DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING, SECOND CYCLE, 30 CREDITS

STOCKHOLM, SWEDEN 2020

Comparing Different Approaches for Solving Large Scale Power Flow Problems on the CPU and GPU with the Newton-Raphson Method

MANOLO DORTO

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Comparing Different Approaches for Solving Large Scale Power Flow Problems on the CPU and GPU with the Newton-Raphson Method

MANOLO D'ORTO

Master in Computer Science
Date: December 14, 2020
Supervisor: Jing Gong
Examiner: Stefano Markidis
School of Electrical Engineering and Computer Science
Host company: Svenska Kraftnät
Swedish title: En jämförande studie om olika tillvägagångssätt för att lösa stora flödesproblem på CPU och GPU med Newton-Raphsons metod

Abstract

Power system modelling is increasing in importance. It is vital for power system operations and transmission grid expansions, and therefore the future energy transition. The Swedish power system is under fast development to face the progress of more flexible demand, a higher share of distributed renewable sources, and updated capacities. In order to ensure the power system's secure physical capacities on a real-time basis in the future, the power system models must be able to handle this increased complexity. Hence, more efficient modelling and reduced computational time are necessary in order to secure the efficient daily operation of the Swedish grid.

This thesis focuses on using the Newton-Raphson method to solve the power flow problem. The most computationally demanding part of the Newton-Raphson method is solving the linear equations at each iteration. Therefore, this study investigates different approaches to solve the linear equations on both CPU and GPU. Six different approaches were developed and evaluated in this thesis. Two of these run entirely on CPU while other two of these run entirely on GPU. The remaining two are hybrid approaches that run on both CPU and GPU. The main difference between the six approaches is where the linear equations are executed. However, all approaches either use LU or QR factorization to solve the linear equations. Two different hardware platforms were used to conduct the experiments, namely one single NVIDIA Quadro T2000 GPU on a laptop and one single NVIDIA V100 GPU on Kebnekaise system at HPC2N.

The results show that the GPU gives better performance compared to the CPU for larger power flow problems. The results also show that the best performing version is a hybrid method where the Jacobian matrix is assembled on GPU; the preprocessing with KLU analysis is preformed on the CPU; and finally the linear equations are solved on the GPU. If the data transfers between the CPU and GPU are not considered, the hybrid version yielded a speedup factor of 46 in comparison with the baseline CPU version using the LU algorithm on the laptop. This speedup was obtained on the largest case with 9241 buses. Furthermore, the execution time of the hybrid version on the Kebnekaise system was approximately 114 times faster than the baseline CPU version on the laptop.

Sammanfattning

Modellering av kraftsystem ökar i betydelse. Det är avgörande för driften av kraftsystemet och utbyggnad av nätet och därmed den framtida energiomställningen. Det svenska kraftsystemet är under snabb utveckling för att möta en mer varierande efterfrågan, en högre andel förnybara energikällor och uppdaterade kapacitetsbestämningsmetoder. För att i framtiden kunna säkerställa kraftsystemets säkra fysiska kapaciteter i realtid, måste kraftsystemmodellerna kunna hantera denna ökade komplexitet. Därför är effektivare modellering och reducerad beräkningstid nödvändig för att säkerställa effektiv daglig drift av det svenska nätet.

Detta examensarbete fokuserar på Newton-Raphsons metod för att lösa stora lastflödesproblem. Den mest beräkningstunga delen av Newton-Raphsons metod är att lösa de linjära ekvationerna vid varje iteration. Därför undersöker denna studie olika metoder för att lösa dessa ekvationer både på CPU och GPU. Sex olika metoder har utvecklats och utvärderats. Två av dessa körs endast på CPU, ytterligare två körs enbart på GPU och de två sista är hybridmetoder som körs både på CPU och GPU. Den största skillnaden mellan de sex versionerna är var de exekverades. Alla tillvägagångssätt använder någon form av LU-faktorisering eller QR-faktorisering för att lösa de linjära ekvationerna. Två olika hårdvaruplattformar användes för att genomföra experimenten, nämligen ett NVIDIA Quadro T2000 GPU på en bärbar dator och ett NVIDIA V100 GPU på HPC2Ns Kebnekaise-system.

Resultaten visar att GPU ger bättre prestanda jämfört med CPU för stora lastflödesproblem. Resultaten visar även att den version med bäst prestanda är en hybridmetod där Jacobimatrisen har konstruerats på GPU; förbehandlingarna sker med hjälp av KLU-analys på CPU; och slutligen löses de linjära ekvationerna på GPU. När dataöverföringarna mellan CPU och GPU inte togs i beaktande, var hybridversionen 46 gånger snabbare än CPU-version med LU-faktorisering på den bärbara datorn i fallet med 9241 bussar. Dessutom var hybridversionen ungefär 114 gånger snabbare när den exekverades på V100 GPU-systemet jämfört med CPU-versionen med LU-faktorisering på den bärbara datorn.

Acknowledgement

I would like to thank my supervisor Jing Gong for taking his time to help me with whatever help I needed. I would also like to thank Svenska kraftnät and especially the spica team, for giving me the opportunity to do my master thesis at their company and putting the time and effort needed to make sure that I succeeded with my master thesis. I would like to thank Lung Sheng Chien at NVIDIA Corporation to share his code for this master thesis.

The computations were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at HPC2N partially funded by the Swedish Research Council through grant agreement no. 2018-05973.

Contents

Chapter 1

Introduction

In today's society, people's lives are partly dictated by electricity. Electricity is everywhere around us, it powers our cars, computers, smart locks, alarms, banking systems, and other critical systems. This makes it extremely important that our electrical infrastructure is robust and works as intended. Our society is constantly trying to reduce the carbon dioxide emissions, which leads to new renewable energy sources being added to our power grid. As more components are added to the power grid, the calculations related to it become more demanding in both resources and computation times.

Svenska kraftnät (SVK) is responsible for the Swedish electrical power infrastructure and they operate by a so-called $N - 1$ criterion. This means that if a single component fails in the power grid, the components remaining in the system must be able to accommodate the new operating conditions without violating the security limits [1]. Consequently, SVK constantly solves the power flow problem to ensure that the system always operates within the necessary safety margins. Figure 1.1 shows the Swedish power grid that illustrates the size of the Swedish electric power infrastructure.

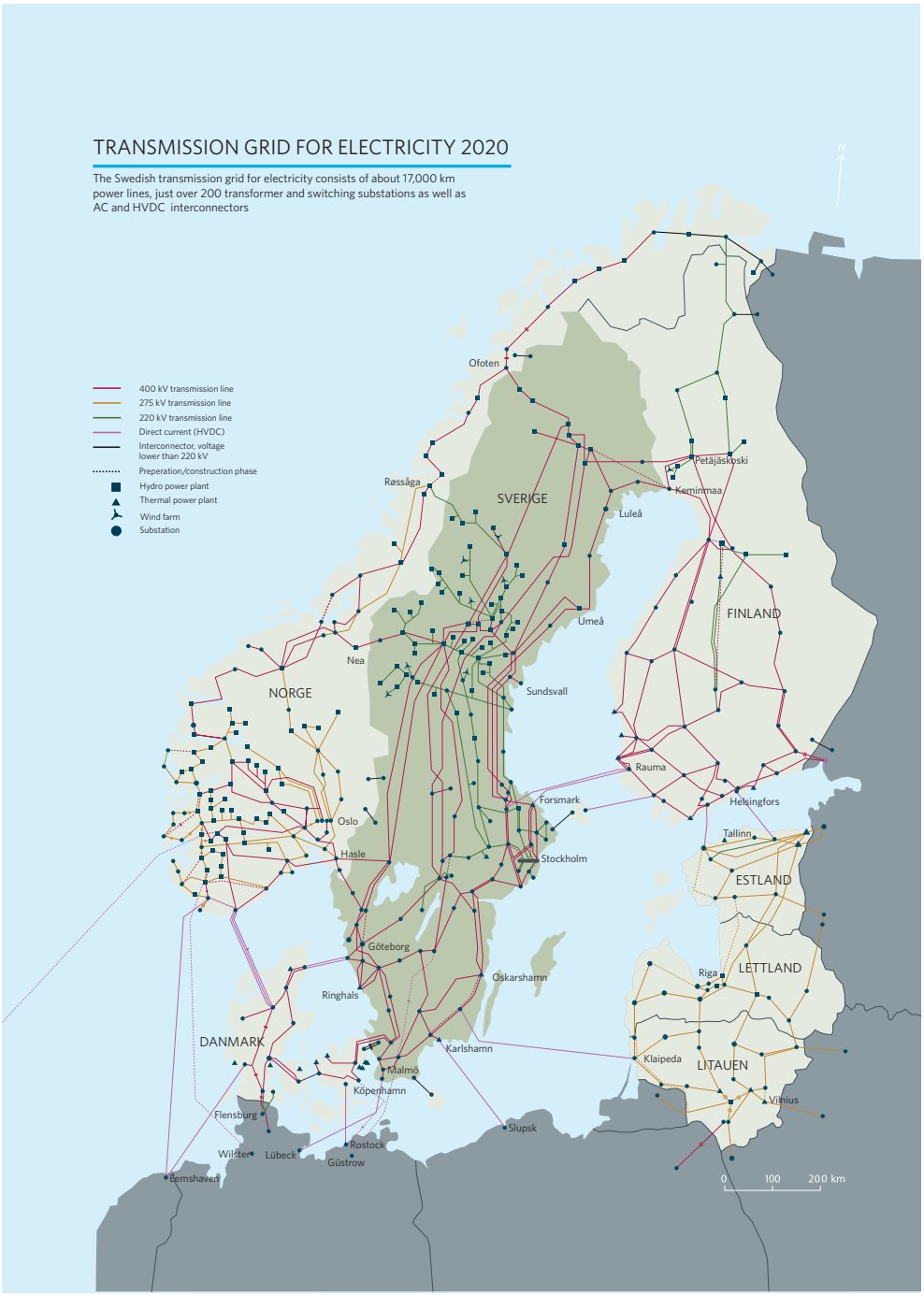

Figure 1.1: The Swedish power grid

1.1 Project Description

At SVK the power flow problems are currently solved on central processing unit (CPU). This is done by employing two different methods, one is based on the Newton-Raphson method and the other is entirely created and developed by Svenska kraftnät. The developers at SVK are interested in speeding up the execution time of the power flow problem by moving parts of, or the whole program, to graphics processing unit (GPU) to see if any significant boost in performance can be gained. The goal of this master thesis is to explore the possibilities of moving the calculations related to the power flow problem on GPU.

1.1.1 Research Question

Will the computation using the Newton-Raphson method applied to large scale power flow problems perform better on the CPU, GPU or a combination of both?

1.2 Methodology

To answer the research question, six different approaches to solve the power flow problems have been developed and evaluated. Two versions were executed on the CPU as a way to benchmark the other versions. The CPU versions replicate the way SVK solve the power flow problems that use the Newton-Raphson method with LU factorization for the linear equations at each iteration.

The four remaining versions included two versions that were partly executed on the CPU and on the GPU and two versions that were executed almost entirely on the GPU. Two different hardware platforms have been used to better benchmark the developed versions. To evaluate the different parts of the developed versions, extensive experiments have been performed to get a better understanding of which parts of the program consume most of the execution times.

1.3 Limitations

This thesis only focused on the Newton-Raphson method but not other methods such as Gauss Seidel's method etc. Furthermore, the linear equations at each iteration were solved using direct rather than iterative techniques, more specifically this study focused on LU and QR factorizations to solve the linear equations. The focus of solving the power flow problem lies on sparse factorization techniques but dense techniques were used for comparative reasons.

1.4 Thesis Outline

In Chapter 2, the relevant background information needed to understand this thesis is presented. This includes an introduction to the power flow problem, how the Newton-Raphson method is applied to the power flow problem, and different approaches on how to solve the linear equations. Chapter 2 also explains the sparse storage scheme used in this thesis, how GPU programming works, and some related work is presented. Chapter 3 presents the method used to answer the research question and detailed information about the different implementations. Chapter 4 presents the results from the different experiments performed and the results are discussed in Chapter 5. Finally, Chapter 6 presents the conclusions made from this thesis and presents some ideas for possible future works.

Chapter 2

Background

2.1 Power Flow Problem

Power flow is the analysis of a steady-state power system [2]. It analyses the flow of electrical power in an interconnected system, also called a network. These systems consist of nodes and branches also referred to as buses and edges. The goal of the power flow problem is to calculate the voltage magnitude and voltage angle for each bus in the system [3]. Once voltage magnitudes and angles are known for all buses, the active and reactive power can be calculated. The solution to the power flow problem is usually obtained by solving nodal power balance equations. These types of equations are non-linear and therefore iterative techniques such as the Newton-Raphson method are commonly used.

The general procedure of how to solve the power flow problem is listed below. Step 1 is explained in more detail in Section 2.1.1 and steps 3 and 4 are presented in depth in Section 2.1.2.

1. The admittance matrix Y is constructed
2. An estimation of all unknown variables is made
3. The power flow equations are calculated using the estimates
4. The resulting non-linear equations are solved using e.g. the Newton-Raphson method. The estimates are updated based on the results of these calculations
5. The power mismatches are evaluated, if they are less than a specified tolerance ε, the iteration ends, otherwise steps 3-5 are repeated.

The power flow problem aims to determine the voltages at each bus and the active and reactive power in each line. Based on these variables, buses are divided into three types [3]:

- Slack bus: For the slack bus, both voltage angle and magnitude are specified while the active and reactive powers are unknown. Because the slack bus serves as a reference for all the other buses, each network needs exactly one slack bus.
- Load bus: For these buses, the injected active and reactive powers are known while the voltage angles and magnitudes are unknown. These buses are referred to as PQ buses.
- Voltage-controlled bus: For these buses, the voltage magnitudes and active powers are known while the voltage angles and reactive powers are unknown. The voltage-controlled buses are referred to as PV buses.

A line or branch is a connection between two buses. Figure 2.1 presents a simple network that contains five buses and five lines. When Bus 1 is considered as the slack bus, Bus 2 and 3 are so-called generator buses or PV buses. Buses 4 and 5 are load buses or PQ buses [4].

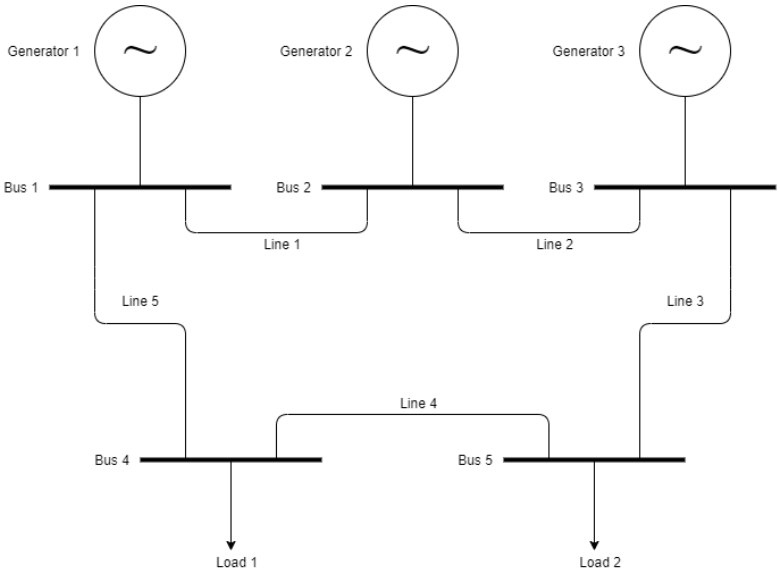

Figure 2.1: A simple network with five buses and five branches

We define N as the total number of buses, N_g as the number of PV buses, and N_l as the number of PQ buses in the network. Consequently, the network in Figure 2.1 has $N = 5$, $N_g = 2$, and $N_l = 2$.

2.1.1 Admittance Matrix and Power Flow Equations

A brief derivation of the power flow equations and an explanation of the admittance matrix is given this Section. A more detailed derivation and explanation can be found in [3][5].

In order to simplify the calculations of the power flow problem, the impedances are converted to admittances as seen in Equation (2.1).

$$y_{ij} = \frac{1}{z_{ij}} = \frac{1}{r_{ij} + jx_{ij}} \tag{2.1}$$

where:

y_{ij} = the admittance between bus i and j

z_{ij} = the impedance between bus i and j

r_{ij} = the resistance between bus i and j

x_{ij} = the reactance between bus i and j

The admittance matrix, usually referred to as the Y-matrix, is an abstract mathematical model of the system [3]. The Y-matrix consists of admittance values of lines and buses. It is square and symmetrical with the number of rows and columns equal to the number of buses in the system.

$$Y = \begin{bmatrix} Y_{11} & Y_{12} & \dots & Y_{1N} \\ Y_{21} & Y_{22} & \dots & Y_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ Y_{N1} & Y_{N2} & \dots & Y_{NN} \end{bmatrix} \tag{2.2}$$

The values of the diagonal elements Y_{ii} are equal to the sum of the admittances connected to bus i as seen in Equation (2.3). The off-diagonal elements Y_{ij} are equal to the negative admittance between bus i and j as seen in Equation (2.4). It is important to note that the Y-matrix is sparse for large systems as most buses do not have a branch between them.

$$Y_{ii} = \sum_{\substack{j=1 \\ j \neq i}}^{N} y_{ij} \tag{2.3}$$

$$Y_{ij} = Y_{ji} = -y_{ij} \tag{2.4}$$

The net injected power at any bus i can be calculated using the bus voltage V_i, its neighbouring bus voltages V_j and the admittances between the neighbouring buses y_{ij}. Using Kirchhoff's current law we get Equation (2.5).

$$I_i = \sum_{j=1}^{N} Y_{ij} V_j \tag{2.5}$$

The power equation at any bus can be written as Equation (2.6).

$$S_i = P_i + jQ_i = V_i I_i^* \tag{2.6}$$

where:

P_i = The active power at bus i

Q_i = The reactive power at bus i

Combining Equations (2.5) and (2.6) we finally get the power flow equations (2.7).

$$\begin{aligned} P_i &= V_i \sum_{j=1}^{N} G_{ij} V_j \cos(\theta_{ij}) + B_{ij} V_j \sin(\theta_{ij}) \\ Q_i &= V_i \sum_{j=1}^{N} G_{ij} V_j \sin(\theta_{ij}) - B_{ij} V_j \cos(\theta_{ij}) \end{aligned} \tag{2.7}$$

where:

θ_{ij} = $\theta_i - \theta_j$, the difference in phase angle between bus i and j,

G_{ij} = The real part of Y_{ij}

B_{ij} = The imaginary part of Y_{ij}

2.1.2 The Newton-Raphson Method

There are several different methods that can be used to solve the power flow problem. The most common methods are the Newton-Raphson method, the Gauss-Seidel method, and the Fast-Decoupled method [6]. The Newton-Raphson method requires less iterations to find a solution compared to the Gauss-Seidel method but the computation time for each iteration is larger [7][8]. For the Newton-Raphson method and the Fast-Decoupled method, the number of iterations needed to find a solution is not dependent of the size of the system. This is not true for the Gauss-Seidel method. The Newton-Raphson method is

more robust compared to the Fast-Decoupled method when it comes to heavily loaded systems and therefore the Fast-Decoupled method was not viable for this thesis [9]. Furthermore, the Newton-Raphson method gives the most accurate result. This thesis focuses on the Newton-Raphson method since it is the most accurate of the three methods and the number of iterations for convergence does not increase with the size of the system.

The Newton-Raphson method is an iterative method. It is based on the Taylor expansion to find the roots of real-valued functions [3]. The unknowns variables in a function can be determined using Taylor expansion approximation. The Newton-Raphson method starts with an initial guess for all the unknown variables. Applying Taylor expansion approximation on the function with unknowns and neglecting the higher order terms, the function can be approximated by the first two terms of the Taylor expansion. This process is repeated until a specified accuracy has been achieved.

When applying Taylor expansion on equation (2.7), and neglecting all higher order terms we get Equation (2.8).

$$\begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} = \begin{bmatrix} J_1 & J_2 \\ J_3 & J_4 \end{bmatrix} \begin{bmatrix} \Delta \theta \\ \Delta V \end{bmatrix} \tag{2.8}$$

where:

$$\Delta P = \begin{bmatrix} P_{2,specified} - P_{2,calculated} \\ \vdots \\ P_{N,specified} - P_{N,calculated} \end{bmatrix}$$

$$\Delta Q = \begin{bmatrix} Q_{2,specified} - Q_{2,calculated} \\ \vdots \\ Q_{N,specified} - Q_{N,calculated} \end{bmatrix}$$

$J_1 = \frac{\partial P}{\partial \theta}$, $J_2 = \frac{\partial P}{\partial V}$, $J_3 = \frac{\partial Q}{\partial \theta}$, $J_4 = \frac{\partial Q}{\partial V}$

To solve for the deviation, the inverse of the Jacobian matrix is required at each iteration, seen Equation (2.9).

$$\begin{bmatrix} \Delta \theta \\ \Delta V \end{bmatrix} = \begin{bmatrix} J_1 & J_2 \\ J_3 & J_4 \end{bmatrix}^{-1} \begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} \tag{2.9}$$

In Equations (2.8) and (2.9) it is important to note that the slack bus is assumed to be bus number 1 which is why the indices start from 2 in Equation (2.8).

The voltage angles and magnitudes are updated at each iteration, and are calculated in Equation (2.10).

$$\begin{bmatrix} \theta^k \\ V^k \end{bmatrix} = \begin{bmatrix} \theta^{k-1} \\ V^{k-1} \end{bmatrix} + \begin{bmatrix} \Delta\theta^k \\ \Delta V^k \end{bmatrix} \tag{2.10}$$

where:

k = current iteration

The iteration is repeated until each element in $[\Delta P^k \quad \Delta Q^k]^T$ is less than a specified tolerance ϵ.

As seen in equation (2.8) the Jacobian matrix consists of four sub-matrices. The Jacobian matrix is a square matrix with the number of rows and columns equal to $N - 1 + N_l$. This is because the slack bus is not included and the voltage magnitudes of the PV buses are known, hence those values are not included. Similar to the Y-matrix, the Jacobian matrix is a sparse matrix when the network is large. In Equations (2.11)-(2.18) we can see how the elements of the Jacobian matrix are constructed.

$$J_1(i,i) = \frac{\Delta\partial P_i}{\Delta\partial\theta_i} = V_i \sum_{\substack{j=1 \\ j\neq i}}^{n} V_j(-G_{ij}\sin(\theta_{ij}) + B_{ij}\cos(\theta_{ij})) \tag{2.11}$$

$$J_1(i,j) = \frac{\Delta\partial P_i}{\Delta\partial\theta_j} = V_i V_j(G_{ij}\sin(\theta_{ij}) - B_{ij}\cos(\theta_{ij})), \quad i \neq j \tag{2.12}$$

$$J_2(i,i) = \frac{\partial P_i}{\partial V_i} = 2G_{ij} + \sum_{\substack{j=1 \\ j\neq i}}^{n} V_j(G_{ij}\cos(\theta_{ij}) + B_{ij}\sin(\theta_{ij})) \tag{2.13}$$

$$J_2(i,j) = \frac{\partial P_i}{\partial V_j} = V_j(G_{ij}\cos(\theta_{ij}) + B_{ij}\sin(\theta_{ij})), \quad i \neq j \tag{2.14}$$

$$J_3(i,i) = \frac{\partial Q_i}{\partial \theta_i} = -V_i \sum_{\substack{j=1 \\ j\neq i}}^{n} V_j(G_{ij}\cos(\theta_{ij}) + B_{ij}\sin(\theta_{ij})) \tag{2.15}$$

$$J_3(i,j) = \frac{\partial Q_i}{\partial \theta_j} = V_i V_j(-G_{ij}\cos(\theta_{ij}) - B_{ij}\sin(\theta_{ij})), \quad i \neq j \tag{2.16}$$

$$J_4(i,i) = \frac{\partial Q_i}{\partial V_i} = -2V_i + \sum_{\substack{j=1 \\ j\neq i}}^{n} V_j(G_{ij}\sin(\theta_{ij}) - B_{ij}\cos(\theta_{ij})) \tag{2.17}$$

$$J_4(i,j) = \frac{\partial Q_i}{\partial V_j} = V_j(G_{ij}\sin(\theta_{ij}) - B_{ij}\cos(\theta_{ij})), \quad i \neq j \tag{2.18}$$

2.1.3 Solving Linear Equations

At each iteration of the Newton-Raphson method, the set of linear equations in Equation (2.8) have to be solved. There are several ways of obtaining the solution, either by using so-called iterative linear solvers or direct linear solvers. Iterative linear solvers start with an estimation and then iterate until they converge [10]. Iterative linear solvers do not guarantee that a solution will be found. On the other hand, direct linear solvers do not start with an estimation and converge toward the solution but rather solve the system straight away. When it comes to large systems, iterative linear solvers might yield better performance but for highly sparse matrices direct linear solvers are better suited. Since the Jacobian matrix is highly sparse this thesis focuses on direct linear solvers.

Traditionally, the linear system is solved with an LU factorization of the Jacobian matrix for the Power flow problems. The most expensive part of the Newton-Raphson method is solving the linear equations at each iteration. Experiments have shown that solving the linear equations with LU factorization took about 85% of the total execution time, the experiments were performed on a system with 9493 buses [11]. Two factorization methods, namely QR and LU factorization have been investigated in the thesis.

LU Factorization

To solve the linear system of equations using LU factorization, a matrix A can be factorized into LU. LU consists of two matrices where L is a lower triangular matrix and U is an upper triangular matrix.

Solving the linear system of equations $LUx = b$ is less computationally heavy than solving $Ax = b$ by finding the inverse of A. To solve $LUx = b$, the

equation $Ly = b$ is solved for y, finally x is solved in the equation $Ux = y$.

The complexity for solving linear equations with LU factorization is proportional to $\frac{2}{3}m^3$, where m is the size of the matrix [12]. Every square non-singular matrix has an LU factorization [13].

QR Factorization

Every invertible matrix A has a QR factorization such that $A = QR$, where Q is an orthogonal ($Q^TQ = QQ^T = I$) matrix and R is an upper triangular matrix. To solve for the linear systems $QRx = b$, equation $y = Q^Tb$ is solved for y. Then x is solved from equation $Rx = y$.

The complexity for solving linear equations with QR factorization is proportional to $\frac{4}{3}m^3$, where m is the size of the matrix.

When solving sparse linear systems, reordering schemes can be applied to minimize fill-in. The fill-in of a matrix are the entries that are zero initially but turn into a non-zero value during the execution of an algorithm [14]. The reordering schemes used in this thesis are explained blow.

- **METIS Reordering** METIS is a software package for computing fill-reducing orderings for sparse matrices [15]. In this thesis only one function is used from the METIS software package, namely `METIS_NodeND`. This function computes the orderings to reduce fill-in based on the multilevel nested dissection paradigm. The nested dissection paradigm is based on computing the vertex separator of the graph corresponding to the sparse matrix. Furthermore, the sparsity pattern of the input matrix needs to be symmetric.

- **AMD Reordering** For the AMD reordering, the sparsity pattern of the input matrix needs to be symmetric [16]. The algorithm is based on the observation that when a variable is eliminated, a clique is formed by its neighbours [17]. Each of the edges within the clique contributes to fill-in. Thus, the AMD reordering aims at minimizing the fill-in by forming the smallest possible clique at each step.

- **COLAMD Reordering** One of the main differences between COLAMD and AMD is that COLAMD does not need the sparsity pattern of the input matrix to be symmetrical [17]. COLAMD computes the column permutations without calculating the normal equations. This makes it

a good choice for QR factorization since the QR factorization does not calculated the normal equations.

- **SYMRCM Reordering** Similarly to COLAMD reordering and METIS reordering, the sparsity pattern of the input matrix needs to be symmetrical [18]. The SYMRCM is based on a breadth-first search algorithm. The aim of SYMRCM is to minimize the bandwidth of the matrix. The number of non-zero diagonals over the main diagonal is called upper bandwidth [19]. The number of non-zero diagonals below the main diagonal is called lower bandwidth.

Special techniques such as sparse partial pivoting and KLU director solver have been employed to solve the linear system raised from the power flow problems.

KLU Direct Sparse Solver

KLU is an algorithm presented by Timothy A. Davis [20] that exploits the fact that the sparsity pattern of the Jacobian matrix applied to the power flow problem is exactly the same at each iteration of the Newton-Raphson method. The algorithm is based on LU factorization and has a total of four steps which are listed below.

1. The matrix is permuted into block triangular form
2. A reordering scheme is applied to each created block to reduce fill-in. This is done to reduce the amount of memory needed and the total number of arithmetic operations
3. The diagonal blocks are scaled and factorized according to Gilbert and Peierls' left looking algorithm with partial pivoting [21]
4. Finally, the linear system is solved using block back substitution

Since the sparsity pattern of the Jacobian matrix is the same at each iteration, the future iterations disregard the first two steps [20]. Furthermore, the third step implements a simplification of the left looking algorithm which does not perform partial pivoting. With this, the depth-first search used in Gilbert and Peierls' algorithm can be omitted. This is because the non-zero patterns of L and U are already known from the first iteration.

Sparse Partial Pivoting

In 1988 John R. Gilbert and Timothy Peierls [21] presented an algorithm that factorizes sparse matrices. This algorithm was based on Gaussian elimination with partial pivoting. The basic idea of the algorithm is to predict enough about the sparsity pattern of the sparse matrix in advance to minimize the computation needed for the factorization. More specifically, the sparsity pattern of each column of the factorization is predicted. By using the information gained, the arithmetic of the computation can be minimized. The sparsity of the matrix is found with a depth-first search approach. A detailed explanation of the algorithm can be found in [21].

GLU Direct Sparse Solver

In 2016 Kai He et al. presented a GPU accelerated LU factorization (GLU) solver to solve linear equations [22]. This algorithm is based on a hybrid right-looking LU factorization and is intended for sparse matrices. The left-looking method shows better performance and is easier to implement than the right-looking method but it does not allow for the same level of parallelization as the right-looking method, especially on GPU platforms. Therefore, the right-looking method was chosen for the GLU implementation.

The GLU algorithm performs a total of 4 steps [23]:

1. The MC64 algorithm is used to find a permutation of the sparse matrix
2. The approximate minimum degree algorithm is used to reduce fill-in
3. A symbolic factorization is performed to determine the structure of L and U
4. The hybrid right-looking LU factorization is performed

Note that the first three steps namely the preprocessing steps are performed on the CPU and only the last step is performed on the GPU. Similar to the KLU direct sparse solver, GLU does only perform the preprocessing at the first iteration since the sparsity pattern is known at the future iterations. This leads to a great improvement in performance when subsequent linear systems are solved with the the same sparsity pattern.

When the study was published in 2016 it showed great speedup compared to existing approaches [22]. Since then, several updates of the GLU algorithm have been released to further increase the performance [24][25].

2.2 Sparse Matrix Storage

Both the Jacobian matrix and the Y-matrix are sparse for large networks. They can both be stored in compressed storage formats. The format used in this thesis is compressed sparse row (CSR). This sparse storage scheme uses three vectors to store the information about a matrix [26]. An example can be seen in Figure 2.2. The row pointer vector contains the offset of the first value on each row, and the last element in the row pointer vector contains the total number of non-zeroes in the matrix. The column indices vector contains the column of each of the non-zero elements of the matrix. Finally the array with values contains all the non-zero values of the original matrix.

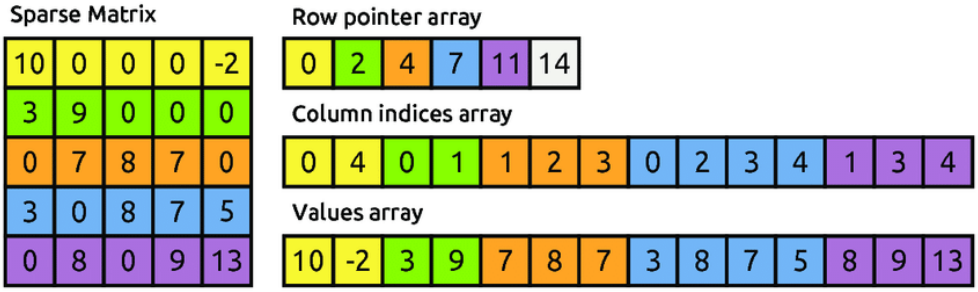

Figure 2.2: A simple example of CSR

2.3 GPU Computing and CUDA

The GPU, or graphics processing unit has come a long way since it was first introduced [27]. GPUs were created to be used as specialized graphics processors that could rapidly produce images for output to a display unit. Today, the GPU is the go-to technology when fast computing is needed since it handles parallel processing. The GPUs have thus been designed to handle a large amount of data, given that the operations are applied to all the data.

CUDA is one of the most popular application programming interfaces for accelerating computing with the GPU [27]. It can rather easily enable code written in C or C++ to run efficiently on GPU for certain problems, e.g. large scale matrix matrix multiplications. The execution time of a program written in C or C++ can be decreased with CUDA if the program has been parallelized on GPU [28]. When working with CUDA the general workflow is as follows:

1. CPU code allocates memory and uploads it to the GPU. The CPU then

launches a function to run on the GPU with a specified number of threads. These types of functions are called kernels

2. The GPU executes the kernel and the CPU either waits for the kernel to finish or it does some other work.

3. Once all the threads running the kernel on the GPU are done, the CPU can synchronize with the GPU and copy the result back from the GPU memory to the CPU memory.

The following sections will describe important concepts when working with CUDA.

2.3.1 Warps, Blocks, and Grids

The threads on the GPU are divided into blocks and grids [27]. There is one or several threads in a block and one or several blocks in each grid. In order to fully utilize the GPU, the optimal balance between the number of threads per block, and the number of blocks per grid needs to be found.

Furthermore, each block is divided into warps. Each warp consists of 32 consecutive threads. When a grid of thread blocks is launched, the thread blocks are distributed to the available streaming multiprocessors (SM). Now each block is divided into warps and all 32 threads are executed in single instruction multiple threads (SIMT) fashion. This means that each thread in the warp executes the same instruction on its own private data.

If branching occurs in the code, it may lead to warp divergence. This is illustrated in the pseudo code below.

if *condition* **then**
| Do something
else
| Do something else
end

Suppose that 16 of the 32 threads in a warp evaluate the condition to true and the remaining 16 threads evaluate it to false. Half of the warp will execute the code in the if block and the other half will execute the code in the else block. This example shows warp divergence i.e. when threads in the same warp execute different instructions. It is evident that this is bad for performance as 16

threads will stay idle while the other 16 execute their condition and vice versa.

It is important to have a good balance between the size of the block and the size of the grid. Ideally, one wants to have enough warps to keep the cores of the device occupied. To keep track of this, the occupancy metric is used. This metric is the ratio between active warps and the maximum number of warps per SM.

2.3.2 GPU Memory

In CUDA, several types of memory are accessible. Some of them are used explicitly by the programmer and some of them are used implicitly. In order for the programmer to achieve high performance, it is important to understand how the GPU memory works. The following sections explain some of the memories available.

Global Memory

Global memory is the most commonly used memory on the GPU. It has the largest memory with the highest latency and it is accessible from any SM throughout the lifetime of the application. Global memory is usually located on the GDDR chips on the GPU.

In order to use this type of memory, the programmer explicitly needs to allocate and free it. This is done with calls to `cudaMalloc` and `cudaFree`. Once memory has been allocated the programmer can copy the data from the CPU to the global memory with `cudaMemcpy`. It takes quite some time to copy data between the CPU and GPU as it goes through the PCIe bus and therefore the number of copy transactions should be minimized.

Global memory is accessible via 32-byte, 64-byte, or 128-byte memory transactions depending on the GPU. This means that each time something is read from the global memory, the memory transaction will always consist of a fixed number of bytes. If the programmer only wants a part of the transaction, the rest of the transaction will be wasted. It is also important that the memory transaction is naturally aligned, meaning that the first address should be a multiple of 32-bytes, 64-bytes, or 128-bytes.

Pinned Memory

Allocated memory on the CPU is by default pageable. This means that the operating system can, at any time, move the physical location of the data. This is bad for the GPU as it cannot safely access data on the CPU as the operating system might have moved it. When transferring data from the CPU to the GPU with `cudaMemcpy`, the CUDA driver first allocates temporary page-locked or pinned memory. This can be avoided by allocating memory with `cudaMallocHost`. However, allocating excessive amounts of pinned memory will eventually result in that the CPU runs out of main memory. It is also important to note that allocating and deallocating pinned memory is more expensive than pageable memory.

Shared Memory

Shared memory is located together with the cache and registers and is therefore much faster than global memory. It is declared in the scope of a kernel but it shares its lifetime with a thread block. When a thread block is finished executing, all the shared memory that was allocated is now freed and will be available for other blocks. The shared memory is shared between all threads in a block meaning that threads outside of the block are not able to access the shared memory of the block.

2.3.3 Coalesced Access

In order to maximize the performance when reading from the global memory, one has to take coalescing into account. Coalesced memory access occurs when all 32 threads in a warp access a continuous chunk of memory. In general, the lower the number of transactions to service a memory request, the better the performance.

2.3.4 Streams

All CUDA operations run in a stream either implicitly or explicitly, this is true for both kernels and data transactions. If nothing is stated the default stream is used, also known as the NULL stream. If the programmer wants to overlap different CUDA operations, streams have to be declared explicitly. Streams can be used to overlap kernels and data transfers. It is important to note that pinned memory needs to be used if one wants to overlap data transfers. When a

stream is created it is a so-called blocking stream, this means that those streams can be blocked waiting for earlier operations in the NULL stream.

2.4 Related Works

The work from Guo et al. [29] investigated different methods to solve the power flow problem with CUDA and C. The different methods were the Gauss-Seidel method, the Newton-Raphson method and the P-Q decoupled method. For solving the linear equations at each iteration of the Newton-Raphson method and the P-Q decoupled method, the Gaussian elimination method was chosen. Furthermore, the sparsity of the Jacobian matrix and Y-matrix were not exploited. The data used in this study was from IEEE and a power flow network called Shandong. The size of the networks can be seen in Table 2.1.

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

Table 2.1: Size of power flow networks used in [29]

Since the method used in this thesis only focused on the Newton-Raphson method, only those results from [29] are presented below in Table 2.2. The table shows the runtimes of the Newton-Raphson method for the different sized networks.

Network	**CPU Runtime(ms)**	**GPU Runtime(ms)**
IEEE9	1.5	9.4
IEEE30	9.8	9.4
IEEE118	313.2	199.7
IEEE300	4689	2684.8
Shandong	583831	10881

Table 2.2: Runtime of the Newton-Raphson method presented in [29]

In the paper *Accelerating Power Flow studies on Graphics Processing Unit* presented by Jaideep Singh and Ipseeta Aruni [30], they investigated and com-

pared the performance of the Newton-Raphson method applied to the power flow problem. The CPU code was written in C++ and the GPU code in CUDA. The CPU version is accelerated with INTEL Math Kernel Library which contains a set of optimized, thread-parallel mathematical functions for solving the nonlinear equations and the matrix multiplications. They did not specify how they solved the linear equations each iteration. However they stated that they used CULA library for the GPU version but it is not clear which method they used within CULA. In this paper they did not consider the sparsity of the Jacobian matrix and Y-matrix.

A summary of the runtime for the creation of the Jacobian matrix can be seen in Tables 2.3 and 2.4, shows the total runtime of the Newton-Raphson method presented in [30].

Jacobian matrix	**CPU Runtime(ms)**	**GPU Runtime(ms)**
14x14	0.022	0.030
53x53	0.133	0.035
181x181	3.131	0.073
551x551	30.311	0.312
1273x1273	171.936	1.320
4702x4702	2441.92	16.524
8877x8877	8802.397	66.449

Table 2.3: Runtime of Creation of the Jacobian Matrix presented in [30]

# of Buses	**CPU Runtime(s)**	**GPU Runtime(s)**
9	0.000261	0.022397
30	0.000597	0.022551
118	0.006403	0.050719
300	0.050898	0.118519
678	0.200661	0.154513
2383	7.972123	0.826617
4766	31.41175	3.061353

Table 2.4: Runtime of the Newton-Raphson method presented in [30]

In 2019 Lukas et al. presented a comparative analysis of three different LU decomposition methods applied to power system simulations [23]. These three

methods were KLU, NICSLU, and GLU2. The size of the matrices explored varied greatly even if the smallest matrix was quite large. The experiments were performed on 8 different matrices with the smallest having 26432 rows and columns, and 92718 non-zero elements. The largest matrix had 220828 rows and columns, and 693442 non-zero elements. The study showed that KLU performed best overall when it came to both preprocessing and factorization time. This was due to the preprocessing step were KLU outperformed both NICSLU and GLU2. Experiments were made to find the best preordering method for KLU and NICSLU, for the KLU algorithm AMD seemed to be the most efficient reordering method and for the NICSLU algorithm AMD ordering and COLAMD ordering seemed to perform similarly. When it came to the re-factorization part, meaning subsequent factorizations of the same matrix, NICSLU outperformed the other two approaches. GLU2 was outperformed in all different scenarios.

In the paper *GPU-Based Fast Decoupled Power Flow With Preconditioned Iterative Solver and Inexact Newton Method* by Xue Li et al. [31] a comparative study was performed to find if the Fast Decoupled method with an iterative linear solver performed better on the GPU compared to on the CPU. The reason why they chose to use an iterative linear solver to solve the linear equations instead of a direct linear solver was that iterative linear solvers are more prone to parallelism. Both the CPU and GPU versions in this paper were developed using Matlab. The GPU version used cuBLAS and cuSPARSE to execute code on the GPU. The iterative linear solver used was the conjugate gradient. The results showed that a speedup factor of 2.86 can be achieved by moving the calculations from the CPU to the GPU. This speedup was achieved when running a test case with 11624 buses.

Chapter 3

Method

This chapter covers the method used to address the research question. A total of six different versions of solving the power flow problem were developed and evaluated. Two of these versions were executed exclusively on the CPU, two were executed on the GPU and the remaining two were hybrids, thus a part of these programs ran on the CPU and GPU respectively.

Section 3.1 gives a brief presentation of the hardware used. Section 3.2 describes the input data used. Section 3.3 explains the most important parts of the versions developed on C++. Section 3.4 explains the key parts of the versions developed in CUDA. Section 3.4.1 explains the key parts of the hybrid versions.

The general outline for all the versions that were implemented can be seen below.

1. Construction of the Y-matrix
2. Calculation of the power flow equations
3. Construction of the Jacobian matrix
4. Preprocessing of the Jacobian matrix
5. Application of a linear solver
6. Update of voltage angle and voltage magnitude

Some of the parts differed in the different implementations. The process of the versions developed in C++, CUDA, and the hybrid versions are explained more thoroughly in sections 3.3, 3.4 and 3.4.1, respectively.

3.1 Software and Hardware Platforms

Two different hardware platforms were used to conduct the experiments. The reason why two platforms were chosen was to better benchmark the code running on the GPU.

The first hardware platform that was used consisted of an HP ZBook 15 G6 computer with CentOS 7 as the operating system. The compiler used for the C++ code was GCC 4.8.5 and the CUDA version was 10.2. The computer had an i7-9850H @2.6GHz CPU, 16GB DDR4 RAM, and an Nvidia Quadro T2000 GPU with 4GB of GDDR5. Further information about the GPU can be found below in Table 3.1.

Architecture	Turing
Compute Capability	7.5
Number of Cores	1024
Amount of GDDR5 Memory	4GB
Number of SMs	16
L1 Cache	64KB per SM
L2 Cache	1024KB

Table 3.1: Summary of Nvidia Quadro T2000

The other hardware that was used in this thesis consisted of an Intel Xeon Gold 6132 CPU and an Nvidia V100 with 32GB of HBM2. The version of the GCC compiler was 8.3.0 and the CUDA version was 10.1.243. The code was executed on Kebnekaise system at the High Performance Computing Center North (HPC2N).

Architecture	Volta
Compute Capability	7.0
Number of Cores	5120
Amount of HBM2 Memory	32GB
Number of SMs	80
L1 Cache	96KB per SM
L2 Cache	6144KB

Table 3.2: Summary of Nvidia V100

3.2 Input Data

All the input data used was taken from MATPOWER's github [32]. MATPOWER is a package of Matlab files for solving power flow problems. The size of the input data, meaning the number of buses, used in this thesis went from a trivial case of 5 buses up to 9241 buses.

The input data was divided into four matrices. However, only the first three of these were of interest. The first two matrices contained data about each bus in the network, where each row corresponded to one bus. The bus data, for each bus, that was taken into account when solving the power flow problem can be seen in the bullet list below. Note that the voltage magnitude and the voltage angle given in the input data were not the final solution, but used as an initial guess.

- Bus number
- Bus type
- Active power
- Reactive power
- Voltage magnitude
- Voltage angle

The third matrix of the input data contained data for each branch in the network. Each row of this matrix corresponded to one branch and the total number of branches was denoted as N_b. The branch data contained all data needed to construct the admittance matrix Y. The number of non-zero (nnz) elements in the Y-matrix is proportional to $2N_b + N$. The Y-matrix was constructed in exactly the same manner for all the different versions.

3.3 The CPU Versions

Two different CPU versions were developed and evaluated. The difference between these versions was that the first one used sparse LU factorization to solve the linear equations while the second version used sparse QR factorization.

The general outline of the CPU versions can be seen in Figure 3.1.

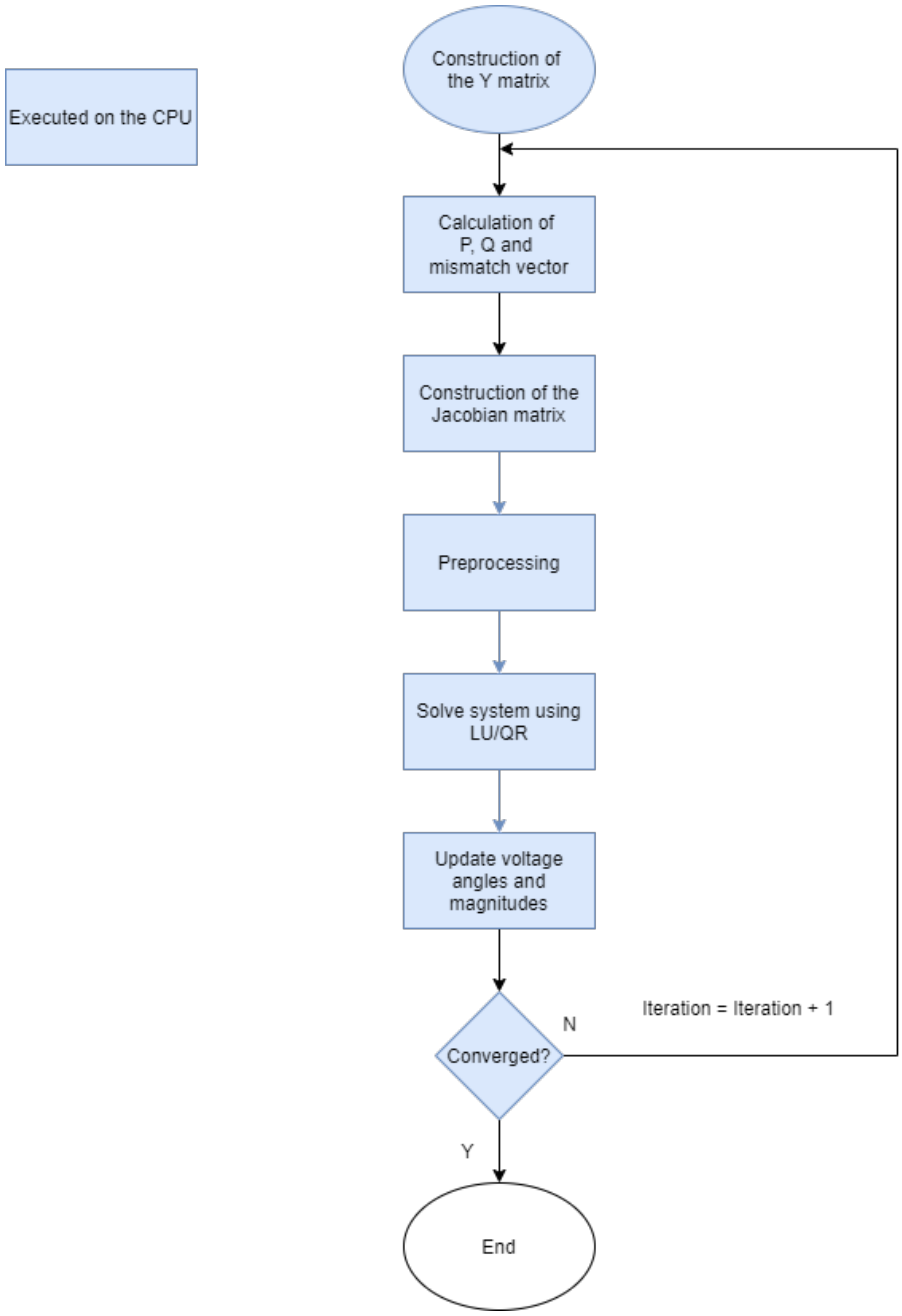

Figure 3.1: Workflow of the Newton-Raphson method with sparse QR/LU factorization on the CPU

Power Flow Equations

As can be seen in Listing 3.1 the power flow equations were solved by iterating over all the non-zero values of the Y-matrix. The reason for this was because if both the real and imaginary part of the Y-matrix were equal to zero, that would result in adding 0 to both P and Q for that iteration. Once the power flow equations were solved, the mismatch vector M was calculated. The M vector contained the difference between the specified values in the input data and the calculated values for each bus, as seen in Listing 3.1. For the PQ buses, this difference was calculated for both the active and reactive powers. However, for the PV buses, only the difference for the active power was calculated since the reactive power was unknown.

```
for (int i = 0; i < yMatrix.outerSize(); i++){
    P[i] = 0.0;
    Q[i] = 0.0;
    for (Eigen::SparseMatrix<complex<double>,
    Eigen::RowMajor>::InnerIterator it(yMatrix,i); it; ++it)
    {
        int j = it.col();
        complex<double> temp = it.value();
        P[i] += voltage[i] * voltage[j] *
        (temp.real() * cos(angle[i] - angle[j])
        + temp.imag() * sin(angle[i] - angle[j]));

        Q[i] += voltage[i] * voltage[j] *
        (temp.real() * sin(angle[i] - angle[j])
        - temp.imag() * cos(angle[i] - angle[j]));
    }
}

for (int i = 1; i < nNodes; i++) {
    dP[i-1] = pTotal[i] - P[i];
    M(i - 1) = dP[i - 1];
}

for (int i = 0; i < pq.size(); i++) {
    dQ[i] = qTotal[i+1] - Q[i+1];
    M(nNodes - 1 + i) = dQ[i];
}
```

Listing 3.1: Calculation of the power flow equations with C++

Assembly of the Jacobian Matrix

The Jacobian matrix was constructed by calculating the four sub-matrices J_1-J_4. When calculating their diagonal elements, P and Q were used to minimize the complexity. This was done by finding a similarity between Equations (2.7) and (2.11) which allowed for less computation. The affected equation can be seen below.

$$J_1 \text{ (i,i)} = \frac{\partial P_i}{\partial \theta_i} = V_i \sum_{\substack{j=1 \\ j \neq i}}^{n} V_j(-G_{ij}\sin(\theta_{ij}) + B_{ij}\cos(\theta_{ij}))$$

$$Q_i = V_i \sum_{j=1}^{n} V_j(G_{ij}\sin(\theta_{ij}) - B_{ij}\cos(\theta_{ij})$$

The simplification is done by negating Q_i and subtracting $V_i^2 B_{ij}$ from equation (2.7) to get $\frac{\partial P_i}{\partial \theta_i}$. This resulted in Equation (3.1).

$$J_1 \text{ (i,i)} = \frac{\partial P_i}{\partial \theta_i} = -Q_i - V_i^2 B_{ii} \tag{3.1}$$

Similarly the diagonal elements of the submatrices J_2-J_4 were calculated as seen in Equations (3.2)-(3.4).

$$J_2 \text{ (i,i)} = \frac{\partial P_i}{\partial V_i} = \frac{P_i}{V_i} + V_i G_{ii} \tag{3.2}$$

$$J_3 \text{ (i,i)} = \frac{\partial Q_i}{\partial \theta_i} = P_i - V_i^2 G_{ii} \tag{3.3}$$

$$J_4 \text{ (i,i)} = \frac{\partial Q_i}{\partial Vi} = \frac{Q_i}{V_i} - V_i B_{ii} \tag{3.4}$$

Just like the Y-matrix, the Jacobian matrix was also stored in CSR format.

The source code for the construction J_1 for the C++ version can be seen below in Listing 3.2.

```
for (int k = 1; k < yMatrix.outerSize(); k++) {
    for (Eigen::SparseMatrix<complex<double>,
    Eigen::RowMajor>::InnerIterator it(yMatrix, k); it; ++it)
    {
        double value = 0.0;
        int i = it.row();
        int j = it.col();
```

```
        complex<double> temp = it.value();
        if (i == j) {
            value = -Q[i];
            value -= voltage[i] * voltage[i] * temp.imag();

        } else if (j > 0) {
            value = voltage[i] * voltage[j] *
            (temp.real() * sin(angle[i] - angle[j]) -
            temp.imag() * cos(angle[i] - angle[j]));
        }
        if (value != 0) {
            triplets.emplace_back(i-1, j-1, value);
            count++;
        }
    }
}
```

Listing 3.2: Construction of J_1 with C++

Linear Solver

The linear solvers used for the C++ version were sparse LU and sparse QR factorization. Both types of factorization were implemented using the Eigen library [33]. This library provides three different reordering schemes to minimize the complexity of the factorization schemes. The reordering schemes were column approximate minimum degree ordering, approximate minimum degree ordering, and natural ordering. These three were all tested to evaluate which one yielded the best performance.

Update of Voltage Values

Finally, the voltage angle was updated for all the buses except for the slack bus as seen in Listing 3.3. Additionally, the voltage magnitude was updated for all the PQ buses. Note that X in listing 3.3 contained the solution to the linear equations.

```
for (int i = 1; i < nNodes; i++) {
    angle[i] += X[i - 1];
    if (abs(M[i-1]) > tol)
        tol = abs(M[i-1]);
}

for (int i = 0; i < pq.size(); i++) {
    voltage[i+1] += X(nNodes - 1 + i);
    if (abs(M[nNodes - 1 + i]) > tol)
```

```
        tol = abs(M[nNodes - 1 + i]);
}
```

Listing 3.3: Update voltage angle and voltage magnitude with C++

3.4 The GPU Versions

Two different versions of solving the power flow problem on the GPU were developed and evaluated. The difference between the two versions was that the first one, let us call it sparse QR, used sparse QR factorization to solve the linear equations. The second GPU implementation, let us call it dense LU, used dense LU factorization to solve the linear equations. The versions that used dense LU factorization was mostly evaluated to better compare the different versions since one of the CPU versions and the hybrid versions use sparse LU factorization.

The general outline of the GPU versions is shown in Figure 4.5.

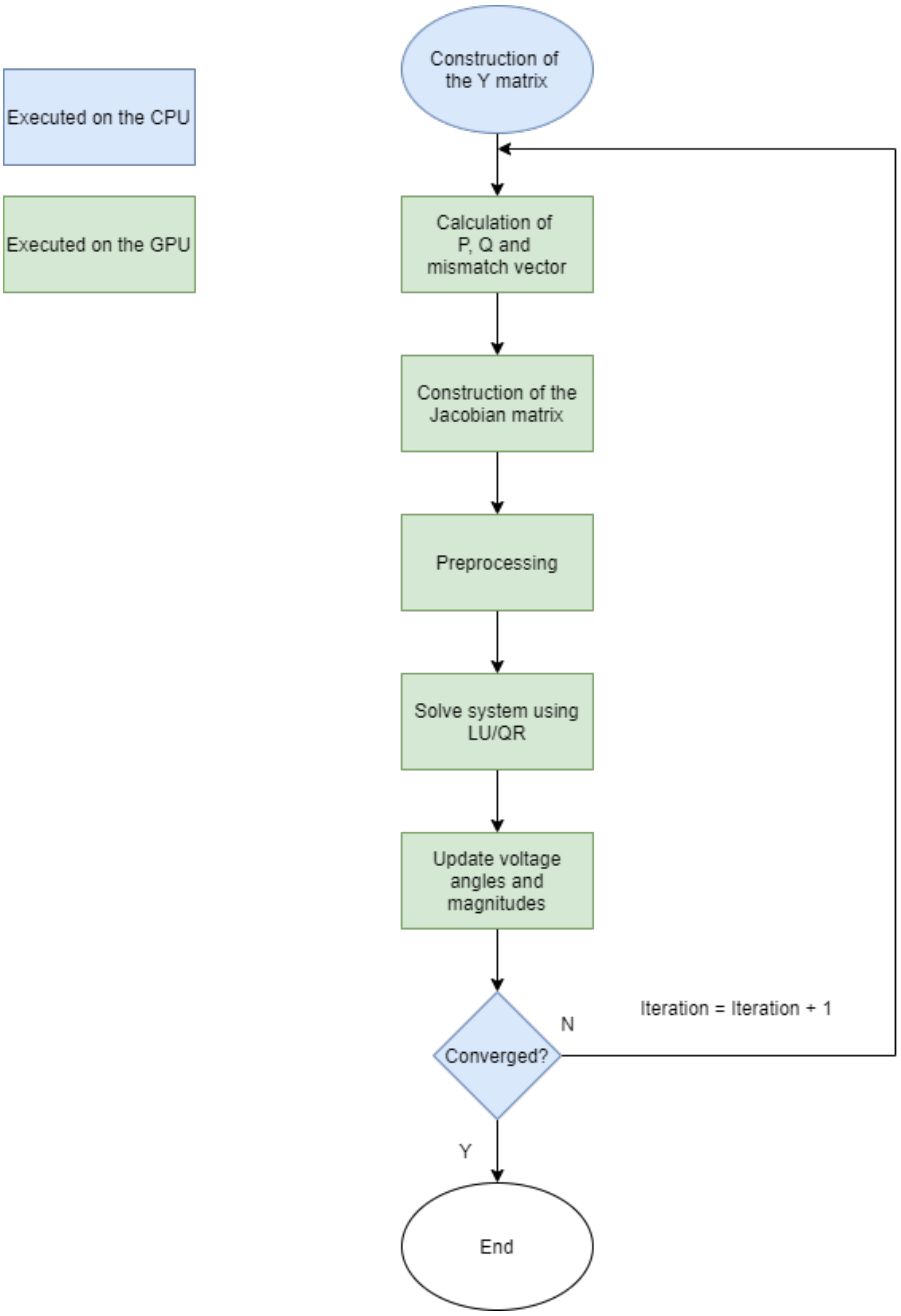

Figure 3.2: Workflow of the Newton-Raphson method executed on GPU

Power Flow Equations

For the calculation of the power flow equations a kernel was used. This kernel can be seen in Listing 3.5. In contrast to how the C++ implementation was executed with being iterated over the nnz elements of the Y-matrix, the CUDA version performed this through launching a kernel running as many threads as there were nnz elements in the Y-matrix which can be seen in Listing 3.4. As a precaution to avoid data races and similar problems, atomic addition was used when the power equations were calculated, as seen in Listing 3.5.

```
dim3 numBlocksY((nnzY+threads-1)/threads);
powerEqn<<<numBlocksY, numThreads>>>(d_pCalc, d_qCalc, d_yRow,
d_yCol, d_yMatrix, d_angle, d_voltage, nnzY);
```

Listing 3.4: Call to the kernel to calculated the power flow equations

```
__global__ void powerEqn(double *P, double *Q, int* d_yRow,
                int* d_yCol, cuDoubleComplex* d_yMatrix,
                double *angle, double *voltage, int nnzY) {
        int ix = blockIdx.x*blockDim.x+threadIdx.x;
        if (ix < nnzY) {
                int i = d_yRow[ix];
                int j = d_yCol[ix];
                atomicAdd(&P[i], (voltage[i] * voltage[j] *
                (cuCreal(d_yMatrix[ix]) * cos(angle[i] -
                angle[j]) + cuCimag(d_yMatrix[ix]) *
                sin(angle[i] - angle[j]))));

                atomicAdd(&Q[i], (voltage[i] * voltage[j] *
                (cuCreal(d_yMatrix[ix]) * sin(angle[i] -
                angle[j]) - cuCimag(d_yMatrix[ix]) *
                cos(angle[i] - angle[j]))));
        }
}
```

Listing 3.5: The kernel that calculated the power flow equations

To calculate the mismatch vector M, two kernels were launched as seen in listing 3.6. The first kernel calculated ΔP and was therefore launched with $N-1$ threads. The second kernel calculated ΔQ and was launched with as many threads as there were PQ buses. The two kernels can be seen in Listing 3.7.

```
dim3 numBlocksNm1((nNodes-1+threads-1)/threads);
missmatchP<<<numBlocksNm1, numThreads, 0, streams[0]>>>
(d_m, d_pTotal, d_pCalc, nNodes-1);

dim3 numBlocksPQ((pq.size()+threads-1)/threads);
```

```
missmatchQ<<<numBlocksPQ, numThreads, 0, streams[1]>>>
(d_m, d_qTotal, d_qCalc, nNodes, d_pq, pq.size());
```

Listing 3.6: Call to the kernel that calculated the missmatch vector

```
__global__ void missmatchP(double *m, double *pTotal,
                double *pCalc, int nNodes) {
        int ix = blockIdx.x*blockDim.x+threadIdx.x;
        if (ix < nNodes){
                m[ix] = pTotal[ix + 1] - pCalc[ix + 1];
        }
}

__global__ void missmatchQ(double *m, double *qTotal,
             double *qCalc, int nNodes, int *pq, int numPQ) {
        int ix = blockIdx.x*blockDim.x+threadIdx.x;
        if (ix < numPQ) {
        m[nNodes + ix - 1] = qTotal[ix+1] - qCalc[ix+1];
        }
}
```

Listing 3.7: The kernels that calculated the missmatch vector

Assembly of the Jacobian Matrix

To construct the four sub-matrices of the Jacobian matrix, the Y-matrix was utilized in four separate ways. These can be seen below denoted as Y_1, Y_2, Y_3 and Y_4.

- Y_1: Contained all elements of the Y-matrix except for the first row and first column as these corresponded to the slack bus and were not needed for the construction of the Jacobian matrix
- Y_2: The first row and column of the Y-matrix were removed. In addition, the columns corresponding to the PV buses were removed since their voltage magnitudes were known
- Y_3: The first row and column of the Y-matrix were removed. Furthermore, the rows corresponding to the PV buses were also removed since their reactive powers were unknown
- Y_4: Y_4 was the intersection of Y_2 and Y_3, with the same logic applied.

The elements in the Jacobian matrix were calculated in the same manner as in the C++ version, meaning that P and Q from the previous step were used.

The main difference was that four kernels were used to calculate the Jacobian matrix, one for each of the four sub-matrices. Each kernel calculated one of the four sub-matrices $J_1 - J_4$ and was launched with as many threads as the number of non-zero elements in its corresponding Y sub-matrices, $Y_1 - Y_4$. The call to the kernel that constructed J_1 can be seen in Listing 3.8. The actual kernel that constructed J_1 can be seen in Listing 3.9.

```
dim3 numBlocksJ1((nnzYJ+threads-1)/threads);
getJ1<<<numBlocksJ1, numThreads, 0, streams[2]>>>(d_jacRow,
d_jacCol, d_jac, d_yJRow, d_yJCol, d_yJ, d_voltage,
d_angle, d_qCalc, d_nnzJ2, nnzYJ);
```

Listing 3.8: Call to the kernel that constructed J_1

```
__global__ void getJ1(int *jacRow, int *jacCol, double *jac,
                int *yRow, int *yCol, cuDoubleComplex* yMatrix,
                double *voltage, double *angle,
                double *q, int *nnzJ2, int nnzJ1) {
        int ix = blockIdx.x*blockDim.x+threadIdx.x;
        if (ix < nnzJ1) {
                int offset = nnzJ2[yRow[ix] - 1];
                if (yRow[ix] != 0 && yCol[ix] != 0) {
                        jacRow[ix + offset] = yRow[ix] - 1;
                        jacCol[ix + offset] = yCol[ix] - 1;

                        if (yRow[ix] == yCol[ix]) {
                                jac[ix + offset] = -q[yRow[ix]]
                                - voltage[yRow[ix]] * voltage[yRow[ix]]
                                * cuCimag(yMatrix[ix]);
                        } else {
                                jac[ix + offset] = voltage[yRow[ix]] *
                                voltage[yCol[ix]]* (cuCreal(yMatrix[ix])
                                * sin(angle[yRow[ix]] - angle[yCol[ix]])
                                - cuCimag(yMatrix[ix])
                                * cos(angle[yRow[ix]] - angle[yCol[ix]]));
                        }
                }
        }
}
```

Listing 3.9: The kernel that constructed J_1

Linear Solver

The library used to solve the linear equations was cuSOLVER which provides several dense and sparse linear solvers. The linear solver used in this thesis

was the sparse QR factorization solver. As opposed to the C++ implementation, the sparse LU factorization solver was not used for the CUDA version since the sparse LU factorization provided by cuSOLVER does not run on the GPU.

Similar to the Eigen library, cuSOLVER provides three different reordering schemes. These are symmetric reverse Cuthill-McKee ordering, Symmetric approximate minimum degree ordering, and metis ordering. These reordering schemes were tested extensively to find the one that gave the best performance.

Update of Voltage Values

Once the linear equations were solved, the voltage angle was updated for all the buses except for the slack bus. The voltage magnitude was updated for all the PQ buses. This was done with one kernel running $N-1$ threads. The kernel used for updating voltage angles and magnitudes can be seen in Listing 3.10

```
__global__ void updateAngleVoltage(double *angle, double
                *voltage, double *x, int nNodes, int pq) {
        int ix = blockIdx.x*blockDim.x+threadIdx.x;
        if(ix<nNodes-1) {
                angle[ix+1] += x[ix];
                if(ix<pq){
                        voltage[ix+1] += x[ix+nNodes-1];
                }
        }

}
```

Listing 3.10: The kernel that updated the voltage angles and magnitudes

Finally, the mismatch vector was copied back to the host (CPU). On the host side, an iteration over all the elements of the mismatch vector was performed to check if all the elements in the vector were less than the specified tolerance ε. If that was the case, it meant that a solution was found and the iteration stopped. If that was not the case, then at least one more iteration was needed. Note that this was the only time during the iteration that memory was copied between the host and the device.

3.4.1 The Hybrid Versions

Two different hybrid versions were developed and evaluated. They were very similar with the only difference being how the preprocessing of the LU factorization was approached. In the first hybrid version, let us call it hybrid 1,

the preprocessing step was based on John R. Gilbert's algorithm [21] and was executed on the CPU. In the second hybrid version, let us call it hybrid 2, the preprocessing step was based on KLU [20] and was executed on the CPU. Both hybrid 1 and hybrid 2 used GLU2 [24] to solve the linear system on the GPU.

The two hybrid versions were both very similar to the GPU versions. They calculated the power flow equations and the Jacobian matrix in exactly the same manner. The two versions also updated the voltage angles and magnitudes in the same way. The difference between the GPU versions and the hybrid versions was how the linear equations were solved. Both the hybrid versions exploited the fact that the sparsity pattern of the Jacobian matrix, the L matrix and the U matrix were exactly the same at each iteration and therefore the preprocessing was only needed in the first iteration.

Workflow diagrams for both hybrid versions can be seen below in Figures 3.3 and 3.4

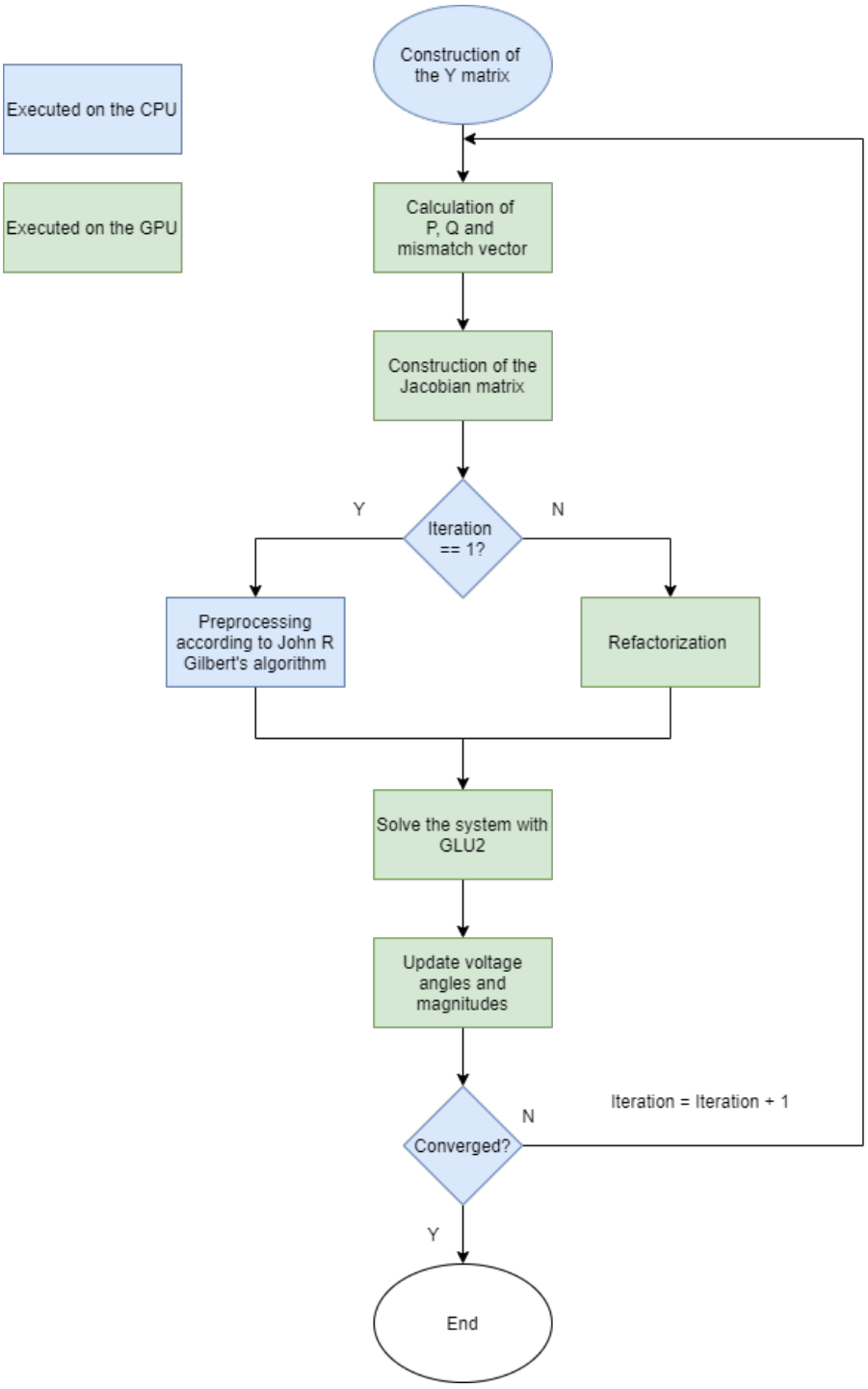

Figure 3.3: The general workflow of hybrid 1

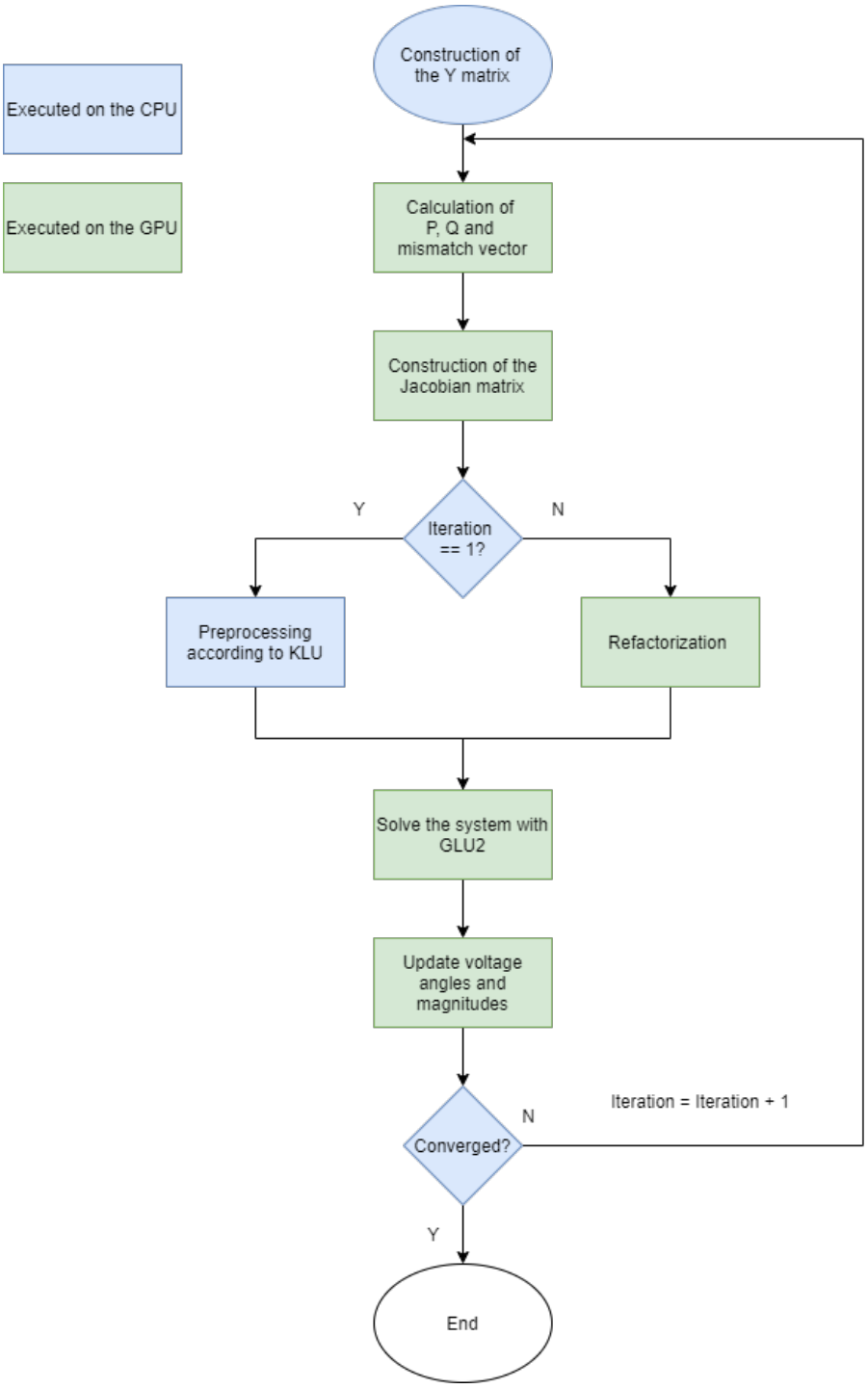

Figure 3.4: The general workflow of hybrid 2

Chapter 4

Results

In this chapter, the results are presented. For all the experiments each code section was executed and timed 10 times and the average was taken. The accuracy of the timer used to measure the execution time was around 0.5 microseconds. If nothing is stated, the experiments were performed on the local hardware platform. In addition, before each experiment a warmup was performed, meaning that the code segment was executed five times before the actual timing started. Furthermore, the GPU kernels were launched with 128 threads for each block. This decision was taken in regard to Figure 4.1.

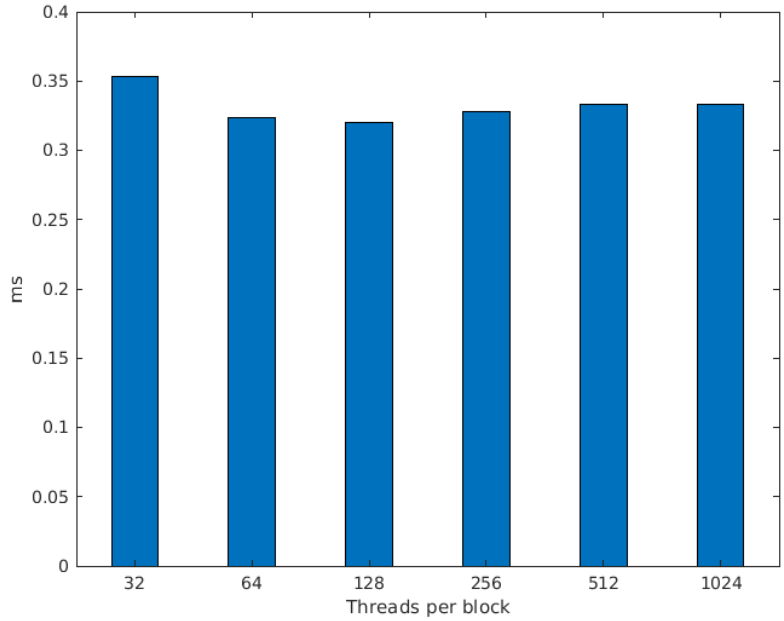

Figure 4.1: Execution time of constructing the Jacobian matrix for different number of threads per block executing the case with 9241 buses

Each of the runs in Figure 4.1 is defined as the time taken to calculate the Jacobian matrix for the largest example available, 9241 buses. The Jacobian matrix was timed because it was the most demanding kernel when not considering the linear solvers. Timing the entire program would lead to excessive overhead as the factorization methods were the most time consuming part of the calculation.

4.1 Performance of Constructing the Jacobian Matrix

In this section, the execution time of the construction of the Jacobian matrix is presented for three different implementations. In Table 4.1 the number of non-zero elements in the Jacobian matrix is presented. The execution time of the Jacobian matrix can be seen in Figure 4.2 which uses logarithmic scaling.

#Buses	5	14	30	69	118	200	2000	3120	9241
#NNZ	17	146	361	808	1051	2412	25652	37040	129412

Table 4.1: Number of non-zero elements in the Jacobian matrix

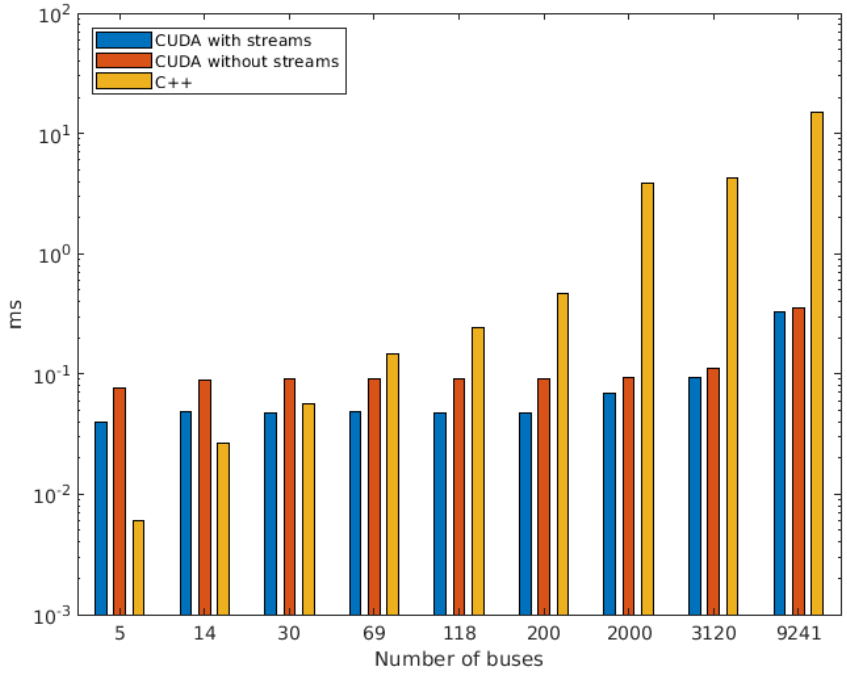

Figure 4.2: Execution time of the construction of the Jacobian matrix

For the CUDA version that uses streams, a total of four streams were used. Each stream calculated a sub-matrix of the Jacobian matrix. Table 4.2 shows the execution time of the construction of the Jacobian matrix.

#Buses	5	14	30	69	118	200	2000	3120	9241
CUDA$_{ST}$	0.04	0.05	0.05	0.05	0.05	0.05	0.07	0.09	0.33
CUDA	0.08	0.09	0.09	0.09	0.09	0.09	0.09	0.11	0.35
C++	0.01	0.03	0.06	0.15	0.24	0.46	3.85	4.28	15.12

Table 4.2: Execution time of the construction of the Jacobian matrix in milliseconds. CUDA$_{ST}$ is the CUDA version with enabling streams.

4.2 Performance of the Factorization Methods

This section presents the performance of the different factorization methods for both the C++, CUDA and hybrid versions. All the figures in this section use logarithmic scaling to better present the results.

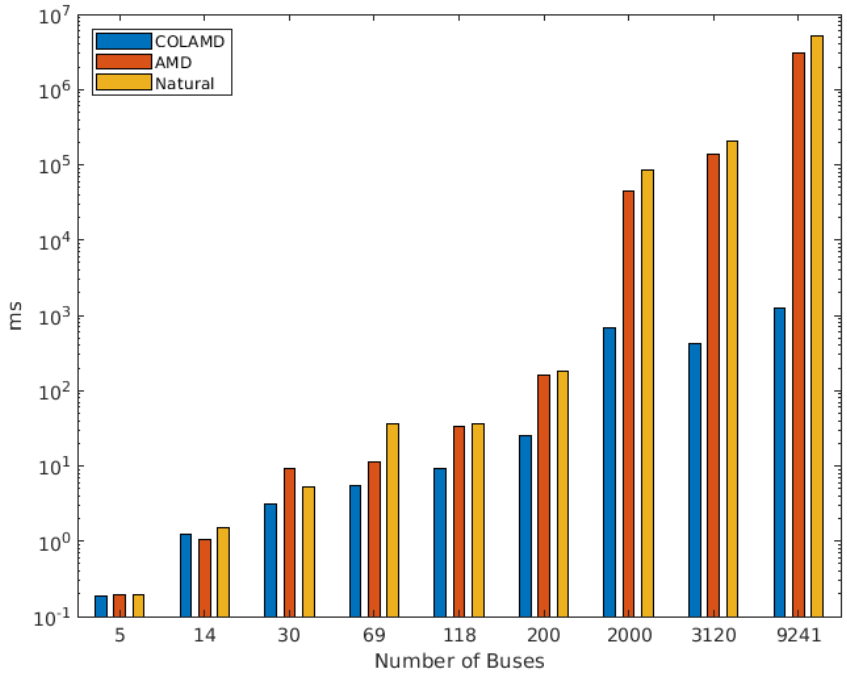

Figure 4.3: Execution time of the C++ version using LU factorization

Figure 4.3 and Table 4.3 show the runtime of the LU factorization on the CPU with the different reordering methods.

#Buses	5	14	30	69	118	200	2000	3120	9241
COLAMD	0.19	1.25	3.14	5.52	9.25	25.69	675.7	425.42	1260
AMD	0.19	1.04	9.4	11.14	33.32	159.5	44949	138403	3085144
Natural	0.19	1.52	5.21	36.51	36.58	178.11	84503	206506	5233278

Table 4.3: Execution time of the C++ version using LU factorization in milliseconds

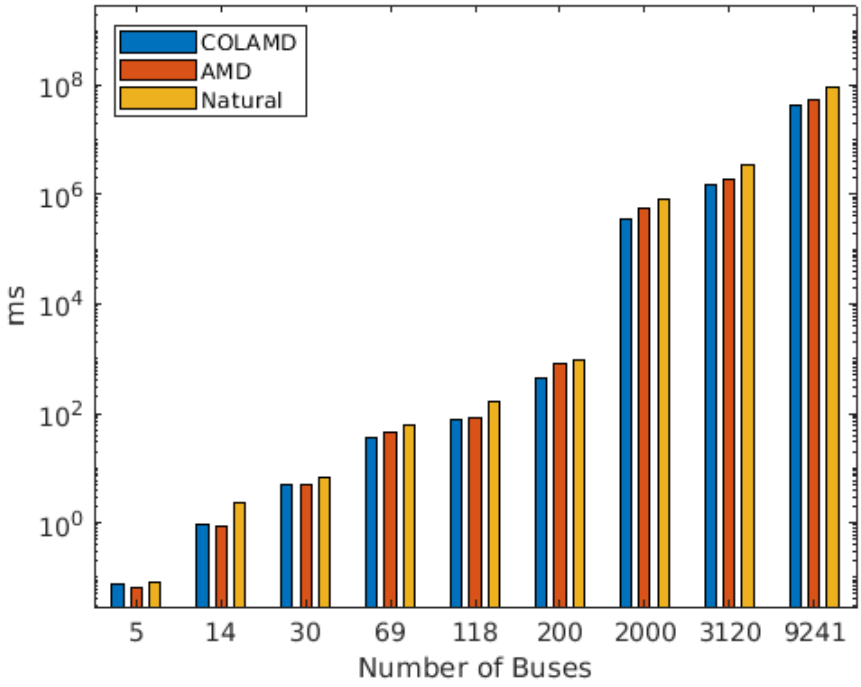

Figure 4.4: Execution time of the C++ version using QR factorization

Figure 4.4 and Table 4.4 show the runtime of the QR factorization on the CPU with the different reordering methods.

#Buses	5	14	30	69	118	200	2000	3120	9241
COLAMD	0.08	0.9	4.81	36.15	74.3	448.5	366342.56	1562543	41833046
AMD	0.07	0.9	4.94	45.85	85.64	796.12	551088.29	1891298	53926402
Natural	0.08	2.39	6.68	61.3	167.65	979.31	806113.5	3613965	92282556

Table 4.4: Execution time of the C++ version using QR factorization in milliseconds

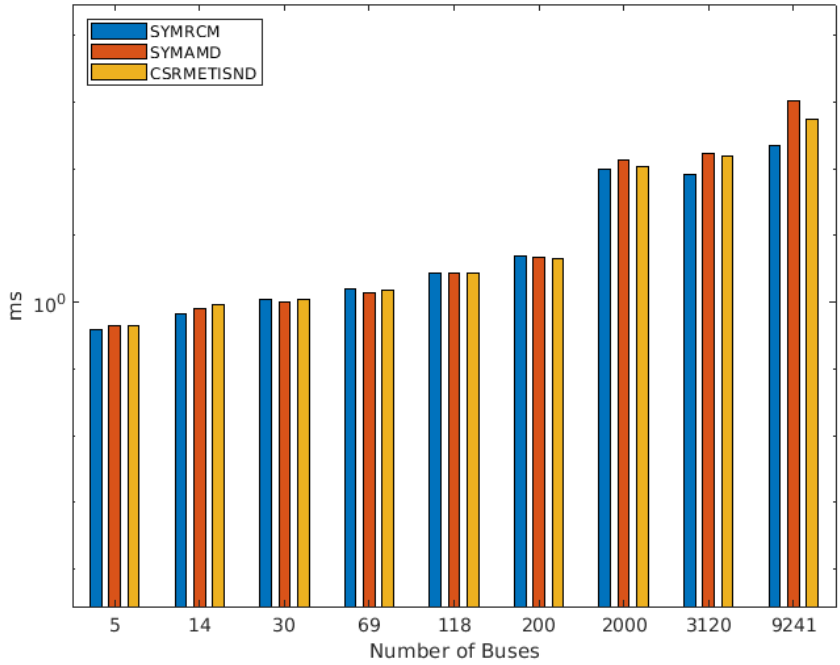

Figure 4.5: Execution time of the CUDA version using QR factorization

Figure 4.5 and Table 4.5 show the execution time of the QR factorization on the GPU with the different reordering methods.

#Buses	5	14	30	69	118	200	2000	3120	9241
SYMRCM	0.39	0.67	1.12	1.62	2.71	4.99	98.17	82.95	225.88
SYMAMD	0.45	0.8	0.99	1.38	2.7	4.62	138.13	172.06	1033.94
CCSRMETISND	0.44	0.92	1.12	1.53	2.7	4.51	108.21	157.29	556.17

Table 4.5: Execution time of the CUDA version using QR factorization in milliseconds

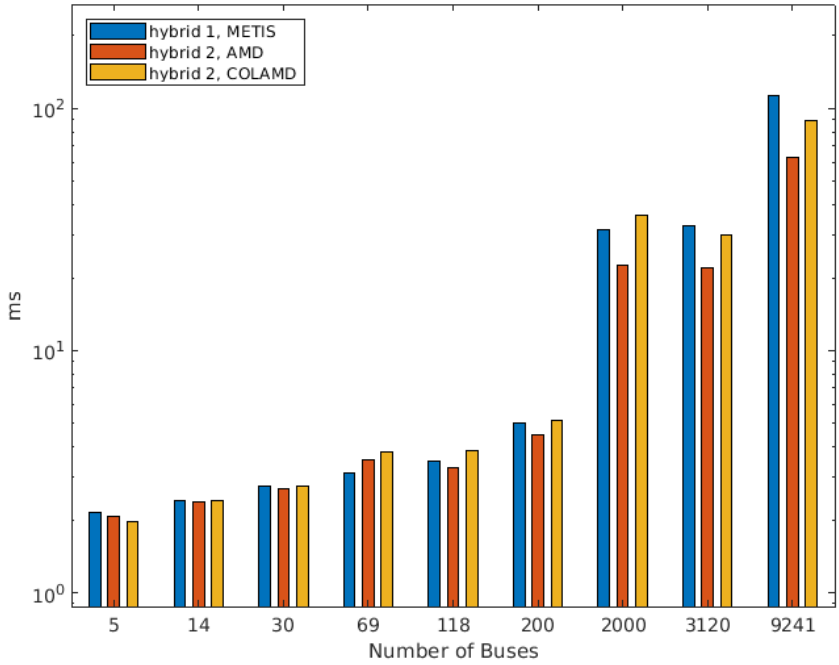

Figure 4.6: Execution time of the first iteration for the hybrid versions

Figure 4.6 and Table 4.6 show the runtime of the first iteration of the Newton-Raphson method, meaning that the preprocessing was done on the CPU and the linear system was solved with GLU2 on the GPU.

#Buses	5	14	30	69	118	200	2000	3120	9241
hybrid 1, METIS	2.16	2.41	2.75	3.11	3.48	5.05	31.54	32.68	113.71
hybrid 2, AMD	2.06	2.36	2.69	3.54	3.30	4.49	22.46	22	62.99
hybrid 2, COLAMD	1.97	2.39	2.77	3.81	3.85	5.18	36.26	29.94	88.9780

Table 4.6: Execution time of the first iteration for the hybrid versions in milliseconds

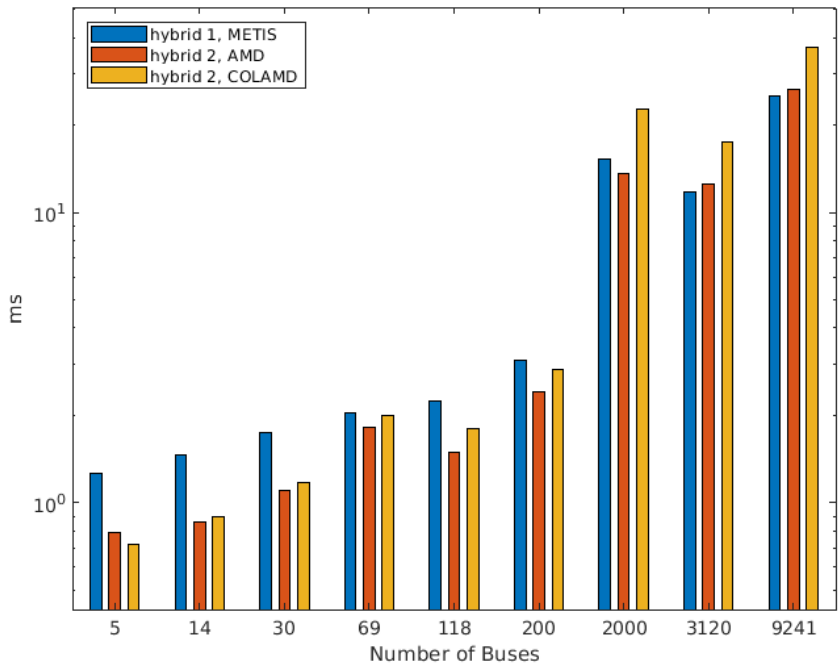

Figure 4.7: Execution time of the refactorization and the linear solver for the hybrid versions

Figure 4.7 and Table 4.7 show the runtime of the refactorization for the hybrid versions on the GPU and the linear solver on the GPU. As explained in Section 3.4.1, this is done each iteration except for the first.

#Buses	5	14	30	69	118	200	2000	3120	9241
hybrid 1, METIS	1.27	1.46	1.74	2.04	2.24	3.1	15.33	11.77	25.16
hybrid 2, AMD	0.79	0.86	1.11	1.82	1.5	2.41	13.59	12.54	26.44
hybrid 2, COLAMD	0.72	0.9	1.17	1.99	1.8	2.86	22.64	17.5	37.05

Table 4.7: Execution time of the refactorization and the linear solver for the hybrid versions in milliseconds

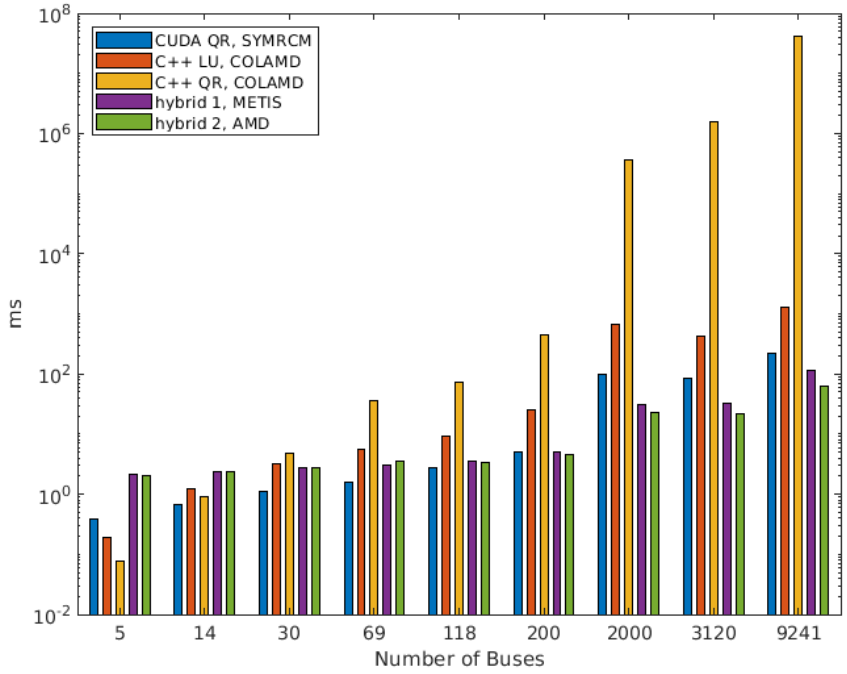

Figure 4.8: Summary of the most efficient reordering methods

Figure 4.8 shows the runtime of the most efficient reordering method for all versions (most efficient runtimes in Figures 4.3-4.6).

4.3 Total Execution Time

This section presents the execution time of the whole execution except for the construction of the Y-matrix as it was constructed in the same manner for all the versions and therefore it will not differ between the versions. The tolerance ε was set to 10^{-5} for all the versions. The number of iterations needed to find a solution is presented in Table 4.8

#Buses	5	14	30	69	118	200	2000	3120	9241
#nrow(Jac)	5	22	53	136	181	350	3514	5890	17036
#nnz(Jac)	17	146	361	808	1051	2412	25652	37040	129412
#Iterations	4	4	4	3	4	3	5	6	6

Table 4.8: Number of iterations needed to find a solution and the number of non-zero elements in the Jacobian

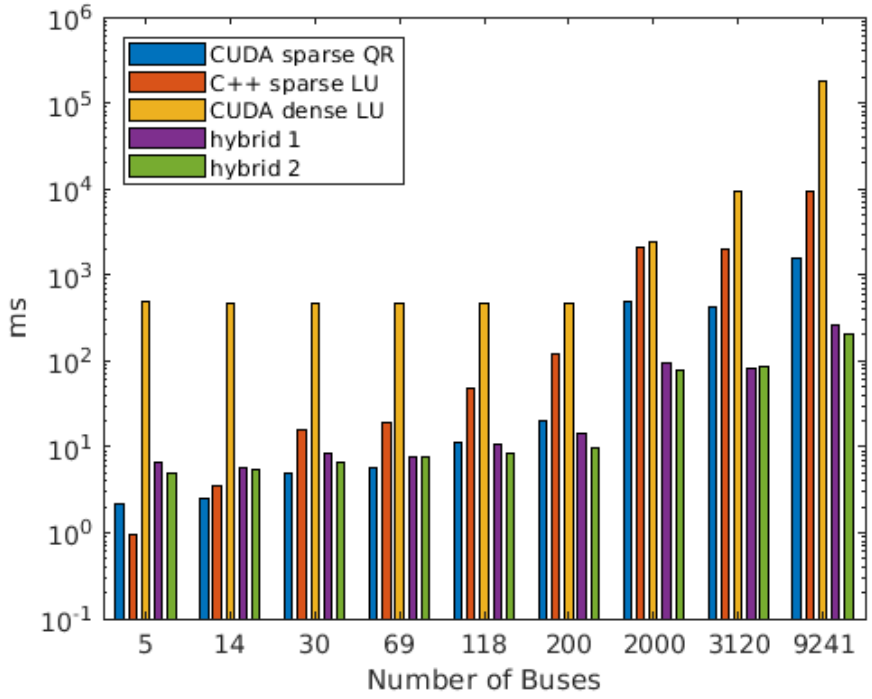

Figure 4.9: Total execution time of the power flow problem on the local hardware platform

Figure 4.9 uses logarithmic scaling and shows the performance of five different versions. Table 4.9 shows the execution time and the standard deviation for the different versions in milliseconds. All the versions used their fastest factorization configuration. This means that the sparse CUDA version used QR factorization with SYMRCM reordering, the dense CUDA version used LU factorization without reordering, the C++ version used LU factorization with AMD reordering, hybrid 1 used METIS reordering and hybrid 2 used AMD reordering. Note that the total execution time was defined as the total time to find a solution, meaning several iterations.

#Buses	5	14	30	69	118	200	2000	3120	9241
CUDA QR	2.12	2.44	4.98	5.57	11.39	20.47	485.49	426.63	1587
CUDA QR, σ	0.29	0.14	0.58	0.43	1.53	1.15	6.51	5.03	3.21
C++ LU SYMRCM	0.96	3.5	16	19	48.31	119.13	2112	2029	9478
C++ LU SYMRCM, σ	0.12	0.5	2.65	1	5.86	16.01	7	3.21	29.77
CUDA Dense LU	496	465	468	468	470	473	2397	9564	182940
CUDA Dense LU, σ	37.11	4.42	5.57	4.32	7.19	6.43	38.63	45.93	186.24
hybrid 1	6.45	5.79	8.43	7.57	10.68	13.98	93.24	79.81	261.77
hybrid 1, σ	0.047	0.041	0.042	0.043	0.021	0.059	1.29	0.35	3.17
hybrid 2	4.94	5.49	6.52	7.65	8.3	9.83	78.79	85.65	206.93
hybrid 2, σ	0.022	0.0086	0.016	0.016	0.06	0.04	0.32	0.23	6.05

Table 4.9: Total execution time and standard deviation of the power flow problem on the local hardware in milliseconds

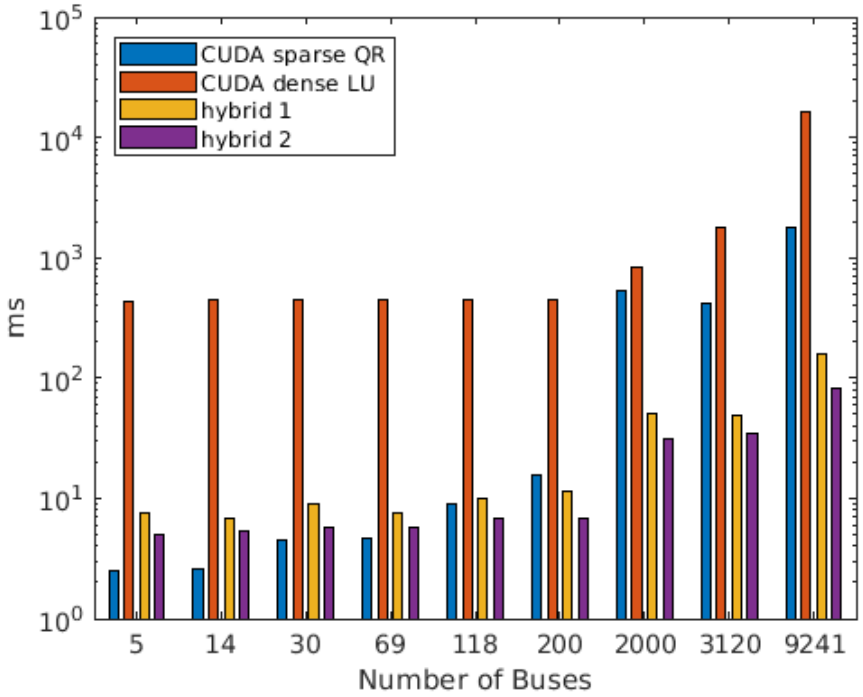

Figure 4.10: Total execution time of the power flow problem on the V100 GPU

Figure 4.10 uses logarithmic scaling. Figure 4.10 and table 4.10 show the performance of four different versions. Table 4.10 also shows the standard

deviation for the different versions. All the versions use the same configuration as in Figure 4.9. The C++ version was disregarded since it does not run on the GPU. Similar to Figure 4.9, Figure 4.10 shows the total execution time, meaning the time until a solution was found.

#Buses	5	14	30	69	118	200	2000	3120	9241
CUDA QR SYMRCM	2.53	2.62	4.47	4.61	8.93	15.78	539.88	418.93	1805.64
CUDA QR SYMRCM, σ	0.032	0.066	0.061	0.081	0.022	0.044	3.98	5.24	3.88
CUDA Dense LU	438	441	440	442	442	450	846	1759	16375
CUDA Dense LU, σ	0.27	0.7	1.61	3.55	0.99	2.38	2.56	12.84	99.36
hybrid 1	7.65	6.74	8.9	7.64	9.92	11.64	50.93	49.52	157.07
hybrid 1, σ	0.3	0.17	0.063	0.33	0.15	0.24	0.91	1.4	2.76
hybrid 2	4.99	5.37	5.68	5.71	6.78	6.76	30.85	34.02	83.50
hybrid 2, σ	0.22	0.023	0.037	0.023	0.044	0.056	0.66	0.19	1.1

Table 4.10: Total execution time and standard deviation of the power flow problem on the V100 GPU in milliseconds

As can be seen in Figure 4.10, hybrid 2 which used pinned memory, AMD reordering and streams performed the best. Figure 4.11 shows the speedup of hybrid 2 compared to the best performing CPU version for each of the nine cases on the local hardware. The C++ version used sparse LU factorization with COLAMD reordering. Figure 4.12 shows the same comparison but on the V100 GPU.

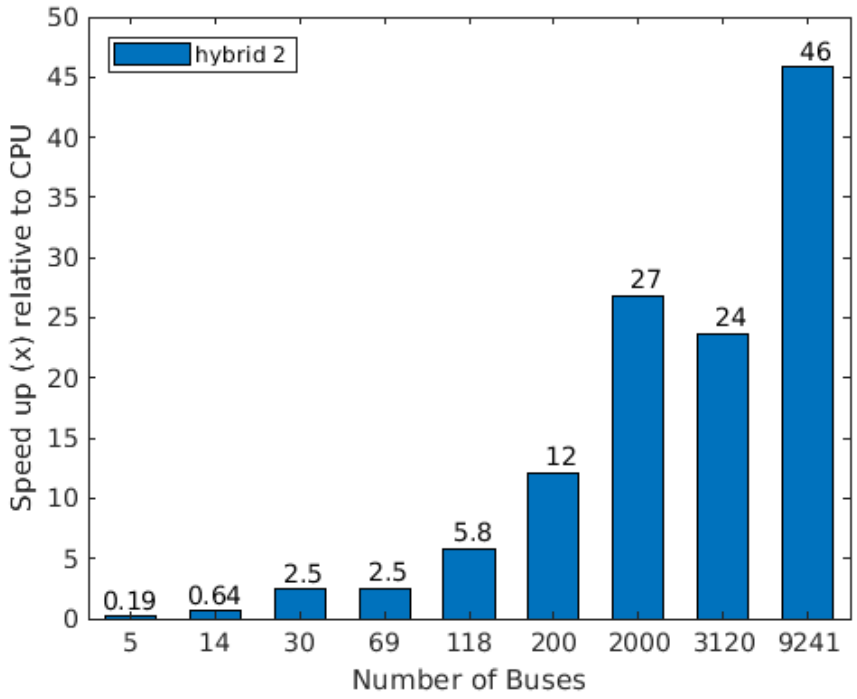

Figure 4.11: The speedup of hybrid 2 in comparison with the baseline CPU version using LU factorization on the local hardware

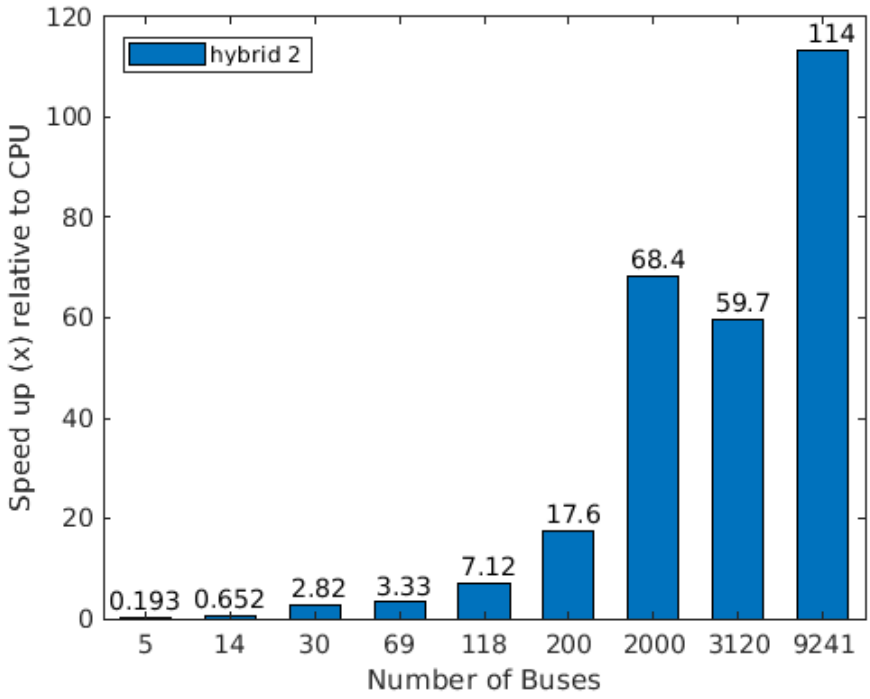

Figure 4.12: The speedup of hybrid 2 in comparison with the baseline CPU version using LU factorization on the V100 GPU

Chapter 5

Discussion

This chapter presents a discussion of the results and methodology of this study. In Section 5.1, the key findings from the results chapter are presented and analyzed. The findings are compared in Section 5.2. In section 5.3, ethics and sustainability issues related to this thesis are discussed.

5.1 Key findings

The results indicate that it is possible to get a boost in performance by performing the computations of the power flow problem on the GPU rather than on the CPU. Furthermore, it seems that a hybrid version might be the best option for solving the power flow problem. As can be seen from the results, all parts of the program perform better on the GPU compared to the CPU when it comes to large scale power flow problems as expected. The best performing hybrid version, hybrid 2, got a speedup factor of 114 compared to the best performing CPU version and a speedup factor of around 19 when compared to the best performing GPU version. Figure 4.9 shows that the GPU outperforms the CPU when it comes to networks with 14 buses or more. When comparing hybrid 2 with the CPU version that use LU factorization, it can be seen that hybrid 2 outperforms the CPU when it comes to networks with 30 buses or more. It can also be seen that hybrid 2 outperforms the GPU for cases with 118 buses and more according to Figure 4.9. The fact that both the CPU and GPU versions recalculate the preprocessing at each iteration while the hybrid versions do not, shows that eliminating the recalculation gives a boost in performance. By comparing Table 4.6 and Table 4.7 it is evident that the hybrid versions get a significant boost in performance when comparing the first iteration with the subsequent iterations, which was expected.

The reason why KLU was used for hybrid 2 was because it has showed promising results in similar studies. The reason why hybrid 2 outperformed the CPU versions and the GPU versions is probably due to the fact that it did not recompute the preprocessing at each iteration. One of the reasons why hybrid 2 outperformed hybrid 1 might be dependent on the reordering scheme since they did not use the same reordering scheme.

In Figure 4.1 it can be seen that 128 threads per block gave the best performance. This is probably due to the streaming multiprocessors keeping busy but not being overloaded with work. This performance was for the largest case, namely the case with 9241 buses. For smaller networks the threads per block should probably be decreased and for the larger cases it should probably be increased. Since the execution time of the construction of the Jacobian matrix was relatively short compared to the execution time for solving the linear equations, further experiments for finding the optimal number of threads per block were not performed.

It was unexpected to see that the GPU versions performed better on the local hardware platform compared to the remote hardware platform. One reason for this could be that the local platform has a newer GPU, but the V100 GPU should still outperform the newer Quadro t2000 GPU. On the other hand, the hybrid versions got a significant boost in performance by moving the execution from the local hardware to the remote hardware. As seen in Tables 4.9 and 4.10, hybrid 2 got a speedup factor of almost 2.5 for the largest case on the local platform compared to the remote platform, while hybrid 1 got a speedup factor of around 1.6.

When analyzing Tables 4.9 and 4.10 it can be seen that the execution time for the case with 2000 buses is actually longer than the case with 3120 buses for some of the versions. This was a surprising outcome but a possible explanation of this could be that the sparsity pattern of the Jacobian matrix for the two cases differed in how dense they were or in the way that they were structured. The fill-in might also have an impact on the execution time meaning that the case with 2000 buses and a specific reordering scheme might have more fill-in compared to the case with 3120 buses.

When running the case with 9241 buses on the GPU more than 90% of the global memory was used and therefore comparisons with larger networks were

not feasible on the local hardware. It would however be interesting to make comparisons with larger networks. One disadvantage of the GPU implementation was that a large portion of the Y-matrix was copied four times as mentioned earlier. This was done in order to facilitate the access to the elements in the Y-matrix and to align the memory transactions. This would not have been possible using only one Y-matrix.

As seen in Figure 4.3, COLAMD reordering outperformed the other reordering schemes significantly when it came to the C++ versions that used LU factorization. The reason for this might be that COLAMD reordering produces less fill-in compared to AMD reordering and natural ordering.

It was unexpected to see that the QR factorization on the CPU version got outperformed by all the other versions when it came to large networks. As seen in Table 4.4, the case with 9241 buses and natural ordering took almost 25 hours to solve. As seen in Figure 4.8 the QR factorization on the CPU starts as the best performing version when the number of buses is small but as the number of buses increase, the execution time increases drastically. It would be interesting to evaluate if other libraries that provide sparse QR factorization on the CPU yield better results on large data compared to the Eigen library.

5.2 Comparison to the related works

The study conducted in the work by Guo et al. [29] included the same network with 118 buses as used in this study. The total runtime on the CPU in the study by Guo et al. for the case with 118 buses was 313.2ms whilst the same case on the GPU had a total runtime of 199.7ms. In the study conducted in this paper the total runtime for the network with 118 buses had an average runtime of 48.3ms on the CPU and an average runtime of 211.3ms on the GPU. This shows that the GPU versions perform very similarly between the two studies but that the CPU version performs better in the study conducted in this paper with the CPU runtime being nearly six times faster than what was found in the work by Guo et al. There may be several reasons for this, though the one that feels most intuitive is that Guo et al. did not exploit the sparsity of the Jacobian matrix and the Y-matrix. Another explanation could be the choice of linear solver.

Comparing the study conducted in this paper with the one by Singh and Aruni [30] was not as straight forward as there were quite a few unknowns in that

study, for example the linear solvers used for the different versions were not specified. However, one of the networks evaluated in this study and the study of Singh and Aruni were the same, namely the one with 118 buses. The creation of the Jacobian matrix for the case with 118 buses in [30] had a runtime of 3.13ms on the CPU and 0.073ms on the GPU. In this study the creation of the Jacobian matrix for the same network had an average runtime of 0.245ms on the CPU and 0.047ms on the GPU. There may be several reasons for why the versions in this study performed better than in the study by Singh and Aruni. One of the factors might be that Singh and Aruni may not have exploited the similarity between the P and Q equations and the diagonal elements of the Jacobian matrix. Another possible explanation could be that they used one Y-matrix which in turn affected the performance in a negative way when reading from it. It is also unclear if streams were used on the GPU for the construction of the Jacobian matrix in [30]. Finally, Singh and Aruni's study was conducted ten years ago, in 2010, which could entail that the hardware might have had an important role in the difference in execution time.

5.3 Sustainability and Ethics

The ethical aspect of this thesis mainly relates to the execution time of the power flow problem. Since this thesis is trying to minimize the execution time of the power flow problem, it means that the error margin of the solution can be minimized and therefore it is more cost efficient for Svenska kraftnät.

The solution to the power flow problem is used by Svenska kraftnät to make sure that the Swedish power grid works as intended. By minimizing the execution time, a solution to a potential fault can be found at an earlier stage, therefore taking the necessary steps to solve the fault can also be done at an earlier stage.

Chapter 6

Conclusions

The research question that this thesis aimed to answer was: *Will the computation using the Newton-Raphson method applied to large scale power flow problems perform better on the CPU, GPU or a combination of both?*

This study shows that the hybrid versions, the versions that use both the CPU and GPU for the computation, performed better than all the other versions developed and evaluated in this study. When comparing the best performing hybrid version with the best performing CPU version, both running the network with 9241 buses, a speedup factor of 114 was achieved. When comparing the best performing GPU version with the best performing CPU version, both running the network with 9241 buses, a speedup factor of 6 was achieved. Finally, when comparing the best performing hybrid version with the best performing GPU version, both running the network with 9241 buses, a speedup factor of 19 was achieved.

Based on the results presented in Chapter 4 the conclusion can be drawn that the hybrid versions are the best suited version for the Newtons-Raphson method applied to large scale power flow problems. For the cases with 69 buses or less, the hybrid version gets outperformed by the GPU version. Furthermore, for the case with 5 buses, the CPU outperforms all the other versions. With this information, the conclusion can be drawn that the hybrid version is not the best suited for small problems.

6.1 Future work

Below follows a list of suggestions for possible future work.

- The different versions presented in this thesis could be evaluated further by executing them on a larger number of hardware platforms and with larger problems.
- It would be interesting to have a comparison between the versions proposed in this thesis, especially the hybrid versions and with the existing KLU and GLU algorithms to see which linear solver has the best performance when it comes to the power flow problem.
- To further evaluate the performance of the hybrid versions, the hybrid versions could be compared to different iterative techniques to solve the linear equations.
- The hybrid versions and the GPU versions in this study could be executed on GPU clusters, meaning that several GPUs could be used to get a solution to the power flow problem. It would be interesting to see how that would impact the performance.
- Another potential future work could be to evaluate how the performance is affected by solving several power flow problems on the same hardware platform at the same time.

Bibliography

[1] *New reliability criteria for power system management*. URL: https://cordis.europa.eu/article/id/221334-new-reliability-criteria-for-power-system-management.

[2] P. Schavemaker and L. van der Sluis. *Electrical Power System Essentials*. Wiley, 2008. ISBN: 9780470987681. URL: https://books.google.se/books?id=-MnG05siaG0C.

[3] Mohammed Albadi. "Power Flow Analysis". In: *Computational Models in Engineering*. Ed. by Konstantin Volkov. Rijeka: IntechOpen, 2020. Chap. 5. DOI: 10.5772/intechopen.83374. URL: https://doi.org/10.5772/intechopen.83374.

[4] Lucian Ioan Dulau and Dorin Bica. "Power Flow Analysis with Loads Profiles". In: *Procedia Engineering* 181 (Dec. 2017), pp. 785–790. DOI: 10.1016/j.proeng.2017.02.466.

[5] P. Kundur, N.J. Balu, and M.G. Lauby. *Power System Stability and Control*. EPRI power system engineering series. McGraw-Hill Education, 1994. ISBN: 9780070359581. URL: https://books.google.se/books?id=2cbvyf8Ly4AC.

[6] Archita Vijayvargia et al. *Comparison between Different Load Flow Methodologies by Analyzing Various Bus Systems*. Tech. rep. 2. 2016, pp. 127–138. URL: http://www.irphouse.com.

[7] *(PDF) POWER FLOW ANALYSIS CONSIDERING NEWTON RAPHSON, GAUSS SEIDEL AND FAST- DECOUPLED METHODS*. URL: https://www.researchgate.net/publication/322113075_POWER_FLOW_ANALYSIS_CONSIDERING_NEWTON_RAPHSON_GAUSS_SEIDEL_AND_FAST-_DECOUPLED_METHODS.

[8] *Compare the Gauss Seidel and Newton Raphson Methods of Load Flow Study*. URL: http://www.brainkart.com/article/Compare-the-Gauss-Seidel-and-Newton-Raphson-Methods-of-Load-Flow-Study_12416/.

[9] F De Leon and A Semlven. *Iterative solvers in the Newton power flow problem: preconditioners, inexact solutions and partial Jacobian updates*. Tech. rep.

[10] *Iterative linear equation solver*. URL: https://abaqus-docs.mit.edu/2017/English/SIMACAEANLRefMap/simaanl-c-itrsolveroverview.htm.

[11] A.J Flueck and Hsiao-Dong Chiang. "Solving the nonlinear power flow equations with a Newton process and GMRES". eng. In: *1996 IEEE International Symposium on Circuits and Systems (ISCAS)*. Vol. 1. IEEE, 1996, 657–660 vol.1. ISBN: 0780330730.

[12] Lloyd N. Trefethen and David Bau. *Numerical Linear Algebra*. SIAM, 1997. ISBN: 0898713617.

[13] *(Sparse) Linear Solvers*. Tech. rep.

[14] Tatsuo Ohtsuki, Lap Kit Cheung, and Toshio Fujisawa. "Minimal triangulation of a graph and optimal pivoting order in a sparse matrix". In: *Journal of Mathematical Analysis and Applications* 54.3 (1976), pp. 622–633.

[15] George Karypis and Vipin Kumar. "METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices". In: (1997).

[16] Doxygen. *OrderingMethods*. https://eigen.tuxfamily.org/dox/group$_{OrderingMethods}{}_{Module.html}$. 2020.

[17] P. Agarwal and E. Olson. "Variable reordering strategies for SLAM". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2012, pp. 3844–3850.

[18] *Reverse Cuthill Mckee Algorithm*. URL: https://www.geeksforgeeks.org/reverse-cuthill-mckee-algorithm/.

[19] *Band matrices*. URL: https://www.extremeoptimization.com/Documentation/Vector-and-Matrix/Structured-Matrix-Types/Band-Matrices.aspx.

[20] Timothy A. Davis and Ekanathan Palamadai Natarajan. “Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems”. In: *ACM Trans. Math. Softw.* 37.3 (Sept. 2010). ISSN: 0098-3500. DOI: 10.1145/1824801.1824814. URL: https://doi-org.focus.lib.kth.se/10.1145/1824801.1824814.

[21] John R Gilbert and Tim Peierls. “Sparse Partial Pivoting in Time Proportional to Arithmetic Operations”. eng. In: *SIAM journal on scientific and statistical computing* 9.5 (1988), pp. 862–874. ISSN: 0196-5204.

[22] Kai He et al. “GPU-Accelerated Parallel Sparse LU Factorization Method for Fast Circuit Analysis”. eng. In: *IEEE transactions on very large scale integration (VLSI) systems* 24.3 (2016), pp. 1140–1150. ISSN: 1557-9999.

[23] L. Razik et al. “A comparative analysis of LU decomposition methods for power system simulations”. In: *2019 IEEE Milan PowerTech*. 2019, pp. 1–6.

[24] W. Lee, R. Achar, and M. S. Nakhla. “Dynamic GPU Parallel Sparse LU Factorization for Fast Circuit Simulation”. In: *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 26.11 (2018), pp. 2518–2529.

[25] Shaoyi Peng and Sheldon X. -D Tan. “GLU3.0: Fast GPU-based Parallel Sparse LU Factorization for Circuit Simulation”. eng. In: (2019).

[26] J. L. Greathouse and M. Daga. “Efficient Sparse Matrix-Vector Multiplication on GPUs Using the CSR Storage Format”. In: *SC ’14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2014, pp. 769–780.

[27] J. Cheng, M. Grossman, and T. McKercher. *Professional CUDA C Programming*. EBL-Schweitzer. Wiley, 2014. ISBN: 9781118739327. URL: https://books.google.se/books?id=q3DvBQAAQBAJ.

[28] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*. 1st. Addison-Wesley Professional, 2010. ISBN: 0131387685.

[29] Chunhui Guo et al. “Performance comparisons of parallel power flow solvers on GPU system”. In: *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2012, pp. 232–239.

[30] J. Singh and I. Aruni. “Accelerating Power Flow studies on Graphics Processing Unit”. In: *2010 Annual IEEE India Conference (INDICON)*. 2010, pp. 1–5.

[31] X. Li et al. “GPU-Based Fast Decoupled Power Flow With Preconditioned Iterative Solver and Inexact Newton Method”. In: *IEEE Transactions on Power Systems* 32.4 (2017), pp. 2695–2703.

[32] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas. “MATPOWER: Steady-State Operations, Planning, and Analysis Tools for Power Systems Research and Education”. In: *IEEE Transactions on Power Systems* 26.1 (2011), pp. 12–19.

[33] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. http://eigen.tuxfamily.org. 2010.

TRITA-EECS-EX-2020:885