April 2011

Power Flow Analysis on CUDA-based GPU

Yizheng Liao
Worcester Polytechnic Institute

Power Flow Analysis on CUDA-based GPU

by

Yizheng Liao

A Major Qualifying Project Report
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
in
Electrical and Computer Engineering
by

May 2011

APPROVED:

Professor Xinming Huang, MQP Advisor

Abstract

This major qualifying project investigates the algorithm and the performance of using the CUDA-based Graphics Processing Unit for power flow analysis. The accomplished work includes the design, implementation and testing of the power flow solver. Comprehensive analysis shows that the execution time of the parallel algorithm outperforms that of the sequential algorithm by several factors.

Acknowledgements

I would like to express my deep and sincere gratitude to my academic and MQP advisor, Professor Xinming Huang, for giving me the professional and insightful comments and suggestions on this project.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The smart grid has been discussed widely in both academic and industry. Many research and development projects have pointed out the functionalities of the smart grid [3]:

- Self-healing
- High reliability and power quality
- Accommodates a wide variety of distributed generation and storage options
- Optimizes asset utilization
- Minimizes operations and maintenance expenses

In order to realize these functions, each unit of the power system, such as the power plant, the grid operator, and the customer, needs to know the on-line power flow. For example, a power plant wants to change its output based on the total power consumption of a city. In order to achieve it, the power plant needs to collect the grida data and compute the power consumption. This is called the power flow analysis.

The power flow analysis has been widely developed since 1960s [4]. Many algorithms have been proposed on this area. However, in the case we presented above, we assume there are at least ten thousands buses in the city-scale grid. Consequently, the computation of the power consumption has a high latency, which may delay the power plant decision.

Therefore, a fast way to compute the grid information is necessary. One approach is using the parallel computation to solve it.

Nowadays, the hardware device has been widely used for high performance computation. The most well known devices include the multi-core CPU, the Field-programmable Gate Array (FPGA), and the Graphic Processing Unit (GPU). The multi-core CPU has very high clock frequency. However, compared with FPGA and GPU, the number of thread on GPU is limited. An FPGA is a custom integrated circuit that generally includes a large number of logical cells [5]. Each logical cell is available for handling a single task from a predefined set of functions. However, the drawback for using FPGA for power flow problem is the computation of floating point on FPGA is not efficient. Similar to the FPGA, the each thread on the GPU can handle a single task based the predefined function. In addition, the recent released CUDA GPU, which is developed by NVIDIA, is available for programming by C, which is more convenient than the hardware language, such as VHDL. In addition, among these three devices, the GPU can be purchased by a very cheap price.

In this project, we investigate the algorithm for using CUDA-based GPU to solve the problem flow problem. Also, we use the developed algorithm to test the IEEE power flow test case. The execution time is compared with the Matlab-based power flow solver, which solves the problem in a sequential way. The performance analysis shows that the execution time of the parallel algorithm is much faster than the sequential algorithm.

Chapter 2

Background

2.1 Power Flow Model

In order to analysis the transmission system, we need to model the buses or nodes interconnected by transmissions links. Generators and loads, which are connected to various buses in the system, inject and remove power from the transmission system. As discussed in [6], we assume that each transmission link is represented by a per phase Π-equivalent circuit. Then for each bus i, the relationship between the complex voltage V_i and the complex current I_i is

$$I_i = Y \times V_i. \tag{2.1}$$

If we extend (2.1) to all the buses, for a n buses system, we can use nodal analysis to relate all the bus currents to the bus voltage. In matrix notation, this relationship is

$$\mathbf{I} = \mathbf{Y}_{bus}\mathbf{V} \tag{2.2}$$

where

$$\mathbf{I} = \begin{bmatrix} I_1 \\ I_2 \\ \vdots \\ I_n \end{bmatrix} \tag{2.3}$$

and

$$\mathbf{V} = \begin{bmatrix} V_1 \\ V_2 \\ \vdots \\ V_n \end{bmatrix}. \tag{2.4}$$

In (2.2), the $n \times n$ matrix $\mathbf{Y}_{bus}$ is called the *bus admittance matrix*. Its element is equal to [6]

- $y_{ii} = \sum$ admittances of Π-equivalent circuit elements incident to the ith bus
- $y_{ik} = -$ admittance of Π-equivalent circuit elements bridging the ith and kth buses

In this project, we created $\mathbf{Y}_{bus}$ by the method given in [7].

2.1.1 Branches

In [7], the branch model is given in Figure 2.1.

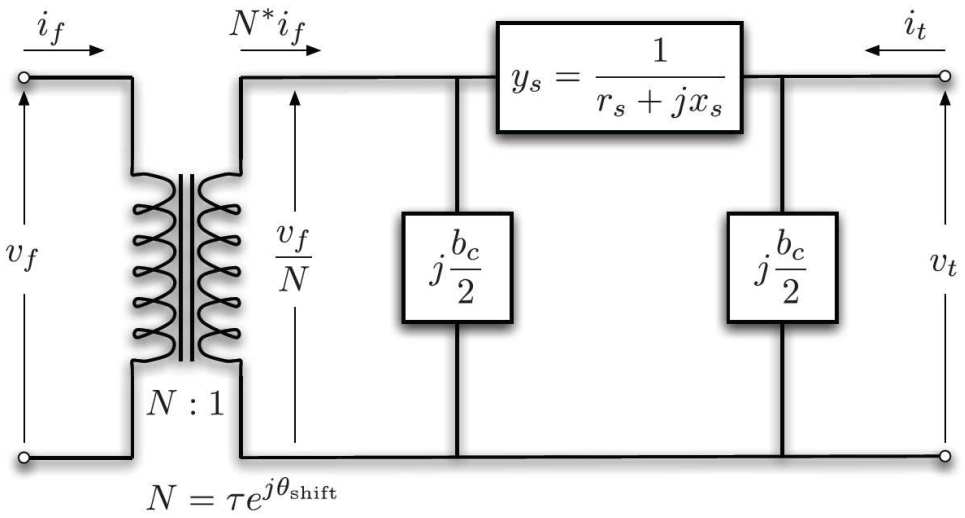

Figure 2.1: Branch Model

For each branch, the complex current injections at the *from* end of branch i_f and the current injections at the *to* end i_t. According to (2.2), the current injections can be expressed in terms of the 2×2 branch admittance matrix $\mathbf{Y}_{br}$ and the respective terminal voltages

v_f and v_t,

$$\begin{bmatrix} i_f \\ i_t \end{bmatrix} = \mathbf{Y}_{br} \begin{bmatrix} v_f \\ v_t \end{bmatrix} \tag{2.5}$$

In (2.5), the branch admittance matrix $\mathbf{Y}_{br}$ can be written as

$$\mathbf{Y}_{br} = \begin{bmatrix} (y_s + j\frac{b_c}{2})\frac{1}{\tau^2} & -y_s \frac{1}{\tau \exp(-j\theta_{shift})} \\ -y_s \frac{1}{\tau \exp(j\theta_{shift})} & y_s + j\frac{b_c}{2} \end{bmatrix} \tag{2.6}$$

where

- $y_s = 1/z_s$ denotes the series admittance and $z_s = r_s + jx_s$ is the series impedance in Π transmission line model
- b_c denotes the total charging capacitance
- τ denotes the magnitude of the transformer tap ratio
- θ_{shift} denotes the phase shift angle of the transformer tap ratio.

For branch i, if the four elements of $\mathbf{Y}_{br}^i$ is labelled as shown in (2.7)

$$\mathbf{Y}_{br}^i = \begin{bmatrix} y_{ff}^i & y_{ft}^i \\ y_{tf}^i & y_{tt}^i \end{bmatrix}, \tag{2.7}$$

therefore, for a transmission system with n_l lines,

$$\mathbf{Y}_{br} = \begin{bmatrix} \mathbf{Y}_{ff} & \mathbf{Y}_{ft} \\ \mathbf{Y}_{tf} & \mathbf{Y}_{tt} \end{bmatrix} \tag{2.8}$$

where each element is a $n_l \times 1$ vector and the ith element of each vector comes from the corresponding element of $\mathbf{Y}_{br}$.

In addition, the connection matrices $\mathbf{C}_f$ and $\mathbf{C}_t$ are used to build the system admittance matrices $\mathbf{Y}_{bus}$ as well. For each branch i connects from bus j to bus k, the (i, j) element of $\mathbf{C}_f$ and the (i, k) element of $\mathbf{C}_t$ are equal to one. All other elements are zero. For a n_b buses transmission system with n_l branches, the demission of $\mathbf{C}_f$ and $\mathbf{C}_t$ is $n_l \times n_b$.

2.1.2 Generators

A generator is connected to a specific bus as the complex power injection. For generator i, the power injection is

$$s_g^i = p_g^i + jq_g^i. \tag{2.9}$$

Therefore, for a system with n_g generators, (2.9) can be expressed in a vector form,

$$\mathbf{S}_g = \mathbf{P}_g + j\mathbf{Q}_g \tag{2.10}$$

where $\mathbf{S}_g$, $\mathbf{P}_g$, and $\mathbf{Q}_g$ are $n_g \times 1$ vectors.

In order to map the generator to the bus, we build the $n_b \times n_g$ generator connection matrix $\mathbf{C}_g$. When generator j is located at bus i, the (i, j) element of $\mathbf{C}_g$ is one. Other elements are zero.

2.1.3 Loads

A load bus is modelled as a complex consumption at a specific bus. For bus i, the load is

$$s_d^i = p_d^i + jq_d^i. \tag{2.11}$$

Hence, for a system with n_b buses, (2.11) becomes to

$$\mathbf{S}_d = \mathbf{P}_d + j\mathbf{Q}_d \tag{2.12}$$

where $\mathbf{S}_d$, $\mathbf{P}_d$, and $\mathbf{Q}_d$ are $n_b \times 1$ vectors.

2.1.4 Shunt Elements

Both capacitor and inductor are called shunt connected element. A shunt element is modelled as a fixed impedance connected to ground at a bus. For bus i, the admittance of the shunt element is given as

$$y_{sh}^i = g_{sh}^i + jb_{sh}^i. \tag{2.13}$$

Thus, for a system with n_b buses, (2.13) is expressed as

$$\mathbf{Y}_{sh} = \mathbf{G}_{sh} + j\mathbf{B}_{sh} \tag{2.14}$$

where $\mathbf{Y}_{sh}$, $\mathbf{G}_{sh}$, and $\mathbf{B}_{sh}$ are $n_b \times 1$ vectors.

2.1.5 Transmission System

Recall the transmission equation (2.2). For a system with n_b buses, all impedance elements of the model are included in a $n_b \times n_b$ complex matrix $\mathbf{Y}_{bus}$, which relates the complex node current injections $\mathbf{I}_{bus}$ and the complex node voltage $\mathbf{V}$.

The similar idea applies to the branch system. For a system with n_l branches, the $n_l \times n_b$ branch admittance matrices $\mathbf{Y}_f$ and $\mathbf{Y}_t$ relate the bus voltages to the branch currents $\mathbf{I}_f$ and $\mathbf{I}_t$ at the *from* and *to* ends of all branches, as shown in (2.15) and (2.16).

$$\mathbf{I}_f = \mathbf{Y}_f \mathbf{V} \tag{2.15}$$

$$\mathbf{I}_t = \mathbf{Y}_t \mathbf{V} \tag{2.16}$$

The admittance matrices can be computed as

$$\mathbf{Y}_f = [\mathbf{Y}_{ff}]\mathbf{C}_f + [\mathbf{Y}_{ft}]\mathbf{C}_t, \tag{2.17}$$

$$\mathbf{Y}_t = [\mathbf{Y}_{tf}]\mathbf{C}_f + [\mathbf{Y}_{tt}]\mathbf{C}_t, \tag{2.18}$$

and

$$\mathbf{Y}_{bus} = \mathbf{C}_f^T \mathbf{Y}_f + \mathbf{C}_t^T \mathbf{Y}_t + [\mathbf{Y}_{sh}] \tag{2.19}$$

where [.] denotes an operator that takes an $n \times 1$ vector and creates a $n \times n$ diagonal matrix with the vector elements on the diagonal, and T denotes the matrix transpose.

The complex power injections now can be computed as functions of the complex bus voltages $\mathbf{V}$ as

$$\mathbf{S}_{bus}(\mathbf{V}) = [\mathbf{V}]\mathbf{I}_{bus}^* = [\mathbf{V}]\mathbf{Y}_{bus}^* \mathbf{V}^*, \tag{2.20}$$

$$\mathbf{S}_f(\mathbf{V}) = [\mathbf{C}_f\mathbf{V}]\mathbf{I}_f^* = [\mathbf{C}_f\mathbf{V}]\mathbf{Y}_f^*\mathbf{V}^*, \tag{2.21}$$

and

$$\mathbf{S}_t(\mathbf{V}) = [\mathbf{C}_t\mathbf{V}]\mathbf{I}_t^* = [\mathbf{C}_t\mathbf{V}]\mathbf{Y}_t^*\mathbf{V}^* \tag{2.22}$$

where $*$ denotes the complex conjugate of each element.

For a AC power model, the total power should balance. Therefore, we have the system power function:

$$gs(\mathbf{V}, \mathbf{S}_g) = \mathbf{S}_{bus}(\mathbf{V}) + \mathbf{S}_d - \mathbf{C}_g\mathbf{S}_g = 0. \tag{2.23}$$

2.2 Power Flow Analysis

The power flow analysis is very important in the electrical grid. In order to ensure the continued operation, we need to avoid line and generator overload, to compute the voltage limits and so on [6]. The goal of the analysis is to obtain the magnitude and phase of each bus voltage in a transmission system for specific load and generator real and voltage conditions [4]. After knowing these information, we can determine the complex power flow on each branch and the reactive power output of each generator.

We know at bus i

$$S_i = P_i + jQ_i \tag{2.24}$$

where $P_i = \Re\{S_i\}$ is the real power and $Q_i = \Im\{S_i\}$ is the reactive power, and

$$V_i = |V_i| \exp(j\theta_i) \tag{2.25}$$

where $|V_i|$ is the magnitude of the complex voltage V_i and $\theta_i = \angle V_i$ is the phase of the complex voltage. Therefore, we can rewrite (2.20) as, for bus i,

$$S_i = P_i + jQ_i = V_i \sum_{m=1}^{n} Y_{bus_{im}}^* V_m^* \tag{2.26}$$

$$= |V_i| \exp(j\theta_i) \sum_{m=1}^{n} Y_{bus_{im}}^* |V_m| \exp(-j\theta_m) \tag{2.27}$$

In (2.26), we have four variables, P_i, Q_i, $|V_i|$, and θ_i. The solution to the power flow problem begins with identifying if each variable of these four are known in the system. In fact, the known and unknown variables depend on the type of bus. In the transmission system, we have three types of bus:

- Load bus: a bus without any generators connected to it
- Generator bus: a bus has at least one generator connected to it
- Slack bus: a reference generator bus

For each type of bus, two out of four variables are known. For load bus, we assume $P_i = -P_d^i$ and $Q_i = -Q_d^i$ are known. In power flow problem, the load bus is usually called as PQ bus. For generator bus, we assume $P_i = P_g^i - P_d^i$ and $|V_i|$ are known. The generator bus is usually called as PV bus in power flow system. For the slack bus, we assume $|V_i|$ and θ_i are known. Therefore, for each PQ bus, we must solve for $|V_i|$ and θ_i. For each PV bus, we must solve for θ_i. For slave bus, none of the variables must be solved. For each bus, after solving $|V_i|$ and θ_i, we can compute P_i and Q_i by (2.26).

2.2.1 Solving Power Flow Problem

As discussed above, the key to solve the power flow problem is finding the complex voltage of each bus. After that, we can easily find the complex power of each bus. In power flow problem, there are two cases of problem. The first case knows $V_1, S_2, S_3, \ldots, S_n$ and solves $S_1, V_2, V_3, \ldots, V_n$.

For (2.26), we assume $i = 1$ is the slack bus. We can separate it from other buses. Hence, we rewrite (2.26) as

$$S_1 = V_1 \sum_{m=1}^{n} Y_{bus_{1m}}^* V_m^* \tag{2.28}$$

$$S_i = V_i \sum_{m=1}^{n} Y_{bus_{im}}^* V_m^* \quad i = 2, 3, \ldots, n \tag{2.29}$$

In (2.28), it we know $V_1, V_2, \ldots, V_n$, we can solve for S_1 explicitly by using (2.28). Since bus 1 is the slack bus, we have known V_1. Therefore, we only need to find $V_2, \ldots, V_n$. These

$n-1$ unknowns can be found by using (2.29). Hence, the key for solving case I problem is finding the solution of $n-1$ implicit equations in the unknown $V_2, V_3, \ldots, V_n$, where V_1 and $S_2, S_3, \ldots, S_n$ are known.

Now let's take complex conjugates of (2.29). Then we have

$$S_i^* = V_i^* \sum_{m=1}^{n} Y_{bus_{im}} V_m \quad i = 2, 3, \ldots, n \tag{2.30}$$

Dividing (2.28) by V_i^* and separating the $Y_{bus_{ii}}$ term, we can rewrite (2.28)

$$\frac{S_i^*}{V_i^*} = \sum_{m=1}^{n} Y_{bus_{im}} V_m \tag{2.31}$$

$$\frac{S_i^*}{V_i^*} = Y_{bus_{ii}} V_i + \sum_{m=1, m\neq i}^{n} Y_{bus_{im}} V_m \quad i = 2, 3, \ldots, n \tag{2.32}$$

or, equivalently,

$$V_i = \frac{1}{Y_{bus_{ii}}} \left[\frac{S_i^*}{V_i^*} - \sum_{m=1, m\neq i}^{n} Y_{bus_{im}} V_m \right] \quad i = 2, 3, \ldots, n \tag{2.33}$$

Thus, now we have a non-linear system which can be solved by using the iteration method, such as the Jacobi method or Gauss-Seidel method. We can rewrite (2.33) in the iterative form, which is shown in (2.34).

$$V_i^{(k)} = \frac{1}{Y_{bus_{ii}}} \left[\frac{S_i^*}{V_i^{*(k-1)}} - \sum_{m=1, m\neq i}^{n} Y_{bus_{im}} V_m^{(k-1)} \right] \quad i = 2, 3, \ldots, n \tag{2.34}$$

The second case of the power flow problem knows $V_1, (P_2, |V_2|), \ldots, (P_l, |V_l|), S_{l+1}, \ldots, S_n$. The unknown variables are $S_1, (Q_2, \theta_2), \ldots, (Q_l, \theta_l), V_{l+1}, \ldots, V_n$. In this case, for the slack bus, we can use (2.28) to find S_1 after solving all the variables of other buses. For other buses, the procedure is similar to the case I.

In case II, we have both PQ buses and PV buses. For PQ buses, since we have known

P_i and Q_i, we can use (2.34) to compute V_i. Therefore, we have the iterative formula:

$$V_i^{(k)} = \frac{1}{Y_{bus_{ii}}} \left[\frac{S_i^*}{V_i^{*(k-1)}} - \sum_{m=1, m\neq i}^{n} Y_{bus_{im}} V_m^{(k-1)} \right] \quad i = l+1, l+2, \ldots, n \tag{2.35}$$

For PV buses, although Q_i is unknown, we can perform a side calculation to estimate it on the basis of the kth step voltage. Thus, by using (2.29), we have

$$\hat{Q}_i = \Im \left[V_i^{(k)} \sum_{m=1}^{n} Y_{bus_{im}^*} (V_m^{(k)})^* \right] \quad i = 2, 3, \ldots, l \tag{2.36}$$

Then, we can compute the complex voltage for PV buses by using (2.37).

$$V_i^{(k)} = \frac{1}{Y_{bus_{ii}}} \left[\frac{S_i^*}{V_i^{*(k-1)}} - \sum_{m=1, m\neq i}^{n} Y_{bus_{im}} V_m^{(k-1)} \right] \quad i = 2, 3, \ldots, l \tag{2.37}$$

2.3 Gauss-Jacobi Iterative Method

Gauss-Jacobi iterative method, usually refers to Jacobi method, is an algorithm for finding the solutions of a linear equations system.

Suppose a system of n linear equations is given. Each equation contains n variables. Therefore, we have

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \ldots a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \ldots a_{2n}x_n = b_2 \\ \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \ldots a_{nn}x_n = b_n \end{cases} \tag{2.38}$$

We can write the linear equations in (2.38) into a matrix form. Then we have

$$\mathbf{Ax} = \mathbf{b} \tag{2.39}$$

where

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}, \tag{2.40}$$

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}, \tag{2.41}$$

and

$$\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{2.42}$$

We can re-write the matrix $\mathbf{A}$ in a form of

$$\mathbf{A} = \mathbf{D} + \mathbf{R} \tag{2.43}$$

where

$$\mathbf{D} = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix} \tag{2.44}$$

and

$$\mathbf{R} = \begin{bmatrix} 0 & a_{12} & \cdots & a_{1n} \\ a_{21} & 0 & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & 0 \end{bmatrix}. \tag{2.45}$$

Now, we can re-write the linear system (2.40) as

$$\begin{aligned}(\mathbf{D}+\mathbf{R})\mathbf{x} &= \mathbf{b}\\ \mathbf{D}\mathbf{x}+\mathbf{R}\mathbf{x} &= \mathbf{b}\\ \mathbf{D}\mathbf{x} &= \mathbf{b}-\mathbf{R}\mathbf{x}\\ \mathbf{x} &= \mathbf{D}^{-1}(\mathbf{b}-\mathbf{R}\mathbf{x}).\end{aligned} \tag{2.46}$$

The Jacobi method is an iterative method that solves the linear system in (2.46). (2.46) can be written in an iterative form as

$$\mathbf{x}^{(k+1)} = \mathbf{D}^{-1}(\mathbf{b}-\mathbf{R}\mathbf{x}^{k}). \tag{2.47}$$

where k is the iterative index.

Since the matrix $\mathbf{D}$ is a diagonal matrix, we can easily compute $\mathbf{D}^{-1}$, as shown in (2.48)

$$\mathbf{D}^{-1} = \begin{bmatrix} \frac{1}{a_{11}} & 0 & \cdots & 0 \\ 0 & \frac{1}{a_{22}} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \frac{1}{a_{nn}} \end{bmatrix} \tag{2.48}$$

Therefore, the element-based formula for the Jacobi method is [8]

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1, j\neq i}^{j=n} a_{ij}x_j^{(k)}\right), \quad i = 1, 2, \ldots, n. \tag{2.49}$$

The stopping criteria for the Jacobi method includes two conditions:

1. $||\mathbf{x}^{(k)}-\mathbf{x}^{(k-1)}||_\infty \leq \epsilon$, where $||.||_\infty$ denotes the infinity norm and ϵ is the error tolerance
2. $k = K$, where K denotes the maximum number of iteration.

When either of them is satisfied, the iteration will stop and output the latest $\mathbf{x}^k$ as the solution of the linear system (2.39).

The algorithm of the Jacobi method is described in Algorithm 1.

Algorithm 1 Jacobi Method

Import $\mathbf{A}, \mathbf{b}, \mathbf{x}^{(0)}$
$\phi = 1, k = 0$
while $\phi \geq \epsilon$ **do**
  $k = k + 1$
  for $i = 1 \rightarrow n$ **do**
    $\tau = 0$
    for $(j = 1 \rightarrow n)$ AND $j \neq i$ **do**
      $\tau = \tau + a_{ij} x_j^{(k-1)}$
    end for
    $x_i^{(k)} = \frac{b_i - \tau}{a_{ii}}$
  end for
  if $(k + 1) == K$ **then**
    BREAK
  end if
  $\phi = \max\{|x_1^{(k)} - x_1^{(k-1)}|, |x_2^{(k)} - x_2^{(k-1)}|, \ldots, |x_n^{(k)} - x_n^{(k-1)}|\}$
end while

2.4 CUDA Overview

Graphic Processing Unit (GPU) is a high performance computing device. Due to its architecture, the number of cores of GPU exceeds that of multi-core CPUS greatly. There are hundreds of cores in mainstream GPUs, therefore, the parallel computation capacity of GPU is much higher than that of multi-core CPU.

CUDA, stands for Compute Unified Device Architecture, is a new computing architecture for GPU, which is introduced by NVIDIA Corporation in 2006 [9]. Compared with the previous GPU architecture, the CUDA GPU provides a more convenient tool for the developers to perform large scale parallel computation via GPU [10].

In this project, we use NVIDIA GeForce GTS 250 CUDA GPU to run the simulation. There are 16 SMs (Streaming Multiprocessors), with 8 scalar processors in each SM. Thus, in total, there are 128 cores in GreForce GTS 250. When a parallel task is launched, the task will be implemented by a function called kernel function, which is run by one thread. These threads are first grouped into blocks and then blocks are organized by one grid. This relationship is shown in Figure 2.2. Threads in the same block are separated into a scheduling unit called warp. On the GPU we used, each warp contains 32 threads. More details are given in Appendix A.

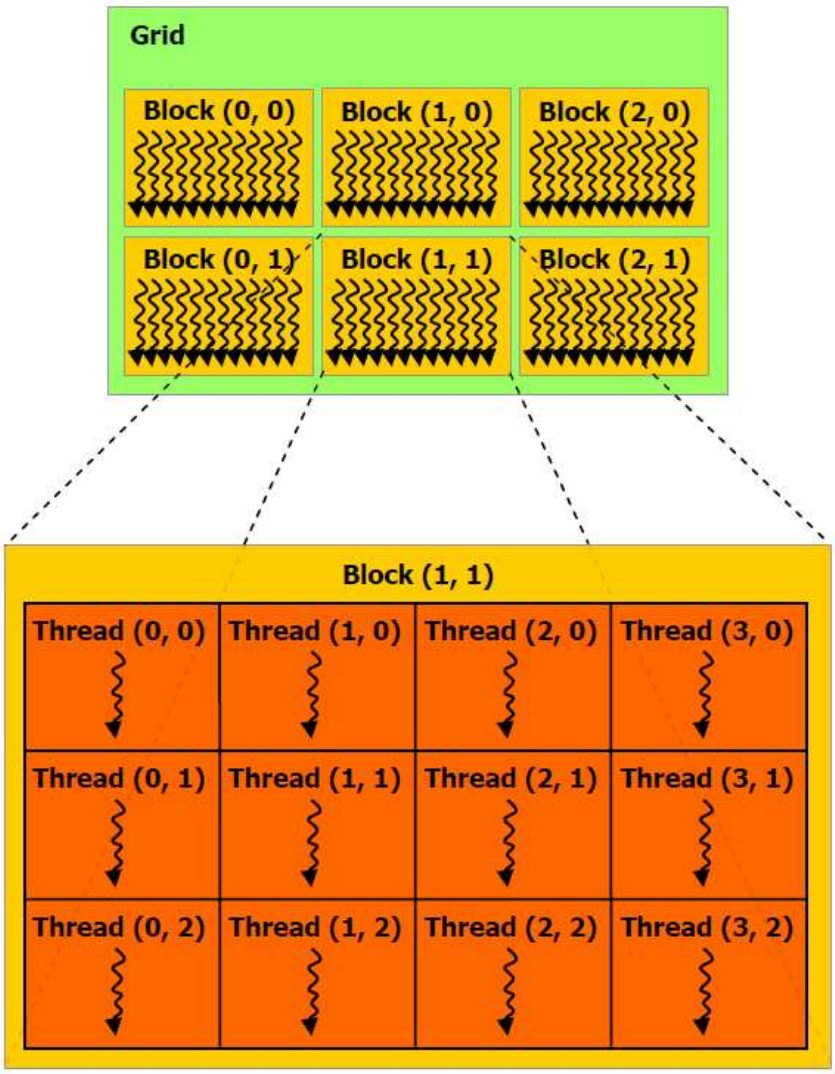

Figure 2.2: Grid of Thread Blocks [1]

In CUDA GPU, there are five types of memory. The following table provides the details about the memory type, the range of threads that can access the memory, which is called scope, and the lifetime, which specifies the lifetime of the variable defined in different type of memory [2]. Figure 2.3 shows the memory model.

Table 2.1: CUDA Memory Type

Memory	Scope	Lifetime
Register	Thread	Kernel
Local	Thread	Kernel
Shared	Block	Kernel
Global	Grid	Application
Constant	Grid	Application

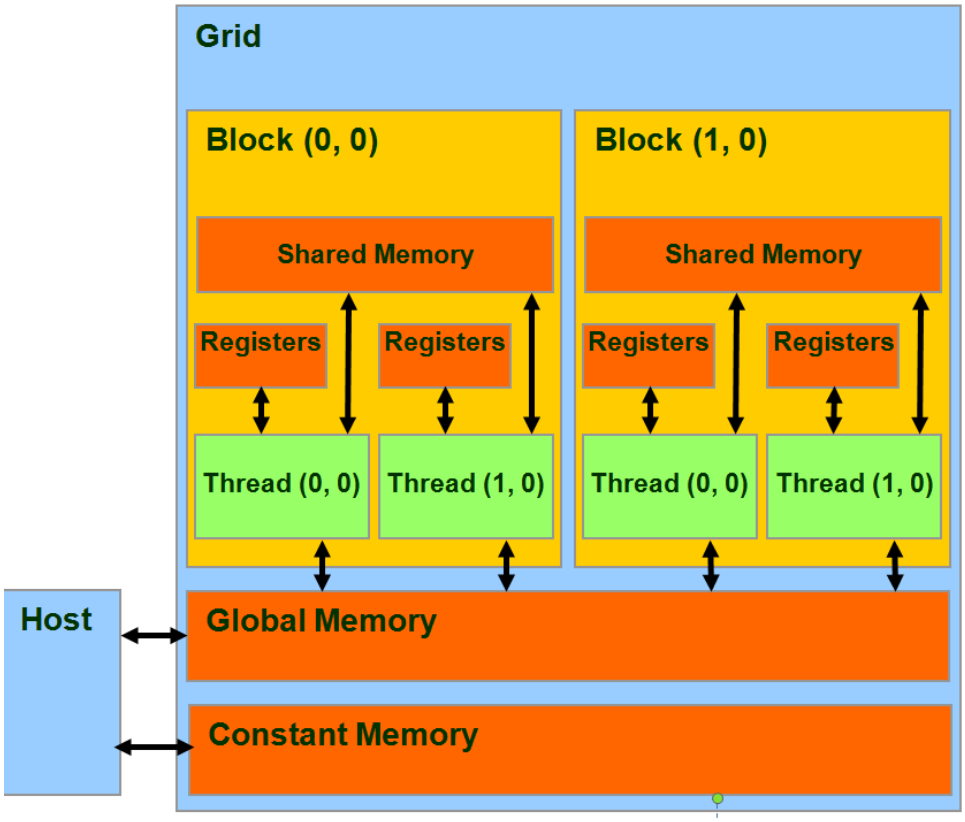

Figure 2.3: Overview of CUDA device memory model [2]

In these memories, the register has the fastest access speed. Each thread is assigned several registers and the assigned register can only be called by its master thread. However,

in CUDA, the number of available registers are limited. On GeForce GTS 250, the available registers for each block is 8192. Therefore, if we launch 512 threads in each block, then each thread is only assigned 16 registers. Therefore, we need to plan ahead to ensure all the available registers are used effectively.

In each block, the shared memory can be accessed by all the threads within the block. In CUDA programming, the shared memory is frequently used to read data from the global memory and then share the data among the threads [11]. However, the available space of the shared memory is a concern. Only 16384 bytes are available for each block. Usually, we store the frequently used data in shared memory.

The global memory has the largest space among all the types of memory. On GeForce GTS 250, the available global memory is about 500 MB. However, the access to the global memory from thread is not efficient. Also, each thread accesses the global memory in a sequential way. Therefore, in CUDA programming, we try to minimize the access from the threads to the global memory.

On CUDA GPU, all the readable data and variables for each thread is stored on the device memory. In CUDA, the bandwidth of the communication between the host, which refers to the PC, and the device, which refers to the GPU, is much lower than the bandwidth of the communication between devices. Therefore, we try to transfer all the necessary data for computation from the host to the device.

Chapter 3

Algorithm and Implementation

This chapter, we will provide detailed study for how to use GPU to solve problem flow problem in parallel.

3.1 Data Structure

In this project, we read the data from a data file and convert them into a defined structure array. We used the data format given in [7] and IEEE power flow test case format. After reading the raw data, we convert the data format to the defined C structure. Table 3.1 shows the bus data format.

Table 3.1: Bus Data

variable name	column	description
bus_i	1	bus number (positive integer)
type	2	bus type (1 = PQ, 2 = PV, 3 = slack, 4 = isolated)
Pd	3	real power demand (MW)
Qd	4	reactive power demand (MVAr)
Gs	5	shunt conductance (MW)
Bs	6	shunt susceptance (MVAr)
area	7	area number (positive integer)
Vm	8	voltage magnitude (p.u.)
Va	9	voltage angle (degrees)
baseKV	10	base voltage (kV)
zone	11	loss zone (positive integer)
Vmax	12	maximum voltage magnitude (p.u.)
Vmin	13	minimum voltage magnitude (p.u.)

Table 3.2 shows the data format of generators.

Table 3.2: Generator Data

variable name	column	description
bus_i	1	bus number (positive integer)
Pg	2	real power output (MW)
Qg	3	reactive power output (MVAr)
Qmax	4	maximum reactive power output (MVAr)
Qmin	5	minimum reactive power output (MVAr)
Vg	6	voltage magnitude setpoint (p.u.)
mBase	7	total MVA base of machine, defaults to baseMVA
status	8	machine status, if it is positive, then machine in service
Pmax	9	maximum real power output (MW)
Pmin	10	minimum real power output (MW)
Pc1	11	lower real power output of PQ capability curve (MW)
Pc2	12	upper real power output of PQ capability curve (MW)
Qc1min	13	minimum reatice power output at PC1 (MVAr)
Qc1max	14	maximum reatice power output at PC1 (MVAr)
Qc2min	15	minimum reatice power output at PC2 (MVAr)
Qc2max	16	maximum reatice power output at PC2 (MVAr)
ramp_{agc}	17	ramp rate for load following/AGC (MW/min)
ramp_{10}	18	ramp rate for 10 minute reserves (MW)
ramp_{30}	19	ramp rate for 30 minute reserves (MW)
ramp_{q}	20	ramp rate for reactive power (2 sec time-scale) (MVAr/min)
apf	21	area participation factor

Table 3.3 shows the data format of branches.

Table 3.3: Branch Data

variable name	column	description
fbus	1	"from" bus number
tbus	2	"to" bus number
r	3	resistance (p.u.)
x	4	reactance (p.u.)
b	5	total line charging susceptance (p.u.)
rateA	6	MVA rating A (long term rating)
rateB	7	MVA rating B (short term rating)
rateC	8	MVA rating C (emergency rating)
ratio	9	transformer off nominal turns ratio, (taps at "from" bus, impedance at "to" bus)
angle	10	transformer phase shift angle (degrees), positive → delay
status	11	initial branch state, 1 = in-service, 0 = out-of-service
angmin	12	minimum angle difference, $\theta_f - \theta_t$ (degree)
angmax	13	maximum angle difference, $\theta_f - \theta_t$ (degree)
PF	14	real power injected at "from" bus end (MW)
QF	15	reactive power injected at "from" bus end (MVAr)
PT	16	real power injected at "to" bus end (MW)
QT	17	reactive power injected at "to" bus end (MVAr)

3.2 Building Bus Admittance Matrix

In order to build the bus admittance matrix $\mathbf{Y}_{bus}$, we follow the equations given in section 2.1. For each element of $\mathbf{Y}_{bus}$, the data type is *cuFloatComplex*, where is a single-precision complex number defined by CUDA library. The following part of this section shows how we construct $\mathbf{Y}_{bus}$.

Firstly, we construct the branch admittance matrix $\mathbf{Y}_{br}$ and the associated elements $\mathbf{Y}_{ff}, \mathbf{Y}_{ft}, \mathbf{Y}_{tf}$, and $\mathbf{Y}_{tt}$. We use the matrix (2.6) to build each element. Here is the C code for doing this step:

```
1 for (unsigned int i = 1; i <= branchsize; i++)
2 {
3         State = make_cuFloatComplex((br[IDX1F(i)].status),(0.0f));
4         Ys.elements[IDX1F(i)] =
5                 cuCdivf(State,
6                 make_cuFloatComplex((br[IDX1F(i)].r),
```

```
7                   (br[IDX1F(i)].x)));
8
9           Bc.elements[IDX1F(i)] =
10                  cuCmulf(State,
11                  make_cuFloatComplex((br[IDX1F(i)].b),(0.0f)));
12
13          if (br[IDX1F(i)].ratio != 0)
14          {
15                  tap.elements[IDX1F(i)] =
16                          make_cuFloatComplex((br[IDX1F(i)].ratio),
17                          (0.0f));
18          }
19          else
20          {
21                  tap.elements[IDX1F(i)] =
22                          make_cuFloatComplex(1,(0.0f));
23          }
24          theta = 180/pi*br[IDX1F(i)].angle;
25
26          tap.elements[IDX1F(i)] =
27                  cuCmulf((tap.elements[IDX1F(i)]),
28                  make_cuFloatComplex(((float)cos(theta)),
29                  ((float)sin(theta))));
30
31
32          Ytt.elements[IDX1F(i)] =
33           cuCaddf(Ys.elements[IDX1F(i)],
34           make_cuFloatComplex((0.0f),
35           (cuCrealf(Bc.elements[IDX1F(i)])/2)));
```

```
36
37          Yff.elements[IDX1F(i)] =
38           cuCdivf(Ytt.elements[IDX1F(i)],
39           cuCmulf(tap.elements[IDX1F(i)],tap.elements[IDX1F(i)]));
40
41          Ys.elements[IDX1F(i)] =
42           make_cuFloatComplex((-1*cuCrealf(Ys.elements[IDX1F(i)])),
43           (-1*cuCimagf(Ys.elements[IDX1F(i)])));
44
45          Yft.elements[IDX1F(i)] =
46                  cuCdivf(Ys.elements[IDX1F(i)],
47                  cuConjf(tap.elements[IDX1F(i)]));
48
49          Ytf.elements[IDX1F(i)] =
50                  cuCdivf(Ys.elements[IDX1F(i)],
51                  tap.elements[IDX1F(i)]);
52
53  }
```

Then we build the shunt admittance $\mathbf{Y}_{sh}$ by using (2.13). Here is the C code:

```
1  for (unsigned int i = 1; i <= bussize; i++)
2  {
3          Ysh.elements[IDX1F(i)] =
4                  make_cuFloatComplex((bus[IDX1F(i)].Gs/baseMVA),
5                  (bus[IDX1F(i)].Bs/baseMVA));
6  }
```

Before implementing (2.17), (2.18) and (2.19), we need to build the branch connection matrix. This listing shows how we build the connection matrix:

```
1  for (unsigned int i = 1; i <= branchsize; i++)
2  {
```

```
3           unsigned int j = br[IDX1F(i)].fbus;
4           unsigned int k = br[IDX1F(i)].tbus;
5
6           Cf.elements[IDX2F(i,j,branchsize)] =
7                   make_cuFloatComplex((1.0f),(0.0f));
8           Ct.elements[IDX2F(i,k,branchsize)] =
9                   make_cuFloatComplex((1.0f),(0.0f));
10  }
```

Now, we have enough for building $\mathbf{Y}_{bus}$ by using (2.17), (2.18) and (2.19). Since in this project, we consider a situation that there are thousands buses in the system. It is not efficient to compute $\mathbf{Y}_f$, $\mathbf{Y}_t$ and $\mathbf{Y}_{bus}$ in a sequence way because a large amount of computation is required. For each element of $\mathbf{Y}_f$ or $\mathbf{Y}_t$, $2n_b$ complex multiplications and $2n_b$ complex additions are needed, where n_b is the number of buses. Therefore, we use the GPU to build $\mathbf{Y}_f$ and $\mathbf{Y}_t$. Each thread computes the value of one element in $\mathbf{Y}_f$ and $\mathbf{Y}_t$. Here is the kernel function:

```
1   __global__ void MakeYfYt(const cVector Y1, const cVector Y2,
2                   const cMatrix Cf, const cMatrix Ct, cMatrix Y)
3   {
4           const unsigned int tx = threadIdx.x+1;
5           cuFloatComplex temp1;
6           __shared__ cuComplex C0;
7           C0 = make_cuFloatComplex((0.0f),(0.0f));
8           __syncthreads();
9
10
11          for (unsigned int j = 1;j <= Y.width; j++)
12          {
13                  Y.elements[IDX2F(tx,j,Y.height)] = C0;
14                  temp1 = cuCmulf(Y1.elements[IDX1F(tx)],
```

```
15                      Cf.elements[IDX2F(tx,j,Cf.height)]);
16
17                      Y.elements[IDX2F(tx,j,Y.height)] =
18                              cuCaddf(Y.elements[IDX2F(tx,j,Y.height)],
19                              temp1);
20
21                      temp1 = cuCmulf(Y2.elements[IDX1F(tx)],
22                              Ct.elements[IDX2F(tx,j,Ct.height)]);
23                      Y.elements[IDX2F(tx,j,Y.height)] =
24                              cuCaddf(Y.elements[IDX2F(tx,j,Y.height)],
25                              temp1);
26
27              }
28
29              __syncthreads();
30
31      }
```

For this kernel function, the dimension of the block is $n_l \times 1$, where n_l is the number of buses, and the dimension of the grid is 1×1. Here we show how we call the kernel function:

```
1   dim3 dimBlockYtYf(YDevice.Yf.height,1);
2   dim3 dimGridYtYf(1,1);
3
4
5   //Yf
6   MakeYfYt<<<dimGridYtYf,dimBlockYtYf>>>
7               (YffDevice, YftDevice, CfDevice, CtDevice, YDevice.Yf);
8   CheckCUDAKernalError("Computer Yf");
9
10
```

```
11 //Yt
12 MakeYfYt<<<dimGridYtYf, dimBlockYtYf>>>
13         (YtfDevice, YttDevice, CfDevice, CtDevice, YDevice.Yt);
14 CheckCUDAKernalError("Computer Yt");
```

For $\mathbf{Y}_{bus}$, we use kernel function to build it as well. Here is the kernel function:

```
1  __global__ void MakeYbus(const cMatrix Cf, const cMatrix Ct,
2  const cMatrix Yf, const cMatrix Yt,
3  const cVector Ysh, cMatrix Ybus)
4  {
5  const int tx = blockIdx.x * blockDim.x + threadIdx.x+1;
6  //const int ty = blockIdx.y * blockDim.y + threadIdx.y+1;
7  cuFloatComplex temp1, temp2, temp3;
8
9  for (unsigned int ty = 1; ty <= Ybus.width; ty++)
10 {
11
12
13         temp1 = make_cuFloatComplex((0.0f),(0.0f));
14         temp2 = make_cuFloatComplex((0.0f),(0.0f));
15         temp3 = make_cuFloatComplex((0.0f),(0.0f));
16         //temp4 = make_cuFloatComplex((0.0f),(0.0f));
17
18         for (unsigned int k = 1; k <= Yf.height; k++)
19         {
20                 temp3 = cuCmulf(
21                         Cf.elements[IDX2F(k,tx,Cf.height)],
22                         Yf.elements[IDX2F(k,ty,Yf.height)]);
23
24                 temp1 = cuCaddf(temp1,temp3);
```

```
25
26                      temp3 = cuCmulf(
27                              Ct.elements[IDX2F(k,tx,Ct.height)],
28                              Yt.elements[IDX2F(k,ty,Yt.height)]);
29
30                      temp2 = cuCaddf(temp2,temp3);
31              }
32
33              __syncthreads();
34
35              Ybus.elements[IDX2F(tx,ty,Ybus.height)] =
36                      cuCaddf(temp1,temp2);
37
38              if (tx == ty)
39              {
40                      Ybus.elements[IDX2F(tx,ty,Ybus.height)] =
41               cuCaddf(
42               Ybus.elements[IDX2F(tx,ty,Ybus.height)],
43               Ysh.elements[IDX1F(tx)]);
44              }
45
46              __syncthreads();
47
48  }
49
50  }
```

For this kernel function, the dimension of the block is $n_b \times 1$ and the grid is 1×1. Here shows how we call the kernel function:

```
1  //Ybus = Cf'Yf+Ct'Yt+[Ysh]
```

```
2  dim3 dimBlockYbus(YHost.Ybus.height,1,1);
3  dim3 dimGridYbus(1,1);
4  MakeYbus<<<dimGridYbus,dimBlockYbus>>>
5  (CfDevice, CtDevice, YDevice.Yf,
6  YDevice.Yt, YshDevice, YDevice.Ybus);
7
8  CheckCUDAKernalError("Computer Ybus");
```

3.3 Solving Power Flow Problem

Here shows the steps for us to compute the solution for power flow problem:

1. Read data from data file
2. Convert the data into C structure
3. Create the bus admittance matrix $\mathbf{Y}_{bus}$
4. Initialize the complex voltage vector $\mathbf{V}^{(0)}$ on host
5. Create the complex power injection vector $\mathbf{S}^{(0)}$ on host
6. Transfer $\mathbf{V}^{(0)}$, $\mathbf{S}^{(0)}$ and $\mathbf{Y}_{bus}$ from host to device
7. Run kernel functions to compute $\mathbf{V}$
8. Compute $\mathbf{S}$ based on $\mathbf{V}$
9. Transfer $\mathbf{V}$ and $\mathbf{S}$ from device to host
10. Free memory space on device
11. Print Results

In this section, we will mainly present how to use the kernel function to compute $\mathbf{V}$. Recall (2.36) and (2.37). Both have been written in an element-based format. Here, we use the similar idea from the computation of $\mathbf{Y}_{bus}$, which uses each thread to compute

one element. Since this solution is founded by the Jacobi method, which is an iterative algorithm, we use the algorithm described in Algorithm 1. Here is the listing shows the algorithm in C:

```
1  for (unsigned int i = 0; i < 1000; i++)
2  {
3  //for PQ bus
4  VDeviceNew1 = VDevice;
5  ComputePQBus<<<dimGridPQ, dimBlockPQ>>>
6          (YDevice.Ybus, BusTypeDevice.pq,
7          SbusDevice, VDevice, VDeviceNew1);
8
9  //for PV bus
10 VDeviceNew2 = VDeviceNew1;
11 ComputePVBus<<<dimGridPV, dimBlockPV>>>
12         (YDevice.Ybus, BusTypeDevice.pv, SbusDevice,
13         SbusDeviceNew, VDeviceNew1, VDeviceNew2, VmDevice);
14
15 if (maxElement(VDevice, VDevice2) < tau)
16         break;
17
18 SbusDevice = SbusDeviceNew;
19 VDevice = VDeviceNew2;
```

In this listing, the maximum number of iteration is 1000. In each iteration, we check the maximum absolute error. If the error is smaller than tau, the Jacobi method converges.

In the listing above, we follow the computation algorithm for power flow system case II, which is given in section 2.2.1. Firstly, we compute the complex voltage for PQ bus by using (2.36). Here is a listing shows the kernel function:

```
1  __global__ void ComputePQBus(const cMatrix Ybus, const rVector pq,
2          const cVector Sbus, const cVector Vold, cVector V)
```

```
3  {
4          unsigned int tx = threadIdx.x+1;
5          cuComplex temp1,temp2,temp3;
6          //tx = 2;
7          unsigned int k = pq.elements[IDX1F(tx)];
8
9
10         temp2 = make_cuFloatComplex(0,0);
11
12         temp1 = cuCdivf(Sbus.elements[IDX1F(k)],
13                 Vold.elements[IDX1F(k)]);
14         temp1 = cuConjf(temp1);
15
16
17
18         for (unsigned int j = 1; j <= Vold.length; j++)
19         {
20                 temp3 =
21                         cuCmulf(
22                         Ybus.elements[IDX2F(k,j,Ybus.height)],
23                         Vold.elements[IDX1F(j)]);
24
25                 temp1 = cuCsubf(temp1,temp3);
26         }
27
28
29         temp1 = cuCdivf(temp1,
30                 Ybus.elements[IDX2F(k,k,Ybus.height)]);
31
```

```
32          V.elements[IDX1F(k)] =
33                  cuCaddf(Vold.elements[IDX1F(k)],temp1);
34  }
```

The dimension of the block is $n_b \times 1$ and the grid is 1×1.

For the PV bus, we follow equation (2.37). This listing shows the kernel function:

```
1  __global__ void ComputePVBus(const cMatrix Ybus, const rVector pv,
2  const cVector Sbusold, cVector Sbus, const cVector Vold,
3  cVector V, const rVectorf Vm)
4  {
5           unsigned int tx = threadIdx.x+1;
6           unsigned int k = pv.elements[IDX1F(tx)];
7           cuComplex temp1,temp2;
8
9           temp1 = make_cuFloatComplex(0,0);
10
11          //Y(k,:)*V
12          for (unsigned int j = 1; j<= Vold.length; j++)
13          {
14                  temp2 =
15                  cuCmulf(Ybus.elements[IDX2F(k,j,Ybus.height)],
16                  Vold.elements[IDX1F(j)]);
17
18                  temp1 = cuCaddf(temp1,temp2);
19          }
20
21
22          temp2 =
23                  cuCmulf(Vold.elements[IDX1F(k)],
24                  cuConjf(temp1));
```

```
25
26              Sbus.elements[IDX1F(k)] =
27                      make_cuFloatComplex(
28                      cuCrealf(Sbusold.elements[IDX1F(k)]),
29                      cuCimagf(temp2));
30
31              temp2 = cuCdivf(Sbus.elements[IDX1F(k)],
32                      Vold.elements[IDX1F(k)]);
33              temp2 = cuConjf(temp2);
34              temp2 = cuCsubf(temp2,temp1);
35              temp2 = cuCdivf(temp2,
36                      Ybus.elements[IDX2F(k,k,Ybus.height)]);
37              temp2 = cuCaddf(Vold.elements[IDX1F(k)],temp2);
38
39              V.elements[IDX1F(k)] =
40                make_cuFloatComplex(
41                cuCrealf(temp2)/cuCabsf(temp2)*Vm.elements[IDX1F(k)],
42                cuCimagf(temp2)/cuCabsf(temp2)*Vm.elements[IDX1F(k)]);
43
44
45  }
```

The dimension of the block is $n_b \times 1$ and the grid is 1×1.

Chapter 4

Performance Analysis

In this project, we use the IEEE power flow test cases to evaluate the performance of the CUDA-based power flow solver. We use the 9-bus case, 14-bus case, 57-bus case, 118-bus case, and 300-bus case. In addition, the Matlab-based power flow software, MATPOWER [12], is used as a benchmark.

Figure 4.1 shows the execution time for CUDA-based power flow solver and MATPOWER power flow solver. The MATPOWER ran on a PC with Intel(R) Pentium(R) Dual CPU. The CPU has two threads and each thread operates at 2 GHz. The operation system is 32-bit Windows 7. The Matlab version is 7.11.0.584 (R2010b). For the power flow solver, the error tolerance τ is 10^{-4}. The MATPOWER uses the Jacobi method as well.

Table 4 provides more details about Figure 4.1. Obviously, the CUDA-based power flow solver is faster than the MATPOWER power flow solver, especially when the number of bus is large. When the number of bus is small, such as 9-bus case and 14-bus case, the execution time about 18 times and about 27 times faster than that of the MATPOWER.

Table 4.1: Execution time for power flow solver

number of bus	CUDA (in second)	MATPOWER (in second)	ratio (MATPOWER/CUDA)	number of iterations
9	0.040	0.7310	18.275	220
14	0.020	0.550	27.500	112
57	0.100	0.901	9.010	540
118	0.188	5.030	26.755	1000
300	0.378	14.752	39.027	1000

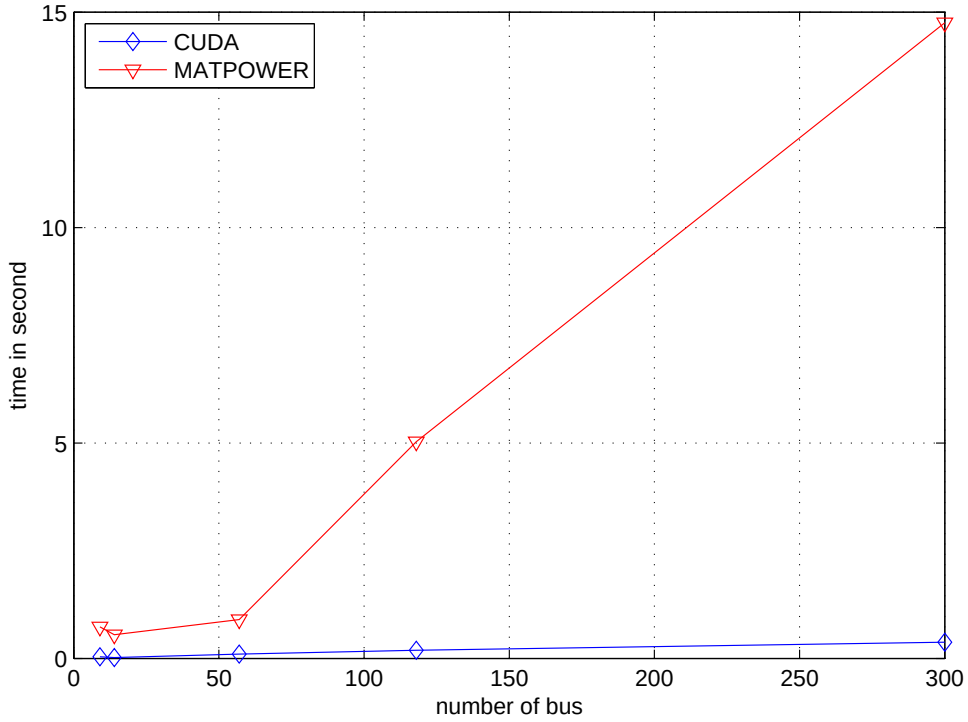

Figure 4.1: Execution time for CUDA-based power flow solver and MATPOWER power flow solver

When the number of bus increases to 57, the execution time ratio is only about 9. The reason is the latency for reading data from global memory is increased. In CUDA GPU, each thread reads the data from the global memory in a sequential way. When the latency of computation on each thread is shorter than the latency of reading data from the memory, the thread will just wait. When the number of buses increases to 118 and 300, the execution time ratio also increases. Although the latency of reading data from global memory also increases, at the same time, the latency of computation on each threads is extended as well. At this point, the computation time is larger than the memory access time. Therefore, the long latency of memory access is hired.

Chapter 5

Conclusion

In summary, in this project, we developed and converted the power flow solver from its classic approach, which is implemented in sequential way, to a new approach, which is implemented in a parallel algorithm. The developed parallel algorithm is implemented on a CUDA-based GPU. Compared with the classic approach, the parallel approach can save the execution time significantly.

There is still a large room for the further development on this algorithm. For example, in the computation of P_i, Q_i, $|V_i|$ and θ_i, the elements of $\mathbf{Y}_{bus}$ are remaining the same. Currently, we save $\mathbf{Y}_{bus}$ in the global memory. If we can save it in the constant memory, the memory access latency can be reduced. In addition, if some elements can be transferred from the global memory or the constant memory to the shared memory, the memory access latency can be reduced as well.

Appendix A

NVIDIA GeForce GTS 250 GPU

Table A.1: NVIDIA GeForce GTS 250 GPU Specification

CUDA Driver Version:	3.20
CUDA Capability Major version number:	1.1
Total amount of global memory:	523829248 bytes
Multiprocessors:	16 MPs
Cores/MP:	8 Cores/MP
Cores:	128 Cores
Total amount of constant memory:	65536 bytes
Total amount of shared memory per block:	16384 bytes
Total number of registers available per block:	8192
Warp size:	32
Maximum number of threads per block:	512
Maximum sizes of each dimension of a block:	$512 \times 512 \times 64$
Maximum sizes of each dimension of a grid:	$65535 \times 65535 \times 1$
Maximum memory pitch:	2147483647 bytes
Clock rate:	1.50 GHz
Host to Device Bandwidth:	1081.5 MB/s
Device to Host Bandwidth:	924.7 MB/s
Device to Device Bandwidth:	48163.1 MB/s

Bibliography

[1] C. Nvidia, "NVIDIA CUDA C Programming Guide Version 3.2," *NVIDIA: Santa Clara, CA*, 2010.

[2] D. Kirk and W. Wen-mei, *Programming massively parallel processors: A Hands-on approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2010.

[3] R. Brown, "Impact of Smart Grid on distribution system design," in *Power and Energy Society General Meeting-Conversion and Delivery of Electrical Energy in the 21st Century, 2008 IEEE*, pp. 1–4, Ieee, 2008.

[4] J. Grainger and W. Stevenson, *Power system analysis*, vol. 621. McGraw-Hill, 1994.

[5] S. Yamshchikov, A. Zhao, *Financial Computations on the GPU*. PhD thesis, WORCESTER POLYTECHNIC INSTITUTE, 2008.

[6] A. Bergen, *Power systems analysis*. Prentice Hall, Inc., Old Tappan, NJ, 1986.

[7] R. Zimmerman, C. Murillo-Sánchez, and R. Thomas, "MATPOWER: Steady-state operations, planning, and analysis tools for power systems research and education," *Power Systems, IEEE Transactions on*, no. 99, pp. 1–8, 2011.

[8] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM, 1994.

[9] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[10] Z. Zhang, Q. Miao, and Y. Wang, "CUDA-Based Jacobi's Iterative Method," in *2009 International Forum on Computer Science-Technology and Applications*, pp. 259–262, IEEE, 2009.

[11] C. Nvidia, "NVIDIA CUDA C Best Practices Guide Version 3.2," *NVIDIA: Santa Clara, CA*, 2010.

[12] R. Zimmerman, "Matpower 4.0 b4 Users Manual," 2010.