration methods, based on CG method, are designed to find numerical solutions for equation (3.23) when the matrix is symmetric and positive definite. Iteration methods are typically conducted using a combination of pre-conditioners and implemented as iterative algorithms for the analysis of large-scale power systems. Specifically, the iteration methods will generate a set of solutions in each iteration, and the solutions are expected to approach the real solution of equation (3.23), gradually, until they satisfy the convergence criteria.

This section will briefly introduce the direct and iteration methods and analyze the pros and cons of these methods in linear system solvers.

3.2.1 Inverse Matrix Linear System Solver

Gaussian elimination is the foundation of inverse matrix solver, and consider the following augmented matrix:

$$\left[\begin{array}{ccc|ccc} a_{11} & a_{12} & a_{13} & 1 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 1 & 0 \\ a_{31} & a_{32} & a_{33} & 0 & 0 & 1 \end{array}\right] \tag{3.33}$$

Or the augmented matrix is expressed in matrix format:

$$AX = I \tag{3.34}$$

Where A is the 3×3 square matrix and I is the identity matrix. Assuming A is not singular, thus a unique inverse matrix A^{-1} can be obtained through Gaussian elimination.

$$\left[\begin{array}{ccc|ccc} 1 & 0 & 0 & t_{11} & t_{12} & t_{13} \\ 0 & 1 & 0 & t_{12} & t_{22} & t_{23} \\ 0 & 0 & 1 & t_{31} & t_{32} & t_{33} \end{array}\right] \tag{3.35}$$

Where T is the inversion of matrix A, i.e., $X = T = A^{-1}$. The solution of linear system of equation (3.32) are evaluated as follows:

$$x = A^{-1}b \tag{3.36}$$

Although this method is straightforward, it would take much time to converge even if utilizing the parallel computing libraries. Evaluating the inverse Jacobian matrix, directly, could generate many fill-ins and allocate substantial space to store the temporary variables, adversely degenerating the performance due to the inherently sparse nature of the Jacobian matrix. Technically, evaluating the inverse matrix of PF system facilitates researchers to analyze the state of large sparse power systems, rather than directly obtaining the solutions of linear equations.

3.2.2 LU Factorization

LU factorization is a classic numerical method for solving linear systems. In the process of PF calculation, the LU solver is utilized in each iteration of the NR or FDPF method, and this direct solver presents the more robustness in ill-conditioned problems [33]. Generally, the LU method is well-suited for the coefficient matrix of the power system due to its square nature. The matrix A is decomposed into two matrices: the lower triangular part L and the upper triangular part U. To improve the stability of the LU method, partial pivoting technique is employed, ensuring reliable solutions with LU factorization [94]. This method includes three steps:

1. Elimination: Factorize matrix A into lower and upper triangular matrices L and U then obtain permutation matrix P, which can be expressed as follows:

$$PA = LU \tag{3.37}$$

2. Forward: Generate the intermediate vector *y* based on P and L in equation (3.37) and they can be written as:

$$Ly = Pb \tag{3.38}$$

3. Backward: solving triangular system, based on vector *y* from equation (3.38), as follows:

$$Ux = y \tag{3.39}$$

Sometimes, Cholesky factorization can be employed on equation (3.37) for simplification, as follows:

$$PA = LL^t \tag{3.40}$$

Where the L^t is the transposition matrix of L. The Cholesky factorization, however, focuses on the symmetric and positive definite matrices and it may usually fail to converge in PF calculation, due to the nature of power system structure. Thus, this method is generally applied to special situations, limiting its broader application in power systems. Additionally, the computational time of the linear system solver contributes a significant portion of the PF calculation, sometimes reaching up to about 80 percent of the total computation time [95]. In power system engineering, researcher often transform matrix A into another simpler matrix format B to reduce the fill-ins and accelerate the linear system solver, as an alternative to using the original matrix A. The transformation is described as:

$$B = QAQ^t \tag{3.41}$$

Where the permutation matrix satisfies $QQ^t = Q^tQ = I$. Therefore, the equation (3.32) is transformed as follows:

$$B\hat{x} = \hat{b} \tag{3.42}$$

Where $\hat{x} = Qx$ and $\hat{b} = Qb$. Although the equation (3.42) has similar form with equation (3.32), the transformation reduces many fill-ins, avoiding the extra computation. Thus, many optimal algorithms have focused on these critical parts [51, 96-99], utilizing parallel methods to reduce the execution time.

Generally, the computation complexity of LU method is $O(n^3)$. For single small-memory platforms, the algorithm takes $O(n+|A|+f)$ time in a sparse matrix format,

where f is the number of floating-point operations performed. Thus, the LU method remains an ideal option for PF calculation.

3.2.3 QR Factorization

QR factorization is also an alternative for linear system solvers, and it is also possible to adopt QR decomposition as an option for leveraging GPU-based parallel processing. Specifically, the QR factorization involves decomposing a matrix with linearly independent columns into a product of an orthogonal matrix [100]. It can be expressed as follows:

$$A = Q \cdot R \tag{3.43}$$

Where Q and R indicate an $m \times m$ unitary matrix and an $m \times n$ upper triangular matrix, respectively. Generally, several methods for implementing the QR decomposition are available, such as Gram-Schmidt process, Householder transformations, and Givens rotations [101]. In practice, compared to LU factorization, the QR decomposition is more stable, because LU decomposition without pivoting may suffer from significant numerical instability, as described in [27]. However, the QR factorization requires almost twice as many operations and involves more complicated steps that are not as parallel as a matrix-matrix product [102, 103]. Despite the potential increase in the computation overhead of linear system solvers, a few studies have still exploited GPU-based QR parallel computing methods, due to their high numerical stability even without pivoting [104].

3.2.4 Iteration Linear Solver

Typically, the iterative methods are considered as the optimal algorithms, designed to find the numerical solutions for linear equation in PF calculation. Usually, the CG method is typically employed to handle large-scale power systems. The conventional CG method minimizes the functions, as follows:

$$f(x) = \frac{1}{2} x^t A x - x^t b \tag{3.44}$$

When its gradient is zero.

$$\nabla f(x) = Ax - b \tag{3.45}$$

Then, x is the solution of $Ax = b$, when the function (3.45) finds its minimum vectors. In the process of iteration, each residual and each newly generated direction vector are A-orthogonal to all the previously selected direction vectors, guaranteeing that the current path is orthogonal to the last step, as illustrated in Algorithm 3.1.

Algorithm 3.1. General CG method.

Algorithm 3.1: General CG method

1: $r = b - Ax_0$; $x = x_0$; $p_1 = p_0$;

2: While $\|r\| > \varepsilon$

3: $\alpha_k = r_{k-1}^t r_{k-1} / p_k^t A p_k$;

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

Algorithm 3.1: General CG method

5: $r_k = r_{k-1} - \alpha_k A p_k$;

6: $\beta_{k+1} = r^t r / r_{k-1}^t r_{k-1}$;

7: $p_{k+1} = r_k + \beta_{k+1} p_k$;

8: End While

The convergence of CG method can be expressed as:

$$\| x_n - x^* \|_A \leq \left(\frac{\kappa(A)-1}{\kappa(A)+1}\right)^n \| x_0 - x^* \|_A \tag{3.46}$$

Where x_n, x^* and x_0 are current solution, exact solution, and initial solution vectors, respectively. $\kappa(A)$ denotes the condition number of A. The inequality suggests that CG may converge slowly when A has a large condition number. Generally, most CG iterations probably struggle to converge in PF calculation, due to the inherent nature of power system. The convergence of these iterative methods is severely dependent on the property of the preconditioner. Researchers often took a long time to validate the preconditioners, repeatedly, until they identified the optimal one. As a result, designing a suitable preconditioner becomes a time-consuming and tedious task. Nevertheless, the parallel CG method has been successfully applied to PF calculation in various studies, including [24, 42, 68].

3.2.5 Preconditioner

Generally, calculating the precise inverse of a matrix, especially a large-scale sparse matrix, is quite a time-consuming task. Hence, a preconditioner is employed to transform the original matrix A into another one, narrowing the eigenvalue spectrum and reducing the condition number of matrix. In the process of preconditioning, the preconditioner progressively approximates A^{-1}, allowing the pseudoinverse of matrix A to be involve in solving the linear system equations and minimizing the computation overhead in PF calculation. As a result, this facilitates an improvement in the convergent rate. The Jacobian preconditioner takes the simplest form among all the widely used preconditioners, as illustrated in equation (3.32), the iteration form can be expressed as:

$$x_{k+1} = M^{-1}Nx_k + M^{-1}b \tag{3.47}$$

Where M and N represent the splitting components of A:

$$A = M - N \tag{3.48}$$

Thus, the iterative form of equation (3.47) can be transformed into:

$$x_{k+1} = Gx_k + f \tag{3.49}$$

Where $f = M^{-1}b$ and

$$G = M^{-1}N = M^{-1}(M - A) = I - M^{-1}A \tag{3.50}$$

Thus, the Jacobian preconditioner G_J is written as:

$$G_J = I - D^{-1}A \tag{3.51}$$

Where D^{-1} is the diagonal inverse of matrix A.

ILU and incomplete Cholesky (IC) factorization are two common methods of incomplete factorization used for preconditioning. The analysis of these two methods

would focus on the ILU method, since IC can be considered as a special case of ILU. Consider an incomplete factorization of the form:

$$A = LU + E \tag{3.52}$$

Where E represents the error, and then the matrices of the various forms of the preconditioned linear system are expressed as:

$$L^{-1}AU^{-1} = I + L^{-1}EU^{-1} \tag{3.53}$$

When the matrix A is diagonally dominant, both L and U are well-condition, and the eigenvalue of $L^{-1}EU^{-1}$ remain confined within reasonable limits [105]. Thus, the preconditioner, $M \approx LU$, can be approximated to A such that $M \approx A$. Conversely, the L^{-1} and U^{-1} may have a large norm, causing the error $L^{-1}EU^{-1}$ to exhibit arbitrarily large eigenvalues and decreasing the ILU instability. Experimental observations suggest that ILU preconditioners can perform poorly in such situations, which commonly occur when the matrices are indefinite or possess large nonsymmetric parts [106].

The sparse matrix approximating inverse method aimed to address the ill-conditioned factorization of ILU. Technically, an intuitive idea for finding approximate inverses of arbitrary sparse matrices is to attempt to find a sparse matrix M, minimizing the Frobenius norm of the residual matrix $I - AM$:

$$f(M) = \| I - AM \|_F^2 \tag{3.54}$$

Determining the preconditioner, M, can be regarded as the process of minimizing the objective function (3.54). Ideally, $f(M)$ should achieve the minimum value, i.e., $f(M) = 0$ and $M = A^{-1}$. In practice, the objective function (3.54) decouples

into the sum of the squares of the 2-norms of the individual columns of the residual matrix $I - AM$ [106], as follows:

$$f(M) = \sum_{j=1}^{n} \| e_j - Am_j \|_2^2 \tag{3.55}$$

Where e_j and m_j are the $j-th$ columns of the identity matrix and of the matrix M, respectively. The independence of the sum in the residual matrix, as indicated by function (3.55), allows for the application of parallel approaches on computers. Furthermore, sparse matrix approximate inverse can be utilized in non-associated cases since factorization-based preconditioners typically require the association of the factorization matrix with the original matrix at some point during the computation. In contrast, the approximate inverse can be directly plugged in with the $MAx = Mb$, and then solved.

Chebyshev preconditioner is a polynomial-based preconditioner, and the inversion of matrix A^{-1} can be expressed as:

$$A^{-1} = \frac{c_0}{2} I + \sum_{k=1}^{k=\infty} c_k T_k(Z) \tag{3.56}$$

Where Z is transform of A spectrum from $[\alpha, \beta]$ to $[-1,1]$. Supposing α is the smallest eigenvalue of A, while the β is largest one. c_k represents the decay rate for the entries of matrix A^{-1}, estimated through the constant condition number q of matrix A. Thus, the iterative form of the Chebyshev preconditioner can be further expressed as:

$$Z = \frac{2}{\beta - \alpha} [A - \frac{\beta + \alpha}{2} I] \tag{3.57}$$

$$\begin{cases} T_0 = I \\ T_1 = Z \\ T_k = 2ZT_{k-1}(Z) - T_{k-2}(Z) \end{cases} \tag{3.58}$$

$$c_k = \frac{1}{\sqrt{\alpha\beta}}(-q)^k \tag{3.59}$$

$$q = \frac{1-\sqrt{\frac{\alpha}{\beta}}}{1+\sqrt{\frac{\alpha}{\beta}}} \tag{3.60}$$

The Chebyshev preconditioner involves matrix operations from the above sets of equations, making it well-suitable for acceleration on the parallel platforms. The accelerating efficacy, however, is very limited. The recursive equations (3.58) cannot be easily split into multiple parallel tasks, due to the data dependency in the recursive formulas. Also, the cost of evaluating the eigenvalue of large-scale sparse matrix is very high, even if the parallel platforms have high floating-point operation capabilities. The specific Chebyshev preconditioning algorithm is illustrated below Algorithm 3.2.

Algorithm 3.2: Chebyshev preconditioning method

1: $\beta = max(eig(A))$;

2: $\alpha = \frac{\beta}{ratio}$;

3: $Z = \frac{2}{\beta-\alpha}A - \frac{\beta+\alpha}{\beta-\alpha}I$;

Algorithm 3.2: Chebyshev preconditioning method

4: $q = (1 - \sqrt{\alpha / \beta}) / (1 + \sqrt{\alpha / \beta})$;

5: $c_0 = \frac{(-q)^0}{\sqrt{\alpha\beta}}$; $c_1 = \frac{(-q)^1}{\sqrt{\alpha\beta}}$;

6: $T_{p0} = I$; $T_{p1} = Z$;

7: $G = \frac{1}{2} c_0 I + c_1 T_{p1}$;

8: For i to r

9: $T = 2ZT_{p1} - T_{p0}$;

10: $c = \frac{(-q)^i}{\sqrt{\alpha\beta}}$;

11: $G = G + cT$;

12: $T_{p0} = T_{p1}$; $T_{p1} = T$;

8: End For

In summary, the preconditioning technique effectively accelerates the solutions of linear equations when employing the CG method. The design of preconditioner, however, remains a complex operation, and the preconditioner has a limited application on special linear system, since its convergence is confined by coefficient matrix property such as diagonally dominant matrix, or symmetric positive definite (SPD) matrix.

3.3 Sparse Matrix Format

Power system buses have limited connections with other buses [35, 107], resulting in matrices from power system exhibiting very high sparsity as the number of

power system buses increases. It is a good choice to utilize the sparse matrix storage formats on a single parallel platform, effectively reducing the memory burden and computation overhead. In power system engineering, sparse formats such as COO, CRS, and CSC are commonly employed. These different sparse storage formats exhibit distinct memory access patterns, potentially impacting the computing performance in the process of PF calculation.

3.3.1 COO Format

COO is a widely used and straightforward storage format. In this format, all nonzero entries are stored sequentially and accompanied by two indices that specify their elements position in the matrix. The information is stored in a structure-of-arrays form, a common approach in sparse matrix storage to enhance cache efficiency [108], as illustrated in Figure 9.

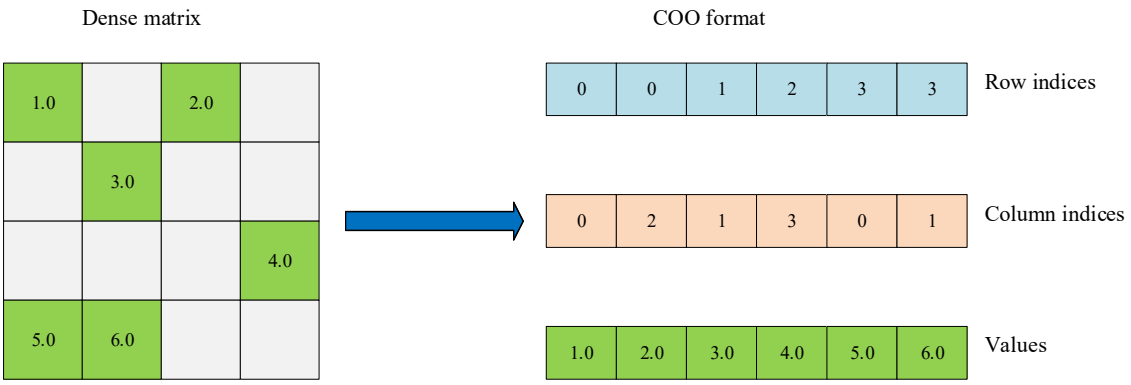

Figure 9. 4×4 sparse matrix in COO format.

3.3.2 CRS Format

CRS format is an evolution of the COO format, where each row is further compressed individually, retaining only additional information about the matrix column index. Figure 10 illustrates the CRS format below. While the CRS format in Figure 9 represents a sparse matrix using three arrays in comparison with COO format, it utilizes relatively less memory.

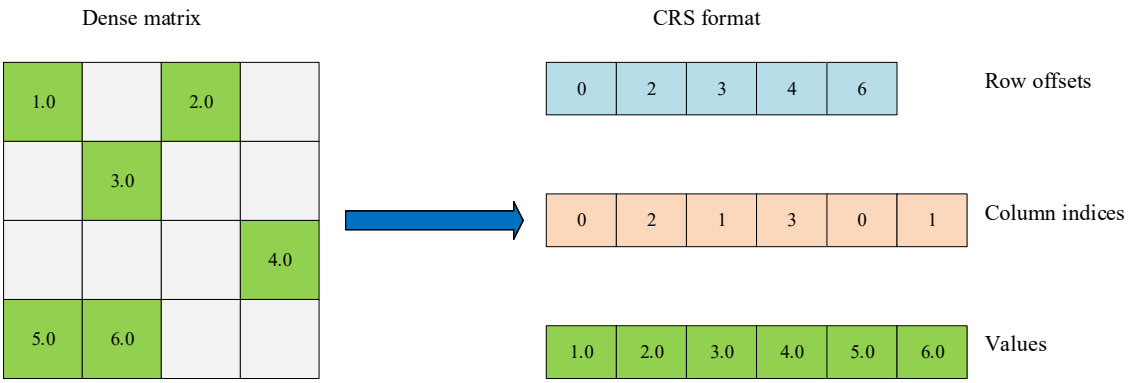

Figure 10. 4×4 sparse matrix in CRS format.

3.3.3 CSC Format

Similarly, CSC format is nearly identical to the CRS format, differing only in that the CSC format compresses each column of the sparse matrix and retains information about each entry's row, as illustrated in Figure 11.

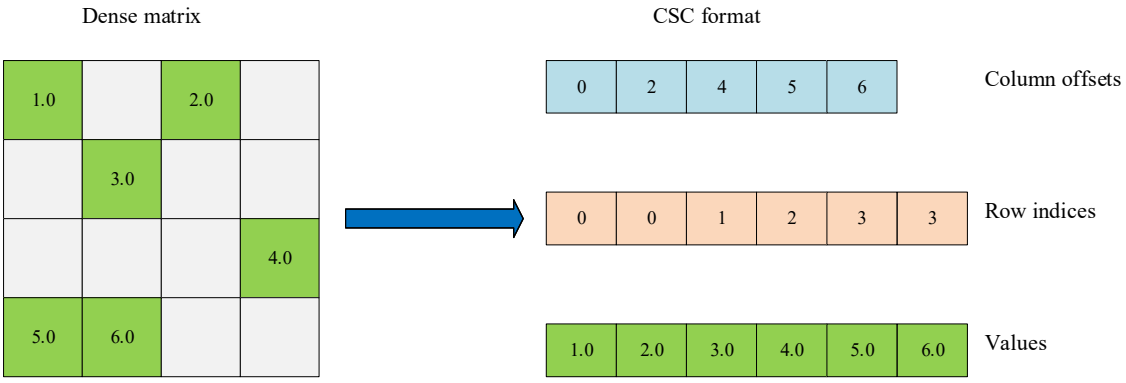

Figure 11. 4×4 sparse matrix in CSC format.

In practice, the CRS or CSC format is commonly used in PF calculation and the nonzero entries are stored in contiguous locations [109], expressed as triple vectors:

$$A_{n\times n} = \left\langle A_x \mid A_j \mid A_p \right\rangle \tag{3.61}$$

Where $A_{n\times n}$ is a square sparse matrix. A_x denotes nonzero entries, and A_j contains the column or row indices. While A_p is the row or column offsets of location. From formula (3.61), the storage space consumption is significantly lower in comparison with the dense matrix format, preventing out of memory issues on the limited space of a single GPU platform. Specifically, only $2 \times nonzeros + n + 1$ values need to be stored in sparse matrix format, rather than n^2 components [34]. However, the irregular patterns of sparse matrix increase the design challenges for parallel applications, requiring careful code tuning and the implementation of proper kernel algorithms. Fortunately, the CUDA toolkits integrate numerous high-performance computing libraries that include highly optimized algorithms and functions [33], such as CUBLAS, CUSPARSE and CUSOLVER.

CHAPTER FOUR

GPU-BASED SPARSE PF STUDIES WITH MODIFIED NEWTON'S METHOD

4.1 Introduction

The Power system is becoming larger and more complex with the development of multiple energy supplies. Efficiently Solving large-scale PF equations plays an crucial role in analyzing power systems and optimizing their performance during normal or contingencies operations. The traditional NR algorithm used for PF calculation is computationally expensive, primarily due to the need for updating Jacobian matrix in each iteration. As alternative to the repetitive updating of the Jacobian matrix, this chapter presents a GPU-based sparse modified Newton's method. Specifically, this method incorporates a fixed Jacobian matrix and leverages vectorization and parallelization techniques to accelerate PF calculation. Additionally, the research in this chapter also explores the performance of the corresponding CPU versions and a MATLAB-based library package, MATPOWER. The comparison of the implementation on the test datasets demonstrates that the GPU variant is more reliable and faster for PF calculation in the large-scale power systems.

4.2 Technical Approach

To enhance the efficiency and stability of the classic NR method, the modification based on Runge-Kutta mathematical algorithm is introduced. This modification incorporates a fixed Jacobian matrix in PF calculation, eliminating the repeated Jacobian matrix calculation and mitigating data transmission overhead between the host and device. The linear system solver is transformed into solving linear equations, instead of

directly inverting the Jacobin matrix, reducing the extra space allocation and improving the performance in PF calculation.

4.2.1Modification of NR method

To accelerate PF calculation and address divergence issues, the PF equations can be formulated as a set of non-linear equations (4.1), as illustrated in section 3.1.2.

$$f(x) = 0 \tag{4.1}$$

Equation (4.1) is expanded with first-order Taylor series at the fixed point following the NR method described above. Subsequently, equation (4.1) is transformed into a set of iterative formulas, which can be expressed as:

$$x_{n+1} = x_n + \Delta x_n \tag{4.2}$$

$$\Delta x_n = -J(x_n)^{-1} \cdot f(x_n) \tag{4.3}$$

Suppose a set of autonomous ordinary differential equations, as follows:

$$\dot{x} = g(x) \tag{4.4}$$

Where $\dot{x} = dx / dh$, equation (4.4) can be solved for x by integration:

$$\int_{x_n}^{x_{n+1}} dx = \int_{h_n}^{h_{n+1}} g(x)dh \tag{4.5}$$

According to the Euler's method, the integration in equation (4.5) can be defined as:

$$\int_{h_n}^{h_{n+1}} g(x)dh \cong \Delta h \cdot g(x_n) \tag{4.6}$$

Where $\Delta h = h_{n+1} - h_n$, substituting (4.5) into (4.4) yields:

$$x_{n+1} = x_n + \Delta x_n \tag{4.7}$$

$$\Delta x_n = \Delta h \cdot g(x_n) \tag{4.8}$$

When step size $\Delta h = 1$, equation (4.3) and (4.8) can be rewritten as:

$$g(x_n) = -J(x_n)^{-1} \cdot f(x_n) \tag{4.9}$$

Where $\dot{x} = g(x)$, the nonlinear equations (4.1) can be eventually converted to:

$$\dot{x}_n = -J(x_n)^{-1} \cdot f(x_n) \tag{4.10}$$

Equation (4.10) is usually regarded as the continuous Newton's method for PF [2]. In practice, when derivatives of the state vector, voltage magnitude and phase angle in equation (4.10), approach zero; $f(x)$ should have the solution vectors x_n as long as $J(x_n)$ is not singular. Consequently, in the modified GPU-based method, the Jacobian matrix does not have to be updated at each iterative step, resulting in a reduction of the computational overhead. The fixed Jacobian matrix J_0 is typically calculated at the beginning of programming, i.e., where $x = x_0$. Thus, equation (4.10) can be further expressed as:

$$\dot{x} = -J_0^{-1} \cdot f(x_n) \tag{4.11}$$

The off-diagonal submatrices in J_0 are omitted as zero to minimize computational overhead and saves memory space, since the zero entries are not engaged in the calculation. This simplification, defined in equation (3.20) and explained in section 3.12, can be represented in matrix format, as follows:

$$J_0 = \begin{bmatrix} J_{11} & 0 \\ 0 & J_{22} \end{bmatrix} \tag{4.12}$$

Although above approximation improves the performance, obtaining Jacobian inverse matrix J_0^{-1}, directly, is computationally expensive task during the PF calculation.

This is due to the potential fill-ins generated in the process of inverting the Jacobian matrix, J_0, resulting in increased much more extra memory allocation for storing these fill-ins, as discussed in section 3.2.1. Specifically, these steps involved in inverting the Jacobian matrix J_0 are divided into three sequential procedures:

1. Allocate the extra memory to store inverse matrix.
2. Implement direct method to decompose the original matrix and fill the nonzero value in the relative location.
3. Combine the result of step 2 to obtain the inversion of original matrix.

These steps, however, are not well-suited for execute in a vectorization and parallelization fashion, due to the dependence of components in the matrix. In fact, almost every entry is probably accessed in the memory space through introduction of numerous conditional statements such as if-statements to search for each entry's exact location in the memory. This leads to a degradation of parallel performance. Despite the GPU's significant capability for floating operation, the presence of many if-statements not only causes the divergence within the same warp, but also imposes a substantial scheduling burden on the GPU, particularly as the number of power system buses increases. To address the issues, the Jacobian inverse matrix can be transformed into solving a set of linear equations, rather than evaluating the inverse matrix, directly. Thus, equation (4.11) can be reformulated as:

$$-J_0 \dot{x}_n = f(x_n) \tag{4.13}$$

Balancing the result precision with computation speed, the second order Runge-Kutta formula is employed to equation (4.13). A set of iterative formulations are defined as:

$$-J_0 \cdot K_1 = f(x_n) \tag{4.14}$$

$$-J_0 \cdot K_2 = f(x_n + \Delta h \cdot K_1) \tag{4.15}$$

$$\Delta x_n = \frac{\Delta h}{2} \cdot (K_1 + K_2) \tag{4.16}$$

$$x_{n+1} = x_n + \Delta x_n \tag{4.17}$$

$$h_{n+1} = h_n + \Delta h \tag{4.18}$$

Where K_1 and K_2 are intermediate values for updating the state vector Δx_n. Reference [110], [111] suggest that local truncation error and global accumulation error are $O(h^{n+1})$ and $O(h^n)$, respectively. The error tends to decrease as the order increases, where the step size $\Delta h \in (0,1)$ plays a crucial role in balancing the computation accuracy and efficiency. Also, the value of Δh not only determines the iteration numbers and calculation time, but it also has associations with the convergence of PF calculation. The iteration numbers and calculation time would decrease as the Δh increases, while the optimal parameter, Δh, is fitting for one power system and probably fails to converge in the others. Weighing the calculation time and convergent rate in PF calculation, the parameter Δh is determined through validating the all the datasets. Furthermore, $\| \Delta x \|_\infty < \varepsilon$ is used as the stop criterion in this GPU-based modified method. Despite the potential increase in iterative numbers and computation overhead, due to the fixed Jacobian matrix, J_0, the high-performance floating operations on the GPU can

compensate for these disadvantages. Importantly, the fixed Jacobian matrix, J_0, significantly reduces the cost of communication between the host and device, avoiding the data transmission overhead rather than utilizing host memory as temporary communicating space in a traditional manner. While the GPU-based takes more iterations to converge, it is usually significantly faster than the classic method since the Jacobian matrices do not involve recomputing in each iteration. Furthermore, as the power mismatch equations themselves remain unchanged, the solution obtained by the GPU-based method is the same as that found with the classic NR algorithm, as illustrated in section 3.1.2.

4.2.2 Vectorization Based on the GPU

The vectorized programs can perform multiple operations concurrently via a single instruction, resulting in the acceleration for the generation of the fixed Jacobian matrix, J_0. The technique, known as coalescing on the GPU platform, lays the foundation for vectorized operation in the sparse matrix format, leveraging GPU's SIMT feature, and is integrated into cupy library. Specifically, the coalescing technique is illustrated in Figure 12. showcasing the implementation of the programs in a vectorization manner. Consider a sparse admittance matrix $D(4,4)$ and four threads in a warp. The entries in the matrix have consecutive locations in the memory and are stored in row-major flat mode. During the process, Dynamic random-access memory (DRAM) divides the sequential sixteen locations into four burst sections. Ideally, all threads in a warp would cover all entries, with just one DRAM request executed, preventing warp divergence, and saving considerable time.

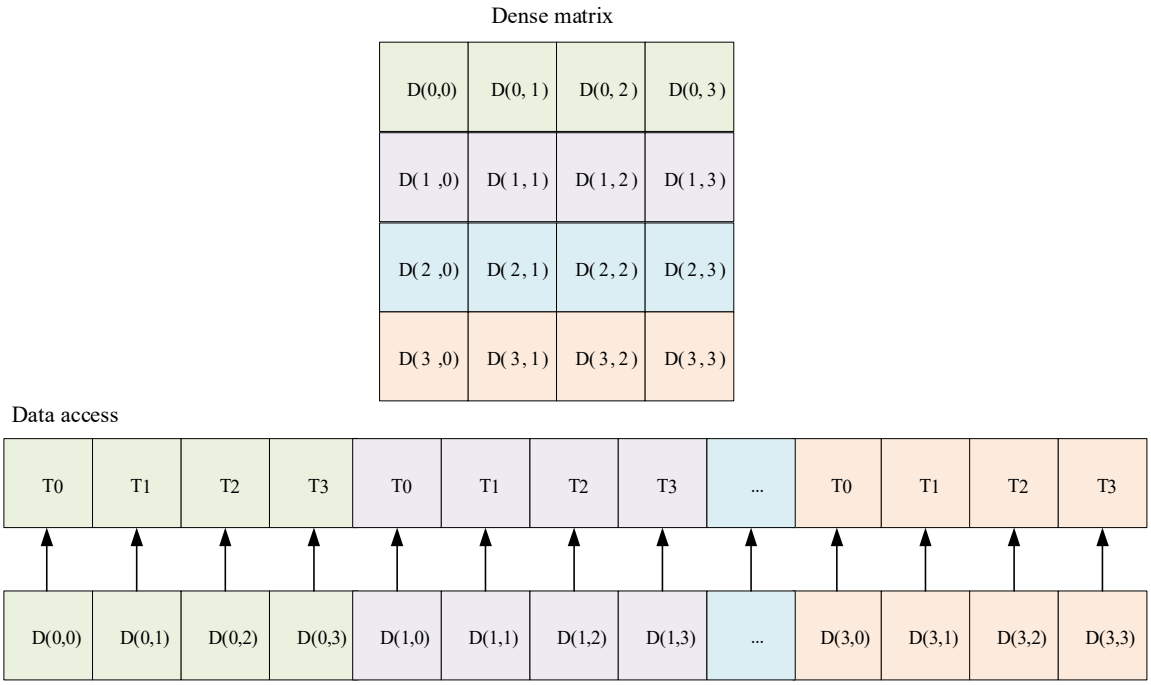

Figure 12. The mechanism of vectorization on the GPU.

In practice, sparse matrix from power system is organized in a flat manner, facilitating the parallelization and verification on the GPU. This method ensures efficient utilization of the GPU's capabilities for processing sparse matrices in PF calculation.

4.3 Implementation on the Single GPU

The heterogeneous architecture and flow chart of GPU-based method are presented in this chapter. Then, the creation of a fixed Jacobian matrix using a vectorization and parallelization technique is elaborately discussed.

4.3.1 Flowchart of Heterogeneous Program

The computing process is separated into two parts, i.e., the host and device parts. Initially, the computationally intensive tasks are offloaded to the device, while the rest are sequentially executed on the host, and the data transmission between the host and device uses PCIe bus, whose limited bandwidth determines the speed. Thus, reducing the communication overhead is critical for the single GPU platform, in comparison with a CPU platform. The time-consuming data transmission occurs only twice in the GPU-based method. Specifically, the first is referred to as loading original datasets, while the second is to receive the final results of PF calculation at the end of programming, as illustrated in Figure 13.

Furthermore, this GPU-forward backward not only alleviates computation burden on the host but also reduces cost of multiple invocations of the GPU's kernel function and the forward-backward data transmission between the host and device, maximizing the overall acceleration efficacy in PF calculation. The steps of GPU-based method are described, as follows:

1. load the datasets and initialize them.
2. Calculate partial derivatives of power injection.
3. Calculate a fixed Jacobian matrix in a vectorization manner.
4. Calculate the power mismatch through $f(x_n)$ in equation (4.13).
5. Evaluate the parameters, Δx and Δh.
6. Update the state vector x and h.
7. Check the stop criterion.
8. Repeat the procedure from 4 to 7 until convergence.

4.3.2 Jacobian Matrix Calculation

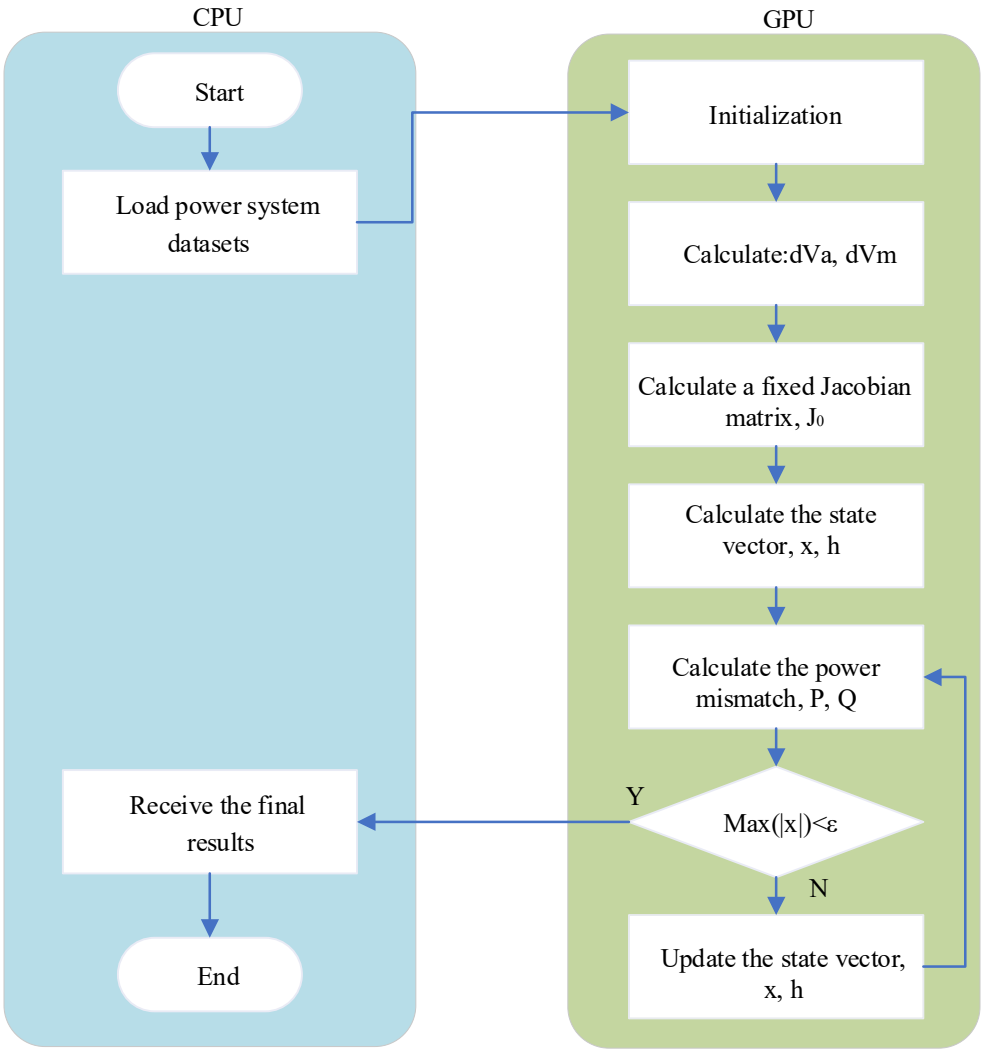

Figure 13. The flowchart of GPU-based method.

Forming a fixed Jacobian matrix consists of two parts, i.e., partial derivatives of power injection for all buses, and creation of the Jacobian matrix. Specifically, the partial derivatives, ∂V_m and ∂V_a in equation (4.22), can be reformulated into a set of vectorized formulas in the flat manner, as explained in section 4.2.2. During the partial derivatives

computation, all threads in a warp execute the same instruction in a lock-step manner. Consequently, vectors can be processed as a cohesive group enhancing the overall performance improvement, rather than being accessed individually, as illustrated in Algorithm 4.1.

$$
\begin{aligned}
&I = Ybus \cdot V \\
&V_{norm} = \frac{V}{\| V \|_2} \\
&\partial V_m = diag(V) \cdot (Ybus \cdot diag(V_{norm}))^* + diag(I)^* \cdot diag(V_{norm}) \\
&\partial V_a = 1j \cdot diag(V) \cdot (diag(I) - Ybus \cdot diag(V))^*
\end{aligned}
\tag{4.19}
$$

Where ∂V_m and ∂V_a represent the partial derivatives of power injection with respect to voltage magnitude and phase angle, respectively. The $diag()$ is denoted as an operation to generate a diagonal matrix from the vectors. Where $Ybus$ is the sparse admittance matrix and unit imaginary number, $1j$.

Algorithm 4.1: Derivatives of power injection

```
1:/* Form a sparse diagonal matrix */
2: diag_sparse(v):
3:    vec_size=size(v)
4:    vec_index=seq(size(v))
5:    vec_ptr=seq(size(v)+1)
6:    diag_mat=csr_mat(flatten(v), flatten(vec_index), flatten(vec_ptr))
7:    return diag_mat
8:/* compute the derivatives */
```

Algorithm 4.1: Derivatives of power injection

```
9: dva_dvm(Ybus, v):
10:   Ibus = Ybus*v
11:   diagV = diag_sparse(v)
12:   diagIbus= diag_sparse(Ibus)
13:   diagVnorm = diag_sparse(v/abs(v))
14:   dVm = diagV*conj(Ybus*diagVnorm)+conj(diagIbus)* diagVnorm
15:   dVa = 1j* diagV*conj(diagIbus -Ybus*diagV)
16:   return dVm, dVa
```

The bold terms in Algorithm 4.1 denote that the corresponding arithmetic operations are operated on GPU using a vectorization parallel mechanism, respectively. Similarly, the computation of a fixed Jacobian, J_0 is carried out in a vectorization manner, depending on the voltage partial derivative vectors, ∂V_m and ∂V_a. The vectorized formulas can be expressed, as follows:

$$\begin{aligned} J_{11} &= real(\partial V_a[pvpq, pvpq]) \\ J_{22} &= imag(\partial V_m[pq, pq]) \end{aligned} \tag{4.20}$$

Where $real$ and $imag$ are the operations to obtain the real and imaginary parts of vectors, respectively. where $pvpq$ and pq correspond to the index vector of PVPQ-buses and PQ-buses. Similarly, the same vectorized approach is also applicable to equation (4.20). However, vectors, $pvpq$ and pq, needs the extra memory allocated to store this index information. The components, in the allocated memory, are organized in

a consecutive flat fashion, avoiding traversing all elements in matrix with for-loops on a term-by-term way, as depicted in Algorithm 4.2. Although this approach incurs additional cost of memory space, it proves worthwhile due to the notable enhancement in computational performance.

Algorithm 4.2: A fixed Jacobian matrix generation

1:/* Compute the components of a fixed Jacobian matrix */

2: diag_spase(dVm, dVa, pv, pq):

3: pvpq=**concate**(pv, pq)

4: J_{11}=**real** (dVa[pvpq, **transpose**(pvpq)])

5: J_{22}=**imag**(dVm[pq, **transpose**(pq)])

6: Jaco=**csr_bmat([J_{11}, None], [None, J_{22}])**

16: return Jaco

Due to the vectorization operation on the GPU, the computational complexity, in Algorithm 4.1 and Algorithm 4.2, can be reduced from $O(n^2)$ to $O(n)$ as demonstrated in comparison with the method in reference [34]. Although the approach in [34] employs the CRS format to mitigate the computation overhead, it still requires traversing all nonzero elements through two for-loops to form the Jacobian matrix on the CPU platform, which causes a negative influence on performance as power systems increase rapidly.

4.4 Experiment Results and Analysis

In the chapter, we will first introduce the computing experiment setup and test cases. Then, the analysis of storage between sparse format and dense format will be discussed. Subsequently, the characteristic curve of forming a fixed Jacobian matrix will be presented. The chapter will also include a comparison of overall performance and speedup. Additionally, the analysis of stability about CPU-based QR and CPU-based LU method will be discussed. Finally, the distribution networks are conducted tests using the GPU-based method in this chapter.

4.4.1 Datasets and Test Device

The experiments in the section were carried out on a workstation, featuring an NVIDIA RTX4000 GPU and 8GB DRAM. The workstation is equipped with an Intel(R)-i9 CPU and 16 GB RAM. Also, the cupy library version used is 10.2, and the CUDA driver version aligns with cupy at 10.2. MATPOWER and Eigen are version 7.1 and 3.4, respectively. The parts of MATPOWER package have been changed into generating a fixed Jacobian matrix instead of a dynamic to maintain experiment consistency. To assure the precision and speed, the stop criterion is set to $1e-4$ for all the power system. The specifications of requirements are detailed in Table 1. The test datasets are sourced from MATPOWER, and are categorized into two groups, as outlined in Table 2 and Table 3.

Table 1. Test platform and software library.

Item	Description
CPU	Intel(R) i9 3.10GHz
Memory	16GB RAM
GPU	NVIDIA RTX400
OS	Windows 10
Complier	NVCC
Tool	PyCharm 2021
CUDA	NVIDIA GPU Computing Toolkit 10.2
Library	Eigen 3.4 MATPOWER 7.1 CUPY 10.2

Table 2. Test case in IEEE power systems.

Case	Notion
Case14	Test case form MATPOWER, IEEE system
Case57	Test case form MATPOWER, IEEE system
Case118	Test case form MATPOWER, IEEE system
Case300	Test case form MATPOWER, IEEE system

Table 3. Test case in other power systems.

Case	Notion
Case1354	Test case form MATPOWER, Pan-European system
Case2848	Test case form MATPOWER, French system
Case3120	Test case form MATPOWER, Polish system
Case6515	Test case form MATPOWER, French system
Case9241	Test case form MATPOWER, Pan-European system
Case13659	Test case form MATPOWER, Pan-European system

4.4.2 Storage Qualitative Analysis

In a power system, the majority of spaces is allocated to store the entries of the admittance matrix and a fixed Jacobian matrix during the program execution. This section presents a comparison of memory usage between dense matrix format and sparse matrix format. Since all data in the program are of single float-point data type, each element in the matrix would occupy four bytes. In fact, the data in memory are stored and organized in a flat manner. Therefore, analyzing the space usage is simplified by considering the combination of the admittance and Jacobian matrices. The memory-consumption of admittance and a fixed Jacobian matrix can be summated for the same power system buses. Table 4 provides specific power system memory-consumption for dense and sparse matrix format, respectively. From Table 4, it suggests that the memory-consumption increases rapidly in dense format.

Table 4. Memory usage (unit: bytes).

Case	Dense	Sparse	Sparsity (%)
Case14	2720	520	80.8824
Case57	57940	2316	96.0028
Case118	186740	4292	97.7016
Case300	1483600	12112	99.1836
Case1354	31284500	51612	99.8350
Case2848	145824320	111744	99.9234
Case3120	182505924	119364	99.9346
Case6515	798285800	265112	99.9668
Case9241	1502485508	415684	99.9723
Case13659	2903875624	561352	99.9807

Furthermore, Figure 14 explicitly illustrates that the sparsity of the power system increases with its scale.

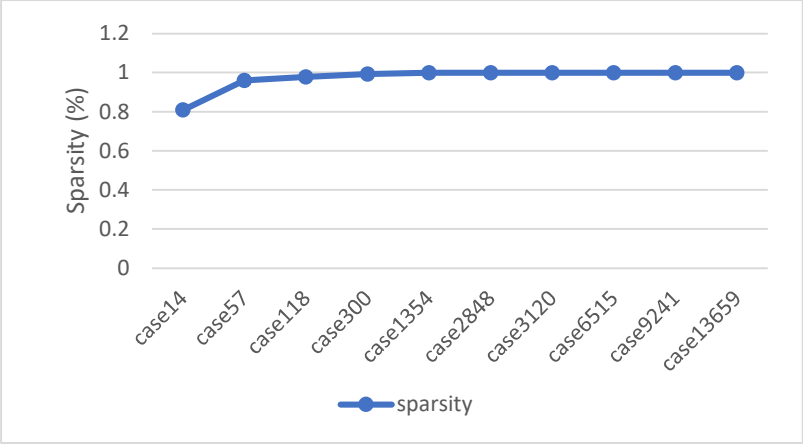

Figure 14. The sparsity of power system.

In addition, Figure 15 depicts the tendency of memory usage as the increase in power system scale, indicating that memory-consumption almost approximates exponential growth in dense format. This can potentially lead to program out-of-memory issues on the single GPU. Given the sparse nature of power system structures, many zeros not only contribute nothing to the computation, but also consume a considerable amount of space.

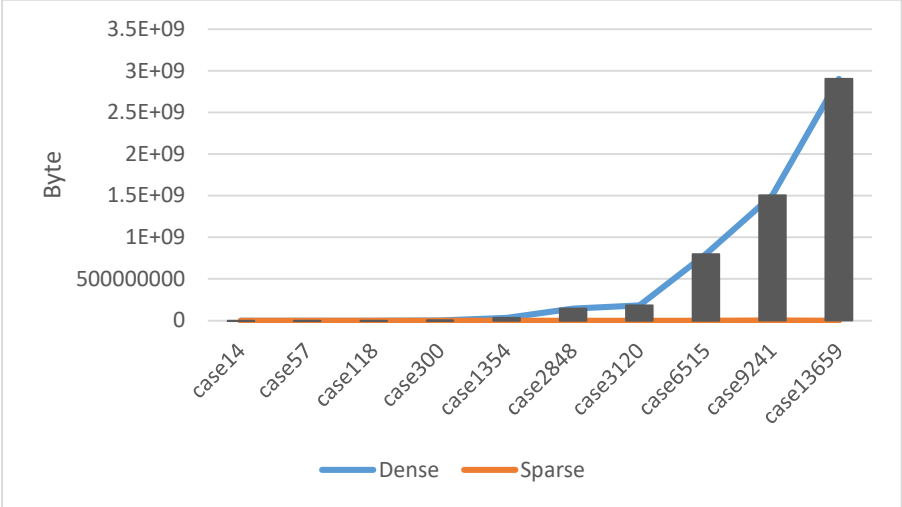

Figure 15. The memory usage in dense and sparse format.

In comparison with the dense format, memory-consumption in sparse format exhibits almost a linear increase. Consequently, the PF calculation would be accelerated in sparse format on the single GPU platform, since many zeros are not involved in the computation. Furthermore, this approach saves considerable space, providing the

opportunity to handle a larger amount of data for the larger-scale power system on the single GPU.

4.4.3 Comparison between the Classic and Modified Method

This subsection highlights the distinctions between classic and the GPU-based method, using IEEE 4-bus system as an illustrative example. The experiment suggests the classic CPU-based method outperforms the modified GPU-based method in terms of the speed, primarily due to communication overhead on the GPU. The power system is relatively small, and the limitation is the communication overhead in heterogenous structure, as explained in section 4.3.1, rather than high computing capability of the GPU. As a result, the calculation time is 0.6ms and 246.8ms for the CPU and the GPU, respectively. Specifically, Table 5 and Table 6 present the results and the power mismatch norms on the different platforms for each iteration.

Table 5. Execution result on the CPU (unit: per unit).

Iteration number	Norm (power mismatch)	Voltage magnitude ($\lvert V_m \rvert$)	Voltage angle ($\lvert V_a \rvert$)
		1	0
		0.98972	0.01921
1	0.455209	0.97871	0.03610
		1.01963	0.02764
		1	0
2	0.066826	0.98167 0.96745 1.01964	0.01795 0.03315 0.02702
		1	0

Table 5. Execution result on the CPU (unit: per unit) – Continued.

Iteration number	Norm (power mismatch)	Voltage magnitude ($\lvert V_m \rvert$)	Voltage angle ($\lvert V_a \rvert$)
		0.98199	0.01664
3	0.016381	0.96810	0.03150
		1.01964	0.02711
		1	0
		0.98230	0.01669
4	0.002514	0.96853	0.03160
		1.01964	0.02711
		1	0
		0.98229	0.01674
5	0.000614	0.96850	0.03166
		1.01964	0.02711
		1	0
		0.98228	0.01674
6	0.000094	0.96849	0.03166
		1.01964	0.02711

Table 6. Execution result on the GPU (unit: per unit).

Iteration number	Norm (power mismatch)	Voltage magnitude ($\lvert V_m \rvert$)	Voltage angle ($\lvert V_a \rvert$)
		1	0
		0.99101	0.00910
1	0.171993	0.98416	0.01693
		1.01991	0.01348
		1	0
		0.98660	0.01326
2	0.008173	0.97631	0.02478
		1.01979	0.02027
		1	0
		0.98442	0.15160
3	0.003867	0.97240	0.02844
		1.01973	0.23680
		1	0
		0.98334	0.01603
4	0.001904	0.97045	0.03015
		1.01968	0.02539
		1	0
		0.98281	0.01642
5	0.000953	0.96947	0.03096

Table 6. Execution result on the GPU (unit: per unit) – Continued.

Iteration number	Norm (power mismatch)	Voltage magnitude ($\lvert V_m \rvert$)	Voltage angle ($\lvert V_a \rvert$)
		1.01967	0.02625
		1	0
		0.98254	0.01660
6	0.000478	0.96898	0.03133
		1.01965	0.02668
		1	0
		0.98241	0.01667
7	0.000239	0.96873	0.03151
		1.01965	0.02689
		1	0
		0.98234	0.01671
8	0.000120	0.96861	0.03159
		1.01964	0.02700
		1	0
		0.98230	0.01674
9	0.000060	0.96852	0.03166
		1.01964	0.02711

Furthermore, Figures 16, 17 and 18 present the profiles of voltage magnitude and phase angle based on the classic NR, the FD, and the GPU-based method on the IEEE 14-bus system, respectively.

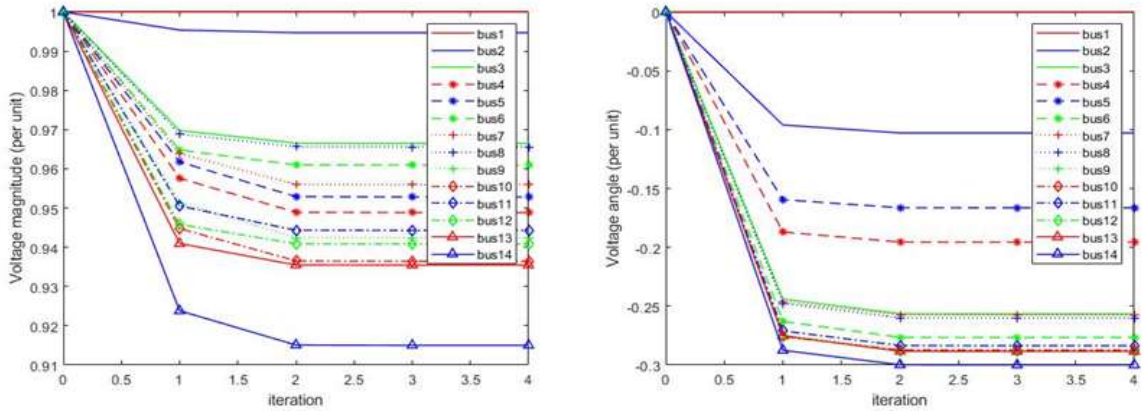

Figure 16. Voltage magnitude and angle with the NR method.

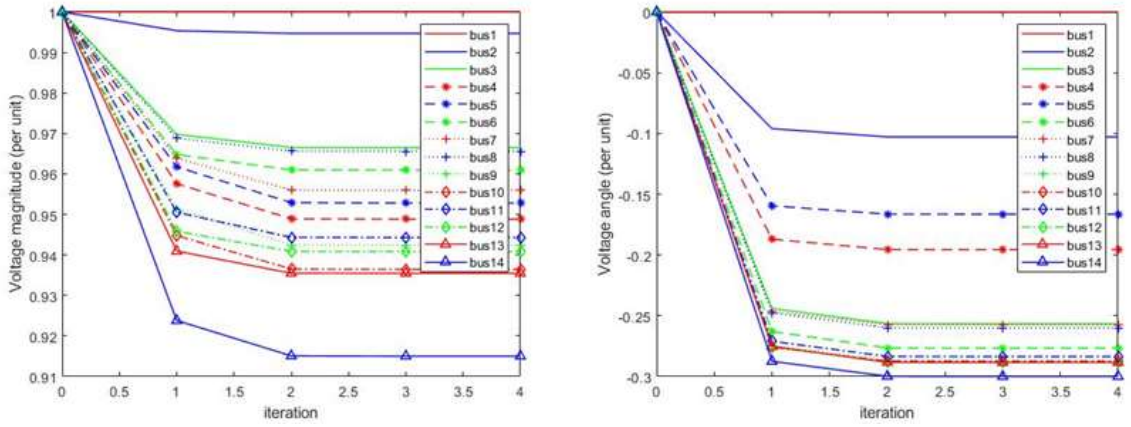

Figure 17. Voltage magnitude and angle with the FD method.

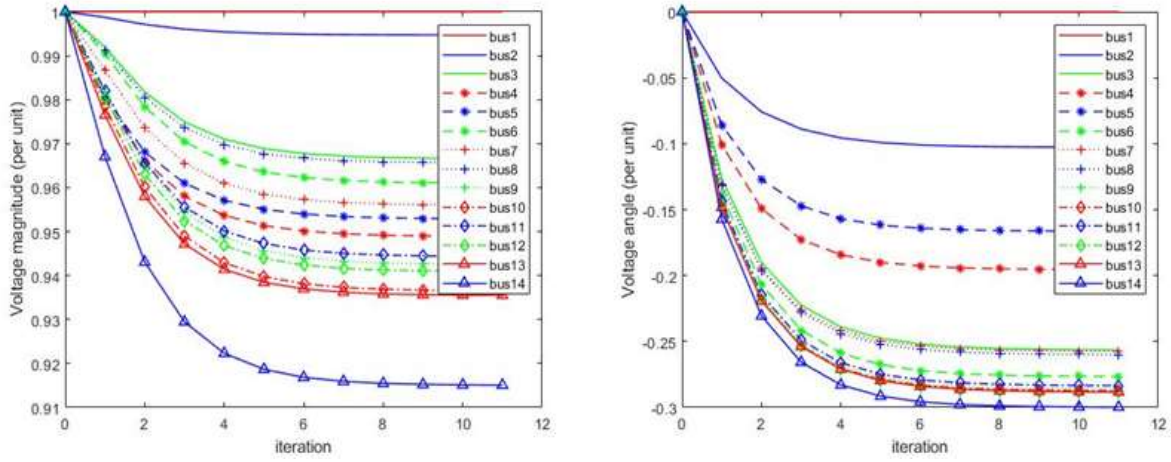

Figure 18. Voltage magnitude and angle with the GPU-based method.

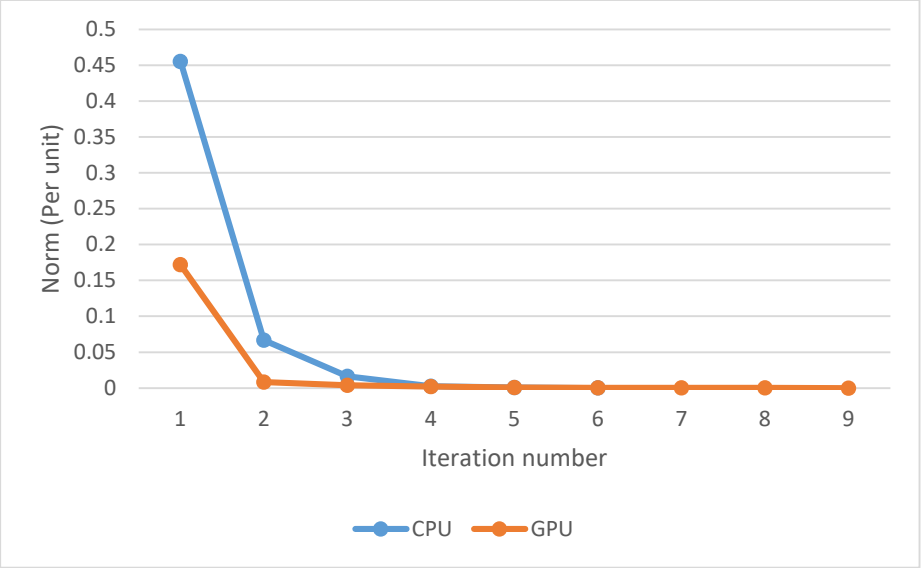

Figure 19. The convergent rate of the NR and GPU-based method.

Besides, all the Jacobian matrices are taken in CRS format in the process of PF calculation. These profiles of voltage magnitude and phase angle also prove that the classic NR method has a faster convergence rate than other methods on the small-scale power system. From Table 5 and Table 6, the GPU-based method can achieve higher precision values compared to the classic method, as indicated by the vector norm. Moreover, Figure 19 illustrates the convergent rate of the classic NR and the GPU-based method, showcasing that the classic method requires fewer iterations than the GPU-based method. Specifically, the local error of the NR method is $O(h^2)$ because the true solution of equation (4.1) is obtained by taking a finite number of terms from the Taylor series, and then it can be viewed as:

$$y_{i+1} = y_i + \dot{y}_i \cdot h + O(h^2) \tag{4.21}$$

Where $h = x_{i+1} - x_i$ and y_i represent the derivatives of PF equations. For the GPU-based method, the local error is $O(h^3)$ because equation (4.1) can be evaluated, based on the second order Runge-Kutta method, as follows:

$$y_{i+1} = y_i + \dot{y}_i \cdot h + \frac{1}{2}\ddot{y}_i \cdot h^2 + O(h^3) \tag{4.22}$$

Furthermore, equations (4.21) and (4.22) suggest that the computational complexity of the classic NR method is less than the GPU-based method. Therefore, it can achieve higher performance than the GPU-based method in the chapter, when the power system has a small scale. Although the classic NR method takes a few iterations to converge, the GPU-based method can provide performance improvement and exact results when the power system is large enough to fully drive parallel potential of the GPU.

4.4.4 Comparison of Generating Jacobian matrix

This section presents the efficiency of generating a fixed Jacobian matrix with GPU-based method and its CPU counterpart, respectively. Specifically, Figures 20 and 21 show the calculation time of creating a fixed Jacobian matrix in bar and line charts, respectively. The GPU-based method demonstrates the overall performance improvement in comparison with other counterparts for the large-scale power system. From Figure 20, the communication overhead is the major limitation of the GPU-based method for the small-scale power system, rather than computing capability of devices. However, the host (CPU) is a homogeneous structure and reduces the extra cost of data transmission, resulting in better performance than GPU-based method on the small-scale power system.

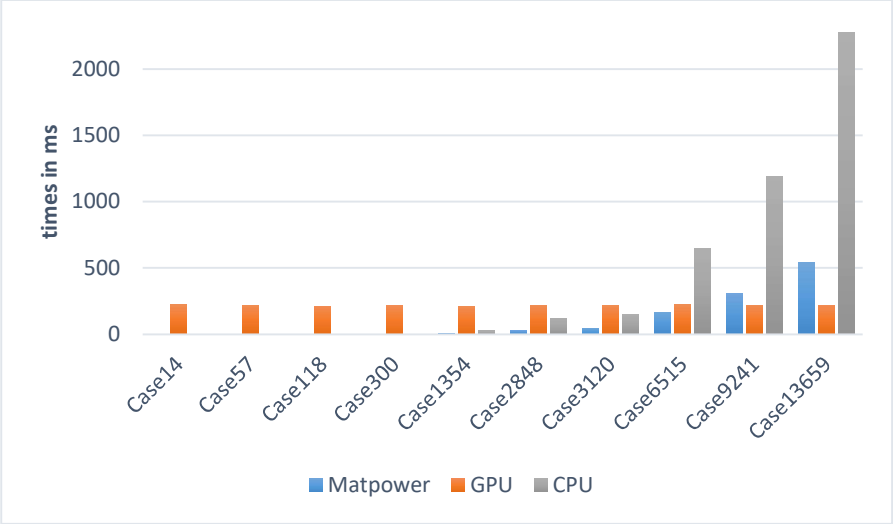

Figure 20. Comparison of creating Jacobian matrix.

Fortunately, the GPU-based method can maintain almost constant time increments even when the power system is sufficiently large, as shown in Figure 21. Conversely, the CPU-based method in Figure 21 exhibits a continuous and fast-increasing tendency as the power system scale increases.

This demonstrates that the GPU-based method can enhance performance improvement, while power systems are large enough to fully exploit the parallel potential of the GPU. Importantly, the GPU-based method not only maintains a linear time increment in the process of forming Jacobian matrix, but also facilitates acceleration on the large-scale power system.

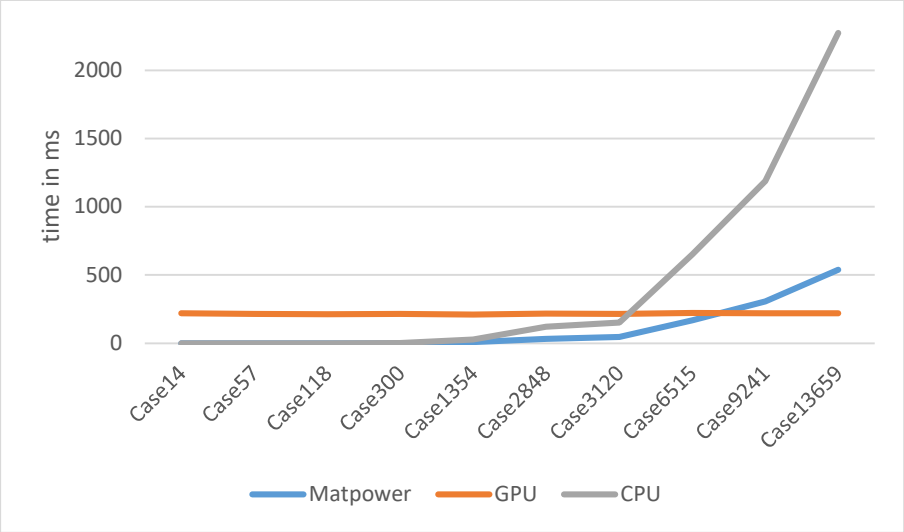

Figure 21. The time-consuming tendency.

4.5 PF Equation Solvers

This section presents the results of comparing different solvers on various hardware platforms. Specifically, the PF linear solvers are the GPU-based LU solver and its CPU counterparts, respectively. Furthermore, the comparison also includes the CPU-based QR solver and the CPU-based LU solver. Tables 7 and 8 provide a comprehensive overview of all implementation results, including a comparison of calculation time and speedup on the GPU and CPU platform, respectively.

Table 7. Execution results (unit: ms).

Case	MATPOWER	CPU-LU	CPU-QR	GPU
Case14	7.3	0.3	0.6	251.5
Case57	9.9	2.1	6.3	261
Case118	12.2	3.8	14.2	262.3
Case300	28.7	19.4	819.8	290.6
Case1354	82.9	92.7	43350	351.9
Case2848	226.6	N/A	31416.7	475.9
Case3120	274.5	358	130952.9	522.7
Case6515	523	N/A	178556.2	653.1
Case9241	1141	1946.4	173306.4	941
Case13659	1866.7	3620.7	315310.5	1343.2

From Table 7, it is evident that the GPU-based method outperforms other methods, particularly as the power system scales up to around 7000 buses. This compensates for the communication overhead, as discussed in section 4.4.3.

Table 8. Speedup result.

Case	GPU (Baseline)	CPU-LU	CPU-QR	MATPOWER
Case14	1	0.001	0.002	0.029
Case57	1	0.008	0.024	0.038
Case118	1	0.014	0.054	0.047
Case300	1	0.067	2.821	0.099
Case1354	1	0.26	123.2	0.235
Case2848	1	N/A	66.02	0.476
Case3120	1	0.68	250.5	0.525
Case6515	1	N/A	273.4	0.800
Case9241	1	2.07	184.2	1.213
Case13659	1	2.68	234.8	1.390

Table 8 demonstrates that the GPU-based approach can achieve, on average, 2.68x, 234.8x and 1.39x, respectively, compared to its sequential CPU counterparts on a power system around 13659 bus. Intuitively, Figure 22, Figure 23 and Figure 24 suggest that the GPU-based method not only maintains a high convergent rate on the large-scale power system, but also improves the stability of the ill-conditioned coefficient matrix in the PF equations, as illustrated in section 4.2.1. Also, the comparison results in Table 7 and Table 8 highlight the CPU-based LU method exhibits poor stability compared to other methods.

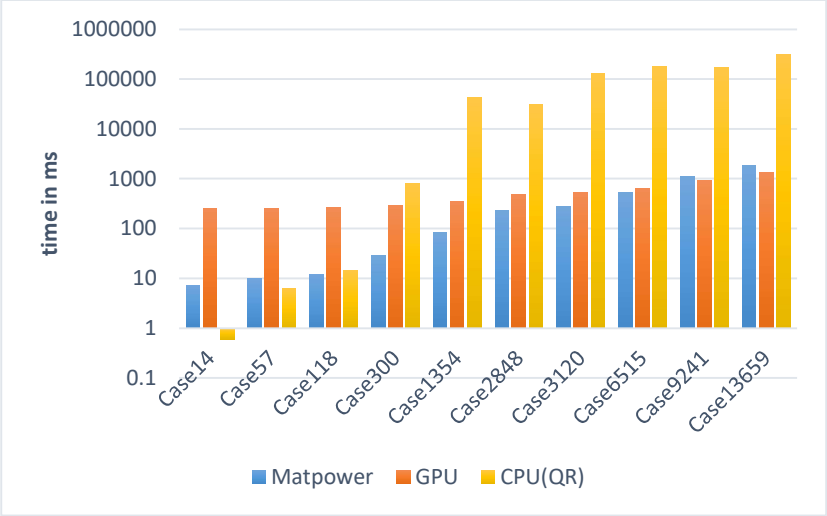

Figure 22. Execution time (MATPOWER, GPU, CPU-QR).

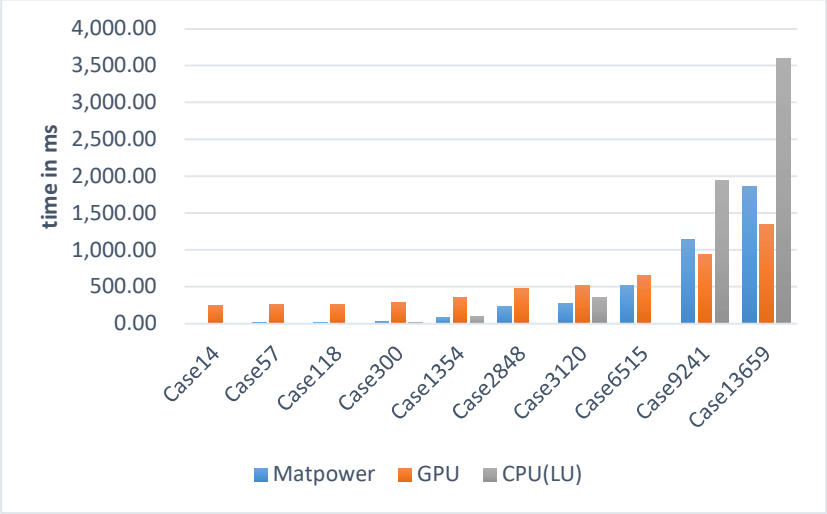

Figure 23. Execution time (MATPOWER, GPU, CPU-LU).

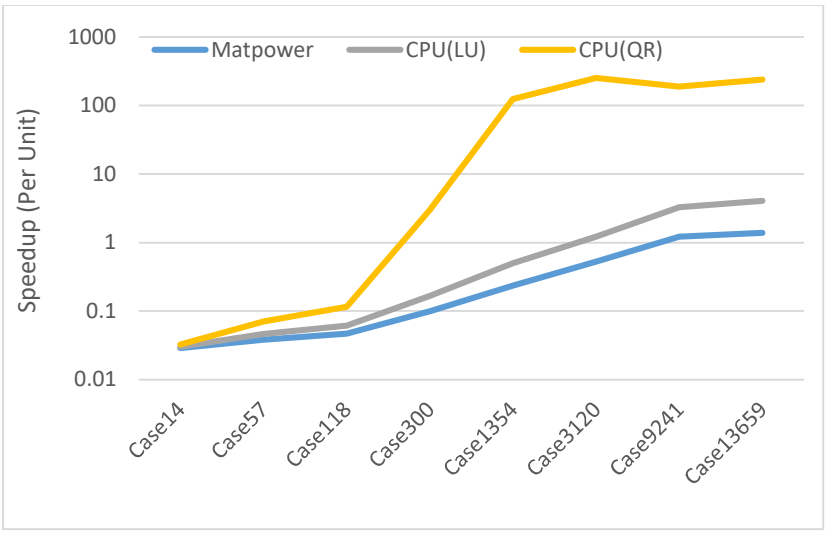

Figure 24. The comparison of speedup.

Specifically, the CPU-based LU method encounters convergence issues on bus 2848 and 6515, primarily due to the numerical instability associated with LU decomposition without pivoting, as explained in section 3.2. Furthermore, the Cholesky factorization demands a positive definite matrix, imposing a more stringent condition than LU factorization. Although the CPU-based QR solver exhibits higher numerical stability even without pivoting, the QR decomposition, usually, would incur double the operational cost compared to LU factorization in solving PF equations, as discussed in section 3.2.3.

4.5.1 Comparison in GPU-based and Iteration Method

This section compares the execution performance between the GPU-based method and the GPU-FDPF method from reference [41], respectively. Additionally, the GPU-based method is assessed on the distribution networks including IEEE 33bw, IEEE

85 and IEEE 141 bus systems. Table 9 lists speedup of GPU-based direct and iterative methods, respectively, against different CPU-based counterparts. Using the naïve CPU-FDPF and MATPOWER as baselines, Table 9 indicates that the speedup of GPU-FDPF iterative method is nearly equivalent to the GPU-based direct method. This is attributed to the GPU-based direct method using MATPOWER as a baseline, which is an optimal PF package, rather than the naïve CPU-FDPF counterpart used in GPU-FDPF. Despite the claim in reference [41] regarding the significant improvement of GPU-FDPF iterative method, it generally takes a longer execution time compared to the GPU-based direct method. Moreover, the GPU-based method demonstrates superior performance improvement over GPU-FDPF, even when the stop criterion is set at $1e-4$ in the PF calculation. The GPU-FDPF method, which focuses on preconditioning and involves transferring the power mismatch back and forth between the host and device, experiences a degradation in overall performance.

Table 9. Comparison results.

Case	GPU-FDPF (Stop criterion:0.001)	GPU-based Method (Stop criterion:0.001)	GPU-based Method (Stop criterion:0.0001)
Case1354	0.38	0.30	0.24
Case2838	0.28	0.53	0.48
Case3012	0.19	0.55	0.43
Case9241	1.04	1.54	1.21

Table 10. Comparison results.

Case	Platform	Iteration number	GPU (ms)
Case33bw	GPU	21	139.0
Case85	GPU	24	156.2
Case141	GPU	33	198.3

From the above Tables 10, the distribution networks exhibit better performance compared to the non-distribution networks. The absence of PV nodes in distribution networks results in a reduced computational cost. Consequently, the GPU-based method can fully focus on leveraging the high computational capability of the GPU, facilitating the acceleration of PF calculation through parallelization and vectorization approaches.

4.6 Summary

The GPU-based sparse modified Newton method demonstrates the remarkable acceleration for large-scale power systems, paving the way for extensive PF applications on a single-GPU platform. Specifically, the contributions of this work can be summarized as follows:

1. Instead of updating the Jacobian matrix in each iteration, a fixed Jacobian matrix is introduced using vectorization parallelization techniques. This approach alleviates the computational burden by avoiding recomputing the Jacobian matrix in each iteration. Although the fixed Jacobian matrix takes more iterations to converge, the high floating-operation capability of the GPU would compensate

for the disadvantages, ensuring a consistent increase in Jacobian matrix generation.

2. The parameter, Δh, is introduced to enhance the stability of ill-conditioned Power system. While the PF fails to converge, Δh are adjusted to approach zero, expanding the searching scope of the solution space, guaranteeing the convergence of PF calculation.
3. In contrast to other high-order numerical methods in PF calculation, the GPU-based method transforms the computation of sparse Jacobian inverse matrix into solving linear equations, eliminating the need for allocating extra memory to store submatrix and many potential fill-ins in the process of Jacobian inverse matrix.
4. The experiment results and analysis demonstrate that the GPU-based method has achieved speedups of 234.8x, 2.68x and 1.39x in comparison with CPU-based QR, CPU-based LU and MATPOWR, respectively, around 13,000 power system buses. Furthermore, the sparse matrix format not only saves much memory but also facilitates efficient processing for large-scale power systems on the single-GPU platform. The analysis of GPU-based iteration and direct methods suggest that sparse GPU-based direct methods can address the memory shortage on the single-GPU platform, yielding better performance compared to the GPU-FDPF method from reference [32]. However, the GPU-based method has limited scalability and proves challenging for implementation on the multiple GPUs, due to the irregular patterns of sparse matrix. Thus, the GPU-based method is most suitable for large-scale power systems on the single-GPU platform.

CHAPTER FIVE

PARALLEL MULTI-GPU IMPLEMENTATION OF FDPF SOLVER WITH HYBRID ARCHITECTURE

5.1 Introduction

Attaining high solution efficiency in conventional sequential computation architecture poses a significant challenge, especially with the increasing of multiple renewable energy sources (RESs). This challenge has become the bottleneck for the application in real-time grid operation, grid planning and analysis of the large-scale and complicated modern power system. Although the GPU-based method in chapter 4 has demonstrates superior performance on the single-GPU platform, it has limited scalability for the multi-GPU system, due to the irregular patterns of sparse matrices. Considering the parallelization and scalability difficulties of sparse matrix format on the muti-GPU system, dense matrix format will be employed on multi-GPU system instead of sparse matrix format. In the multi-GPU systems, memory space is not the primary limiting factor for PF calculation. Consequently, this chapter proposes a parallel multi-GPU and multi-process FD method to accelerate the PF calculation, reducing the system responsive time and ensuring real-time performance in large-scale modern power systems. Two hierarchy architectures, namely task parallelism and data parallelism, are designed to optimize the FD solver parallelization in PF calculation. Additionally, GPUDirect technology is employed to enhance the efficiency of data transmission and significantly reduce copy overhead. The proposed method in this chapter achieves an impressive

average speedup ranging from 9x ~ 33x when compared to the single GPU, demonstrating its efficacy on a sample large-scale power system.

5.2 Technical Approach

The section explains the derivation of task parallelism and data parallelism from the FD method for PF calculation. To enhance memory access speed, dense matrices are rearranged in a coalesce manner. In the process of data migration, one master and multi-salve communication manner is proposed, contributing to performance improvement.

5.2.1 Task Parallelism of the FD Method

PF is a nonlinear problem addressed through iterative numerical method to a set of linear equations $Ax = b$, as defined in section 3.1.2. These linear equations can be further split into two groups, namely independent equations (3.30) and (3.31) in section 3.1.2. While the FD method is derived from the NR method, it stands out for its algorithmic simplicity and efficiency [4]. One key advantage is that the coefficient matrices, B' and B'', are fixed and reusable, while the classic NR method must update these coefficient matrices in each decomposition process [4]. Additionally, equations (3.30) and (3.31) can be divided into two parallel tasks, and then each task is assigned to its own CPU and maintained by their own processes, respectively.

Figure 25 depicts the architecture of a parallel multi-GPU hybrid system. Each process in Figure 25 is assigned to a CPU and manages its unique task, acting as an individual computing unit to achieve high parallelism. Also, data race issues can be efficiently avoided since each process possesses its respective copies of the data, distinguishing it from the shared-memory programming model. Additionally, Unified

Virtual Addressing (UVA) maps CPU system memory and device global memory into the single virtual address space [70], as illustrated in Figure 25. The Combination of the peer-to-peer (P2P) CUD APIs with UVA allows seamless access to memory on any device and eliminates the need for explicit management of separate memory buffers.

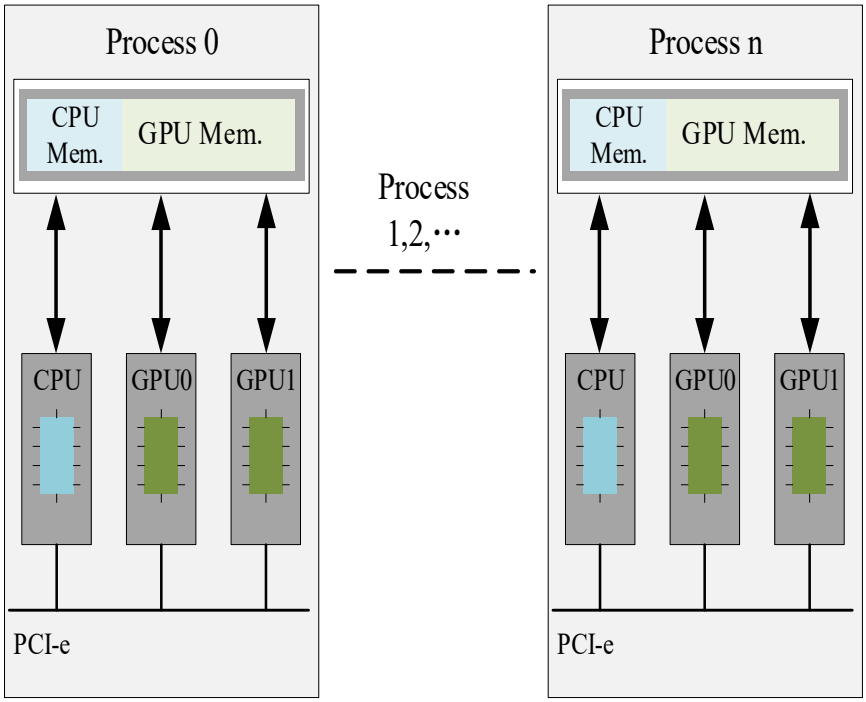

Figure 25. The multi-GPU and multi-process architecture.

In practice, two independent tasks consist of updating voltage magnitude and angle, and then they are calculated simultaneously on the independent process unit, which brings further parallelism advantages for acceleration, as following:

1. Reduce the data race by operating on the same datasets.

2. Alleviate the complexity associated with computing system scheduling.
3. Isolate the essential data.
4. Maintain the independence of the individual tasks.

5.2.2 Data Parallelism of the FD Method

The utilization of the dense matrix provides opportunity to leverage data parallelism for the coefficient matrices $\dot{B}$ and $\ddot{B}$ across multiple GPUs. These input matrices, $\dot{B}$ and $\ddot{B}$, are stored in a column-major format conforming to the Fortran and LAPACK standards [112]. As mentioned in the previous subsection, each task can be considered as solving a set of equations, $Ax = b$. Therefore, the essential bottleneck lies in optimizing the parallelization of the coefficient matrix A.

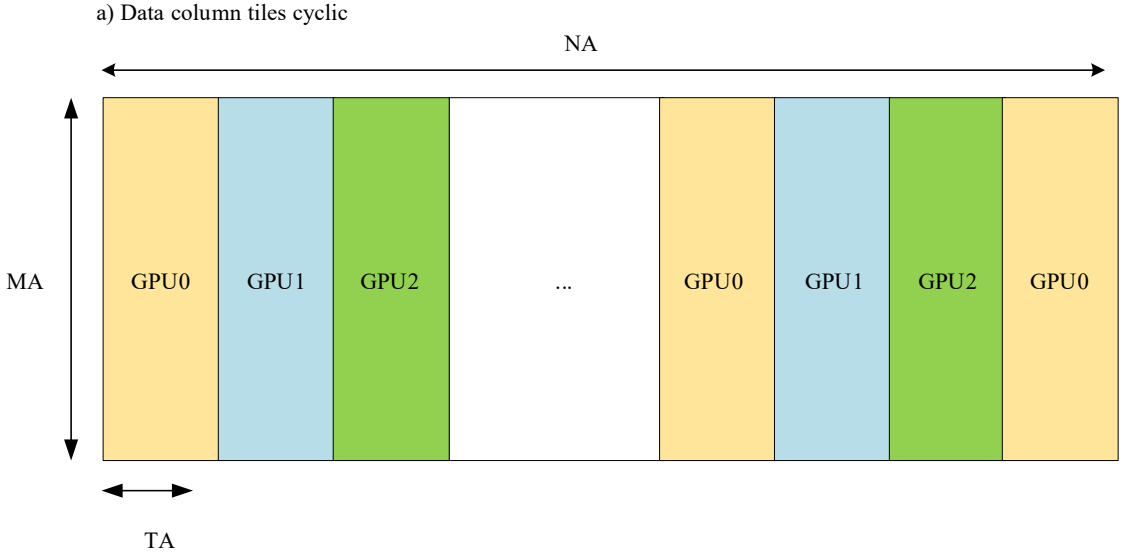

Figure 26a. Data parallelism on multiple GPUs.

b) Data in packed and unpacked format

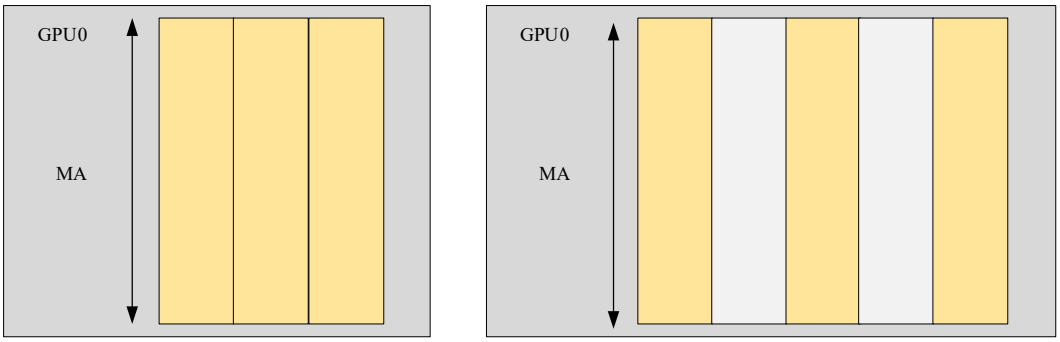

Figure 26b. Data parallelism on multiple GPUs.

Figure 26a depicts a partition of matrix A with dimension $NA \times MA$. Each block column tile has TA columns, and it is assigned to the multiple GPUs in a cyclic fashion. In conventional data partitioning strategies, the tiles of matrix A typically has two possible formats for storage on the different GPU spaces. The first format, known as the unpacked format, involves distributing data tiles across different memory blocks discretely. On the other hand, the packed format aggregates the column tiles in a continuous memory, as illustrated in Figure 26b. Specifically, the packed format tends to deliver superior performance compared to the unpacked format, benefiting from the aligned and coalesced memory access.

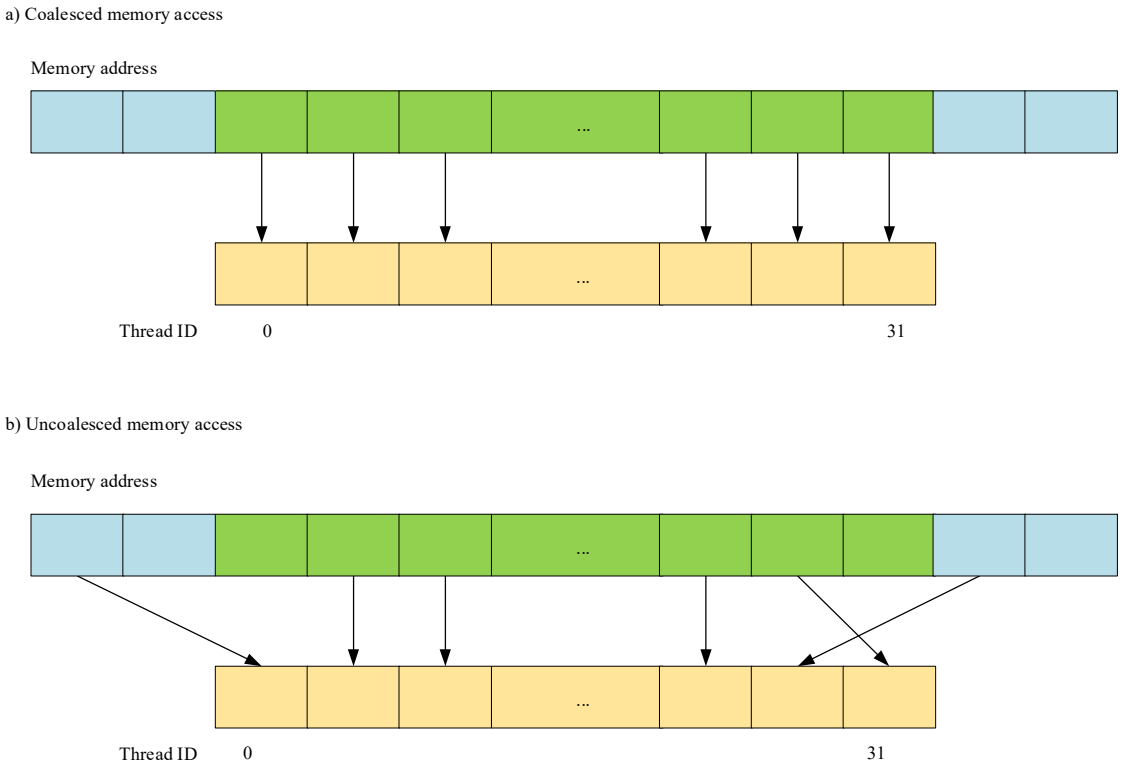

Figure 27. The comparison of coalesced and uncoalesced memory access.

Figure 27a illustrates an ideal scenario of aligned and coalesced memory access. These consecutive locations in the memory can be accessed and delivered at high speed, referred to as dynamic random-access memory (DRAM) burst. Each thread in a warp can load the maximum bytes of data, when only one DRAM request is implemented. While the column tiles scattered across global memory in an uncoalesced manner, the data transmission overhead significantly increases, due to multiple DMA requests, adversely impacting the overall performance, as illustrated in Figure 27b.

5.2.3 Zero Communication Overhead

In the heterogeneous multi-GPU system, the communication overhead is the major limited factor, rather than the computing capability. The GPUDirect technology provides direct communication between NVIDIA GPUs in the remote systems, eliminating the need for the system CPUs and the required buffer copies of data via the host memory. This reduction in the time-consuming communication between host and devices leads to a 10x improvement in performance. To address the challenge of managing and scheduling the multi-GPU systems, one master and multi-salve communicating fashion is proposed, as illustrated in Figure 28.

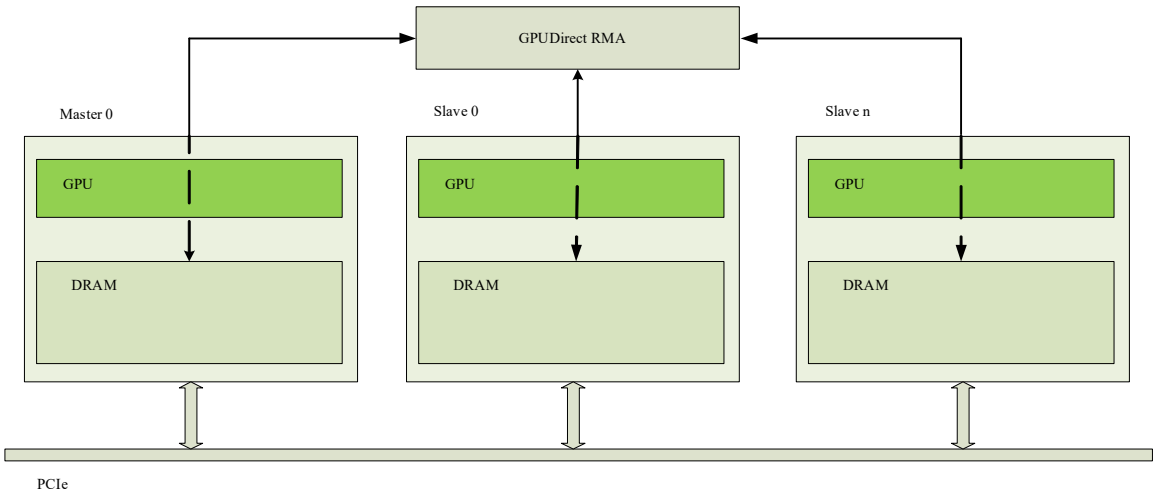

Figure 28. The communication model through GPUDirect.

In this architecture, one of the GPUs in each process unit can not only broadcast and collect results from other GPUs, but also manages the distribution of the power

mismatch as a master. Instead of taking host as temporary transmitting media, GPUs can mutually access each other's GPU memories directly within their own process units. This method eliminates the forward and backward transmission between host and devices, compared to the naïve CUDA kernel. Additionally, data migration is transparent between devices with the GPUDirect technology, resulting in overall performance improvement.

5.3 Implementation on the Hybrid Architecture

In this section, the discussion begins with an exploration of the naive CUDA kernel and multi-GPU CUDA kernel, providing detailed flow charts for their implementation.

5.3.1 The Naïve CUDA Kernel

Parallel multi-GPU method is a computational technique that leverages a heterogeneous structure, comprising a host and multiple devices, to accelerate computational tasks. Based on the computing capability of GPUs, the computationally intensive calculations are offloaded to the devices, while the host manages and schedules two tasks in a parallelism fashion. However, conventional communication between GPUs involves using the host as temporary storage media to transmit data, leading to the performance severely degradation due to the lower PCIe connection. Additionally, in the process of each task iteration, the power mismatch had to be repeatedly transmitted forward and backward between the host and devices. Therefore, the bottleneck is time-consuming data transmission rather than computation. Figure 29 depicts the naïve kernel of PF calculation in a multi-thread and multi-GPU fashion. Two threads are created on the host, which then schedule two independent tasks and manage multiple GPUs.

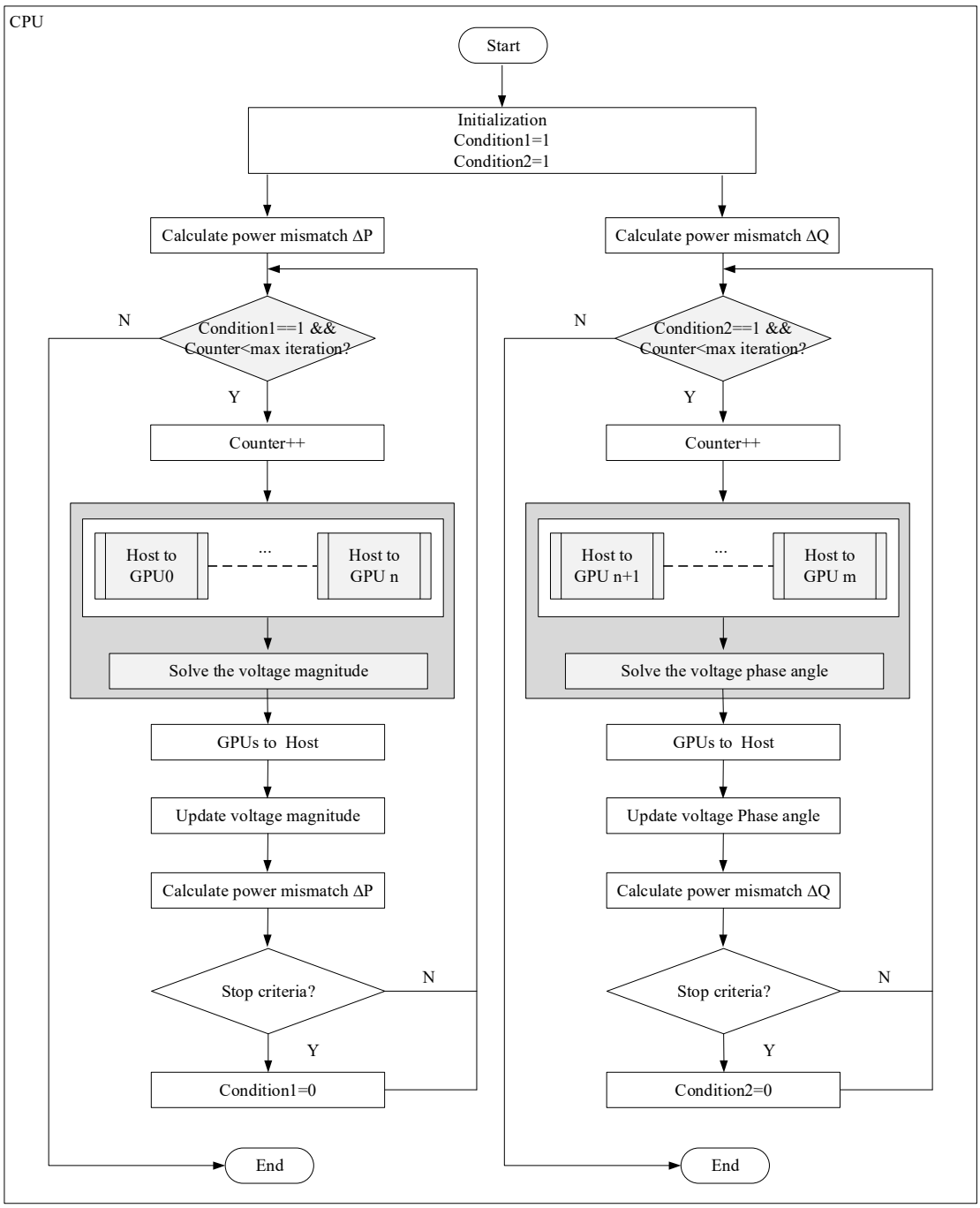

Figure 29. The flowchart of naïve CUDA kernel.

This architecture is fundamentally based on shared-memory programming model, where all data are stored and shared in the same spaces. Consequently, each thread can access them with minimal overhead, as the data is transparent to all threads on the same host. When dealing with a larger-scale power system, each thread must frequently access the same location in the shared memory to calculate the voltage magnitude and angle. Consequently, the parallel operation on the data is transformed into a time-consuming sequential process. Despite the significant boost in floating-point processing capacity provided by multiple GPUs for accelerating PF calculation [113], performance degradation can still occur due to the occurrence of data races. Furthermore, during each iteration, numerous time-consuming data transmissions occurred between the host and devices, significantly impacting overall performance.

5.3.2 The Multi-GPU CUDA Kernel

To address the previous challenges, the multi-process and multi-GPU architecture is proposed, leveraging the Matilda HPC cluster. This high-performance computing server provides flexibility in configuring hardware resources. Specifically, the parallel multi-GPU CUDA kernel is structured as a distributed architecture, as illustrated in Figure 30. In this study, we adopted a parallel execution approach where individual tasks were distributed across separate processing units and executed independently. To mitigate data race issue, each dataset was stored separately in its own memory space, as opposed to a shared memory model. In contrast to the naïve kernel, the time-consuming data transmission just occur at the beginning and ending of the program on the multi-GPU architecture, which further facilitates hiding the execution time by the scheduling of multiple process units, improving the degree of parallelization. To reduce the design

complexity and communication overload between devices, one master-and-multi-slave communicating fashion is proposed, as illustrated in section 5.2.3.

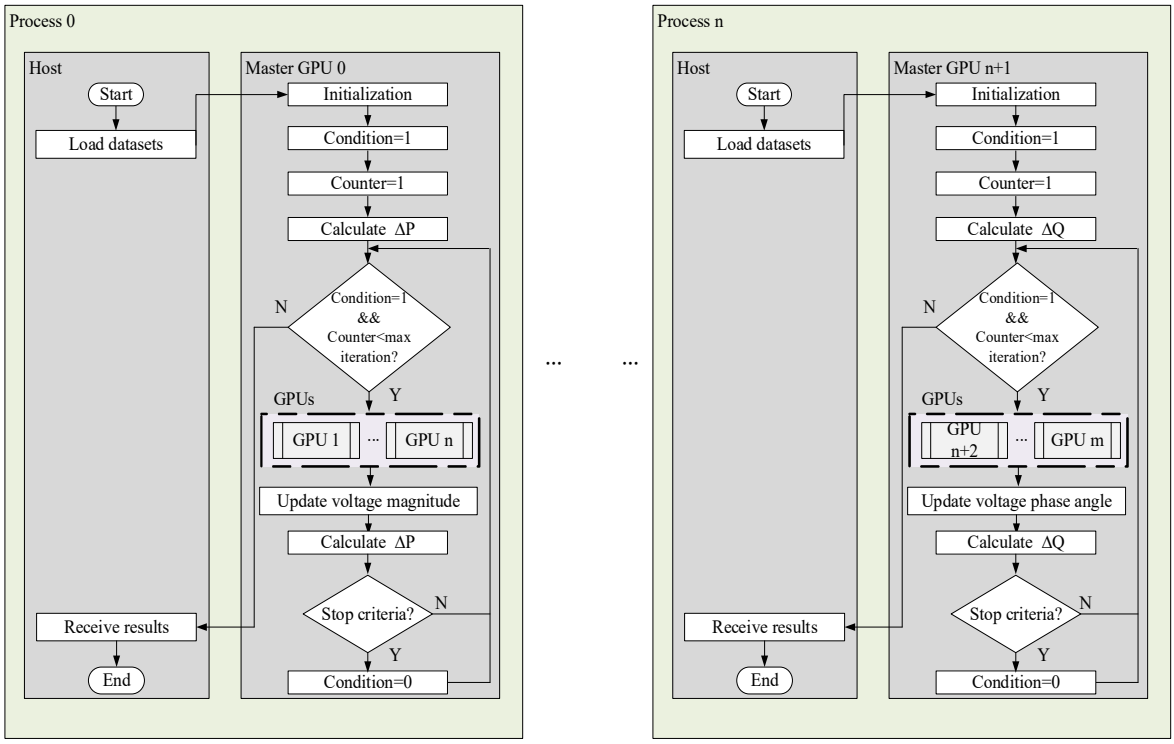

Figure 30. The flowchart of multi-process and multi-GPU CUDA kernel.

In this proposed architecture, a single GPU within each processing unit serves the master, is responsible not only for broadcasting and collecting results from the other GPUs, but also for managing the distribution of power mismatches. Instead of taking host as temporary transmitting media, GPUs can directly access each other's memories within their respective process units. This method eliminates the forward and backward

transmission between the host and devices, compared to the naive CUDA kernel. Furthermore, data migration is transparent between devices with the GPUDirect technology, resulting in performance improvement.

5.4 Experiment Results and Analysis

In this section, the configuration of computing resources and test cases will be introduced. Then, the analysis and validation of linear equation solutions in each iteration will be presented, and the implementation about speedup of different GPU numbers will be presented. Also, the analysis and comparison of speedup on this multi-GPU parallel architecture will be discussed.

5.4.1 The Configuration of Experiment Setup

The computing experiments in this paper are carried out on the Matilda HPC cluster, equipped with four NVIDIA Tesla V100 GPU nodes. The CPU on the Matilda HPC cluster is Intel(R) Xeon(R) with 2.50 GHz. To fully leverage the high floating-point computing capabilities of multiple GPUs, version 10.2 of the CUDA toolkit was utilized. The stable Red Hat Enterprise version 8.6 was selected as the operating system. To ensure precision in computing results, the CUDA implementation exclusively utilizes double-precision floating operations. Besides, the convergence tolerances are set to $1e-4$ for the entire PF calculation. Furthermore, the large-scale test cases are taken from MATPOWER package, as outlined in Table 11.

Table 11. MATPOWER test cases.

Case	Notion
Case1354	Test case form MATPOWER, Pan-European system
Case2848	Test case form MATPOWER, French system
Case3120	Test case form MATPOWER, Polish system
Case6515	Test case form MATPOWER, French system
Case9241	Test case form MATPOWER, Pan-European system
Case13659	Test case form MATPOWER, Pan-European system

5.4.2 The Validation of Linear System Solver

During PF calculation, even slight errors at each iteration can lead to significant divergence. As a result, such errors can hinder the convergence of the PF calculation. In this subsection, taking the calculation of voltage magnitude on Case 13,659 as example. The MATPOWER solution x^* will be as a point of comparison, and then the relative error can be expressed as:

$$err = \frac{\| x - x^* \|}{\| x^* \|} \tag{5.1}$$

In equation (5.1), where both x^* and x are vectors, Chebyshev norm is employed to quantify the error between vectors. Table 12 presents the absolute and relative errors at each iteration during the calculation of voltage magnitude, as follows:

Table 12. Validation of multi-GPU linear solver.

Iteration	Absolute Error (1e-12)	Relative Error (1e-7)
1	0.6502	0.0000
2	0.2773	0.0002
3	0.4391	0.0027
4	0.4777	0.0258
5	0.2686	0.1278
6	0.3448	0.9266

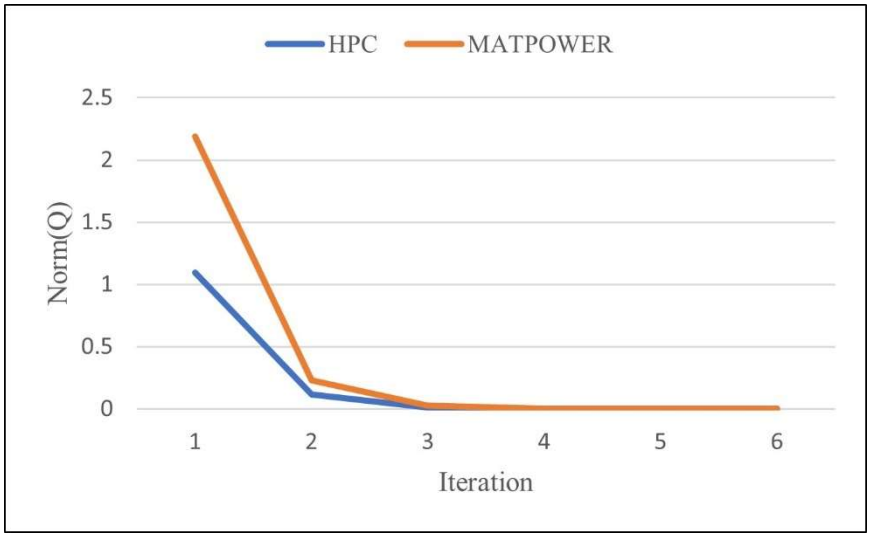

Figure 31. The convergence comparison of power mismatch Q on bus 13,659.

From Table 12, both errors are almost close to zero, indicating that the solutions obtained through multi-GPU approach satisfy the precision requirements and guarantee the convergence of PF calculation. Figure 31 presents the iteration process of the norm Q

on Case 13,659. The multi-GPU solution tracks are almost as precise as that of MATPOWER. While two tracks in Figure 31 have a slight gap between the two tracks at the beginning of the iteration, both tracks ultimately converge with the same numbers of iterations. It turns out the inner multi-GPU solver has no adverse impact on outer iteration of PF calculation. Therefore, it is demonstrated that the multi-GPU solver is robust, maintaining similar precision and convergence rates.

5.4.3 The Implementation Results

For comparison purposes, the multi-GPU method was implemented on the Matilda HPC cluster using varying numbers of GPUs. Meanwhile, the single GPU still takes the naive kernel in the process of PF calculation, necessitating the use of host memory as temporary storage for data transmission under the same multi-GPU architecture. Table 13 presents implementation results of the multi-GPU method with different GPU numbers, including a comparison of execution time and speedup, respectively. From Table 13, it is obvious that the multi-GPU approach outperformances the single GPU, showcasing significant performance improvements with an increasing number of GPUs for the large-scale power system.

Table 13. Execution Results on Multiple GPUs.

Case	GPU number	Time(s)	Speedup
Case 1354	1	0.653791	1
	2	0.472641	1.383272
	3	0.195968	3.336213
	4	0.066216	9.873611
Case 2848	1	8.962783	1
	2	7.551165	1.186864
	3	1.505995	5.951403
	4	0.538820	16.63409
Case 3120	1	12.44643	1
	2	10.68490	1.164862
	3	1.774424	7.014349
	4	0.733948	16.95819
Case 6515	1	43.25135	1
	2	38.63750	1.119414
	3	5.551443	7.791010
	4	1.858600	23.27093
Case 9241	1	132.2529	1
	2	91.82130	1.440329
	3	39.76525	3.325841
	4	4.005890	33.01461
Case 13,659	1	320.7924	1
	2	254.4930	1.260516
	3	65.48202	4.898939
	4	10.490380	30.57967

The speedup observed in Case 9241 is faster than Case 13,659, as indicated in Table 13. This discrepancy can be attributed to the impact of matrix structure on computing performance, as depicted in Figure 32. Further discussion on this aspect will be provided in the subsequent section.

The curves in Figure 33 depict the correlation between the number of GPUs and speedup across the various power systems. The performance exhibits a nearly consistent linear increment under three GPUs, while the speedup has a substantial improvement when utilizing four GPUs.

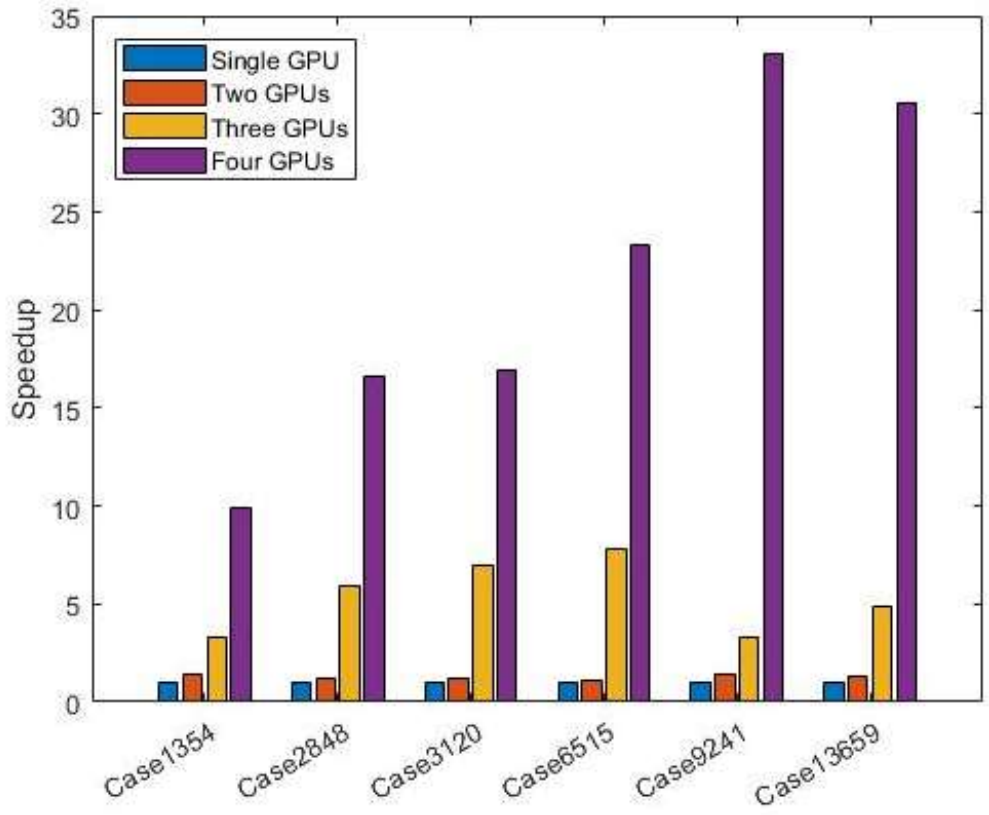

Figure 32. Speedup on different GPU numbers.

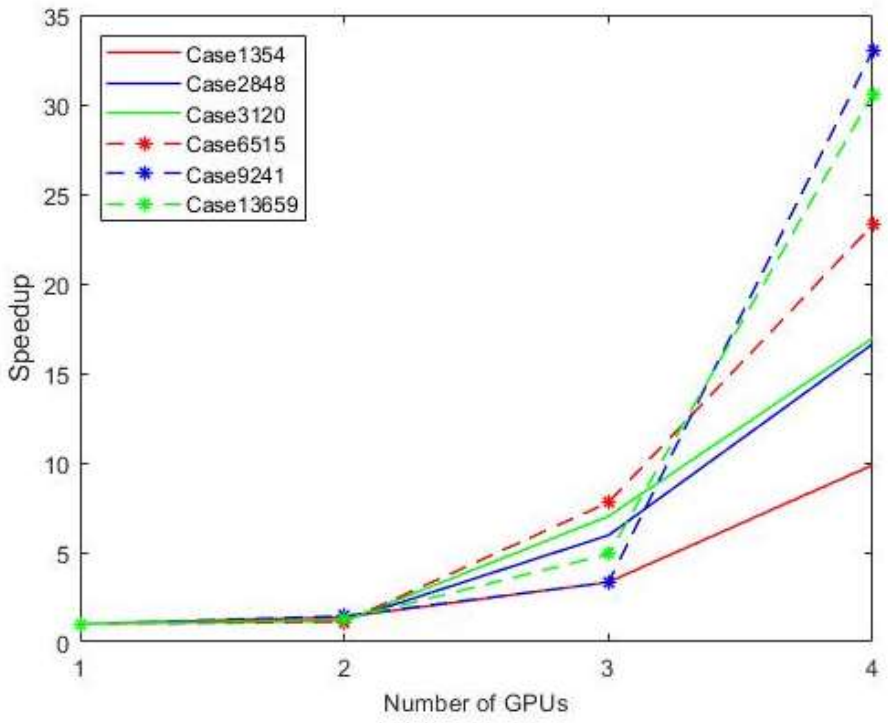

Figure 33. Speedup on different GPU numbers.

5.4.4 The Analysis and Discussion

In this section, the implementation results are discussed and analyzed. Due to the dense matrix format in this method, the CPU-GPU sparse based counterpart is used to validate the accelerating computing efficacy in comparison with the multi-GPU based method.

5.4.4.1 Analysis of Implementation Results

From the above results, the speedup of Case 9241 is faster than that of Case 13,659. The discrepancy can be attributed to two distinct factors, i.e., the matrix structure and the scheduling mechanism of the GPU, respectively. In certain instances, the

coefficient matrices of the power system are in an ill-conditioned state, leading to the divergence of PF calculation.

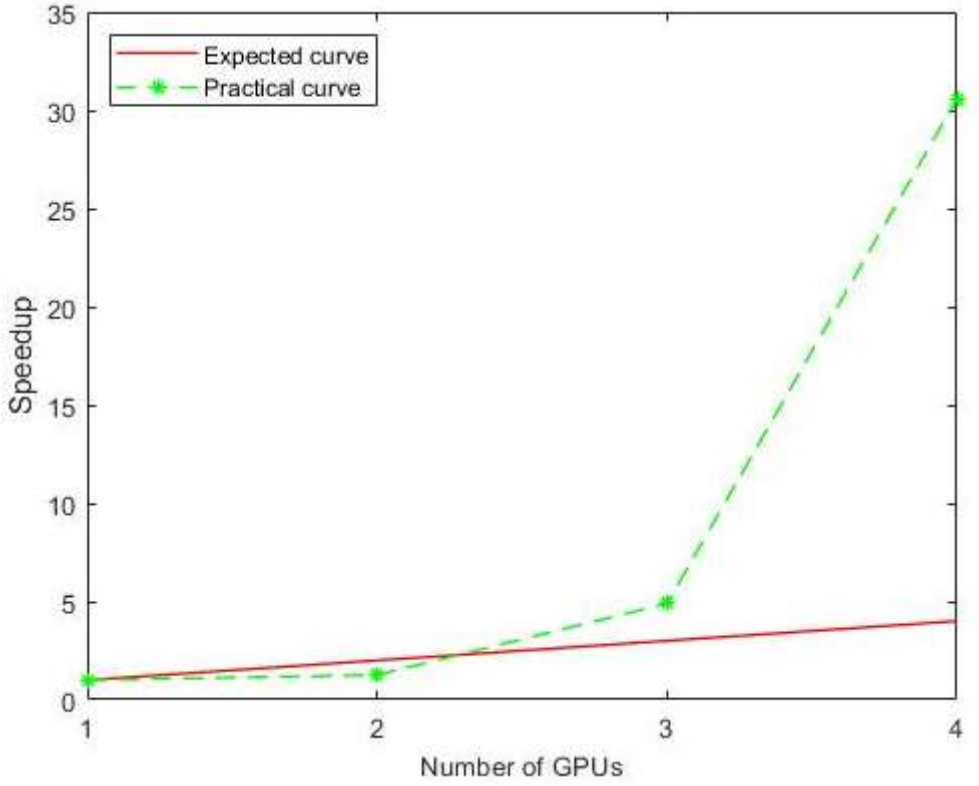

Figure 34. Expected and practical curve of speedup on Case 13,659.

Although Case 13,659 does not experience severe divergence, it still takes more iterations for updating the power mismatch during the PF calculation process to convergence. This feature of matrices will degrade the performance of the system on Case 13,659. From the view of schedule of GPU, each SM partitions the thread blocks

assigned to it into 32-thread warps which are subsequently scheduled for execution on available hardware resources [70]. All threads in a warp executed the same instruction simultaneously, following a parallel fashion.

In the case of Case 9241, the time-consuming task is the calculation of voltage angles. The size of the matrix in the task is 9240 × 9240, which is a multiple size of a warp. Therefore, the matrix transmission and loading can be efficiently performed without warp divergence. On the contrary, in the process of Case 13,659, the warp divergence occurs, converting the parallel computing into a sequential manner. As a result, the speedup of Case 13,659 is slightly slower than that of Case 9241, despite the larger size of the former. Therefore, the conclusion is that not every matrix can obtain the maximum optimization with the multi-GPU method.

Figures 33 and 34 illustrate that the significant improvement in speedup when the number of GPUs is four. Furthermore, the practical speedup curve in Figure 34 exhibits a steeper slope than that of the expected linear speedup curve on for Case 13,659, as outlined in the statistical fashion [114]. This result is primarily attributed to the GPU assignment strategy. When the PF calculation is allocated on the single GPU, parallel tasks must utilize hardware resource of the single GPU in a shared fashion and, ultimately, executed in a sequential manner, accessing the GPU's resources.

Furthermore, instead of taking GPU P2P transmission, data migration on the single GPU platform necessitates the use of host memory as temporary media for communication between the host and devices, causing the performance decline due to slower PCIe bandwidth. Although the speedup of calculation is better with two and three GPUs, the acceleration efficacy is limited. With two GPUs, each GPU is assigned to its

own process unit, and the data transmission cannot fully utilize GPU P2P transmission, due to only one GPU on its own process unit. Therefore, the limited improvement can be observed in Figure 34, despite the high floating operation capability of the GPU. Nevertheless, the practical speedup remains below the expected linear speedup. When the number of GPUs is three, two GPUs are allocated to one process unit, while the other process unit has only one GPU. The task involving two GPUs performs better, while the other task on the single GPU still operates inefficiently. Hence, the practical speedup is slightly higher than that of expected linear speedup in Figure 34, due to the imbalanced commutating distribution among GPUs. When the number of GPUs is equal or greater than two on each process unit, each task can exploit the features of GPUDirect to communicate in a P2P fashion between devices, resulting in great improvement at four GPUs in Figure 34.

In summary, each process unit with two or more GPUs can directly transmit data from device to device, eliminating transmission overhead between the host and devices, and significantly improving the overall performance.

5.4.4.2 Discussion of the Matrix Format

Although the sparse format effectively reduces the computing overhead due to the nature of power system, it is still hard to exploit the parallelism of GPU because of irregular storage. Furthermore, extending to additional GPUs becomes difficult due to its inherent structure and traversal complexity of sparse formats. A recent reference [36] adopts a sparse format and employs a multithread and multi-GPU architecture, distributing PF calculation across two GPUs and achieving the optimal performance. In [36], the speedup is nearly twice that of a single GPU platform. In contrast, the multi-

process and multi-GPU method in this paper demonstrates a remarkable improvement, achieving 9x ~ 33x under the same comparison, as indicated in Table 14.

Table 14. Execution Results on Multiple GPUs.

Case	GPU number per task	Method in [36]		Multi-GPU method	
		Time(s)	Speedup	Time(s)	Speedup
Case 1354	2	0.342000	1.868421	0.066216	9.873611
Case 9241	2	6.525000	1.633410	4.005890	33.01461

The aforementioned results highlight the efficacy of the multi-process and multi-GPU method in maintaining high computational efficiency, particularly when dealing with matrices in dense formats. The method outlined in this paper successfully mitigates data race issues and enhances scalability by reducing interdependencies between tasks and isolating memory within the process unit, resulting in an overall improvement in performance.

5.5 Summary

In this chapter, a multi-GPU method for parallel acceleration of the PF calculation is proposed and demonstrated achieving optimal parallelization. The method utilizes a two-level hierarchy architecture, namely task and data parallelism, and simplifies the overall complexity. On the first level, each task is executed simultaneously and scheduled by its own process unit. On the second level, the computationally intensive data from

each task is further assigned to multiple GPUs, which are managed by their corresponding process units, improving data parallelization. To prevent data races, storage spaces on different process units are isolated, enabling each computing unit to function as a minicomputer, and accelerating PF calculation. This paper introduces GPUDirect to reduce communication overhead between hosts and devices. Data migration becomes nearly transparent for devices on the same process unit, eliminating time-consuming data transmission between the host and devices. In addition, a communication model with one master and multiple slaves for GPUs is proposed to broadcast and collect data in a coalesced manner, improving the efficiency of data transmission. The results presented in this paper demonstrate the outstanding performance of the multi-GPU method, even when coefficient matrices adopt a dense format. This method delivers an overall performance improvement for large-scale systems. Although this method may consume more space than the sparse matrix method, the acceleration efficiency compensates for the space shortage. Therefore, this method can be beneficial for the development of cloud storage. As a future research path, the hybrid storage format will be considered, combing the advantages of the dense and sparse matrix formats, and further exploiting the parallelism of PF calculation.

CHAPTER SIX

SUMMARY

6.1 Conclusion

The real-time PF calculation is critical for the power system analysis, planning and operation, as the development of multiple energy supplies. However, the power system, modeled as a nonlinear system, needs to be transformed into a series of linear equations for solving. The dissertation exploits the GPU-based high-performance method for acceleration of PF calculation on both single-GPU and multi-GPU platforms, respectively.

This works presents GPU-based sparse modified method, building upon the NR method, but with a focus on acceleration and stability for the ill-conditioned matrices. Leveraging the vectorization and parallelization of GPUs, a fixed Jacobian matrix is created in a vectorization manner, maintaining a constant time increase even for the large-scale power systems. Instead of obtaining the coefficient matrix inverse, the linear system is directly solved using the high-order numerical method. Although this numerical method requires more iterations to converge, the high-performance floating operation capability of the GPU compensates for the disadvantages, resulting in superior outperformance in PF calculation. Due to taking the sparse format matrix on the single-GPU, the proposed GPU-based sparse method can handle the large-scale power systems, avoiding the out-of-memory issues on the single-GPU.

The GPU-based sparse method, however, has a limited scalability, causing parallelization difficulties for the applications on multi-GPU system. In multi-GPU

system, the memory space is not a major limiting factor for PF calculation. To address this, and to maintain speedup, improve the scalability, and simplify the PF calculation, the parallel multi-GPU hybrid method, based on dense matrix format, is firstly proposed to exploit the GPU parallelization for accelerating the PF calculation. Specifically, two hierarchy architectures, task parallelism and data parallelism, are designed to optimize the FD solver parallelization. Importantly, each process has its own separate space to isolate the critical data, preventing the data races inherent in a shared-memory manner. The communication overhead is a major limiting factor, rather than computing capability in the multi-GPU system. Thus, dense matrices are rearranged and distributed across multi-GPU in a coalesced fashion for memory access acceleration. GPUDirect technology is employed to enhance efficiency of data transmission, significantly reducing copy overhead, and offering one master and multi-salve communicating manner to manage the other GPUs, collect data, and broadcast the results. The parallel multi-GPU based method demonstrates an overall performance improvement for a large-scale power system, achieving, on average, 9x ~ 33x speedup in comparison to the naïve single-GPU platform.

6.2 Future Work

Although the multi-GPU based method has achieved high parallelization scalability and remarkable speedups for PF acceleration even if compared to other CPU-GPU based sparse approaches. Nevertheless, this method may be slower than the single-GPU platform, particularly when dealing with numerous entries in computations. This scenario can increase the computation overhead, especially as the power systems significantly extend in scale in the future. Thus, based on the advantages of the proposed

method in the dissertation, the parallel sparse multi-node method will be proposed to further enhance the PF calculation optimization. To refine the granularity of parallelization in PF calculation, the coefficient matrix in the task parallelism will be factorized into sparse triangular matrix and distributed across the different GPU nodes. The multi-node partitioned global address space programming model is designed to facilitate efficient fine-grained communication and substantially reduce synchronization overhead, as depicted in Figure 35.

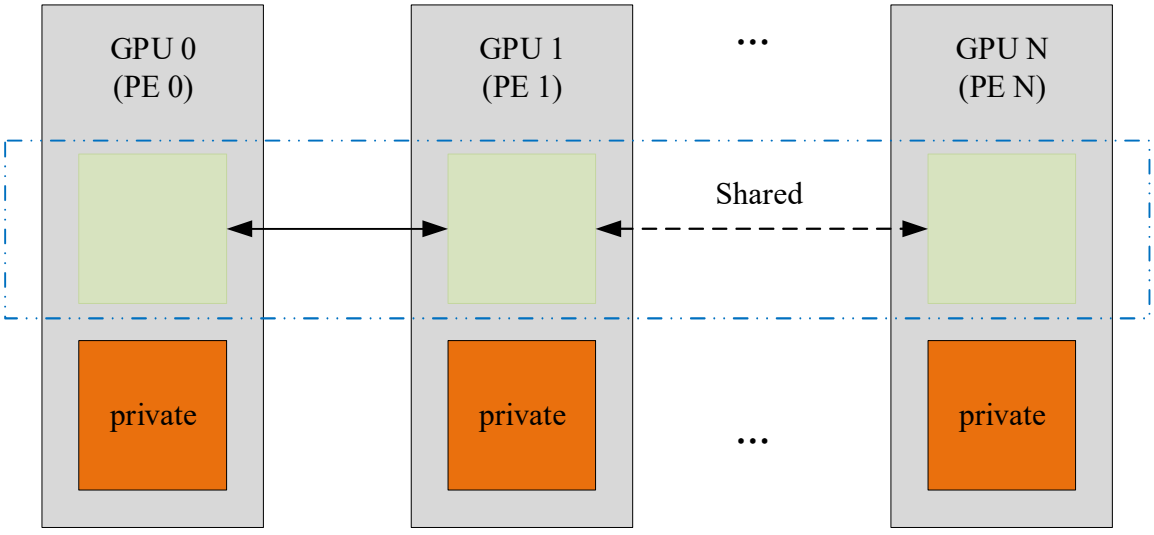

Figure 35. The multi-node partitioned global address space programming model.

In each GPU node, the memory is divided into shared and private spaces, with data in shared memory transparent and accessible to all GPUs, minimizing the expense with data migration. Additionally, private space isolates the critical elements, preserving

the dependence and security of parallel tasks. Consequently, the primary focus for future improvements lies in significantly reducing communication overhead and enhancing sparse matrix format parallelization in the multi-node systems.

REFERENCES

[1] Yoon, D.-H., Han, Y, "Parallel Power Flow Computation Trends and Applications: A Review Focusing on GPU," *Energies,* 2020, pp. 2147.

[2] M. Wang, Y. Xia, Y.Chen, and S. Huang, "GPU-Based Power Flow analysis with continuous Newton's method," *in 2017 IEEE Conference on Energy Internet and EnergySystem Integration (E12)*, Nov. 2017, pp. 1-5.

[3] L. Zeng, S. G. Alawneh, and S. A. Arefifar, "GPU-Based Sparse Power Flow Studies with Modified Newton's Method," *IEEE Access*, Nov. 2021, pp. 153226–153239.

[4] Wang, X., Song, Y., Irving, M, *modern power systems analysis.* Springer: New York, NY, USA, 2008.

[5] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Boston, MA: PWS, 2004.

[6] K. H. Jin, M. T. McCann, E. Froustey and M. Unser," Deep Convolutional Neural Network for Inverse Problems in Imaging," *in IEEE Transactions on Image Processing*, Sept. 2017, vol. 26, no. 9, pp. 4509-4522.

[7] J. W. H. Liu, "The multifrontal method for sparse matrix solution: Theory and practice," *SIAM Rev*., Mar. 1992, vol. 34, no. 1, pp. 82-109.

[8] Dense Linear Algebra on GPUs. Accessed: Mar. 2018. [Online]. Available: https:// developer.nvidia.com/cublas.

[9] S. Jin et al," Understanding GPU-Based Lossy Compression for Extreme-Scale Cosmological Simulations," 2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS), May. 2020, pp. 105-115.

[10] L. Lai, Q. Zhang, H. Tsai and W. -T. Cheng," GPU-based Hybrid Parallel Logic Simulation for Scan Patterns," *2020 IEEE International Test Conference in Asia (ITC-Asia),* Sept. 2020, pp. 118-123.

[11] W.-K. Lee, R. C.-W. Phan, G.-S. Poh, and B.-M. Goi, "SearchaStore: Fast and secure searchable cloud services," *Cluster Comput*., June. 2018, pp. 1-14.

[12] M. Sabbagh, Y. Fei and D. Kaeli," A Novel GPU Overdrive Fault Attack," *2020 57th ACM/IEEE Design Automation Conference (DAC),* Jul. 2020, pp. 1-6.

[13] V. M. van Santen, F. L. F. Diep, J. Henkel and H. Amrouch," Massively Parallel Circuit Setup in GPU-SPICE," *in IEEE Transactions on Computers*, Oct. 2020.

[14] X. Liu, H. Yu and S. X. -. Tan," A GPU-Accelerated Parallel Shooting Algorithm for Analysis of Radio Frequency and Microwave Integrated Circuits," *in IEEE Transactions on Very Large-Scale Integration (VLSI) Systems*, Mar. 2015, pp. 480-492.

[15] C. Zhao, Z. Zhou and D. Wu," Empyrean ALPS-GT: GPU-accelerated Analog Circuit Simulation," *2020 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, Nov. 2020, pp. 1-3.

[16] B. D. de Vos, J. M. Wolterink, P. A. de Jong, T. Leiner, M. A. Viergever and I. Iˇsgum, "ConvNet-Based Localization of Anatomical Structures in 3-D Medical Images*," in IEEE Transactions on Medical Imaging*, Feb. 2017, pp. 1470-148.

[17] L. Rakai and W. Rosehart," GPU-Accelerated Solutions to Optimal Power Flow Problems," *2014 47th Hawaii International Conference on System Science*s, Jan. 2014, pp.2511-2516.

[18] Falcão, D.M, "High performance computing in power system applications." *In International Conference on Vector and Parallel Processing*, Sept.1996, pp. 1-23.

[19] Ramesh, V, "On distributed computing for on-line power system applications." *Int. J. Electr. Power Energy Syst*. Nov1996, pp. 527–533.

[20] Baldick, R., Kim, B.H., Chase, C., Luo, Y., "A fast distributed implementation of optimal power flow." *IEEE Trans. Power Syst*. Aug. 1999, pp. 858–864.

[21] Li, F., Broadwater, R.P., "Distributed algorithms with theoretical scalability analysis of radial and looped load flows for power distribution system." *Electr. Power Syst. Res*. May. 2003, pp. 169–177.

[22] Green, R.C., Wang, L., Alam, M., "High performance for electric power systems: Applications and trends*". In Proceedings of the 2011 IEEE Power and Energy Society General Meeting*, Jul. 2011, pp. 1-8.

[23] Saad, Y*., Iterative Methods for sparse Linear Systems, 2ed*; PWS: Boston, MA, 2004.

[24] Dag, H., Semlyen, A., "A new preconditioned conjugate gradient power flow." *In IEEE Transactions on Power Systems*. Nov. 2003, pp. 1248-1255.

[25] Chen, Y., Shen, C., "A Jacobian-free Newton-GMRES(m) method with adaptive preconditioner and its application for power flow calculations." *In IEEE Transactions on Power Systems*. Jul. 2006, pp. 1096-1103.

[26] Flueck, A.J., Chiang, H.-D., "Solving the nonlinear power flow equations with a Newton process and GMRES." *IEEE trans. Power Syst*. May. 1998, pp. 267-273.

[27] Davis, T.A., *Direct methods for Sparse Linear Systems*; SIAM: Philadelphia, PA, 2006.

[28] Singh, J., Aruni, I., "Accelerating Power Flow Studies on Graphics Processing Unit." *2010 Annual IEEE India Conference (INDI-CON)*, Dec. 2010, pp. 1-5.

[29] Guo, C., Jiang, B., Yuan, H., Yang, Z., Wang, L., Ren, S., "Performance Comparison of Parallel Power Flow Solvers on GPU System." *2012 IEEE International Conference on Embedded and Real-time Computing Systems and Applications*, Aug. 2012, pp. 232-239.

[30] Li, X., Li, F.-X., Clark, J.M., "Exploration of multifrontal method with GPU in power flow computation." *2013 IEEE Power & Energy Society General Meeting*, Jul. 2013, pp. 1-5.

[31] Gopal, A., Niebur, D., Venkatasubramanian S., "DC Power Flow Based Contingency Analysis Using Graphics Processing Units." *2007 IEEE Lausanne Power Tech*, Jul. 2007, pp. 731-736.

[32] Su, X., He, C., Liu, T., Wu, L., "Full Parallel Power Flow Solution: A GPU-CPU-Based Vectorization Parallelization and Sparse Techniques for Newton-Raphson Implementation." *In IEEE Transactions on Smart Grid*, Sept. 2020, pp. 1833-1844.

[33] Huang, S., Dinavahi, V., "Performance analysis of GPU-accelerated fast decoupled Power Flow using direct linear solver*." 2017 IEEE Electrical Power and Energy Conference (EPEC)*, Oct. 2017, pp. 1-6.

[34] Schäfer, F., Braun, M., "An efficient open-source implementation to compute the jacobian matrix for the Newton-Raphson Power flow algorithm*." 2018 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe)*, Oct. 2018, pp. 1-6.

[35] Gnanavignesh, R., Shenoy, U.J., "Parallel Sparse LU Factorization of Power Flow Jacobian using GPU." *TENCON 2019 – 2019 IEEE Region 10 Conference (TENCON),* Oct. 2019, pp. 1857-1862.

[36] Wang, Z.-Q., Wende, S., Berg, V., Braun M., "Fast Parallel Newton-Raphson Power flow Solver for Large Number of System Calculations with CPU and GPU". *CoRR*, Sept. 2021.

[37] Chen, X., Wu, W., Wang. Y., Yu, H., Yang. H., "An EScheduler-Based Data Dependence Analysis and Task Scheduling for Parallel Circuit Simulation*." IEEE Trans. Circuits Syst.II Express Briefs*, Sept. 2011, pp. 702-706.

[38] Benzi, M., Tuma, M., "A sparse approximate inverse preconditioner for nonsymmetric linear systems." SIAM J. Scientific Comput, May. 1998, pp. 968-994

[39] De Leon, F., Semlyen, A., "Iterative solvers in the newton power flow problem: preconditioners, inexact solutions and partial jacobian updates." *IEE Proc. Gen., Transm. and Distrib.,* Jul. 2002, pp. 479-484.

[40] Dag, H., Semlyen, A., “A new preconditioned conjugate gradient power flow.” *IEEE Trans. Power Syst.,* Nov. 2003, pp. 1248-1255.

[41] Li, X., Li, F., Yuan, H., Cui, H., Hu, Q., “GPU-Based Fast Decoupled Power Flow with Preconditioned Iterative Solver and Inexact Newton Method.” *In IEEE Transactions on Power Systems*, Oct. 2017, pp. 2695-2703.

[42] Li, X., Li, F., "GPU-based power flow analysis with Chebyshev Preconditioner and Conjugate gradient method." *Elect. Power Syst. Res.*, Nov. 2014, pp. 87-93.

[43] Li, X., Li, F., "GPU-based two-step preconditioning for conjugate gradient method in power flow." *2015 IEEE Power & Energy Society General Meeting*, Jul. 2015, pp. 1-5.

[44] W. Lee, R. Achar and M. S. Nakhla,” Dynamic GPU Parallel Sparse LU Factorization for Fast Circuit Simulation,” *in IEEE Transactions on Very Large-Scale Integration (VLSI) Systems,* Nov. 2018, pp. 2518-2529.

[45] Sooknanan, D.J., Joshi, A., “GPU Computing Using CUDA in the Deployment of Smart Grids.” *2016 SAI Computing Conference*, Jul. 2016, pp. 1260-1266.

[46] Demmel, J.W, Eisenstat, S.C, Gilbert, J.R., Li, X.S, Liu, J.W., “A supernodal approach to sparse partial pivoting.” *SIAM J. Matrix Anal. Appl*. 1999, pp. 720-755.

[47] Schenk, O., Gärtner, K., “Solving unsymemetric sparse systems of linear equations with PARDISO*.” Futrure Cener. Comput. Syst*. Apr. 2004, pp. 475-487.

[48] Christen, M., Schenk, O., Burkhart, H., “General-purpose Sparse Matrix Building Blocks Using the NVIDIA CUDA Technology Platform*.” In First workshop on general purpose processing on graphics processing units*. Oct. 2007, pp. 32.

[49] Zeng, L., Alawneh, S.G., Arefifar, S.A., "GPU-based Sparse Power Flow Studies with Modified Newton’s Method." *IEEE Access*. Nov. 2021, pp. 153226-153239.

[50] Zhou, G, Feng, Y.-J., Bo, R., Zhang, T., "GPU-accelerated sparse matrices parallel inversion algorithm for large-scale power systems." *International Journal of Electrical Power & Energy System*. Oct. 2019, pp. 34-43.

[51] Wu, J.Q, Bose, A., “Parallel solution of large sparse matrix equations and parallel power flow*.” IEEE Trans. Power Syst*. Aug. 1995, pp. 1343-1349.

[52] He, K., Sheldon, X.-D.T., Wang, H., Shi, G., “GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis.” *IEEE Transactions on Very Large-Scale Integration (VLSI) Systems.* Apr. 2016, pp. 1140-1150.

[53] Peng, S, Sheldon, X.-D.T., "GLU3.0: Fast GPU-based Parallel Sparse LU Factorization for Circuit Simulation*." IEEE Design & Test*. Feb. 2020, pp. 78-90.

[54] Chen, X., Ren, L., Wang, Y., Yang, H., "GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling." *IEEE Transactions on Parallel and Distributed Systems*. Mar. 2015, pp. 786-795.

[55] Lee, W.-K., Achar, R., Nakhla, M.S., "Dynamic GPU Parallel Sparse LU Factorization for Fast Circuit Simulation." *IEEE Trans-actions on Very Large-Scale Integration (VLSI) Systems.* Aug. 2018, pp. 2518-2529.

[56] Gao, J.-Q, Chen, Q., He, G.-X., "A thread-adaptive sparse approximate inverse preconditioning algorithm on multi-GPUs." *Parallel Computing*. Apr. 2021, pp. 102724.

[57] Xie, C.-H., Chen, J.-Y., Firoz, J., Li, J.-J., Song, S.-W. L, Barker, K., Raugas, M., Li, A., "Fast and Scalable Sparse Triangular Solver for Multi-GPU Based HPC Architecture." *Association for Computing Machinery*. Aug. 2021, pp. 1-11.

[58] Ma, W., Hu, Y., Yuan, W., Liu, X., "Developing a multi-GPU-enabled preconditioned GMRES with inexact triangular solves for block sparse matrices*." Mathematical Problems in Engineering.* Feb. 2021, pp. 1-17.

[59] Lin, S., Xie, Z., "A Jacobi_PCG solver for sparse linear systems on multi-GPU cluster." *The Journal of Supercomputing*. Jan. 2017, pp. 433-454.

[60] Ding, N., Liu, Y., Williams, S., Li, X.-Y. S., "A Message-Driven, Multi-GPU Parallel Sparse Triangular Solver." *Proceedings of the 2021 SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21).* 2021, pp. 147-159.

[61] Amano, M., Zecevic, A. I., Siljak, D. D., "An improved block-parallel newton method via epsilon decompositions for load-flow calculations." *IEEE Trans.Power Syst.,* Aug. 1996, pp. 1519-1527.

[62] Chen, S.-D., Chen, J.-F., "Fast load flow using multiprocessors*." International Journal of Electrical Power and Energy Systems*. Aug. 2000, pp. 231-236.

[63] Wang, X., Ziavras, S. G., Nwankpa, C., Johnson, J., Nagvajara, P., "Parallel solution of newton's power flow equations on configurable chips." International Journal of Electrical Power and Energy Systems, Jun. 2007, pp. 422-431.

[64] Koester, D. P., Ranka, S., Fox, G. "Parallel block-diagonal-bordered sparse linear solvers for electrical power system applications." *In Scalable Parallel Libraries Conference*, Oct. 1993, pp. 195-203.

[65] D. Ablakovic, I. Dzafic, S. Kecici, "Parallelization of radial three-phase distribution power flow using GPU," *2012 3rd IEEE PES Innovative Smart Grid Technologies Europe (ISGT Europe)*, Oct. 2012, pp. 1-7.

[66] Y. Wallach, "Gradient methods for load-flow problems." *IEEE Trans. Power App. Syst.,* pp. 1314-1318

[67] Galiana, F. D., Javidi, H., McFee, S. "On the application of a preconditioned conjugate gradient algorithm to power network analysis." *IEEE Trans. Power Syst.,* pp. 629-636.

[68] Garcia, N. "Parallel power flow solutions using a biconjugate gradient algorithm and a newton method: A gpu-based approach*." In IEEE Power and Energy Society General Meeting*, Jul. 2010, pp. 1-4.

[69] Li, Z., Donde, V. D., Tournier, J. C., Yang, F., "On limitations of traditional multi-core and potential of many-core processing architectures for sparse linear solvers used in large-scale power system applications*." In IEEE Power and Energy Society General Meeting*, Jul. 2011, pp. 1-8.

[70] John Cheng, Max Grossman, Ty McKercher, *Professional CUDA C Programming.* Indianapolis: John Wiley & Sons, 2014.

[71] Zeng, L., Alawneh, S.G., Arefifar, S.A., "Parallel Multi-GPU Implementation of Fast Decoupled Power Flow Solver with Hybrid Architecture." *Computing Cluster*. Jun. 2022, pp. 1-12.

[72] Zhang, Y., Owens, J.D., "A quantitative performance analysis model for GPU architectures." *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. Feb. 2011, pp. 382-393.

[73] Cook, S. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs.* Morgan Kaufmann: MA, USA, 2012, pp. 84-89.

[74] Geer, D., "Chip Makers Turn to Multicore Processor." *Computer*. May. 2005, 38, pp. 11-13.

[75] David, B.K., Hwu, W.-M. W., *Programming Massively Parallel Processors: A Hands-on Approach, 3rd.* Cambridge: MA, USA, 2010, pp. 3-6.

[76] Kuon, I., Tessier, R., Rose, J., "FPGA Architecture: Survey for DC power flow solution using LabVIEW language." *AADV. Electr. Eng*. 2012, pp. 68-74.

[77] Murach, M., Nagvajara, P., Johnson, J. and Nwankpa, C., "Optimal power flow utilizing FPGA technology." *In Proceedings of the 37th Annual North American Power Symposium.* Oct. 2005; pp. 97-101.

[78] Wills, A., Mills, A., Ninness, B., "FPGA implementation of an interior-point solution for linear model predictive control." *Proc. 18th IFAC World Congr*. Jan. 2011, pp. 14527-14523.

[79] Jerez, J.L., Constantinides, G.A., Kerrigan, E.C., Ling, K.-V., "Parallel Mpc for real-time FPGA-based implementation*." IFAC Proc*. Jan. 2014, pp. 3238-3251.

[80] Ling, K.V., Wu, B.F., Maciejowski, J.M., "Embeded model predictive control (MPC) using a FPGA." *Proc. 17th IFAC World Congr*. Jan. 2008, pp. 15250-15255.

[81] Abughalieh, K.M., Alawneh, S.G., "A survey of parallel implementations for model predictive control." *IEEE Access*. Mar. 2019, pp.34348-34360.

[82] Yaseen, A.Y., Abbood A.A., "Study of Power System Load Flow Using FPGA and LabView." *Engineering and Technology Journal*. May. 2020, 38, pp. 690-697.

[83] Ahmadi, A., Jin, S., Smith, M.C., Collins, E.R., Goudarzi, A. "Parallel Power Flow based on OpenMP." *2018 North American Power Symposium (NAPS)*. Sept. 2018, pp. 1-6.

[84] Zimmerman, R.D., Murillo- Sánchez, C.E, Thomas, R.J. MATPOWER, "Steady-State Operations, Planning and Analysis Tools for Power Systems Research and Education." *IEEE Transactions on Power Systems*. Jun. 2011, pp. 12-19.

[85] Ao, L., Cheng, B., Li, F., "Research of Power Flow Parallel Computing Based on MPI and P-Q Decomposition Method." *In 2010 International Conference on Electrical and Control Engineering.* Jun. 2010, pp.2925-2928.

[86] Alvarado, F.L., "Computational complexity in power systems." *IEEE Trans. Power App. Syst*. Jul. 1976, pp. 1028-1027.

[87] Dagum, L., Menon, R. "OpenMP: an industry standard API for shared memory programming." *IEEE Computational Science and Engineering*. Jan. 1998, pp. 46-55.

[88] Mehrotra, S., "On the Implemetation of a Primal-Dual Interior Point Method." *SIAM Journal on Optimization*. Nov. 1992, pp. 575-601.

[89] Wang, X., Ziavras, S.G., "Parallel LU. Factorization of sparse matrices on FPGA-based configurable computing engines." *Concur Comp: Practice Exper*. Apr. 2004, pp. 319-43.

[90] Wang X., Ziavras, SG., "Parallel direct solution of linear equations on FPGA-based machines." *InProceedings International Parallel and Distributed Processing Symposium*. Apr. 2003.

[91] Glover, J.-D, Overbye, T.-J., Sarma, M.-S., *Power System Analysis & Design, 6th*. Cengage Learning: Boston, USA, 2015, pp. 349-353.

[92] Kundur, P., Balu, N.J., Lauby, M.G., *Power System Stability and Control*. McGraw-hill: New York, NY, USA, 1994.

[93] Stoot, B., Alsac, O., "Fast decoupled load flow." *IEEE Trans. Power Appar. Syst.* May. 1974, pp. 859-869.

[94] Chapra, S.-C, *Applied Numerical Methods with MATLAB® for Engineers and Scientists, 3rd.* McGraw-Hill: New York, USA, 2012, pp. 260-261.

[95] Zhou, G., Zhang, X., Lang, Y., Bo, R., Jia, Y., Lin, J., Feng, Y. "A novel GPU-based strategy for contingency screening of static security analysis." *Int. J. Electr. Power Energy Syst.* Dec. 2016, pp. 33-39.

[96] Lau, K., Tylavsky, D.J., Bose, A., "Coarse grain scheduling in parallel triangular factorization and solution of power system matrices." *IEEE Trans. Power Syst*. May. 1991, pp. 708-714.

[97] Amano, M., Zecevic, A., Siljak, D., "An improved block-parallel Newton method via epsilon decompositions for load-flow calculations." *IEEE Trans. Power Syst*. Aug. 1996, pp. 1519-1527.

[98] El-Keib, A., Ding, H., Maratukulam, D., "A parallel load flow algorithm." *Electr. Power Syst. Res.* Sept. 1994, pp. 203-208.

[99] Fukuyama Y., Nakanishi Y., Chiang HD., "Parallel power flow calculation in electric distribution networks." *IEEE International Symposium on Circuits and Systems (ISCAS)* May. 1996, pp. 669-672.

[100] Luo, C., Zhang, K., Salinas, S., Li, P., "SecFact: Secure Large-scale QR and LU Factorizations." *IEEE Transactions on Big Data*. Dec. 2017, pp. 796-807.

[101] Golub, G.H, Van Loan, C.F., *Matrix Computation, 4th.* John Hopkins University Press: Baltimore, USA, 2013; pp. 233-255.

[102] Buttari, A., Langou, J., Kurzak, J., Dongarra, J. "A class of parallel tiled linear algebra algorithms for multicore architectures." *Parallel Computing.* Jan. 2009, pp. 38-53.

[103] Gregorio, Q.-O., Enrique, S.Q.-O., Robert, A., Van, D.G., Field, G.V.-Z., Ernie, C. "Programming matrix algorithms-by-blocks for thread-level parallelism." *ACM Trans. Math. Softw.* Jul. 2009, pp. 1-26.

[104] Zhou, G., Feng, Y., Bo, R., Chien, L., Zhang, X., Lang, Y., Jia, Y., Chen, Z., "GPU-accelerated batch-ACPF solution for N-1 static security analysis." *IEEE Trans. Smart Grid.* Aug. 2016, pp. 1406-1416.

[105] Saad Y., "Preconditioning techniques for nonsymmetric and indefinite linear systems." *Journal of computational and applied mathematics*. Nov.1988, pp. 89-105.

[106] Saad Y., "Iterative methods for sparse linear systems." *Society for Industrial and Applied Mathematics*. Jan. 2003, pp. 336-337.

[107] J. Sanders, E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Boston, MA, USA: Addison-Wesley, 2010.

[108] Flegar, G., *Sparse linear system solvers on gpus: Parallel preconditioning, workload balancing, and communication reduction*. 2019.

[109] M. M. A. Abdelaziz, "OpenCL-Accelerated Probabilistic Power Flow for Active Distribution Networks," *in IEEE Transactions on Sustainable Energy*. Jul. 2018, pp.1255-1264.

[110] Richard L. Burden, J. Douglas Faires, *Numerical Analysis*. Boston, MA, USA: Brooks/Cole, 2011, pp. 294-296.

[111] M. Faverge, J. Herrmann, J. Langou, B. R. Lowery, Y. Robert and J. Dongarra, "Designing LU-QR Hybrid Solvers for Performance and Stability," *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. May. 2014, pp. 1029-1038.

[112] Yulu Jia, Piotr Luszczek, and Jack Dongarra, "Multi-GPU Implementation of LU Factorization," *Procedia Computer Science*. June. 2012, pp. 106-115.

[113] Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Kru¨ger, J., Lefohn, A.E., Purcell, T.J. "A survey of general-purpose computation on graphics hardware". *Computer Graphics Forum*. Mar. 2007, pp. 80–113.

[114] Xie C, Chen J, Firoz J, Li J, Song SL, Barker K, Raugas M, Li A., "Fast and scalable sparse triangular solver for multi-gpu based hpc architectures." *InProceedings of the 50th International Conference on Parallel Processing*. Aug. 2021, pp. 1-11.