mathematics

MDPI

Review

A Review of High-Performance Computing Methods for Power Flow Analysis

Shadi G. Alawneh *, Lei Zeng and Seyed Ali Arefifar

Electrical & Computer Engineering, Oakland University, 115 Library Drive, Rochester, MI 48309, USA; leizeng@oakland.edu (L.Z.); arefifar@oakland.edu (S.A.A.)
* Correspondence: shadialawneh@oakland.edu

Abstract: Power flow analysis is critical for power systems due to the development of multiple energy supplies. For safety, stability, and real-time response in grid operation, grid planning, and analysis of power systems, it requires designing high-performance computing methods, accelerating power flow calculation, obtaining the voltage magnitude and phase angle of buses inside the power system, and coping with the increasingly complex large-scale power system. This paper provides an overview of the available parallel methods to fix the issues. Specifically, these methods can be classified into three categories from a hardware perspective: multi-cores, hybrid CPU-GPU architecture, and FPGA. In addition, from the perspective of numerical computation, the power flow algorithm is generally classified into iterative and direct methods. This review paper introduces models of power flow and hardware computing architectures and then compares their performance in parallel power flow calculations depending on parallel numerical methods on different computing platforms. Furthermore, this paper analyzes the challenges and pros and cons of these methods and provides guidance on how to exploit the parallelism of future power flow applications.

Keywords: multi-cores; multi-GPUs; FPGA; iterative and direct methods; power flow calculation; parallelism

MSC: 68U01

check for updates

Citation: Alawneh, S.G.; Zeng, L.; Arefifar, S.A. A Review of High-Performance Computing Methods for Power Flow Analysis. *Mathematics* **2023**, *11*, 2461. https://doi.org/10.3390/math11112461

Academic Editor: Xiangmin Jiao

Received: 1 April 2023
Revised: 19 May 2023
Accepted: 24 May 2023
Published: 26 May 2023

Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Power flow (PF) analysis is popularly employed to analyze electrical power systems by estimating the voltage and phase angle of buses inside the power system, while the PF calculation is designed to repeatedly apply several numerical methods for solving nonlinear equations of such electric power systems [1]. Power flow calculation plays a crucial role in power system analysis, planning, and operation [2], as does the development of multiple renewable energy supplies. Furthermore, with the introduction of solar and wind energy, the spread of electric vehicles (EVs) is continually expanding and increasing power system complexity. Additionally, the application of smart grids and advanced electronic devices creates a need for more electrical power of higher quality, while the spread of EVs is evaluated to increase power consumption in the future. Therefore, there has been a greater need for more accurate power flow analysis [1]. Power system analysis and modeling, however, have been challenging for power engineers due to the increasing scale and heavier loading of the power system, which not only puts great pressure on power flow calculations [3] but also motivates engineers to design high-performance methods to fix the issues.

Generally, PF calculations repeatedly utilize several numerical methods for linearizing non-linear equations and solving these linearized equations of such electrical power systems; the PF calculation usually takes a considerable amount of execution time due to updating the voltage magnitude and angle on each iterative process [4]. In practice, the classic algorithm usually uses the Newton–Raphson (NR) method in the process of power

flow calculation, but the efficacy of acceleration is limited due to the sequential execution style of the program. With the advancement of parallel devices, numerous highly efficient heterogeneous computing systems have emerged, particularly on the GPGPU platform. Khairy, M. et al. [5] provided a survey about GPUs in terms of performance, programmability, and heterogeneity. Although the authors focused on the detailed parallel techniques of GPUs, these methods, including memory and data locality and warp-level scheduling, provided guidance for parallel acceleration in PF calculations. Thus, researchers proposed various methods to improve the parallelization of the large-scale PF calculation depending on different hardware platforms and computing architectures. In particular, the use of high-performance computing (HPC) machines has been an active study for many years to reduce the execution time of PF calculations [6–10]. From the perspective of numerical computing methods, most of them can be classified into two categories: iteration methods [11–14] and direct methods [15–25]. As for the iteration method, X. Li et al. [26–28] designed a preconditioner to reduce the condition number of the Jacobian matrix, improving the stability of solving linear equations when applying the NR method. Although these iteration methods can accelerate the calculation of large-scale power systems with the help of parallel devices, the execution time severely depends on the design of the preconditioner. Generally, the PF calculation easily fails to converge due to the sparsity and singularity of the power system. To overcome this shortage, researchers need to validate the preconditioner repeatedly until they obtain a stable and robust preconditioner for solving PF equations. Thus, the preparation of the iteration method is a time-consuming and tedious task. In addition, the iterative method can be regarded as an optimization algorithm used to find the optimal numerical solutions for linear equations in PF calculations. Specifically, the process of solving linear equations in PF calculations is a problem of unconstrained optimization. Zhou Y. et al. [29,30] introduced a parallel ant colony optimization approach for the traveling salesman problem (TSP), leveraging various parallel devices, which led to superior performance. Nevertheless, this method typically requires a substantial amount of time to find an optimal solution due to the inherent randomness associated with ant movement. Thus, the method can be employed in the linear solver of PF calculation in the future, particularly when the shortage and limitations can be mitigated by the advancements in computing capabilities of parallel devices due to the method's inherent potential for parallelism.

Most direct approaches utilize either the NR or fast decoupled power flow (FDPF) method, solving the linearized PF equations on the parallel devices, operating the coefficient matrices in a vectorization parallelization manner, and improving the program's performance. References [3,16] took the dense NR method on the single GPU platform and indeed achieved almost a speedup of 8x compared with the CPU counterpart. Similarly, D.J. Sooknanan et al. [31] concentrated on accelerating the calculation of the Jacobian and bus admittance matrix on a single GPU. In addition, Guo, C., et al. [17] analyzed the performance of a parallel PF solver, proving the outperformance of the NR method. These dense parallel methods leveraged the high floating-point operations and throughput of the GPUs to achieve the high speed-up. For the dense matrix format, these methods, however, can fail to converge while the power system has a large scale, resulting in out-of-bounds memory access on the single GPU platform. To overcome the issue, in [20,24,32–37], researchers took the sparse matrix format and exploited the parallelism of the power system in sparse format. Although the GPU-based sparse method saved a large amount of global memory and reduced the computation involved by zero entries in sparse matrices, the single-GPU architecture limited the scalability of computing units due to the strong data dependence and irregular memory access patterns of sparse matrices. Furthermore, the GPU-based sparse linear solver has been studied in connection with linear equation solvers in PF engineering. In [23,38–41], most of them attempted the left-looking or the hybrid column-based right-looking algorithm to perform a parallel LU decomposition, accelerating the sparse linear solver. However, multi-GPU methods have also been proposed to improve the performance of sparse linear solves [42–46] with the development of parallel devices. The

cost of communication, however, has become the main bottleneck rather than computing capability. Therefore, the parallel multi-GPU implementation [47], based on the FDPF method, was proposed, and it took task parallelism and data parallelism to reduce the cost of communication between GPUs, improving the performance of the program.

According to the survey of HPC methods for PF calculation, most parallel implementations for power systems are based on the GPU platform. Thus, this paper introduces overall parallel methods for power systems on different parallel devices, including multi-cores, hybrid CPU-GPU, and FPGA heterogeneous parallel devices.

The rest of this paper is organized as follows: Section 2 briefly reviews the background of power flow modeling and analysis, power flow methods, and the linear system solver. In Section 3, the methods and performance, based on different parallel platforms, are reviewed. Section 4 briefly discusses and analyzes the pros and cons of these parallel methods and provides guidance for parallelization in future PF applications. Finally, the overall conclusion and review will be presented in the last section.

2. Background

2.1. PF Model and Analysis

A PF model and analysis are a representation of engineering modeling and a numerical method that estimates the electrical power flow of an interconnected power system, which is designed to repeatedly execute a specific numerical method to obtain the voltage magnitudes and angles of all the buses in a power system and calculate the real and reactive powers of peripheral equipment connected to the buses [1]. To solve a nonlinear power flow problem using admittance Y_{bus}, the nodal equations for a power system network are expressed as follows:

$$I = Y_{bus} V \tag{1}$$

where I is the N vector of source currents injected into each bus and V is the N vector of bus voltages. For bus k, the kth equation in (1) is written as:

$$I_k = \sum_{n=1}^{N} Y_{kn} V_n \tag{2}$$

where Y_{kn} is admittance between bus k and bus n. I_k is the current at bus k. V_n is the voltage at bus n. The complex power delivered to bus k is expressed as:

$$S_k = P_k + jQ_k = V_k I_k^* \tag{3}$$

where S_k is the complex power on bus k. P_k and Q_k are real and reactive power, respectively, using (2) and (3),

$$P_k + jQ_k = V_k \left[\sum_{n=1}^{N} Y_{kn} V_n \right]^* \tag{4}$$

with the following notation,

$$V_n = V_n e^{j\delta_n} \tag{5}$$

$$Y_{kn} = Y_{kn} e^{j\theta_{kn}} = G_{kn} + jB_{kn} \tag{6}$$

where G_{kn} and B_{kn} are conductance and susceptance between bus k and bus n, respectively. δ_n is the angle of voltage at bus n. θ_{kn} is the angle difference between bus k and bus n. Therefore, Equation (4) can be written as:

$$P_k + jQ_k = V_k \sum_{n=1}^{N} Y_{kn} V_n e^{j(\delta_k - \delta_n - \theta_{kn})} \tag{7}$$

Taking the real and imaginary parts of (7), the power balance equations are written as either:

$$P_k = V_k \sum_{n=1}^{N} Y_{kn} V_n cos(\delta_k - \delta_n - \theta_{kn}) \tag{8}$$

$$Q_k = V_k \sum_{n=1}^{N} Y_{kn} V_n sin(\delta_k - \delta_n - \theta_{kn}) \tag{9}$$

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

$$P_k = V_k \sum_{n}^{N} V_n [G_{kn} \cos(\delta_k - \delta_n) + B_{kn} \sin(\delta_k - \delta_n)] \tag{10}$$

$$Q_k = V_k \sum_{n=1}^{N} V_n [G_{kn} \sin(\delta_k - \delta_n) - B_{kn} \cos(\delta_k - \delta_n)] \tag{11}$$

From the PF model, solving the power balance equations facilitates power system engineers maneuvering the voltage magnitude and phase angle at each bus and analyzing whether the power system is under balanced three-phase steady-state conditions [48].

2.2. PF Methods

The PF problem is the computation of voltage magnitude and phase angle at each bus in a power system under balanced three-phase steady-state conditions [48]. Thus, numerous approaches have been developed to fix the issue, including Gauss–Seidel (GS), NR [49], and FDPF [50] methods. Among them, the NR method is currently prevalent for power flow analysis and calculation due to its fast convergent property. Equations (10) and (11) employ Taylor serial techniques for linearization, and then the set of non-linear equations can be formulated as follows:

$$\Delta f = J \cdot \Delta x \tag{12}$$

where J is a Jacobian matrix. Specifically, Equation (12) can further be expressed in the form of a matrix:

$$\begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} = \begin{bmatrix} \frac{\partial \Delta P}{\partial \theta} & \frac{\partial \Delta P}{\partial V} \\ \frac{\partial \Delta Q}{\partial \theta} & \frac{\partial \Delta Q}{\partial V} \end{bmatrix} \cdot \begin{bmatrix} \Delta \theta \\ \Delta V \end{bmatrix} \tag{13}$$

where ΔP and ΔQ are the power mismatches, respectively. $\partial P/\partial \theta$ and $\partial P/\partial V$ are derivatives of ΔP to $\Delta \theta$ and derivatives of ΔP to ΔV, respectively. Similarly, $\partial Q/\partial \theta$ and $\partial Q/\partial V$ are derivatives of ΔQ to $\Delta \theta$ and derivatives of ΔQ to ΔV, respectively. Initially, assuming the arbitrary values of V and θ introduced into the NR iteration of (13), we updated the voltage magnitude, angle, and power mismatch in the process of each iteration, repeatedly, until the mismatch satisfied the convergent condition. For the sake of clarity, Figure 1 depicts the iterative process of PF calculation depending on the NR method. The computationally expensive calculation involves updating both Jacobian coefficient matrices and the power mismatch. Nevertheless, the NR method performed the PF calculation much faster than the GS method, obtaining the results in fewer iterations [1,17]; thus, the NR method is also widely used by some researchers on large-scale systems [1,48]. Furthermore, the Jacobian matrix of the NR method can provide an index for sensitivity analysis or some other control problems, which is why some studies have centered on parallel algorithms for acceleration of the GPU-based NR method.

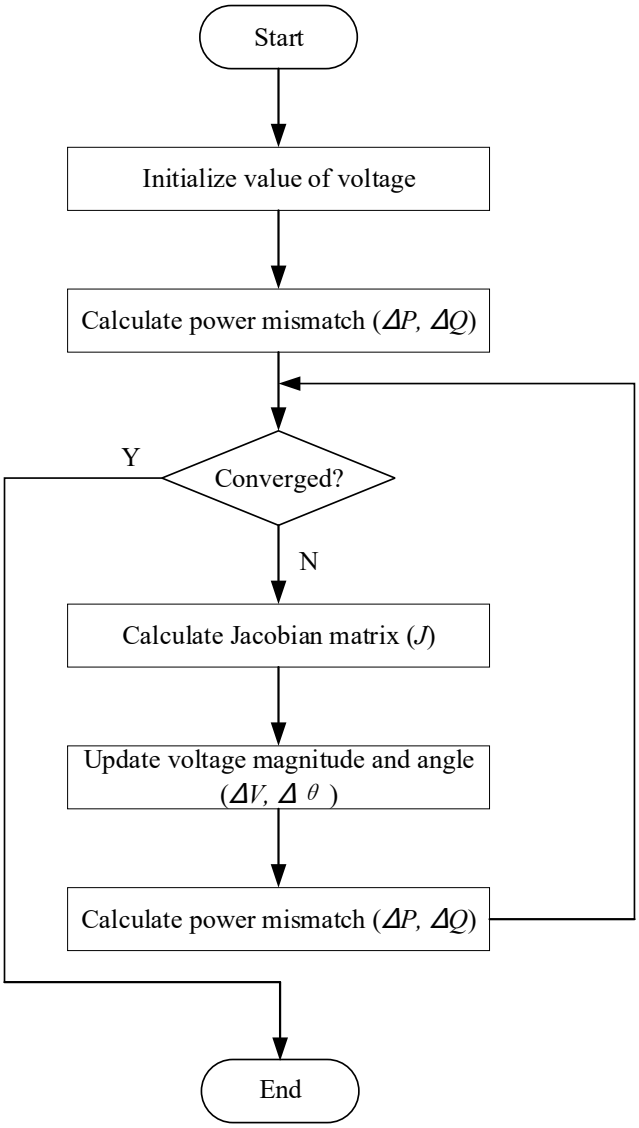


Figure 1. The flow chart of the NR method.

The FDPF method, which is based on the NR method, simplifies the complex calculation of Jacobian coefficient matrices. Thus, the formula can be modified from (13):

$$\begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} = \begin{bmatrix} \frac{\partial \Delta P}{\partial \theta} & 0 \\ 0 & \frac{\partial \Delta Q}{\partial V} \end{bmatrix} \cdot \begin{bmatrix} \Delta \theta \\ \Delta V \end{bmatrix} \tag{14}$$

Since the transmission line has a small resistance, the adjacent buses tend to have a tiny phase angle difference [35]. Thus, sub-diagonal components of the Jacobian matrix can be ignored as zero. Although the FDPF method is derived from the NR method, it is much simpler and more efficient algorithmically [4]. One of the reasons is that the non-zero Jacobian coefficient matrices are fixed and reusable, while the NR method has to update the Jacobian coefficient matrices on each decomposition process [4]. As a result, Equation (14) can further be divided into two independent tasks as follows:

$$\Delta P = \frac{\partial \Delta P}{\partial \theta} \cdot \Delta \theta \tag{15}$$

$$\Delta Q = \frac{\partial \Delta Q}{\partial V} \cdot \Delta V \tag{16}$$

According to Equations (15) and (16), two tasks, solving voltage magnitude and phase angle equations, are suitable for assigning to different respective parallel devices, such as multi-GPUs and multi-cores, which improves the granularity of parallelization. Specifically, in the process of linear equations of PF calculation, the Jacobian matrix can be stored in a packed format and accessed in contiguous blocks rather than scattered throughout memory. That means threads in a warp can access data while fewer DRAM (dynamic random-access memory) requests are executed.

2.3. Linear System Solver

For solving the power flow calculation by utilizing either the NR or FDPF methods, the solution of a linear system of matrix equations is critical [1]. The solvers can be classified into direct and iteration methods, respectively. Specifically, the direct method consists of LU factorization and QR decomposition, while the iteration methods are based on conjugate gradient (CG) methods with the design of a preconditioner, which is typically implemented for the analysis of large-scale power systems.

2.3.1. LU Factorization

LU decomposition is a classic numerical method for solving linear systems. In the process of PF calculation, the LU solver is utilized on each iteration, and this direct solver is more robust on ill-conditioned problems [21]. Generally, the LU method is suited for the coefficient matrix of the power system because the matrix is square. Similarly, just as for standard Gauss elimination, the partial pivoting technique is used to obtain reliable solutions with LU factorization [51]. This method includes three steps:

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

$$PA = LU \tag{17}$$

2. Forward: Generate the intermediate vector y depending on P and L in Equation (17), and they can be written as:

$$Ly = Pb \tag{18}$$

3. Backward: solving a triangular system, based on the vector y of Equation (18), as follows:

$$Ux = y \tag{19}$$

Sometimes, Equation (17) can be applied to Cholesky factorization for acceleration, as follows:

$$PA = LL^t \tag{20}$$

where the L^t is the transposition matrix of L. In practice, the Cholesky factorization, however, focuses on the symmetric and positive definite matrix, and this factorization more often fails to converge in PF calculation due to the nature of the power system. Thus, this method is usually applied to special situations, while its disadvantage limits its application to power systems. Furthermore, the calculation time of the linear system solver takes up a large proportion of the power flow calculation. It was found to take up to about 80 percent of the total computation time in [52]. Thus, many optimal algorithms focused on these critical parts [37,53–56], which utilized parallel methods to reduce the execution time.

2.3.2. QR Factorization

QR factorization is also an alternative for power flow calculation. Specifically, the QR factorization is to decompose a matrix with linearly independent columns into a product of an orthogonal matrix [57]. It can be written as follows:

$$A = Q \cdot R \tag{21}$$

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 the Gram–Schmidt process, Householder transformations, and Givens rotations [58]. In practice, compared to LU factorization, QR decomposition is more stable because LU decomposition without pivoting may suffer from significant numerical instability, as described in [15]. However, QR factorization requires almost twice as many operations and more complicated steps that are not as parallel as a matrix-matrix product [59,60]. Although QR factorization might increase the cost of computation, a few studies still exploit the GPU-based parallel computing method due to its high numerical stability, even without pivoting [61].

2.3.3. Iterative Method

Generally, the iterative method can be treated as the optimal algorithm, designed to find the numerical solutions of linear equations in PF calculations. Usually, the conjugate gradient (CG) method is utilized to handle large-scale power systems. Most of them probably failed to converge due to the nature of the power system, and then the convergence of these iterative methods severely depends on the property of the preconditioner. Research often took a long time to validate the preconditioners repeatedly until it found the optimal one. Thus, the design of a preconditioner is a time-consuming and tedious task. Nevertheless, the parallel CG method has been applied to PF calculations, including [12,27,62].

3. Parallel Method and Performance

3.1. The Multi-Core CPU Architecture

The design philosophy of a multi-core platform is to minimize the execution latency of a single thread. Typically, the multi-core processor consists of a shared memory with multiple central processing units (CPUs) on the chip, providing good support for multi-threaded applications and improving overall performance by handling more tasks in a parallel fashion [63]. Figure 2 depicts the classic outline of the multi-core CPU architecture. Reviewing the CPU architecture, large last-level on-chip caches are designed to capture frequently accessed data and convert some of the long-latency memory access into short-latency cache accesses, and the design philosophy of arithmetic units and operand data delivery focuses on minimizing the effective latency of operation at the cost of increased use of chip area and power [64].

In distributed memory fashion, each CPU with its own local memory accelerates computation-intensive tasks, and memory is scalable as the number of processors increases. Thus, each processor can rapidly access its own local memory without an interface, avoiding the overhead of maintaining cache coherency. Specifically, the researchers are responsible for many of the details associated with data communication between processors. Furthermore, it might be difficult to map the multiple mutually dependent tasks into the distributed architecture. Conversely, the multi-threads in Figure 2b can operate independently but share the same memory in a shared-memory fashion. The data changes in the shared memory are visible to all processors instead of time-consuming broadcast operations in Figure 2a. Shared-memory parallel devices are widely used due to their user-friendly programming perspective on memory. However, this shared memory architecture lacks scalability between memory and CPUs. The programmers sometimes have to introduce the time-consuming atomic operation, ensuring the correct results, because each CPU has the same opportunity to access the critical data.

3.2. The Multi-Core Parallel in PF Studies

In the process of PF calculation, the multi-core method is designed to allow greater performance achievement through task parallelism [65]. The utilization of memory can be summarized as shared and distributed memory fashion. In [65], OpenMP (Open Multi-Processing) used coarse-grained parallelism to accelerate the linear system solver in a

shared-memory fashion. To find the solution of PF equations at each iteration, the author first factorizes matrix L and U of the Jacobian matrix using the Gaussian elimination technique, and then the forward-back substitution method is applied to find the solution of the linear system [65]. Further, Algorithm 1 depicts the implementation of this approach with the respective OpenMP directives.

Algorithm 1 Solving dx = J\F [65]

```
1 #pragma omp parallel for schedule (guided, chunk_size) ...
2 for k = 0, 1, ... , n{
3       for j = k + 1, k + 2, ... , n {
4             x = J_ik/J_kk;
5                for I = k, k + 1, ... , n
6                   J_ji = J_ji − x × J_ki;
7             J_ik = x;
8       }
9 }
10 #paragma omp parallel for schedule (guided, chunk_size)
11 for m = 0, 1, ... , n {
12                d[m] = 1.0;
13                if(m! = 0) d[m − 1] = 0;
14              for i = 0, 1, ... , n {
15                   sum = 0.0;
16                      for j = 0, 1, ... , i − 1 sum = sum + J_ij × y_j;
17                   y_i = d_i − sum;
18                }
19                for i = n, n − 1, ... , 0 {
20                   sum = 0.0;
21                      for j = i + 1, i + 2, ... , n sum = sum + J_ij × dx_jm;
22                   dx_jm = (y_i − sum)/J_ii;
23                }
24 }
```

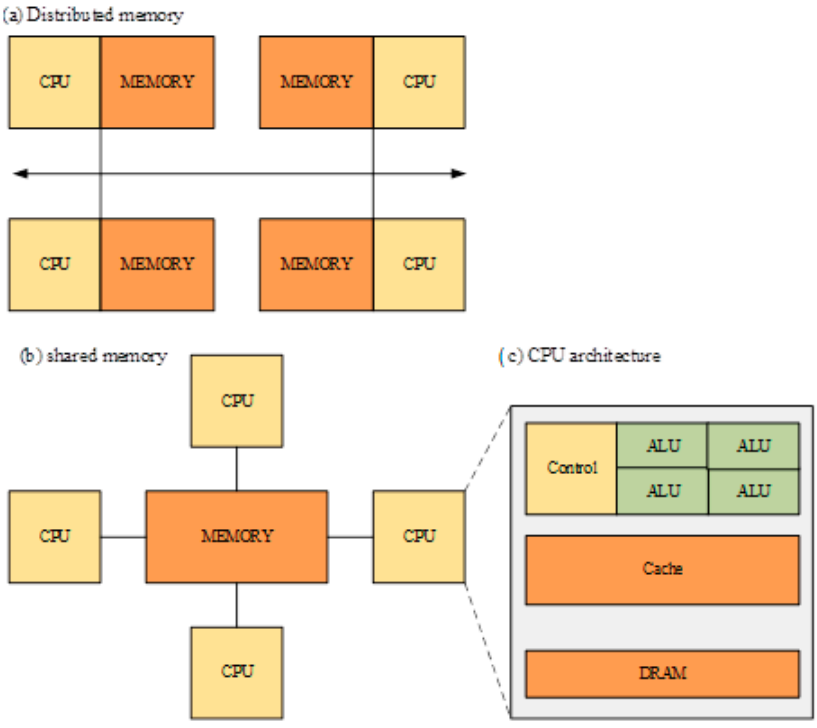


Figure 2. The classic multi-core parallel architecture.

The authors tested the implementation on an Intel Core i5-2400 Quad CPU at 3.1 GHz. The proposed approach achieves, on average, a 2.0× speedup compared to the sequential MATPOWER [66]. From Algorithm 1, only the outer loop has been parallelized due to the data dependencies. Although it improved the performance of the linear system solver, the efficiency of the program might degrade as the power system scales, caused by the data race. To reduce the data race, Li, F., et al. [67] proposed parallel PF calculations based on distributed memory using the MPI (message passing interface) library. The authors took the triangular decomposition of the parallel algorithm and distributed the data on the multiple computing units as shown in Figure 2a. To avoid the deadlock caused by the asynchrony, this method took data independence and processed a number independence strategy [67]. Specifically, it can be considered a one-master, multi-slave model. Each computing unit calculates its own independent task, and then the result of the child computing unit is sent back to the main computing unit depending on the size of the data and the result obtained, as described in Figure 3.

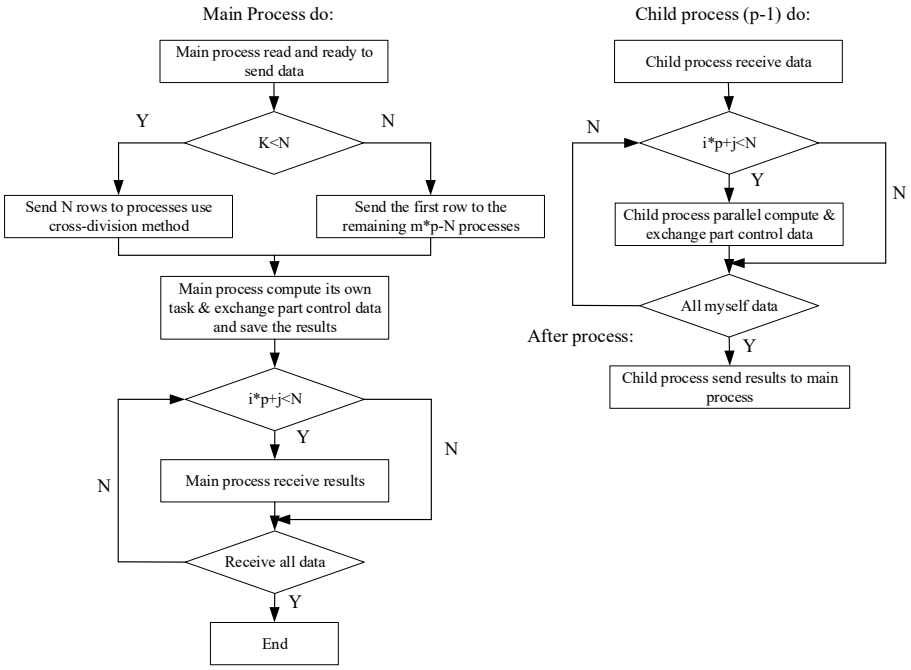


Figure 3. The schematic diagram of the power grid [67].

In [67], the speedup of implementation on the pc-cluster system self-built is almost 3× depending on the IEEE datasets. However, the authors only tested and analyzed the IEEE 30, IEEE 118, and IEEE 300 buses. Thus, this method cannot ensure high efficiency as the power system scales due to the high cost of communication between different computing units in a distributed memory fashion.

3.3. The GPU Architecture

GPU parallel computing architecture is a heterogeneous architecture, consisting of host and device, respectively. Specifically, the GPUs are designed as parallel, throughput-oriented computing engines, and the computing-intensive tasks are transmitted to the device due to the highly efficient floating-operation capabilities of the GPUs. Figure 4 depicts this heterogeneous programming mode in a 2D fashion. Massive threads lay the foundation of single instruction multiple threads (SIMT), and they can be invoked simultaneously and organized in a two-level hierarchy, with each level supporting up to three dimensions [24].

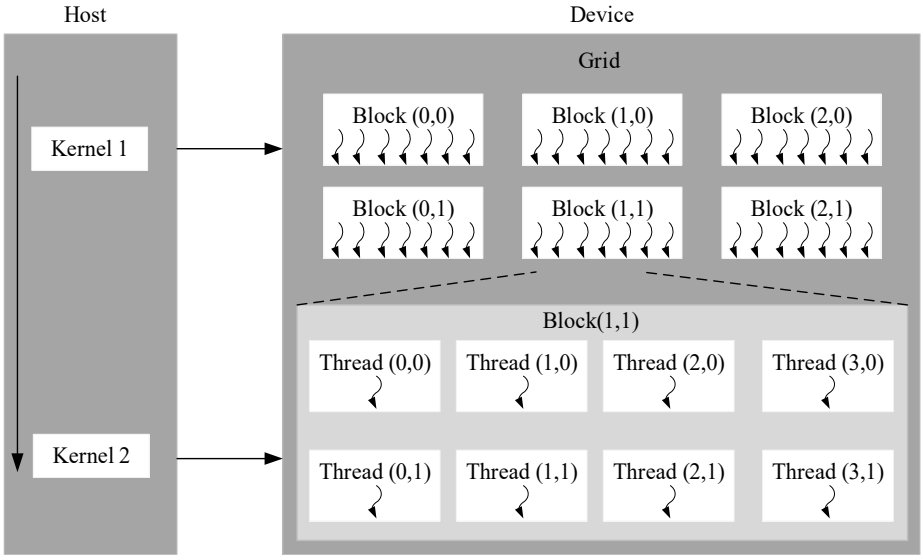


Figure 4. The heterogeneous programming model [47].

When the kernel is launched, the massive threads in the first level are grouped into a block and assigned to a streaming multiprocessor (SM) for execution [23]. All threads in a block can implement synchronization with an explicit synchronization barrier [47]. Similarly, the blocks are also organized as a grid on the second level. During the execution of a CUDA kernel, every set of 32 consecutive threads within one block is integrated into the minimum execution unit called a warp [68]. All threads of a warp execute instructions in a lock-step manner in accordance with the single program, multiple data (SPMD) concept [69]. While some threads in the same warp encounter the if-statement, it might cause thread divergence, keeping these threads in a blind-wait state and losing the advantage of the parallel devices [47]. Thus, threads should access the data from memory in a coalesced manner due to the features of the GPUs, and the program should avoid multiple launches of kernel functions to reduce the time-consuming communication between host and device.

3.4. The Hybrid CPU-GPU Parallel in PF Studies

Research on heterogeneous GPU-based parallel PF calculation has always been active for years, and these methods can also be classified into iteration and direct methods. Essentially, most iteration methods in PF calculation are considered optimal algorithms based on the CG method. Assuming the values of voltage magnitude and phase angle, the program finds the optimal solution depending on the optimal path. These methods, however, have limited application and focus on symmetric and positive definite matrices.

Additionally, to ensure the convergence of the PF calculation, a preconditioner is designed to reduce the condition number of the matrix, improving the program's performance. In [26], Li X. et al. proposed a GPU-based FDPF method and transmitted the Jacobian matrix to the GPU at the beginning of the program. Furthermore, the authors designed the inner iteration, the inexact Newton method, to approach the solution of linear equations with speed and convergence.

Although the inexact Newton method cannot obtain a precise solution for the inner CG iteration, it facilitates the outer iteration to gradually approach the accurate solution [26], as described in Figure 5. In the process of the PF calculation, communication between the host and device occurred multiple times through the PCI Express (PCIE) bus, causing the degradation of the program, as shown in Figure 6. Still, the proposed approach achieves a 2.86× speedup with over 10,000 buses, compared to traditional FDPF on the 8-core Xeon E5607 2.27GHz CPU equipped with the NVIDIA Tesla M2070 GPU. Similarly, in [27,28], the Chebyshev preconditioner is designed to reduce the condition number of sparse matrices and relieve the computation overhead, which achieves almost 4× and 8× speedups, respectively, through the CUBLAS and CUSPARSE libraries.

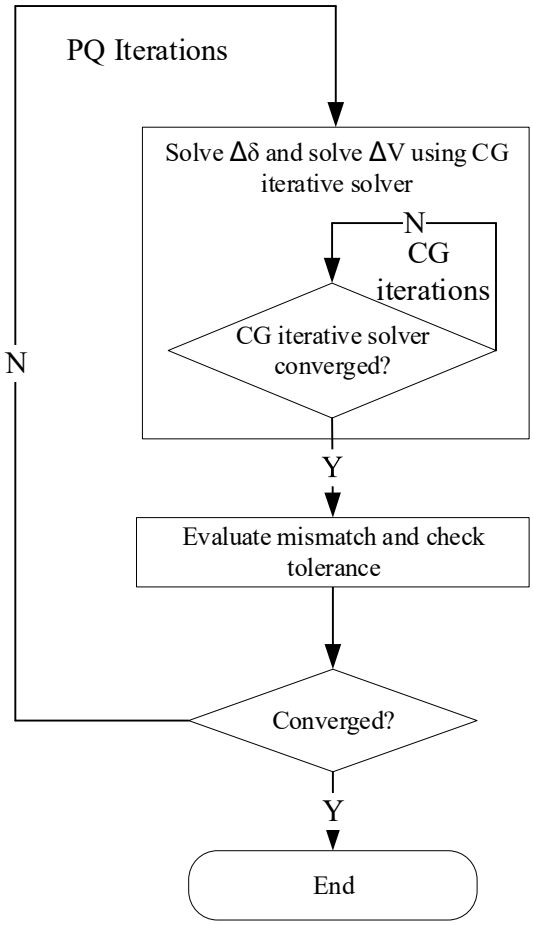


Figure 5. The flowchart of the FDPF iteration method [26].

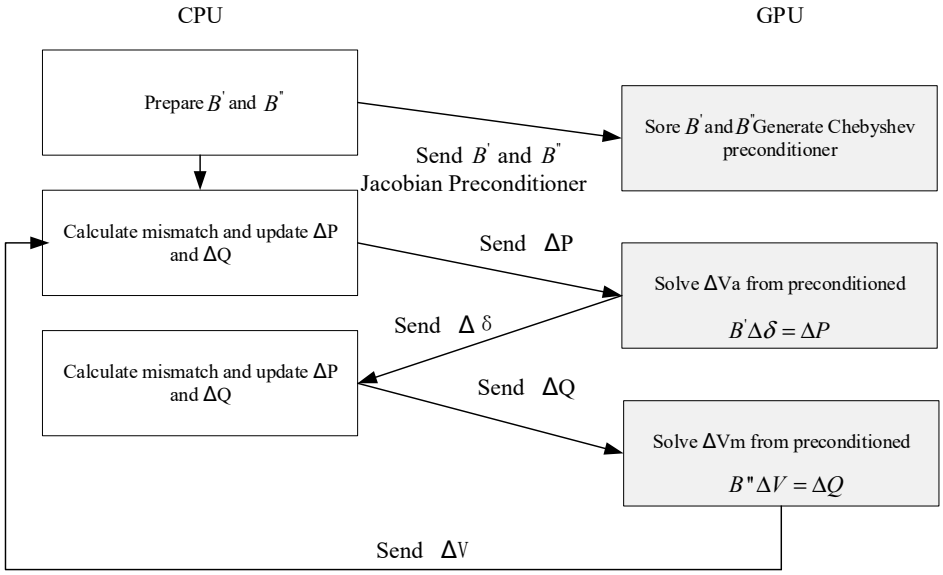


Figure 6. The flowchart of the FDPF iteration method [26].

Generally, the direct method is more robust for ill-conditioned problems of power flow calculation [21]; however, this method consumes much more memory compared to iterative methods. Thus, researchers usually take a sparse approach to distributing parallel tasks on a single GPU platform when the power system is large scale. In [16,31], researchers took a dense matrix format and distributed the computing-intensive tasks on a single GPU platform. Specifically, in [16], several systems with 14, 118, 300, and 2383 buses were tested

on the Intel i7-4500U CPU equipped with NVIDIA GeForce GT745, obtaining a significant speedup of 10×–40× compared to the CPU counterpart. Although the result demonstrated the efficacy of the parallel acceleration, the kernel easily crashed as a result of out-of-bounds memory access when the size of the power system continued to increase [16].

Similarly, Sooknanan, D.J., et al. [31] tested the same datasets on the Intel Xeon Quad Core with the NVIDIA TESLA C1006 GPU, using the same matrix format and achieving almost 7× faster than the CPU for the power system with 2383 buses due to the high parallelism of the dense matrix format. The application of dense format is expected to be applied to the small size of the power system on the single GPU platform, avoiding the overflow of the memory. Thus, most researchers exploited the parallelization of PF calculation in the sparse format, avoiding memory out of bounds and improving the scalability of the power system. In [20], researchers explored a CPU-GPU-based parallel power flow approach, combing sparse matrix techniques, and the datasets were organized in compressed row storage (CRS) and compressed column storage (CSC) format, reducing the computational complexity from O(N^3) to O ($N^{1.4}$) for $N \times N$ Jacobian matrix [70]. Compared to [26,27,71], Su, X. et al. [20] not only focused on solving linear equations in parallel but also optimized the creation of Jacobian matrices in a parallel vectorization fashion through the Arrayfire library. Vectorization parallelization refers to converting a scalar program to a vectorized one, and then vectorized programs can run multiple operations concurrently via a single instruction, whereas scalar programs can only operate on pairs of operands [20]. Similarly, researchers used the same method to accelerate the generation of the Jacobian matrix in [22].

In a different approach [21], generally, the lower and upper triangular sparse matrices are still sparse after LU factorization [15]. Thus, Huang, S. et al. focused on the linear system solver and reordered the coefficient matrix, A, shifting A to B, and this transformation can be expressed as follows:

$$B = QAQ^T \tag{22}$$

where Q is the permutation matrix, following the below property:

$$QQ^T = Q^TQ = I \tag{23}$$

where I is the identity matrix, the fill-ins of the A can be greatly reduced by this transformation. Therefore, the linear equations, $Ax = b$, can be rewritten as follows:

$$Ax = b \rightarrow AQ^TQx = b \rightarrow QAQ^TQx = Qb \tag{24}$$

For brevity, Equation (24) can be rewritten as:

$$B\hat{x} = \hat{b} \tag{25}$$

This proposed method was tested on the Intel Xeon E5-2620 CPU with the NVIDIA GeForce Titan Black GPU and achieved 4.16× faster than its Matlab counterpart over the 13,659 buses.

In [3,35], researchers used the continuous Newton's method for robust convergence of PF calculations based on dense and sparse matrix formats, respectively. Specifically, the PF calculation was converted into finding solutions to autonomous ordinary differential equations, as follows:

$$\dot{x}_n = -J(x_n)^{-1} \cdot f(x_n) \tag{26}$$

where $f(x)$ is the nonlinear power flow equations. $\dot{x}_n$ and x_n are the derivatives of state vectors and state vectors, respectively. In fact, if $\dot{x}_n$ approaches zero, then $f(x)$ has the solution, as long as $J(x_n)$ is not singular. The second order Runge–Kutta formula is applied to (26), and the computing formulations are written as follows:

$$-J_0 \cdot k_1 = f(x_n) \tag{27}$$

$$-J_0 \cdot k_2 = f(x_n + \Delta h \cdot k_1) \tag{28}$$

$$\Delta x_n = \frac{\Delta h}{2} \cdot (k_1 + k_2) \tag{29}$$

$$x_{n+1} = x_n + \Delta x_n \tag{30}$$

$$h_{n+1} = h_n + \Delta h \tag{31}$$

where k_1 and k_2 are the intermediate values for updating sate vector Δx_n [3]. Additionally, the step size Δh is also called the fixed time step, determining the iteration number and calculation time [35]. In [35], instead of inverting the matrix directly, the time-consuming inversion of the matrix is transformed to solve the linear equations for the reduction of fill-ins due to the sparsity of the power system. Although the fixed Jacobian matrix and high-order numerical method would increase iterative numbers and computation overhead, the high floating-point operation and parallel capability of the GPU can compensate for the disadvantage. The researchers tested the implementation on the Intel i9 CPU with the NVIDIA RTX400. The report achieved a speedup of 2.68×, compared with a single CPU platform.

Wang, Z.-Q. et al. [24] combined the advantages of the GPU and CPU to improve the parallelism of PF calculations. Researchers used the multi-threaded task level to realize parallelization with OpenMP [72]. The parts of the CPU are responsible for the initialization of sparse matrix and LU refactorization and schedule transmission between the CPU and GPUs, hiding the scheduling time in the task schedule. Furthermore, the computationally intensive tasks are distributed across multi-GPUs. The result of this method shows a speedup of over 100× compared to the open-source tool Pandapower after performing the hybrid CPU-GPU-based parallel program with sparsity patterns on both CPU and GPU [24]. Nevertheless, the high dependency on sparse matrix still limited the scalability of multi-GPUs. Although this method has high speed, the execution time is still slower on the Intel i7-8700k with two NVIDIA GTX 1080 GPUs. The main limitation is probably the communication overhead rather than the computing capabilities of the GPUs.

In [47], researchers proposed a parallel, multi-GPU, and multi-process FDPF method to accelerate the PF calculation. In the process of PF calculation, two hierarchy architectures, task parallelism, and data parallelism, are designed to optimize the FDPF solver parallelization [47]. In addition, GPUDirect technology is introduced to enhance the efficiency of data transmission between multi-GPUs. From (15) and (16), the calculation of voltage magnitude and phase angle can be divided into two independent tasks. These tasks are further managed by the processes in a distributed memory manner, avoiding a serious data race in shared memory, as shown in Figure 7. From Figure 7, each process managed its own GPUs, and then the one master and multi-salve communicating fashion is proposed for reduction of the design complexity and communication overhead. Furthermore, due to the introduction of GPUDirect technology, data migration seemed transparent between GPUs, resulting in performance improvements. Thus, this method achieved a speedup of 9×–33× on the HPC cluster with Intel Xeon and four NVIDIA Tesla V100 GPU nodes, compared to the single GPU counterpart on a large-scale power system.

Furthermore, this method overcame the scabble limitations and avoided the data race in [24], but it consumed more space than the method in [24]. Thus, this multi-process and multi-GPU method can further benefit the development of cloud storage.

3.5. The FPGA Architecture

The field programmable gate array (FPGA) is an array of configurable logic blocks (CLBs) linked through a programmable interconnection network with configurable inputs/outputs [73], as illustrated in Figure 8.

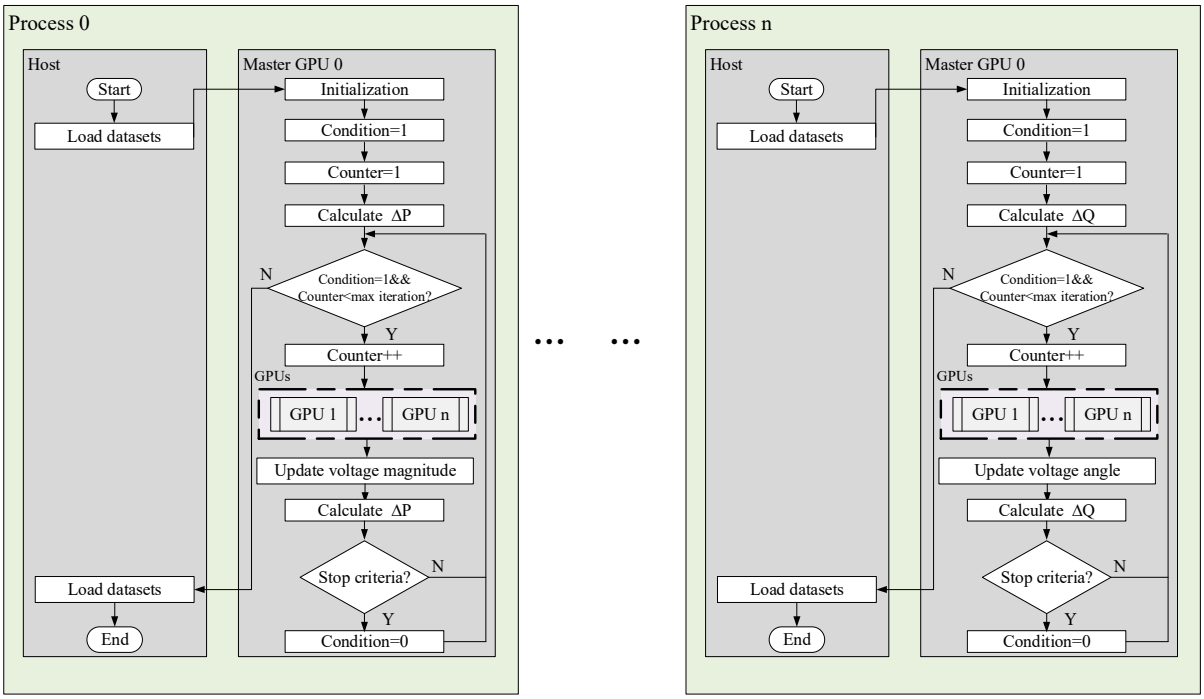


Figure 7. The architecture of multi-process and multi-GPUs [47].

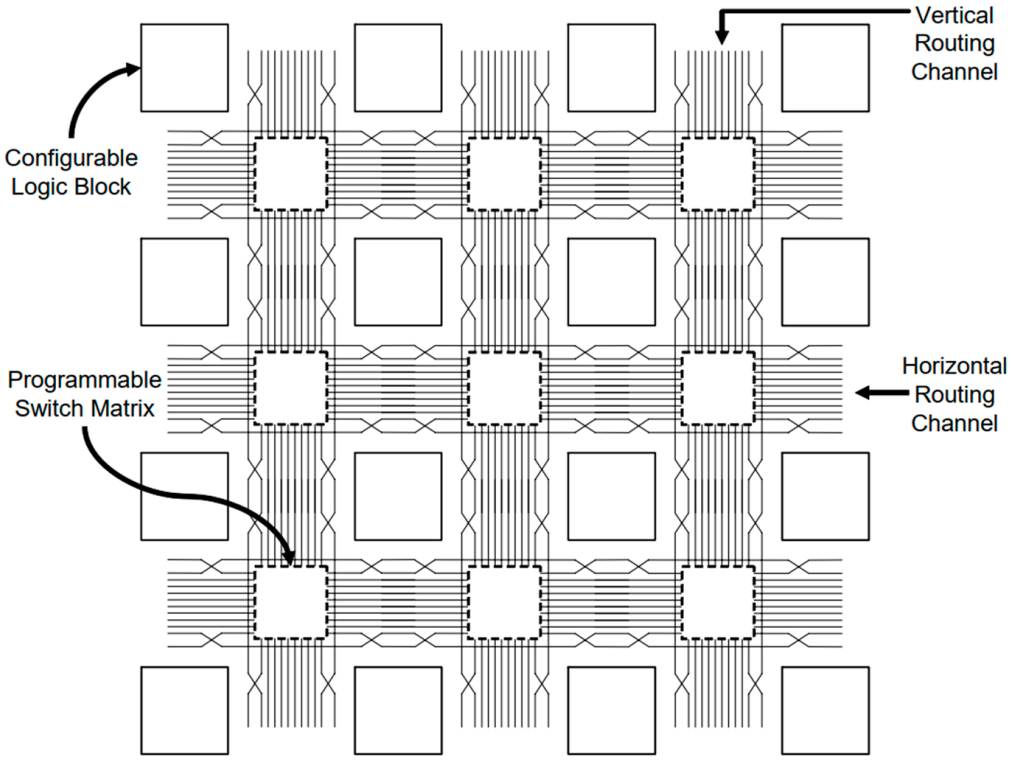


Figure 8. The FPGA architecture [74].

Typically, a programmer specifies the desired functionality in VHDL or Verilog, and this hardware description is then mapped to logic blocks and memory in the FPGA using a compiler [75]. In addition, the FPGA technology supports optimized use of parallel computations, including pipelining [76,77] and for-loop optimization [78], which provides a performance level higher than other fixed architectures such as microcontrollers [79].

3.6. The FPGA Parallel in PF Studies

Although the FPGA parallel method has been applied to power and industrial applications for a long time, most of them center on the acceleration of LU factorization for PF calculation and analysis.

Murach, M. et al. [75] used the FPGA to optimize the power flow analysis using the primal-dual interior point method (PDIPM). Essentially, the authors treated it as a general-purpose optimization problem and leveraged the pipeline and parallelism of the FPGA to improve the performance. Thus, the method in [75] can be classified as an iterative method, the CG method. In the process of LU decomposition, lp_solve [80], an open-source package, is used to accelerate LU factorization. This proposed approach indicated that LU performance can be enhanced by 6x and the overall large-scale OPF computation by at least 3× using FPGA technology over general-purpose workstations [75]. Wang, X.F. et al. proposed a novel partitioning scheme for the nonsymmetric Jacobian matrices appearing in Newton's method, resulting in highly efficient parallelization of LU factorization and the subsequent solution of the power flow equations [81]. 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 by the parallel DBBD LU factorization algorithm described in [82], after the Jacobian matrix is reordered into the DBBD form. Then the lower and upper triangular matrices were calculated by parallel forward reduction and backward substitution [83], as follows:

$$\begin{Bmatrix} J_{11} & 0 & 0 & 0 & J_{1n} \\ 0 & J_{22} & 0 & 0 & J_{2n} \\ 0 & 0 & J_{33} & 0 & J_{3n} \\ 0 & 0 & 0 & \dots & \dots \\ J_{n1} & J_{n2} & J_{n3} & \dots & J_{nn} \end{Bmatrix} = \begin{Bmatrix} L_{11} & 0 & 0 & 0 & 0 \\ 0 & L_{22} & 0 & 0 & 0 \\ 0 & 0 & L_{33} & 0 & 0 \\ 0 & 0 & 0 & \dots & 0 \\ L_{n1} & L_{n2} & L_{n3} & \dots & L_{nn} \end{Bmatrix} \cdot \begin{Bmatrix} U_{11} & 0 & 0 & 0 & U_{1n} \\ 0 & U_{22} & 0 & 0 & U_{2n} \\ 0 & 0 & U_{33} & 0 & U_{3n} \\ 0 & 0 & 0 & \dots & \dots \\ 0 & 0 & 0 & \dots & U_{nn} \end{Bmatrix} \tag{32}$$

where:

$$J_{kk} = L_{kk}U_{kk} \tag{33}$$

$$U_{kn} = L_{kk}^{-1}J_{kn} \tag{34}$$

$$L_{nk} = J_{nk}U_{kk}^{-1} \tag{35}$$

$$L_{nn}U_{nn} = J_{nn} - \sum_{k=1}^{n-1} L_{nk}U_{kn} \tag{36}$$

The authors tested the implementation on the ES2S180 platform and achieved almost the same speedup of 7×, compared to the EP20KE1500 counterpart.

4. Results and Comparison

Studies about parallel computing are shown in Table 1 on different hardware platforms. Specifically, Table 1 shows the speedup results for random access memory (RAM) on different hardware and in a matrix format with different methods. Then it indicates that the direct method is the most popular, and most of them focus on the optimization of linear system solvers. Generally, the accelerating efficacy of the direct method is better than the iterative method; however, the latter is suited for large-scale power systems on a single GPU platform due to the limitations of the hardware space.

Table 1. Comparison of performance and speedup of the reviewed work.

References	Method	Speedup	Matrix Format	Hardware Platform
[65]	Direct method	2.0× (MATPOWER counterpart)	Dense	CPU: Intel Core i5-2400 RAM: 12GB
[67]	Direct method	3.0× (CPU counterpart)	Dense	HPC cluster self-built RAM: not available
[26]	Iterative method	2.86× (MATPOWER counterpart)	Sparse	CPU: Intel Xeon E5607 GPU: NVIDIA Tesla M2070 RAM: 24GB
[27]	Iterative method	4× (MATPOWER counterpart)	Sparse	CPU: Intel Xeon E5607 GPU: NVIDIA Tesla M2070 RAM: 24GB
[28]	Iterative method	8× (MATPOWER counterpart)	Sparse	CPU: Intel Xeon E5607 GPU: NVIDIA Tesla M2070 RAM: 24GB
[16]	Direct method	10×–40× (CPU counterpart)	Dense	CPU: Intel i7-4500U GPU: NVIDIA GeForce GT745 RAM: not available
[31]	Direct method	7× (CPU counterpart)	Dense	CPU: Intel Xeon Quad GPU: NVIDIA TESLA C1006 RAM: 8GB
[21]	Direct method	4.16× (MATPOWER counterpart)	Sparse	CPU: Intel Xeon E5-2620 GPU: NVIDIA Ge-Force Titan RAM: 32GB
[35]	Direct method	2.68× (CPU counterpart)	Sparse	CPU: Intel i9 GPU: NVIDIA RTX400 RAM: 16GB
[24]	Direct method	100× (PandaPower counter)	Sparse	CPU: Intel i7-8700k GPU: NVIDIA GTX 1080 GPU: NVIDIA GTX 1080 RAM: 32GB
[47]	Direct method	9×–33× (Single GPU counterpart)	Dense	CPU: Intel Xeon GPU: NVIDIA Tesla V100 (4 nodes) RAM: 32GB
[75]	Direct method	3× (CPU counterpart)	Sparse	FPGA RAM: not available
[81]	Direct method	7× (EP20KE1500 counterpart)	Sparse	ES2S180 RAM: 9,383,040 bits

Furthermore, researchers usually need to design preconditioners to reduce the condition number of the matrix for the convergence of the PF calculation while using the iterative method. Table 1 also presents the fact that the speedup of sparse formats is not always

faster than dense formats. Although sparse formats can reduce computation complexity, they also increase the difficulties of parallelization due to their high dependency. During the LU decomposition, the sparse matrix might generate many fill-ins without reordering the sparse matrix, causing degradation of program performance. Although the dense matrix can consume much space, it can be benefited by the development of cloud storage for the improvement of parallelization. In addition, the hybrid CPU-GPU architecture is a heterogeneous programming model, and the performance, however, cannot have great improvement even if the platform is equipped with better GPU devices. It means that the main limitation is the communication overhead rather than the computing capability of the GPUs. Thus, Table 1 presents parallel devices that indeed aid acceleration when performing power flow calculations and provides researcher guidance on the PF calculation on different parallel platforms, as follows:

- The direct method is usually more robust than the iterative method.
- The direct method of sparse format is suited for small-memory platforms, such as PCs, laptops, and workstations.
- The iterative method performs better on almost all kinds of platforms, but it needs a preconditioner for enhanced stability in PF calculation.
- The direct method of dense format can provide an overall performance improvement for large-scale power systems on large-memory parallel platforms.

5. Conclusions

In this paper, an overview of the current research regarding power flow calculations on different parallel computing devices is presented.

First, the background of PF calculation is presented, including the PF model, PF methods, and linear system solver, which lay the parallelization foundation for PF calculation and analysis. Then the different models of parallel devices and their corresponding direct and iterative methods are introduced in this paper. In addition, the pros and cons of these parallel methods are discussed, and the speedup results are summarized in a table based on the different parallel devices. Specifically, the PF calculation based on the multi-thread method reduces the communication overhead in a shared-memory fashion. Data access is almost transparent for all threads on the same single parallel device. As for heterogeneous parallel platforms, the computationally intensive tasks are distributed on different devices for acceleration. The host, however, is responsible for the scheduling and management of the devices. This method indeed improves the overall performance of PF calculations but limits improvement when multiple transmissions occur between host and devices, losing the advantage of parallelization and the high capability of floating operation on the GPUs.

Although significant speedup achievements were sufficiently convincing in this study, the optimum algorithm, direct or iterative, has not been decided in this paper, which depends on the specific situations and hardware platforms. Specifically, the direct method is more robust than the iterative method without the preconditioner on the PF calculation. The iterative method, however, is the optimal method for a large-scale power system. In future research, it is possible to extend the parallel optimization approaches discussed in [29,30] to the linear system solver used in PF calculation, leveraging its inherent robustness and parallel nature. Additionally, the quantitative analysis and algorithmic optimization presented in [29,30] can be further explored and applied to the design of various parallel architectures in PF calculation, aiming to achieve enhanced performance. Thus, this paper summarizes the main features and speedup results of power flow calculation on the different parallel devices. It demonstrated that the parallel device indeed aids speedup and provided researchers with guidance on how to exploit the parallelism of the extremely massive numerical computations for the future large-scale and complex power system.

Author Contributions: Writing—original draft, S.G.A., L.Z. and S.A.A. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: Not Applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Yoon, D.-H.; Han, Y. Parallel Power Flow Computation Trends and Applications: A Review Focusing on GPU. *Energies* **2020**, *13*, 2147. [CrossRef]
2. Grainger, J.; Stevenson, W. *Power System Analysis*; McGraw-Hill: New York, NY, USA, 1999; pp. 329–368.
3. Wang, M.; Xia, Y.; Chen, Y.; Huang, S. GPU-based power flow analysis with continuous Newton's method. In Proceedings of the 2017 IEEE Conference on Energy Internet and Energy System Integration (EI2), Beijing, China, 26–28 November 2017; pp. 1–5.
4. Wang, X.; Song, Y.; Irving, M. *Modern Power Systems Analysis*; Springer: New York, NY, USA, 2008.
5. Khairy, M.; Wassal, A.G.; Zahran, M. A survey of architectural approaches for improving GPGPU performance, programmability, and heterogeneity. *J. Parallel Distrib. Comput.* **2019**, *127*, 65–88. [CrossRef]
6. Falcão, D.M. High performance computing in power system applications. In Proceedings of the International Conference on Vector and Parallel Processing, Porto, Portugal, 25–27 September 1996; Springer: Berlin, Germany, 1996; pp. 1–23.
7. Ramesh, V. On distributed computing for on-line power system applications. *Int. J. Electr. Power Energy Syst.* **1996**, *18*, 527–533. [CrossRef]
8. Baldick, R.; Kim, B.H.; Chase, C.; Luo, Y. A fast distributed implementation of optimal power flow. *IEEE Trans. Power Syst.* **1999**, *14*, 858–864. [CrossRef]
9. 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.* **2003**, *65*, 169–177. [CrossRef]
10. 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, Detroit, MI, USA, 24–28 July 2011; pp. 1–8.
11. Saad, Y. *Iterative Methods for Sparse Linear Systems*, 2nd ed.; PWS: Boston, MA, USA, 2004.
12. Dag, H.; Semlyen, A. A new preconditioned conjugate gradient power flow. *IEEE Trans. Power Syst.* **2003**, *4*, 1248–1255. [CrossRef]
13. Chen, Y.; Shen, C. A Jacobian-free Newton-GMRES(m) method with adaptive preconditioner and its application for power flow calculations. *IEEE Trans. Power Syst.* **2006**, *21*, 1096–1103. [CrossRef]
14. Flueck, A.J.; Chiang, H.-D. Solving the nonlinear power flow equations with a Newton process and GMRES. *IEEE Trans. Power Syst.* **1998**, *13*, 267–273. [CrossRef]
15. Davis, T.A. *Direct Methods for Sparse Linear Systems*; SIAM: Philadelphia, PA, USA, 2006.
16. Singh, J.; Aruni, I. Accelerating Power Flow Studies on Graphics Processing Unit. In Proceedings of the 2010 Annual IEEE India Conference (INDICON), Kolkata, India, 17–19 December 2010; pp. 1–5.
17. Guo, C.; Jiang, B.; Yuan, H.; Yang, Z.; Wang, L.; Ren, S. Performance Comparison of Parallel Power Flow Solvers on GPU System. In Proceedings of the 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, Seoul, Republic of Korea, 19–22 August 2012; pp. 232–239.
18. Li, X.; Li, F.-X.; Clark, J.M. Exploration of multifrontal method with GPU in power flow computation. In Proceedings of the 2013 IEEE Power & Energy Society General Meeting, Vancouver, BC, Canada, 21–25 July 2013; pp. 1–5.
19. Gopal, A.; Niebur, D.; Venkatasubramanian, S. DC Power Flow Based Contingency Analysis Using Graphics Processing Units. In Proceedings of the 2007 IEEE Lausanne Power Tech, Lausanne, Switzerland, 1–5 July 2007; pp. 731–736.
20. 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. *IEEE Trans. Smart Grid* **2020**, *11*, 1833–1844. [CrossRef]
21. Huang, S.; Dinavahi, V. Performance analysis of GPU-accelerated fast decoupled Power Flow using direct linear solver. In Proceedings of the 2017 IEEE Electrical Power and Energy Conference (EPEC), Saskatoon, SK, Canada, 22–25 October 2017; pp. 1–6.
22. Schäfer, F.; Braun, M. An efficient open-source implementation to compute the jacobian matrix for the Newton-Raphson Power flow algorithm. In Proceedings of the 2018 IEEE PES Innovative Smart Grid Technologies Conference Europe (ISGT-Europe), Sarajevo, Bosnia and Herzegovina, 21–25 October 2018; pp. 1–6.
23. Gnanavignesh, R.; Shenoy, U.J. Parallel Sparse LU Factorization of Power Flow Jacobian using GPU. In Proceedings of the TENCON 2019–2019 IEEE Region 10 Conference (TENCON), Kochi, India, 17–20 October 2019; pp. 1857–1862.
24. 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* **2021**, *27*, 100483. [CrossRef]
25. 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* **2011**, *58*, 702–706. [CrossRef]
26. Li, X.; Li, F.; Yuan, H.; Cui, H.; Hu, Q. GPU-Based Fast Decoupled Power Flow with Preconditioned Iterative Solver and Inexact Newton Method. *IEEE Trans. Power Syst.* **2017**, *31*, 2695–2703. [CrossRef]
27. Li, X.; Li, F. GPU-based power flow analysis with Chebyshev Preconditioner and Conjugate gradient method. Elect. *Power Syst. Res.* **2014**, *116*, 87–93. [CrossRef]
28. Li, X.; Li, F. GPU-based two-step preconditioning for conjugate gradient method in power flow. In Proceedings of the 2015 IEEE Power & Energy Society General Meeting, Denver, CO, USA, 26–30 July 2015; pp. 1–5.

29. Zhou, Y.; He, F.; Hou, N.; Qiu, Y. Parallel ant colony optimization on multi-core SIMD CPUs. *Future Gener. Comput. Syst.* **2018**, *79*, 473–487. [CrossRef]
30. Zhou, Y.; He, F.; Qiu, Y. Dynamic strategy based parallel ant colony optimization on GPUs for TSPs. *Sci. China Inf. Sci.* **2017**, *60*, 068102. [CrossRef]
31. Sooknanan, D.J.; Joshi, A. GPU Computing Using CUDA in the Deployment of Smart Grids. In Proceedings of the 2016 SAI Computing Conference, London, UK, 13–15 July 2016; pp. 1260–1266.
32. 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**, *20*, 720–755. [CrossRef]
33. Schenk, O.; Gärtner, K. Solving unsymemetric sparse systems of linear equations with PARDISO. *Future Gener. Comput. Syst.* **2004**, *20*, 475–487. [CrossRef]
34. Christen, M.; Schenk, O.; Burkhart, H. General-purpose Sparse Matrix Building Blocks Using the NVIDIA CUDA Technology Platform. In Proceedings of the First Workshop on General Purpose Processing on Graphics Processing Units, Boston, MA, USA, 4 October 2007; p. 32.
35. Zeng, L.; Alawneh, S.G.; Arefifar, S.A. GPU-based Sparse Power Flow Studies with Modified Newton's Method. *IEEE Access* **2021**, *9*, 153226–153239. [CrossRef]
36. Zhou, G.; Feng, Y.-J.; Bo, R.; Zhang, T. GPU-accelerated sparse matrices parallel inversion algorithm for large-scale power systems. *Int. J. Electr. Power Energy Syst.* **2019**, *111*, 34–43. [CrossRef]
37. Wu, J.Q.; Bose, A. Parallel solution of large sparse matrix equations and parallel power flow. *IEEE Trans. Power Syst.* **1995**, *10*, 1343–1349.
38. He, K.; Sheldon, X.-D.T.; Wang, H.; Shi, G. GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis. *IEEE Trans. Very Large-Scale Integr. VLSI Syst.* **2016**, *24*, 1140–1150. [CrossRef]
39. Peng, S.; Sheldon, X.-D.T. GLU3.0: Fast GPU-based Parallel Sparse LU Factorization for Circuit Simulation. *IEEE Des. Test* **2020**, *37*, 78–90. [CrossRef]
40. Chen, X.; Ren, L.; Wang, Y.; Yang, H. GPU-Accelerated Sparse LU Factorization for Circuit Simulation with Performance Modeling. *IEEE Trans. Parallel Distrib. Syst.* **2015**, *26*, 786–795. [CrossRef]
41. Lee, W.-K.; Achar, R.; Nakhla, M.S. Dynamic GPU Parallel Sparse LU Factorization for Fast Circuit Simulation. *IEEE Trans. Very Large-Scale Integr. VLSI Syst.* **2018**, *26*, 2518–2529. [CrossRef]
42. Gao, J.-Q.; Chen, Q.; He, G.-X. A thread-adaptive sparse approximate inverse preconditioning algorithm on multi-GPUs. *Parallel Comput.* **2021**, *101*, 102724. [CrossRef]
43. 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. In Proceedings of the 50th International Conference on Parallel Processing, Lemont, IL, USA, 9–12 August 2021.
44. Ma, W.; Hu, Y.; Yuan, W.; Liu, X. Developing a multi-GPU-enabled preconditioned GMRES with inexact triangular solves for block sparse matrices. *Math. Probl. Eng.* **2021**, *2021*, 6804723. [CrossRef]
45. Lin, S.; Xie, Z. A Jacobi_PCG solver for sparse linear systems on multi-GPU cluster. *J. Supercomput.* **2017**, *73*, 433–454. [CrossRef]
46. Ding, N.; Liu, Y.; Williams, S.; Li, X.-Y.S. A Message-Driven, Multi-GPU Parallel Sparse Triangular Solver. In Proceedings of the 2021 SIAM Conference on Applied and Computational Discrete Algorithms (ACDA21), Virtual, 19–21 July 2021; pp. 147–159.
47. Zeng, L.; Alawneh, S.G.; Arefifar, S.A. Parallel Multi-GPU Implementation of Fast Decoupled Power Flow Solver with Hybrid Architecture. *Comput. Clust.* **2022**. *under review*.
48. Glover, J.-D.; Overbye, T.-J.; Sarma, M.-S. *Power System Analysis & Design*, 6th ed.; Cengage Learning: Boston, MA, USA, 2015; pp. 349–353.
49. Kundur, P.; Balu, N.J.; Lauby, M.G. *Power System Stability and Control*; McGraw-hill: New York, NY, USA, 1994; Volume 7.
50. Stoot, B.; Alsac, O. Fast decoupled load flow. *IEEE Trans. Power Appar. Syst.* **1974**, *PAS-93*, 859–869. [CrossRef]
51. Chapra, S.-C. *Applied Numerical Methods with MATLAB® for Engineers and Scientists*, 3rd ed.; McGraw-Hill: New York, NY, USA, 2012; pp. 260–261.
52. 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.* **2016**, *83*, 33–39. [CrossRef]
53. Lau, K.; Tylavsky, D.J.; Bose, A. Coarse grain scheduling in parallel triangular factorization and solution of power system matrices. *IEEE Trans. Power Syst.* **1991**, *6*, 708–714. [CrossRef]
54. Amano, M.; Zecevic, A.; Siljak, D. An improved block-parallel Newton method via epsilon decompositions for load-flow calculations. *IEEE Trans. Power Syst.* **1996**, *11*, 1519–1527. [CrossRef]
55. El-Keib, A.; Ding, H.; Maratukulam, D. A parallel load flow algorithm. *Electr. Power Syst. Res.* **1994**, *30*, 203–208. [CrossRef]
56. Fukuyama, Y.; Nakanishi, Y.; Chiang, H.D. Parallel power flow calculation in electric distribution networks. In Proceedings of the 1996 IEEE International Symposium on Circuits and Systems, Circuits and Systems Connecting the World (ISCAS 96), Atlanta, GA, USA, 15 May 1996; Volume 1, pp. 669–672.
57. Luo, C.; Zhang, K.; Salinas, S.; Li, P. SecFact: Secure Large-scale QR and LU Factorizations. *IEEE Trans. Big Data* **2021**, *7*, 796–807. [CrossRef]
58. Golub, G.H.; Van Loan, C.F. *Matrix Computation*, 4th ed.; John Hopkins University Press: Baltimore, MD, USA, 2013; pp. 233–255.

59. Buttari, A.; Langou, J.; Kurzak, J.; Dongarra, J. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.* **2009**, *35*, 38–53. [CrossRef]
60. 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.* **2009**, *36*, 1–26.
61. 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.* **2017**, *8*, 1406–1416. [CrossRef]
62. Garcia, N. Parallel power flow solutions using a biconjugate gradient algorithm and a Newton method: A GPU-based approach. In Proceedings of the Power and Energy Society General Meeting, Providence, RI, USA, 25–29 July 2010; pp. 1–4.
63. Geer, D. Chip Makers Turn to Multicore Processor. *Computer* **2005**, *38*, 11–13. [CrossRef]
64. David, B.K.; Hwu, W.-M.W. *Programming Massively Parallel Processors: A Hands-on Approach*, 3rd ed.; Elsevier: Cambridge, MA, USA, 2010; pp. 3–6.
65. Ahmadi, A.; Jin, S.; Smith, M.C.; Collins, E.R.; Goudarzi, A. Parallel Power Flow based on OpenMP. In Proceedings of the 2018 North American Power Symposium (NAPS), Fargo, ND, USA, 9–11 September 2018; pp. 1–6.
66. 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 Trans. Power Syst.* **2011**, *26*, 12–19. [CrossRef]
67. Ao, L.; Cheng, B.; Li, F. Research of Power Flow Parallel Computing Based on MPI and P-Q Decomposition Method. In Proceedings of the 2010 International Conference on Electrical and Control Engineering, Tuxtla Gutierrez, Mexico, 25–27 June 2010; pp. 2925–2928.
68. Zhang, Y.; Owens, J.D. A quantitative performance analysis model for GPU architectures. In Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture, San Antonio, TX, USA, 12–16 February 2011; pp. 382–393.
69. Cook, S. *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*; Morgan Kaufmann: Burlington, MA, USA, 2012; pp. 84–89.
70. Alvarado, F.L. Computional complexity in power systems. *IEEE Trans. Power Appar. Syst.* **1976**, *95*, 1028–1037. [CrossRef]
71. Zhang, B.; Chen, S. *Advanced Power System Network Analysis*; Tsinghua University Press: Beijing, China, 1996.
72. Dagum, L.; Menon, R. Openmp: An industry standard api for shared-memory programmming. *IEEE Comput. Sci. Eng.* **1998**, *5*, 46–55. [CrossRef]
73. Kuon, I.; Tessier, R.; Rose, J. FPGA Architecture: Survey for DC power flow solution using LabVIEW language. *AADV Electr. Eng.* **2012**, *10*, 68–74.
74. Yaseen, A.Y.; Abbood, A.A. Study of Power System Load Flow Using FPGA and LabView. *Eng. Technol. J.* **2020**, *38*, 690–697. [CrossRef]
75. Murach, M.; Nagvajara, P.; Johnson, J.; Nwankpa, C. Optimal power flow utilizing FPGA technology. In Proceedings of the 37th Annual North American Power Symposium, Ames, IA, USA, 23–25 October 2005; pp. 97–101.
76. Wills, A.; Mills, A.; Ninness, B. FPGA implementation of an interior-point solution for linear model predictive control. *IFAC Proc. Vol.* **2011**, *44*, 14527–14532. [CrossRef]
77. Jerez, J.L.; Constantinides, G.A.; Kerrigan, E.C.; Ling, K.-V. Parallel Mpc for real-time FPGA-based implementation. *IFAC Proc. Vol.* **2014**, *59*, 3238–3251. [CrossRef]
78. Ling, K.V.; Wu, B.F.; Maciejowski, J.M. Embeded model predictive control (MPC) using a FPGA. *IFAC Proc. Vol.* **2008**, *41*, 15250–15255. [CrossRef]
79. Abughalieh, K.M.; Alawneh, S.G. A survey of parallel implementations for model predictive control. *IEEE Access.* **2019**, *7*, 34348–34360. [CrossRef]
80. Mehrotra, S. On the Implemetation of a Primal-Dual Interior Point Method. *SIAM J. Optim.* **1992**, *2*, 575–601. [CrossRef]
81. Wang, X.F.; Ziavras, S.G.; Nwankpa, C.; Johnson, J. Nagvajara Prawat. Parallel Solution of Newton's Power Flow Equations on Configurable Chips. *Electr. Power Energy Syst.* **2007**, *29*, 422–431. [CrossRef]
82. Wang, X.; Ziavras, S.G. Parallel LU. Factorization of sparse matrices on FPGA-based configurable computing engines. *Concurr. Comput. Pract. Exp.* **2004**, *16*, 319–343. [CrossRef]
83. Wang, X.; Ziavras, S.G. Parallel direct solution of linear equations on FPGA-based machines. In Proceedings of the Seventieth IEEE International Parallel and Distributed Processing Symposium IPDPS2003 (Eleventh IEEE International Workshop on Parallel and Distributed Real-Time Systems), Nice, France, 22–26 April 2003.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.