2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications

Performance Comparisons of Parallel Power Flow Solvers on GPU System

Chunhui Guo, Baochen Jiang*, Hao Yuan, Zhiqiang Yang
School of Mechanical, Electrical & Information Engineering
Shandong University at Weihai
Weihai, 264209, P. R. China
guochunhui3024@mail.sdu.edu.cn, {jbc, yuanhao}@sdu.edu.cn,
yangzhiqiang@ieslab.com.cn

Li Wang, Shangping Ren†
Department of Computer Science
Illinois Institute of Technology
Chicago, IL 60616, USA
{lwang64, ren}@iit.edu

***Abstract*—This paper transforms sequential power flow problem to a parallel problem and solves it on GPU. In particular, we implement parallel Gauss-Seidel solver, Newton-Raphson solver, and P-Q decoupled solver using CUDA (Compute Unified Device Architecture) on GPU. The aim is to investigate the performance of the three different parallel power flow solvers. We use four IEEE standard power systems and one actual running power system from Shangdong Province as the test cases when comparing the speedups that a GPU system can provide. The results show that Newton-Raphson solver has the best speedup when it is operated on GPU, Gauss-Seidel solver performs the worst, and P-Q decoupled solver is in the middle. The test results also indicate that when the size of the system is small, GPU does not seem to have advantages over CPU from computation time perspective. However, as the size of the system increases, the advantages of GPU becomes more clear. For instance, when the system has close to one thousand bus counts, the GPU can provide as high as over fifty-three times speedup.**

***Index Terms*—Power Flow; GPU; CUDA; Newton-Raphson Method; P-Q Decoupled Method; Gauss-Seidel Method; Cyber-Physical Systems**

I. Introduction

Power flow is often used to describe the steady state of the power system. It gives a power system's parameter values (i.e., the magnitude and angle of the voltage, real and reactive power, etc.) when the system is in a steady state. A power system's power flow is calculated based on the power network structure and running status (i.e., the power of generator, the voltage of the slack bus, etc.) [1], [2]. With real-time system power flow information, we can predict whether the changes in the load and network structure may compromise the safety of the power system, and whether the elements in the system, i.e., lines, transformers, etc., are overloaded, and decide what preventive measures should take place in the event of overloading, etc. In addition, through power flow analysis, we can also verify if all running requirements made in the planning stage are met. Hence, it is not difficult to see that the power systems' power flow not only plays an important role in optimizing real-time control of operating power systems, it also provides essential information for design future extensions of existing power system and new power systems.

Aside from the importance mentioned above, power flow is also fundamental to short circuit diagnostic analysis, transient stability computing, etc. Hence, accelerated power flow calculation will also speedup the process of these analysis and computation [3]. However, due to the complexity of large size power systems, thousands of equations are involved in the power flow calculation. Therefore, our goal is to increase the computing speed so that the real-time flow analysis can become possible.

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA for graphics processing. In CUDA programming model, CPU works as a host, and GPU works as an assistant processor or device. The computing architecture is considered to be heterogeneous. In this architecture, CPU mainly processes highly logical transactions and serial computation because of its powerful control logics. On the other hand, as GPU has abundant computation resources but uses simple logic, it is used to execute highly parallel tasks. The functions that are executed by GPU are called *kernel*. Unlike the functions which are executed on CPU, when the kernel is called, it is executed multiple times by the CUDA thread; while the functions executed by CPU are executed only once.

The rest of the paper is organized as follows. The mathematical presentation of power flow is given in Section III. In Section IV, we analyze and compare three algorithms in power flow: Gauss-Seidel algorithm, Newton-Raphson algorithm, and P-Q decoupled algorithm. The parallel implementation of these three algorithms is shown in Section V. In Section VI, we evaluate the speedup performance of these three algorithms using four IEEE standard power systems and one real power system. Finally, we conclude and point out future work in Section VII.

II. Related Work

Parallel computing is a good solution to increasing the computation speed. Currently, a commonly used approaches to accelerating the speed of power flow solver are through distributed multi-threading, parallel machines, or distributed

*Corresponding author: jbc@sdu.edu.cn
†The work is supported in part by NSF CAREER CNS 0746643 and NSF CNS 1018731.

978-0-7695-4824-1/12 $26.00 © 2012 IEEE
DOI 10.1109/RTCSA.2012.36

systems [4], [5], [6]. However, these approaches are often constrained by the need of special hardware support, high cost, and limited speed improvement.

GPUs have been developed rapidly in recent years. Because of the high computing efficiency and low prices, they are widely used in sparse matrix solvers [7], finite element analysis [8], electromagnetic [9], biology [10], etc. In addition, GPUs are also used in parallel power flow. For instance, in [11], Gopal et al. implemented the failure analysis for power systems on GPU. In [12], Xia et al. presented a simplified Newton-Raphson power flow solver. Singh et al. [13] discussed pre-processing steps in using Newton power flow solver. Garcia [14] showed GPU-based approach to the power flow problem that integrate biconjugate gradient algorithm and Newton method; while Vilacha et al. [15] presented Jacobi method for power flow problem based on SIMD-processors. The work discussed above compared the execution time of obtaining the power flow on both CPU and GPU and showed that a good speedup ratio can be achieved. However, they did not compare the execution time of different algorithms after parallelism. The main difference between our work and the work discussed above is that we implement three parallel Gauss-Seidel solver, Newton-Raphson solver, and P-Q decoupled solver, and compare their speedup ratios when running on GPU and find out the appropriate approach for solving the power flow problem.

III. Power Flow Model and Problem Definition

In a power system, the relationship between bus injection current and bus voltage is represented by bus admittance matrix. That is

$$Y_{ij} = |Y_{ij}|(\cos\theta_{ij} + j\sin\theta_{ij}) = G_{ij} + jB_{ij} \tag{1}$$

where Y_{ij} whose magnitude and angle are $|Y_{ij}|$ and θ_{ij}, respectively, refers to the admittance between bus i and bus j, and it can be further divided into electrical conductance G_{ij} and susceptance B_{ij}. When i is equal to j, Y_{ij} becomes the self-admittance. For the power systems with n independent buses, the dimensions of their bus admittance matrix is $n \times n$.

For bus i, its complex power is

$$S_i = V_i \sum_{k=1}^{n} \overset{*}{Y_{ij}} \overset{*}{V_k} = P_i - jQ_i \tag{2}$$

where "*" refers to conjugate complex, and its voltage V_i is

$$V_i = |V_i|(\cos\delta_i + j\sin\delta_i) \tag{3}$$

where $|V_i|$ and δ_i refer to magnitude and angle of voltage, respectively.

In addition, complex power can be further divided into real power P_i, and reactive power Q_i, where

$$P_i = \sum_{k=1}^{n} |V_i V_k Y_{ik}| \cos(\theta_{ik} + \delta_k + \delta_i) \tag{4}$$

and

$$Q_i = -\sum_{k=1}^{n} |V_i V_k Y_{ik}| \sin(\theta_{ik} + \delta_k + \delta_i) \tag{5}$$

In (4) and (5), the values of both Y_{ik} and θ_{ik} are known. For different types of buses, only two out of four variables' values are known. Therefore, for both (4) and (5), they are non-linear, and they are used as power equation in the power system.

For power systems with n independent bus, the number of (4) and (5) is n, both. Therefore, in order to obtain a power system's power flow, we need to solve the nonlinear equation systems with $2n$ equations.

IV. Power Flow Solver

The essence of obtaining the power flow of a power system is to solve the set non-linear equations given by (4) and (5) in Section III. In our work, we take the iteration approach. The commonly used iteration methods for solving the power flow problem include Gauss-Seidel method, Newton-Raphson method, and P-Q decoupled method.

The main flow for obtaining a power system's power flow are as follows:

1) Obtain the aggregate data from a power system. The aggregate data includes bus data, branch data, and identifiers of slack bus, etc. The bus data includes identifier, voltage, real and reactive power, and the branch data includes starting bus identifier, the ending bus identifier, resistance, reactance, and transformation ratio, etc.
2) Rearrange the bus by the order of PQ bus (the buses with known real and reactive power), PV bus (the buses with real power and voltage), slack bus (the buses with known voltage and zero angle, as well as those which are used to guarantee that generated power is equal to the consumed power in the power system). For example, if a power system has n buses, and m buses among them are PQ buses, after rearrangement, the first m buses are PQ buses, the $(m+1)$th to $(n-1)$th buses are PV buses, and the last n buses are slack buses.
3) Calculate the admittance matrix for each bus.
4) Initialize the voltage value and angle for each bus, and set the iteration times to zero.
5) Iteratively calculate the voltage value, angle, real power, reactive power, and the increment of each parameter between current iteration and previous iteration until each parameter obtained is smaller than the predefined error ε.
6) Calculate the power of slack buses and branches.

A. The Characteristics of Three Iterative Algorithms

Gauss-Seidel algorithm: the Gauss-Seidel algorithm uses the latest iteration value, i.e., once a new value of a parameter has been obtained, the new value will be used in the next iteration to accelerate the speedup.

The iterative format of Gauss-Seidel algorithm using bus voltage V_i and reactive power Q_i is given below:

$$V_i^{(k+1)} = \frac{1}{Y_{ii}}\left(\frac{P_i - jQ_i}{\overset{*}{V_i^{(k)}}} - \sum_{j=1}^{i-1} Y_{ij}V_j^{(k+1)} - \sum_{j=i+1}^{n} Y_{ij}V_j^{(k)}\right) \quad (6)$$

$$Q_i^{(k)} = -Im[\overset{*}{V_i^{(k)}}\left(\sum_{j=1}^{i-1} Y_{ij}V_j^{(k+1)} + \sum_{j=i}^{n} Y_{ij}V_j^{(k)}\right)] \quad (7)$$

where i and j denote the identifiers of buses, k is the number of iterations, n is the total number of buses, the symbol "*" above the letter refers to the conjugate complex number, and the *Im* is the imaginary part of a complex number.

Newton-Raphson algorithm: in the Newton-Raphson algorithm, for each iteration, the voltage correction ΔU, voltage angle correction $\Delta\delta$, real power correction ΔP, and reactive power ΔQ are used in the power flow equation. Hence, the power equation is expanded by using Taylor series and only the linear part of the correction is kept. In this way, the non-linear equation is converted into linear equation of correction, and it is called correction equation. In the correction equation, its coefficient matrix is called Jacobian matrix, and each item in the correction equation is the first partial derivative. Therefore, we only need to solve the linear equation, i.e., correction equation, and modify the corresponding power equation variables after obtaining the correction value. For each iteration, the Jacobian matrix needs to be recalculated to accelerate the speedup.

As the bus voltage can be presented in different ways, therefore, Newton-Raphson algorithm has two presentations, i.e., the *polar* form representation and *rectangular* form representation. The calculation in the rectangular form is simple, while the calculation in polar form is complex because of the trigonometric functions involved in the calculation. However, the number of correction equations in polar form is less comparing to the rectangular form representation. In our work, we use polar form in the Newton-Raphson algorithm.

P-Q decoupled algorithm: the P-Q decoupled algorithm is a simplified version of Newton-Raphson algorithm. It uses the imaginary part of the bus admittance matrix to replace Jacobian matrix to get linear real power correction equations and linear reactive power correction equations. In addition, the coefficient matrix in the correction equations remains unchanged. Therefore, the computation cost of Jacobian matrix in each iteration is decreased, however, the convergence speed slows down. Additionally, P-Q decoupled algorithm must meet the simplification requirements. They are: (1) the first order partial derivative of real power increment with respect to the voltage is much larger than the first order partial derivative with respect to the angle; (2) the first order partial derivative of reactive power increment with respect to the angle is much larger than the first order partial derivative with respect to the voltage; and (3) for each element in the bus admittance matrix, the value of real part is much smaller than the value of imaginary part.

In summary, we have the following observations when the three algorithms operate sequentially:

- Computation speed: P-Q decoupled algorithm is the fastest when the size of the system is not too small, while Gauss-Seidel algorithm is the slowest.
- Total number of iterations: Newton-Raphson algorithm requires the least iterations, while Gauss-Seidel algorithm requires the most iterations.
- Requirement of the initial values: For Gauss-Seidel algorithm, its initial values can be random. However, for both Newton-Raphson algorithm and P-Q decoupled algorithm, their initial values are usually fixed, i.e., the voltage magnitude is 1.0, and the voltage angle is 0.0.
- Pratical use: Newton-Raphson algorithm is the most commonly used, while Gauss-Seidel algorithm is the least commonly used.

B. Speedup Analysis

For each algorithm, as the number of iterations before and after parallelism remains the same, and for each iteration, the speedup of the algorithm is also the same, therefore, we only need to analyze the speedup in one iteration. In our work, we use the number of multiplication to estimate the computation cost. If the power system has total n buses, and m out of n buses are PQ buses, we assume values of n and m are in the same order of magnitude for large scale power system. When approximating the computation cost, we use n to replace m, and only keep the quadratic parts in the equation.

The total number of multiplications for Gauss-Seidel solver is

$$\sum_{1}^{n-1}(n-1+2) + \sum_{1}^{n-m-1}(n+1) \approx n^2 \quad (8)$$

while the total number of multiplications after parallelism becomes

$$\sum_{1}^{n-1}(1+2) + \sum_{1}^{n-m-1}(1+1) \approx 5n \quad (9)$$

Suppose the computation cost of trigonometric function is r times as much as multiplication. As the value of r is usually much larger than 1, hence, we will keep r but ignore other small coefficients when approximating the computation cost.

The total number of multiplications for Newton-Raphson solver to calculate Jacobian matrix is

$$\sum_{1}^{(n+m)^2}(2r+4) + \sum_{1}^{n+m}(2r+3) \approx 8rn^2 \quad (10)$$

and the total number of multiplications after parallelism becomes

$$(2r+4) + \sum_{1}^{n+m}(2r+3) \approx 4rn \quad (11)$$

As Newton-Raphson solver applies Gaussian elimination method to solve linear equations, therefore, the total number of multiplications is

$$\sum_{k=1}^{n}(n+1-k)^2 + \sum_{k=n}^{1}(n-k) \approx 4n^2/3 \quad (12)$$

and the total number of multiplications after parallelism becomes

$$(2n+1)+n \approx 3n \tag{13}$$

The speedup is calculated by dividing the number of multiplications after parallelism by the number of multiplications before parallelism. The speedup value for each solver is shown in Table I, where n refers to the bus count of power system. From Table I, we can see that under parallelism, the Newton-Raphson solver performs the best, while the Gauss-Seidel solver performs the worst.

TABLE I
THEORETICAL SPEEDUPS OF DIFFERENT POWER FLOW SOLVERS

Power Flow Solver	Speedup
Gauss-Seidel Solver	$0.2n$
Newton-Raphson Solver	$2n$
P-Q Decoupled Solver	$0.4n$

It is worth pointing out that the the communication cost between CPU memory and GPU memory is not considered when theoretically calculating the speedup value for each solver.

V. TRANSFORMING SEQUENTIAL POWER FLOW SOLVER TO PARALLEL POWER FLOW SOLVER

In this section, we discuss how the sequential power flow solvers are transformed to parallel programs on GPU. The steps involved in the transformation are

1) Allocate GPU memory for the program.
2) Copy the original data from CPU to GPU.
3) Set the thread structure in GPU and call the kernel to process the data.
4) Copy the processed data from GPU back to CPU.
5) Release the allocated GPU memory.

In this paper, we parallelize the calculation of bus admittance matrix and the iteration process in the power flow solver. The following subsections gives the detailed description of the implementation.

A. Linear Equations Solver

As both Newton-Raphson algorithm and P-Q decoupled algorithm need to solve linear equations, we choose a commonly used method, i.e., the Gaussian elimination method, to solve the linear equations. Gaussian elimination method consists of two parts: forward elimination and back substitution.

For augmented matrix

$$A = \begin{pmatrix} a_{1,1} & \cdots & a_{1,k} & \cdots & a_{1,n} & a_{1,n+1} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ a_{k,1} & \cdots & a_{k,k} & \cdots & a_{k,n} & a_{k,n+1} \\ \vdots & \ddots & \vdots & \ddots & \vdots & \vdots \\ a_{n,1} & \cdots & a_{n,k} & \cdots & a_{n,n} & a_{n,n+1} \end{pmatrix} \tag{14}$$

the procedure of the kth ($1 \le k \le n$) forward elimination is

$$a_{kj} = a_{kj}/a_{kk}, \qquad j = k+1, \cdots, n+1 \tag{15}$$

$$a_{ij} = a_{ij} - a_{ik} \times a_{kj}, \qquad i = k+1, \cdots, n, \quad j = k+1, \cdots, n+1 \tag{16}$$

Equation (15) and (16) can be calculated separately. In our implementation, we define two CUDA kernel functions, i.e., *GaussKernelA* and *GaussKernelB* to calculate (15) and (16), respectively. It is worth pointing out that in the main process, we cannot calculate (16) until the calculation of (15) is completed. Algorithm 1 and Algorithm 2 show the detailed implementation for function *GaussKernelA* and *GaussKernelB*, respectively.

The inputs to both function *GaussKernelA* and *GaussKernelB* are the same, which are

- The augmented matrix $augMatrixGPU$ in GPU memory.
- The number of rows n in matrix $augMatrixGPU$.
- The kth Gauss forward elimination step.

Algorithm 1 GAUSS ELIMINATION CUDA KERNEL A

Input: Augmented matrix in GPU memory: $augMatrixGPU$, number of rows in matrix $augMatrixGPU$: n, the Gauss forward elimination step: k.

1: $i \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
2: $j \leftarrow blockIdx.y * blockDim.y + threadIdx.y$
3: **if** $i == k$ and $j > k$ and $j < n+1$ and $augMatrixGPU[k \times (n+1)+k] \neq 0.0$ **then**
4: $augMatrixGPU[k \times (n+1)+j] \leftarrow augMatrixGPU[k \times (n+1)+j] / augMatrixGPU[k \times (n+1)+k]$
5: **end if**

A brief explanation of Algorithm 1 is as follows. In Algorithm 1, the augmented matrix is first expanded by row, and it is stored in an one-dimentional array $augMatrixGPU$. The index number of the element in the one-dimentional array is $i \times (n+1)+j$, where i, j, and $(n+1)$ refers to the row number, column number, and total number of columns in augmented matrix [1], respectively.

In Line 1 and Line 2, the type of CUDA built-in varaibles, i.e., *blockIdx*, *blockDim*, and *threadIdx*, is *dim3* [2]. The *blockIdx* refers to the block index in the grid, and *blockDim* denotes the block dimension, and *threadIdx* refers to the thread index. From Line 3 to Line 5, the algorithm performs the operation shown in (15).

The explanation of Algorithm 2 is similar to Algorithm 1. The main difference between Algorithm 2 and Algorithm 1 is that Algorithm 2 implements (16) (See Line 4), while Algorithm 1 implements (15).

[1] In Algorithm 2 and Algorithm 3, the augmented matrix is stored in the same way.

[2] *dim3* is a vector type and it is defined based on *unit3*. *dim3* is equivalent to the *struct* structure which consists of three unsigned int variables.

Algorithm 2 GAUSS ELIMINATION CUDA KERNEL B

Input: Augmented matrix in GPU memory: $augMatrixGPU$, number of rows in matrix $augMatrixGPU$: n, the Gauss forward elimination step: k.

1: $i \leftarrow blockIdx.x * blockDim.x + threadIdx.x$
2: $j \leftarrow blockIdx.y * blockDim.y + threadIdx.y$
3: **if** $i > k$ and $i < n$ and $j > k$ and $j < n+1$ and $augMatrixGPU[k \times (n+1) + k] \neq 0.0$ **then**
4: $augMatrixGPU[i \times (n+1) + j] \leftarrow augMatrixGPU[i \times (n+1) + j] - augMatrixGPU[i \times (n+1) + k] \times augMatrixGPU[k \times (n+1) + j]$
5: **end if**

Algorithm 3 GAUSS FORWARD ELIMINATION

Input: Augmented matrix in GPU memory: $augmentMatrix$, number of rows in matrix $augmentMatrix$: n.

1: $cudaMalloc((void**)\&aguMatrixGPU, sizeof(float) \times n \times (n+1))$
2: $cudaMemcpy2D(aguMatrixGPU, sizeof(float) \times (n+1), aguMatrix, sizeof(float) \times (n+1), sizeof(float) \times (n+1), n, cudaMemcpyHostToDevice)$
3: $dim3\ blockDim(22, 22)$
4: $dim3\ gridDim((n+blockDim.x-1)/blockDim.x, (n+1+blockDim.y-1)/blockDim.y)$
5: **for** $k \leftarrow 0$ to $n-1$ **do**
6: $GaussKernelA <<< gridDim, blockDim >>> (aguMatrixGPU, n, k)$;
7: $GaussKernelB <<< gridDim, blockDim >>> (aguMatrixGPU, n, k)$;
8: **end for**
9: $cudaMemcpy2D(aguMatrix, sizeof(float) \times (n+1), aguMatrixGPU, sizeof(float) \times (n+1), sizeof(float) \times (n+1), n, cudaMemcpyDeviceToHost)$
10: $cudaFree(aguMatrixGPU)$

A brief explanation of Algorithm 3 is as follows. Line 1 allocates GPU memory for the augmented matrix *aguMatrixGPU*. In Line 2, the augmented matrix is copied from CPU memory to GPU memory by CUDA memory copy function *cudaMemcpy2D*. Line 3 and Line 4 set CUDA thread structure. The dimension of each block is 22×22, namely, each block consists of 484 threads (a block can have up to 512 threads in CUDA). In each grid, it consists of $((n + blockDim.x - 1)/blockDim.x) \times ((n + blockDim.x - 1)/blockDim.x)$ blocks. The reason we use $(n+blockDim.x-1)/blockDim.x$ to represent the dimension of grid is to ensure the number of blocks in each grid is an integer. From Line 5 to Line 8, the main process repeatly invokes CUDA kernels, i.e., *GaussKernelA* and *GaussKernelB*, to eliminate the augmented matrix. In Line 9, the eliminated augmented matrix is copied from GPU memory to CPU memory by the same CUDA memory copy function *cudaMemcpy2D*. Line 10 releases GPU memory allocated for the augmented matrix.

B. Compute Bus Admittance Matrix

The diagonal element Y_{ii} in the bus admittance matrix Y is called *self-admittance*, and its value is equal to the summation of the admitance of the branches which connect to the bus i. While the non-diagonal element Y_{ij} ($i \neq j$) is called *mutual admittance*, and its negative value is equal to the admittance of branch which connects to both bus i and j.

Based on the branch information (the starting bus identifier, ending bus identifier, branch equivalent admittance), we can calculate bus admittance matrix. In the bus admittance matrix, each element is independent and can be calculated separately. We define CUDA kernel function *computeAdmittanceMatrixKernel* to calculate the bus admittance matrix in parallel. For grounded branches, i.e., the identifier of the starting bus is equal to the identifier of the ending bus, we only need to calculate the element in the bus admittance matrix whose row number and column number is equal to the identifier of the starting bus, and its value is equal to its original value plus the branch equivalent admittance. For ungrounded branches, we need to calculate four elements in the bus admittance matrix. The four elements are

- The element whose row number and column number are equal to the identifier of the starting bus, and the value of this element is equal to its original value plus the branch equivalent admittance.
- The element whose row number and column number are equal to the identifier of the ending bus, and the value of this element is equal to its original value plus the branch equivalent admittance.
- The element whose row number is equal to the starting bus identifier, and its column number is equal to the identifier of the ending bus, and the value of this element is equal to the negtive branch equivalent admittance.
- The element whose row number is equal to the ending bus identifier, and its column number is equal to the identifier of the starting bus, and the value of this element is equal to the negative branch equivalent admittance.

The input data for calculating CUDA kernel in bus admittance matrix is

- Number of branches l.
- Starting buses' identifiers array $startNumVectorGPU$ in GPU memory.
- Ending buses' identifiers array $endNumVectorGPU$ in GPU memory.
- Branch equivalent admittance array $branchAdmittanceVectorGPU$ in GPU memory.

The procedure of calculating GPU parallel bus admittance matrix is as follows.

- Allocate GPU memory for bus admittance matrix, starting buses' identifiers array, end buses' identifiers array, and branch equivalent admittance array.

- Copy bus admittance matrix, starting buses' identifiers array, ending buses' identifiers array, and branch equivalent admittance array from CPU memory to GPU memory.
- Call CUDA kernel function *computeAdmittanceMatrixKernel* to calculate bus admittance matrix.
- Copy the obtained bus admittance matrix from GPU memory to CPU memory.
- Release the memory allocated for bus admittance matrix, starting buses' identifiers array, end buses' identifiers array, and branch equivalent admittance array.

C. Gauss-Seidel Solver

For (6) and (7), when calculating the voltage and reactive power of a bus, the data needed for summation operations is fixed and independent. Therefore, summation operations can be performed separately. In our work, we define a CUDA kernel function $GaussSumKernel$ to perform the summation operations in parallel. The input data includes

- Total number of buses n.
- Number of PQ buses m.
- Current bus identifier i.
- Bus voltage array $voltageMagnitudeVectorGPU$ in GPU memory. Assume the current number of iteration is $(k+1)$, and then the 1st to the $(i-1)$the buses' voltages are obtained by the $(k+1)$th iteration, and the remaining buses' voltages, i.e., the ith to nth buses' voltage are obtained by the kth iteration.
- The row i, i.e., $admittanceVectorGPU$, of the bus admittance matrix in GPU memory. It is worth pointing out that when calculating the voltage or reactive power of bus i, we only need the data in row i in bus admittance matrix. Therefore, we only transmit row i from CPU to GPU to save time.

The output is a summation array *sumGPU* which contains two elements: the first element is the summation value needed when calculating the bus voltage; the second element is the summation value needed when calculating the reactive power. For PQ bus, there is no need to calculate reactive power, therefore, the second element in array *sumGPU* is 0.

The procedure of calculating GPU parallel Jacobian matrix is as follows.

- Allocate GPU memory for bus voltage array, bus admittance matrix, and summation array.
- Copy bus voltage array and bus admittance matrix from CPU memory to GPU memory.
- Call CUDA kernel function *GaussSumKernel* to calculate the summation value needed by the Gauss-Seidel iteration.
- Copy the obtained summation array from GPU memory to CPU memory.
- Release the memory allocated for bus voltage array, bus admittance matrix, and summation array.

D. Newton-Raphson Solver

The main computation cost in the Newton-Raphson iteration comes from two parts, i.e., Jacobian matrix calculation and linear equations solving. The discussion regarding solving linear equations is given in Section V-A. In this subsection, we discuss the parallel implementation of Jacobian matrix calculation.

Suppose there are n buses in the power system, and the number of PQ buses is m. In the Newton-Raphson (polar form) solver, the order of Jacobian matrix J is $(n+m-1)$, and it can be presented as

$$J = \begin{pmatrix} H & N \\ K & L \end{pmatrix} \tag{17}$$

where the order of matrix H, N, K, and L is $(n-1)$, $(n-1)\times m$, $m\times(n-1)$, and m, respectively. Although the calculation approaches of these four sub-matrices are different, all of them contains first partial derivatives. In our work, we define CUDA kernel function *compuateJacobianMatrixKernel* to calculate the Jacobian matrix in parallel. For each element in the Jacobian matrix, we use one of the eight formulas (18)-(25) to calculate its value.

$$\begin{aligned} J_{ij} &= -U_i U_j (G_{ij}\sin\delta_{ij} - B_{ij}\cos\delta_{ij}) \\ i &= 1,\cdots,n-1; j = 1,\cdots,n-1; i \neq j \end{aligned} \tag{18}$$

$$\begin{aligned} J_{ij} &= U_i \sum_{\substack{k=1 \\ k\neq i}}^{n} U_k (G_{ik}\sin\delta_{ik} - B_{ik}\cos\delta_{ik}) \\ & i = 1,\cdots,n-1; i = j \end{aligned} \tag{19}$$

$$\begin{aligned} J_{ij} &= -U_i U_{j'} (G_{ij'}\cos\delta_{ij'} + B_{ij'}\sin\delta_{ij'}) \\ i &= 1,\cdots,n-1; j = n,\cdots,n+m-1; \\ & i \neq j; j' = j-n+1 \end{aligned} \tag{20}$$

$$\begin{aligned} J_{ij} = -U_i \sum_{\substack{k=1 \\ k\neq i}}^{n} U_k (G_{ik}\cos\delta_{ik} + B_{ik}\sin\delta_{ik}) - 2U_i^2 G_{ii} \\ i = 1,\cdots,n-1; i = j \end{aligned} \tag{21}$$

$$\begin{aligned} J_{ij} &= U_{i'} U_j (G_{i'j}\cos\delta_{i'j} + B_{i'j}\sin\delta_{ij}) \\ i &= n,\cdots,n+m-1; j = 1,\cdots,n-1; \\ & i \neq j; i' = i-n+1 \end{aligned} \tag{22}$$

$$\begin{aligned} J_{ij} = -U_{i'} \sum_{\substack{k=1 \\ k\neq i'}}^{n} U_k (G_{i'k}\cos\delta_{i'k} + B_{i'k}\sin\delta_{i'k}) \\ i = n,\cdots,n+m-1; i = j; i' = i-n+1 \end{aligned} \tag{23}$$

$$\begin{aligned} J_{ij} &= -U_{i'} U_{j'} (G_{i'j'}\sin\delta_{i'j'} - B_{i'j'}\cos\delta_{i'j'}) \\ i &= n,\cdots,n+m-1; j = n,\cdots,n+m-1; \\ & i \neq j; i' = i-n+1; j' = j-n+1 \end{aligned} \tag{24}$$

$$\begin{aligned} J_{ij} = -U_{i'} \sum_{\substack{k=1 \\ k\neq i'}}^{n} U_k (G_{i'k}\sin\delta_{i'k} - B_{i'k}\cos\delta_{i'k}) + 2U_i'^2 G_{i'i'} \\ i = n,\cdots,n+m-1; i = j; i' = i-n+1 \end{aligned} \tag{25}$$

In these formulas, G_{ij} and B_{ij} refer to real part and imaginary part, respectively, of the bus admittance matrix element whose row number is i and column number is j, U_i and δ_{ij} refer to voltage magnitude and voltage angle of the bus i, respectively. The selection of the formula is based on element's row number i and column number j.

The input data includes

- Total number of buses n.
- Dimention of the Jacobian matrix r.
- Bus voltage array $voltageMagnitudeVectorGPU$ in GPU memory.
- Bus voltage angle array $voltageAngleVectorGPU$ in GPU memory.
- The real part $admittanceMatrixRealGPU$ of the bus admittance matrix in GPU memory.
- The imaginary part $admittanceMatrixImaginaryGPU$ of the bus admittance matrix in GPU memory.

The procedure of calculating GPU parallel Jacobian matrix is as follows.

- Allocate GPU memory for Jacobian matrix, the real and imaginary part of the bus admittance matrix, bus voltage array and bus voltage angle array.
- Copy Jacobian matrix, the real and imaginary part of the bus admittance matrix, bus voltage array and bus voltage angle array from CPU memory to GPU memory.
- Call CUDA kernel function *compuateJacobianMatrixKernel* to calculate Jacobian matrix.
- Copy the obtained Jacobian matrix from GPU memory to CPU memory.
- Release the memory allocated for Jacobian matrix, the real and imaginary part of the bus admittance matrix, bus voltage array and bus voltage angle array.

E. P-Q Decoupled Solver

The P-Q decoupled solver is a simplified version of the Newton-Raphson solver, i.e., it ignores the calculation of Jacobian matrix. Hence, the computation cost of P-Q decoupled solver is solving linear equations. In Section V-A, we have already explained the parallel implementation of solving linear equations, we will not restate the procedure in this section.

VI. Performance Evaluation

This paper implemented these three sequential and parallel power flow solvers using C and CUDA on Windows operating system. A PC workstation with Intel i3-2100 CPU at 3.10GHz and 2G RAM runs the sequential solvers and works as the host for GPU. A NVIDIA GeForce GTS 450 GPU with 192 CUDA cores is used to compute parallel power flow solvers. We use standard IEEE9, IEEE30, IEEE118, and IEEE300 systems, and an actual running power system from Shandong Province as the benchmark to evaluate the GPU speedup of the three power flow solvers and compare the performances of the three parallel power flow solvers. Table II gives the number of bus and branches each of the benchmark system. The sequential solver's runtime on CPU, parallel solver's runtime on GPU, and speedup of Gauss-Seidel solver, Newton-Raphson solver and P-Q decoupled solver are shown Table III, Table IV, and Table V, respectively. In the power flow solving process, the iteration error is set to 10^{-5}, and the maximum iteration number is set to 100.

TABLE II
Power System Parameters

System	Bus Count	Branch Count
IEEE9	9	9
IEEE30	30	41
IEEE118	118	186
IEEE300	300	357
Shandong	974	1449

TABLE III
Execution Time and Speedup of Gauss-Seidel Solver

System	CPU Runtime(s)	GPU Runtime(s)	Speedup
IEEE9	0.0001	0.3276	0.0003
IEEE30	0.002	0.7051	0.0028
IEEE118	0.023	3.2963	0.007
IEEE300	0.3428	7.2992	0.047
Shandong	1.2147	19.603	0.062

TABLE IV
Execution Time and Speedup of Newton-Raphson Solver

System	CPU Runtime(s)	GPU Runtime(s)	Speedup
IEEE9	0.0015	0.0094	0.1596
IEEE30	0.0098	0.0094	1.0426
IEEE118	0.3132	0.1997	1.5684
IEEE300	4.689	2.6848	1.7465
Shandong	583.831	10.881	53.656

TABLE V
Execution Time and Speedup of P-Q Decoupled Solver

System	CPU Runtime(s)	GPU Runtime(s)	Speedup
IEEE9	0.0047	0.0047	1.0
IEEE30	0.0081	0.0125	0.648
IEEE118	0.1137	0.117	0.9718
IEEE300	1.5107	1.1606	1.3017
Shandong	148.974	5.5068	27.0527

We repeat the experiments under each scenario for ten times and take the average of the ten execution times as the final execution time for comparison. The speedup is sequential execution time divided by parallel execution time. The comparison of three parallel power flow solvers is plotted in Fig. 1. As can be seen from the figure that the speedup of Gauss-Seidel decoupled solver is almost coincident with the abscissa. In other words, it is very small, close to zero. From Fig. 1, we can also see that the Newton-Raphson solver has the best speedup, Gauss-Seidel solver performs the worst, and P-Q decoupled solver is in the middle. Due to the frequent data transmission between CPU and GPU, the parallel runtime is larger than sequential runtime of Gauss-Seidel solver. Furthermore, both speedup and speedup differences of all the solvers increase with the system size increases.

As can be seen from Fig. 1 that when the system size is larger than 300, the speedup increases dramatically. Although

we do not know exactly why the speedup surge happens at size 300, we believe the turning point is application dependent. Our future work will further investigate this.

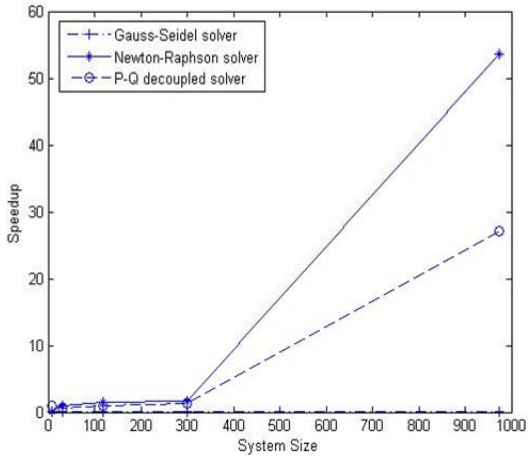

Fig. 1. Speedup Comparison

In addition, compared to the theoretical speedup obtainable by different algorithms when communication and memory access costs are not taken into consideration (Table I), there is still a big gap between the empirically obtained speedup and the theoretical potential speedup.

VII. Conclusion

This paper has discussed three parallel implementations of power flow solvers on GPU, i.e., Gauss-Seidel solver, Newton-Raphson solver, and P-Q decoupled solver, and compared the speedups of these three different methods. The parallel Gauss-Seidel solver parallelizes the summation process of each iteration, while for Newton-Raphson solver, we parallelize the Jacobian matrix computation when transforming non-linear equations to linear equations, and using Gauss elemination slover to solve the linear equations. The P-Q decoupled solver is a simplified version of Newton-Raphson solver, where Gauss elimination solver used solve linear equations is parallelized on GPU. We use four IEEE standard power systems and one actual running power system from Shangdong Province to evaluate and compare the speedups of three parallel power flow solvers. The results show that Newton-Raphson solver has the best speedup, Gauss-Seidel solver performs the worst, and P-Q decoupled solver is in the middle. According to the results, both speedup and speedup differences of all the solvers increase with the system size increases.

Although the empirical results have shown clear speedup when we transition the power flow slover on GPU, there is sitll a big gap between the obtained speedups and their potentials. One of the reasons is that in our theoretical analysis, we assumed there is no communication and memory access time cost, but in practice, such cost is not negligible. Our immediate next work is to study the equation distribution strategy so that the communication among different cores on GPU is minimized and hence further improves the speedups. Furthermore, as both the bus admittance matrix and Jacobian matrix of Newton-Raphson solver are sparse matrices, we can use sparse matrix technology to reduce the execution time and save memory usage. In addition, we will study different applications and investigate the speedup turning point.

Acknowledgement

The authors would like to thank Songzhao Xie for his work to implement the sequential P-Q decoupled power flow solver.

References

[1] J. J. Grainger and W. D. Stevenson. *Power System Analysis*. McGraw-Hill, 1994.

[2] S. Wang and Y. Liu. Review of load flow calculation methods in power systems. *Shandong Electric Power*, pages 8–12, Sept. 1996.

[3] Daniel J. Tylavsky and Anjan Bose. Parallel processing in power systems computation. *Power Systems, IEEE Transactions on*, 7(2):629–638, May 1992.

[4] J. Fong and C. Pottle. Parallel processing of power system analysis problems via simple parallel microcomputer structures. *Power Apparatus and Systems, IEEE Transactions on*, PAS-97(5):1834–1841, Sept. 1978.

[5] A. Gomez and R. Betancourt. Implementation of the fast decoupled load flow on a vector computer. *Power Systems, IEEE Transactions on*, 5(3):977–983, Aug. 1990.

[6] V.C. Ramesh. On distributed computing for on-line power system applications. *International Journal of Electrical Power and Energy Systems*, 18(8):527–533, 1996.

[7] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröoder. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, Jul. 2003.

[8] Z.A. Taylor, M. Cheng, and S. Ourselin. High-speed nonlinear finite element analysis for surgical simulation using graphics processing units. *Medical Imaging, IEEE Transactions on*, 27(5):650–663, May 2008.

[9] N. Goanddel, N. Nunn, T. Warburton, and M. Clemens. Scalability of higher-order discontinuous galerkin fem computations for solving electromagnetic wave propagation problems on gpu clusters. *Magnetics, IEEE Transactions on*, 46(8):3469–3472, Aug. 2010.

[10] G. Chalkidis, M. Nagasaki, and S. Miyano. High performance hybrid functional petri net simulations of biological pathway models on cuda. *Computational Biology and Bioinformatics, IEEE/ACM Transactions on*, 8(6):1545–1556, Dec. 2011.

[11] A. Gopal, D. Niebur, and S. Venkatasubramanian. Dc power flow based contingency analysis using graphics processing units. In *Power Tech, 2007 IEEE Lausanne*, pages 731–736, Jul. 2007.

[12] J.-F. Xia, F. Yang, J. Li, and X.-Y. Zheng. Implementation of parallel power flow calculation based on gpu. *Power System Protection and Control*, 38:100–103, Sept. 2010.

[13] J. Singh and I. Aruni. Accelerating power flow studies on graphics processing unit. In *India Conference (INDICON), 2010 Annual IEEE*, pages 1–5, Dec. 2010.

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

[15] C. Vilacha, J.C. Moreira, E. Miguez, and A.F. Otero. Massive jacobi power flow based on simd-processor. In *Environment and Electrical Engineering (EEEIC), 2011 10th International Conference on*, pages 1–4, May 2011.