Delft University of Technology
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Load Flow Problem Parallel Programming on the GPU with MATLAB

A thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfillment of the requirements

for the degree

MASTER OF SCIENCE
in
Education and Communication

by

Erik Berkhof

Delft, the Netherlands
2015

MSc THESIS Education and Communication

“Load Flow Problem Parallel programming on the GPU with MATLAB”

Erik Berkhof

Delft University of Technology

Daily supervisor	**Responsible professor**
Prof. dr. ir. C. Vuik	Prof. dr. ir. C. Vuik

Other thesis committee members

Prof. dr. ir. H.X. Lin	Dr. J. G. Spandaw

January 26, 2015 Delft, the Netherlands

Foreword

This document is my thesis for the master Science Education and Communication. The topic of the study is solving a load flow problem by using the GPU, where we use the program language MATLAB. This thesis is carried out in the Numerical Analysis group of the Delft Institute of Applied Mathematics, at Delft University of Technology in Delft.

My special thanks goes to Prof. dr. ir. C. Vuik, my supervisor at the Technical University of Delft. He gave me the time and the confidence I needed during the research. When I needed help, he always made time to look at my work.

I would also like to thanks Prof. dr. ir. H.X. Lin and Dr. J. G. Spandaw for being member of my Thesis Committee.

Thank you!

Delft, February 6, 2015

Erik Berkhof

Table of Contents

1 Introduction

In order to fulfill everybody's electricity needs, a complex electricity network exists. A lot of generators and cables are needed for generation and tranportation of the electrical current. If there is a problem, say a power cut, then it is important that the network is functioning as good as possible. In order to get this done, we will solve load flow problems. Ideally, when a transmission line of an electrical network is broken, the system must continue to operate. It takes a lot of time to compute a solution of this problem, since the network can be very large. Programming this problem on the GPU might be a solution, since many algoritms work a lot faster when they are parallel programmed on the GPU.
There has been more research to solve the load flow problem by using the GPU. We will refer to the thesis of Shiming Xu [1].

In chapter 2 we will look at what is a power system model and which matrices are used.
In chapter 3 we will describe what is a load flow problem, i.e. the problem we want to solve. We will use the Newton-Rapshon method to solve the resulting non-linear system.
In chapter 4 we describe two iterative methods, Bi-CGSTAB and GMRES. Further in this document we will only work with the Bi-CGSTAB method, but we will also give another option. We give here also a couple of preconditioners.
In chapter 5 we give a short description of a GPU. In this chapter we shall in particular discuss how we use the GPU in MATLAB and the MATLAB tool *arrayfun*. We shall here, and also in the following chapters, use the Poisson matrix and describe how we can program the preconditioners in parallel.
We get examples of load flow problems by the program Matpower, which we give in chapter 6. With the results of the last chapter we know that using full matrices we do not speed up if we use the GPU in MATLAB, it is even slower. So we give in this chapter a negative result.

The admittance matrix is a sparse matrix (the matrix we use by solving the load flow problem). The matrix appears to be unstructured. What if the matrix have a clear structure, is it possible using the GPU and only MATLAB to have speed up? This is the reason why we

use in the next chapters the Poisson matrix. So in chapter 7 we use an important and sparse matrix with a clear structure, the Poisson matrix. We describe in this chapter the speed of computation of (multiple) matrix vector multiplications by using the GPU or the CPU. We give a couple of codes and look what gives us the most speed up.
The results that we found in the last chapter, will be used in chapter 8. We want to solve a linear system $A\underline{x} = \underline{b}$, where A is the Poisson matrix. There is a code for the Bi-CGSTAB method in MATLAB, but we want find a faster code in MATLAB with using the GPU. In this chapter we use the structure of the Poisson matrix.
In chapter 9 we conclude that we do not found a faster code in MATLAB to solve the load flow problem by using the GPU. In the last chapter we have a code that is faster than the standard code for Bi-CGSTAB method in MATLAB, but only when we use the diagonal scaling preconditioner. Our code is faster if we used the CPU. So we conclude that when we use MATLAB to solve a linear system $A\underline{x} = \underline{b}$, using the GPU is not a good option if you want speed up our code.

In this document we compare the time of different codes to compute the solution of a linear system, with and without using the GPU. The results depend on the type of CPU and GPU we use. The CPU we use is an 8 core i7 CPU 920 @2.67GHz and has the following memory specifications: maximum memory size of 24 GB, three channels and a max bandwidth of 25.6 GB/sec.The computer has 6 GB RAM memory.
The GPU we use is an NVIDIA GeForce GTX 650 with the specifications: Clock 5.0 Gbps, bandwidth 80.0 GB/sec and a standard memory configuration of 1 GB.
If we had used another CPU or GPU, the results maybe different.

2 Powersystems

A network of electrical components where electric current is transported, is called a power system. A power system can described as a grid, where generators supply power and the transmission system carries that power to the loads. In this chapter we make a simple form of a power system model. First we use Euler's identity for descriping the sinusoidal voltage and current, whereupon we give the complex representation of voltage and current. We give an explanation of buses and give the admittance matrix.

2.1 Power, Voltage and Current

In a DC circuit (Direct current circuit) the circuit voltages and currents are constant, so independent of time. We can say that a DC circuit has no memory, the circuit voltage or current is independent of the past values of voltage or current. In a DC circuit the following expression holds:

$$P = \frac{V^2}{R} = I^2 R$$

where P is the power, V the voltage, I the current and R the resistance.

In an AC circuit (Alternating current circuit) the circuit voltages and currents are time-dependent. All the voltages and currents are sinusoidal and have the same frequency:

$$v(t) = \sqrt{2}|V| \sin(\omega t) \quad \text{and} \quad i(t) = \sqrt{2}|I| \sin(\omega t)$$

where ω the angular frequency is ($\omega = 2\pi f$) [rad/s]. The power in an AC circuit is:

$$p(t) = \frac{v^2(t)}{R} = i^2(t) R$$

and the avarage power is:

$$P = \frac{1}{T}\int_0^T \frac{v^2}{R} dt = \frac{1}{T}\int_0^T i^2 R dt$$

where T the period of the sine wave is ($T = \frac{1}{f} = \frac{2\pi}{\omega}$) [s]. When the average power in the DC circuit and the average power in the AC circuit are assumed to be equal, then

$$V^2 = \frac{1}{T}\int_0^T v^2 dt \quad \text{and} \quad I^2 = \frac{1}{T}\int_0^T i^2 dt.$$

With substitution we get

$$V = \sqrt{\frac{1}{T}\int_0^T v^2 dt} = \sqrt{2}|V|\sqrt{\frac{1}{T}\int_0^T \sin^2(\omega t) dt} = \sqrt{2}|V|\sqrt{\frac{1}{2}} = |V|$$

and

$$I = \sqrt{\frac{1}{T}\int_0^T i^2 dt} = \sqrt{2}|I|\sqrt{\frac{1}{T}\int_0^T \sin^2(\omega t) dt} = \sqrt{2}|I|\sqrt{\frac{1}{2}} = |I|$$

$|V|$ is the RMS or effective value of the alternating voltage.
$|I|$ is the RMS or effective value of the alternating current.

The voltage and current of a single-phase inductive load can writen as

$$v(t) = \sqrt{2}|V|\cos(\omega t) \quad \text{and} \quad i(t) = \sqrt{2}|I|\cos(\omega t - \varphi)$$

Now we use Euler's identity

$$e^{j\varphi} = \cos(\varphi) + j\sin(\varphi)$$

and the sinusoidal voltage and current can written as:

$$\begin{aligned} v(t) &= \text{Re}\{\sqrt{2}|V|e^{jwt}\} = \text{Re}\{\sqrt{2}Ve^{jwt}\} \quad \text{with} \quad V = |V| && (1)\\ i(t) &= \text{Re}\{\sqrt{2}|I|e^{j(wt-\varphi)}\} = \text{Re}\{\sqrt{2}Ie^{jwt}\} \quad \text{with} \quad I = |I|e^{-j\varphi} && (2)\end{aligned}$$

2.2 Active and Reactive Power

We know that $v(t) = \sqrt{2}|V|\cos(\omega t)$ and $i(t) = \sqrt{2}|I|\cos(\omega t - \varphi)$. The value $\varphi = \delta_V - \delta_I$ is called the power factor angle. So we can find the instantaneous power $p(t)$:

$$\begin{aligned} p(t) &= v(t)i(t)\\ &= \sqrt{2}|V|\cos(\omega t)\sqrt{2}|I|\cos(\omega t - \varphi)\\ &= 2|V||I|\cos(\omega t)\cos(\omega t - \varphi)\\ &= 2|V||I|\cos(\omega t)[\cos\varphi\cos(\omega t) + \sin\varphi\cos(\omega t)]\\ &= |V||I|[2\cos\varphi\cos^2(\omega t) + 2\sin\varphi\sin(\omega t)\cos(\omega t)]\\ &= |V||I|\cos\varphi[2\cos^2(\omega t)] + |V||I|\sin\varphi[2\sin(\omega t)\cos(\omega t)]\\ &= |V||I|\cos\varphi[1 + \cos(2\omega t)] + |V||I|\sin\varphi[\sin(2\omega t)]\\ &= P[1 + \cos(2\omega t)] + Q[\sin(2\omega t)] && (3)\end{aligned}$$

$P = |V||I|\cos\varphi$ and $Q = |V||I|\sin\varphi$.

The term $P[1 + \cos(2\omega t)]$ describes an unidirectional component of the instantaneous power with average value P. This value is called the active power or also real or average power. The cosine represents the phase shift between the voltage and current. We might also say: the cosine of the phase angle between the voltage and current phasor. The active power is defined as $P = |V||I|\cos(\varphi)$.
The term $Q[\sin(2\omega t)]$ is alternately positive and negative and has an average value of zero. When this term has a positive sign, the power flow is toward the load. When it is negative, the power flows from the load back to the source of supply. The amplitude of this oscillating power is called reactive power or imaginary power. The reactive power is defined as $Q = |V||I|\sin(\varphi)$.

Now we can use the complex representation of voltage and current. Remember that $V = |V|$ and $I = |I|e^{-j\varphi}$.

$$\begin{aligned} P &= |V||I|\cos\varphi = \text{Re}\Big(|V||I|e^{j\varphi}\Big) = \text{Re}\Big(V\overline{I}\Big), && (4)\\ Q &= |V||I|\sin\varphi = \text{Im}\Big(|V||I|e^{j\varphi}\Big) = \text{Im}\Big(V\overline{I}\Big). && (5)\end{aligned}$$

Where $\overline{I}$ is the complex conjugate of I. Let $S = V\overline{I}$.

2.3 Impedance and Admittance

Impedance is the measure of the opposition that a circuit presents of a current when a voltage is applied. An impedance is the extention of the notion resistance and is denoted by $Z = R + jX$ and measured in Ohm (Ω). We call R the resistance and X the reactance, where $R > 0$.
If $X > 0$, then the reactance is called inductive and we can write $jX = j\omega L$, where $L > 0$ and called the inductance.
If $X < 0$, then the reactance is called capactive and we can write $jX = \frac{1}{j\omega C}$, where $C > 0$ and called the capacitance.

The inverse of impedance is called the admittance and is denoted by $Y = G + jB$. We can write $Y = \frac{1}{Z} = \frac{R}{|Z|^2} - j\frac{X}{|Z|^2}$ and the measure is siemens (S). We call $G = \frac{R}{|Z|^2} \geq 0$ the conductance and $B = -j\frac{X}{|Z|^2}$ the susceptance.
The voltage drop over an impedance Z is equal to $V = ZI$. This is the extension of Ohm's law to AC circuits. We can also write

$$I = \frac{1}{Z}V = YV.$$

The power consumed bij the impedance is

$$S = V\overline{I} = ZI\overline{I} = |I|^2 Z = |I|^2 R + j|I|^2 X.$$

2.4 Kirchhoff's circuit laws

To calculate the voltage and current in an electrical circuit, we use Kirchhoff's circuit laws.

Kirchhoff's current law (KCL)
At any point in the circuit that does not represent a capacitor plate, the sum of currents flowing towards that point is equal to the sum of currents flowing away from that point, i.e., $\sum_k I_k = 0$.

Kirchhoff's voltage law (KVL)
The directed sum of the electrical potential differences around any closed circuit is zero. i.e., $\sum_k V_k = 0$.

2.5 Power System Model

A power system model as a network of buses (nodes) and lines (edges). At each bus i for electrical magnitudes are of importance:

$$\begin{aligned} |V_i| &\quad \text{, the voltage amplitude,} \\ \delta_i &\quad \text{, the voltage phase angle,} \\ P_i &\quad \text{, the injected active power,} \\ Q_i &\quad \text{, the injected reactive power.} \end{aligned}$$

In the network we have generators and loads. In the model are generator buses (or PV-buses), where P_i and $|V_i|$ are specified and Q_i and δ_i are unknown. And in the model are load buses

(or PQ-buses) where P_i and Q_i are specified and $|V_i|$ and δ_i are unknown. A load will have specified negative injected active power P, and specified reactive power Q.
A generator should control P and $|V|$. However there are generators who can not control them. An example of the latter is a wind turbine. So we modeled those ones as a load with a positive injected active power P. In case that a PV generator and a PQ load are connected to the same bus, this results in a PV-bus with:

- a voltage amplitude equal to that of the generator,
- an active power equal to the sum of the active power of the generator and the load.

Buses without a generator or load connected (such as transmission substations) are modeled as load with $P = Q = 0$.

In a power system we deal with system losses. These losses have to be taken into account. They are dependent on the power flow, but these losses are not known in advance. So a generator bus has to be assigned to supply these unknown losses, called slack bus. For the slack bus we cannot specify the real power P, but we can specify the magnitude $|V|$. For a slack bus it is generally specified that $\delta = 0$.

Lines are the network representation of the transmission that connect buses in the power system. A transmission line from i to j has some impedance. The total impedance over the line is modeled as a single impedance z_{ij} of the line. From section 2.3 we know that the admittance of that line is $y_{ij} = \frac{1}{z_{ij}}$. There is shunt admittance from the line and the ground. For the model we distribute this total shunt admittance over buses i and j. There is no conductance between line and ground, but there are losses. This means that the shunt admittance is only due to the electrical field between line and ground. So we have $y_s = jb_s$ with $b_s > 0$. See Figure 1.

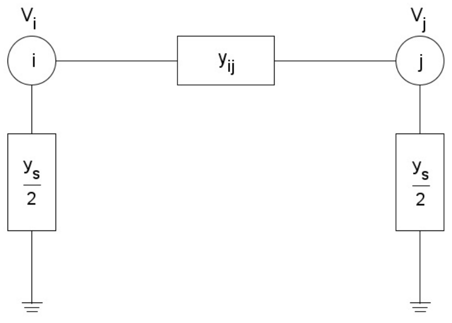

Figure 1: Transmission line model

There are also three other devices in power systems: shunts, tap transformers and phase shifters.

Shunt capacitors can be used to inject reactive power (resulting in a higher node voltage) and shunt inductors consume reactive power (resulting in a lower node voltage). A shunt is modeled as a reactance $z_s = jx_s$ between de bus and the ground. The shunt admittance is $y_s = \frac{1}{z_s} = -j\frac{1}{x_s} = jb_s$. The shunt is inductive if $x_s > 0$, and is capacitive is $x_s < 0$.

A tap transformer is a transformer that can be set to different turns ratios. Tap transformers are generally used to control the voltage magnitude, dealing with fluctuating industrial and domestic demands or with the effects of switching out a circuit for maintenance.
Phase shifters are devices that can change the voltage phase angle, while keeping the voltage magnitude constant. They can be used to control the active power.

2.6 Admittance Matrix

Let I be the vector of injected currents at each bus and V the vector of bus voltages. from section 2.3 we know that $I = YV$. So we say that matrix Y is the admittance matrix. We also know that $Z = Y^{-1}$, the impedance matrix.

First we look at the injected current I_i at each bus i. Is $I_i > 0$ power is generated, if $I_i < 0$ power is consumed and if $I_i = 0$ there is no current injected. The Kirchoff's current law says

$$I_i = \sum_k I_{ik}.$$

Let y_{ij} be the admittance of the line between buses i and j. If there is no line between buses i and j, then $y_{ij} = 0$. If there is only one line between buses i and j, then we have $I_{ij} = y_{ij}(V_i - V_j)$. In this situations it also holds that $I_{ij} = -I_{ji}$. We can write it in matrix notation

$$\begin{bmatrix} I_{ij} \\ I_{ji} \end{bmatrix} = y_{ij} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} V_i \\ V_j \end{bmatrix}$$

If the power system only consists of simplified lines, then the admittance matrix for that system is a Laplacian matrix [3], given by

$$Y_{ij} = \begin{cases} \sum_{k \neq i} y_{ik} & \text{if } i = j, \\ -y_{ij} & \text{if } i \neq j. \end{cases}$$

Then we have

$$I_i = \sum_k I_{ik} = \sum_k y_{ik}(V_i - V_k) = \sum_{k \neq i} y_{ik} V_i - \sum_{k \neq i} y_{ik} V_k = \sum_k Y_{ik} V_k = (YV)_i$$

If a shunt s is connected to bus i, then $I_{is} = y_s(V_i - 0) = y_s V_i$. This means that in the admittance matrix Y, an extra term y_s has been added to Y_{ii}.

Knowing how to deal with shunts, it is easy to incorporate the line shunt admittance model as depicted in Figure 1. For a transmission line with shunt admittance y_s we find

$$\begin{bmatrix} I_{ij} \\ I_{ji} \end{bmatrix} = \left(y_{ij} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} + y_s \begin{bmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{bmatrix} \right) \begin{bmatrix} V_i \\ V_j \end{bmatrix}.$$

The influence on the admittance matrix of a device t between buses i and j is either a tap transformer or a phase shifter. Let E be the voltage induced by t, then $V_i = TE$.

Then the current from bus j to t in the direction of bus i is:

$$I_{ji} = y_{ij}(V_j - V_i) = y_{ij}(V_j - E) = y_{ij}(V_j - \frac{V_i}{T}).$$

With the conservation of power we get

$$V_i\overline{I_{ij}} = -E\overline{I_{ji}} \Rightarrow T\overline{I_{ij}} = -\overline{I_{ji}} \Rightarrow \overline{T}I_{ij} = -I_{ji}.$$

Then the current from bus i to t in the direction of bus j is:

$$I_{ij} = -\frac{I_{ji}}{\overline{T}} = y_{ij}(\frac{V_i}{|T|^2} - \frac{V_j}{\overline{T}}).$$

If the device t that connects bus i to bus j is a tap transformer, then $\overline{T} = T$ and $|T|^2 = T^2$. Then we find

$$\begin{bmatrix} I_{ij} \\ I_{ji} \end{bmatrix} = y_{ij} \begin{bmatrix} \frac{1}{T^2} & -\frac{1}{T} \\ -\frac{1}{T} & 1 \end{bmatrix} \begin{bmatrix} V_i \\ V_j \end{bmatrix}.$$

If instead, t is a phase shifter, then $\overline{T} = e^{-j\delta_T} = \frac{1}{T}$ and $|T|^2 = 1$. Then we find

$$\begin{bmatrix} I_{ij} \\ I_{ji} \end{bmatrix} = y_{ij} \begin{bmatrix} 1 & -T \\ -\overline{T} & 1 \end{bmatrix} \begin{bmatrix} V_i \\ V_j \end{bmatrix}.$$

3 Load Flow Problem

Computing the flow of electrical power in a power system in steady state is called the load flow problem. This means that we calculate all node voltages and line currents in the power system. First we describe the load flow model, what gives us a system of non-linear real equations. To solve this system of non-linear equations we use an iterative technique to approximate the solution, the Newton-Raphson method.

3.1 Load Flow Model

The power consumed by the impedance is given by matrix S. From section 2 we can find

$$S_i = V_i \overline{I_i} = V_i (\overline{YV})_i = V_i \sum_{k=1}^{N} \overline{Y}_{ik} \overline{V}_k.$$

The admittance matrix Y is easy to obtain and generally very sparse. We know that $V = ZI$, where Z is the impedance matrix. The impedance matrix is generally not sparse and harder to obtain. For each bus i where $S_i = 0$ (i.e. no injected power), the injected current $I_i = 0$. So we use the lineair $(YV)_i = 0$ and then we eliminate the variable V_i.

We know that $Y = G + jB$. Let $\delta_{ij} = \delta i - \delta_j$ such that $V_i = |V_i| e^{j\delta i}$ (from which follows that $I_i = |I_i| e^{-j\delta_j}$, see section 2.1). This gives

$$\begin{aligned} S_i &= |V_i| e^{j\delta_i} \sum_{k=1}^{N} (G_{ik} - jB_{ik}) |V_k| e^{-j\delta_k} \\ &= \sum_{k=1}^{N} |V_i||V_k| (\cos\delta_{ik} + j\sin\delta_{ik})(G_{ik} - jB_{ik}) \\ &= \sum_{k=1}^{N} |V_i||V_k| (G_{ik}\cos\delta_{ik} + B_{ik}\sin\delta_{ik}) + j \sum_{k=1}^{N} |V_i||V_k| (G_{ik}\sin\delta_{ik} - B_{ik}\cos\delta_{ik}). \end{aligned}$$

Now we define a real vector of the voltage variables of the load flow problem as

$$\mathbf{V} = [\delta_1, \ldots, \delta_N, |V_1|, \ldots, |V_N|]^T.$$

We define, for a more comfortable notation

$$\begin{aligned} P_{ik}(\mathbf{V}) &= |V_i||V_k| (G_{ik}\cos\delta_{ik} + B_{ik}\sin\delta_{ik}), \\ Q_{ik}(\mathbf{V}) &= |V_i||V_k| (G_{ik}\sin\delta_{ik} - B_{ik}\cos\delta_{ik}). \end{aligned}$$

And so we get

$$S_i = \sum_{k=1}^{N} P_{ik}(\mathbf{V}) + j \sum_{k=1}^{N} Q_{ik}(\mathbf{V}).$$

Now we have a real and an imaginary part:

$$\begin{aligned} P_i &= P_i(\mathbf{V}) = \sum_{k=1}^{N} P_{ik}(\mathbf{V}), \\ Q_i &= Q_i(\mathbf{V}) = \sum_{k=1}^{N} Q_{ik}(\mathbf{V}). \end{aligned}$$

These equations relate the complex power in each node to the node voltages, using the admittance matrix of the power system network. We now have $2N$ non-linear real equations. Each node has four variables: $|V_i|, \delta_i, P_i$ and Q_i. In section 2.5, we say that in each node two of these have a specified value. So we have $2N$ non-linear real equations and $2N$ unknowns variables.

3.2 Newton-Rapshon method

To solve this system of non-linear equations we use an iterative technique to approximate the solution. We will use the Newton-Raphson method to obtain a solution. We define a real vector of the voltage variables of the load flow problem:

$$\mathbf{V} = [\delta_1, \ldots, \delta_N, |V_1|, \ldots, |V_N|]^T.$$

Now we define a set of functions $\mathcal{F}_i$ as:

$$\mathcal{F}_i(\mathbf{V}) = \left[\begin{array}{c} \Delta P_i(\mathbf{V}) \\ \Delta Q_i(\mathbf{V}) \end{array} \right] = \left\{ \begin{array}{lll} P_i - P_i(\mathbf{V}), & i = & 1 \ldots \text{N}, \\ Q_i - Q_i(\mathbf{V}), & i = & \text{N+1} \ldots 2\ \text{N}. \end{array} \right.$$

The function $\mathcal{F}$ is called the power mismatch.

We start with a vector $\mathbf{V^0}$ and update it iteratively with a function Φ, such that:

1. $\mathbf{V^{k+1}} = \Phi(\mathbf{V^k})$
2. $\Phi(\mathbf{V}) = \mathbf{V} \Leftrightarrow \mathcal{F}(\mathbf{V}) = \mathbf{0}$

with $\mathcal{F}$ the vector of functions $\mathcal{F}_i$.

So there is a non-singular matrix $A(\mathbf{V})$ such that $\Phi(\mathbf{V}) = \mathbf{V} - A(\mathbf{V})^{-1}\mathcal{F}(\mathbf{V})$. The Newton-Raphson's method is based on the first order Taylor expansion of $\mathcal{F}$, so $A(\mathbf{V}) = J(\mathbf{V})$, where J the Jacobian of $\mathcal{F}$ is.

The Newton-Rapshon method is defined as

$$\mathbf{V^{k+1}} = \mathbf{V^k} + \Delta\mathbf{V^k},$$

where $\Delta\mathbf{V^k}$ is the solution of the linear system of equations

$$-J(\mathbf{V^k})\Delta\mathbf{V^k} = \mathcal{F}(\mathbf{V^k}).$$

So

$$\mathbf{V^{k+1}} = \mathbf{V^k} - J^{-1}(\mathbf{V^k})\mathcal{F}(\mathbf{V^k}). \tag{6}$$

In the case of the linear system, each bus i of the power system gives two equations, row i and row $N+i$. In the linear system of equations, the δ_i and $|V_i|$ are unknown. If bus i is a load bus, then δ_i and $|V_i|$ are unknown. In case we have a generator bus or a slack bus we have another situation.

For a slack bus, δ_i and $|V_i|$ are known and P_i and Q_i are unknown. We can eliminate the variables δ_i and $|V_i|$ from the system and substistute their values into the coefficient matrix. Row i corresponds to the equation for P_i, and row $N+i$ corresponds to the equation for Q_i. As soon the Newton-Rapshon process has converged, P_i and Q_i are easily calculated by substituting the found solution into the original equations i and $N+i$.

If we have a generator bus, then P_i and $|V_i|$ are known and Q_i and δ_i are unknown. With $|V_i|$ known, we can eliminate this variable and the column $N+i$ from the linear system. The Q_i is unknown, so we take row $N+i$ out of the system like we did in the case of a slack bus.

Now we calculate the Jacobian. Let N_1 and N_2 be the dimensions after the elimination of the slack and generator buses. The strucure of the Jacobian is

$$J(\mathbf{V}) = -\begin{bmatrix} \frac{\partial P_1(\mathbf{V})}{\partial \delta_1} & \cdots & \frac{\partial P_1(\mathbf{V})}{\partial \delta_{N_1}} & \frac{\partial P_1(\mathbf{V})}{\partial |V_1|} & \cdots & \frac{\partial P_1(\mathbf{V})}{\partial |V_{N_2}|} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial P_{N_1}(\mathbf{V})}{\partial \delta_1} & \cdots & \frac{\partial P_{N_1}(\mathbf{V})}{\partial \delta_{N_1}} & \frac{\partial P_{N_1}(\mathbf{V})}{\partial |V_1|} & \cdots & \frac{\partial P_{N_1}(\mathbf{V})}{\partial |V_{N_2}|} \\ \frac{\partial Q_1(\mathbf{V})}{\partial \delta_1} & \cdots & \frac{\partial Q_1(\mathbf{V})}{\partial \delta_{N_1}} & \frac{\partial Q_1(\mathbf{V})}{\partial |V_1|} & \cdots & \frac{\partial Q_1(\mathbf{V})}{\partial |V_{N_2}|} \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ \frac{\partial Q_{N_2}(\mathbf{V})}{\partial \delta_1} & \cdots & \frac{\partial Q_{N_2}(\mathbf{V})}{\partial \delta_{N_1}} & \frac{\partial Q_{N_2}(\mathbf{V})}{\partial |V_1|} & \cdots & \frac{\partial Q_{N_2}(\mathbf{V})}{\partial |V_{N_2}|} \end{bmatrix}. \tag{7}$$

Let $i \neq j$.

$$\begin{aligned} \frac{\partial P_i(\mathbf{V})}{\partial \delta_j} &= |V_i||V_j|(G_{ij}\sin\delta_{ij} - B_{ij}\cos\delta_{ij}) = Q_{ij}(\mathbf{V}) \\ \frac{\partial P_i(\mathbf{V})}{\partial \delta_i} &= \sum_{k\neq i} |V_i||V_k|(-G_{ik}\sin\delta_{ik} + B_{ik}\cos\delta_{ik}) = -Q_i(\mathbf{V}) - |V_i|^2 B_{ii} \\ \frac{\partial Q_i(\mathbf{V})}{\partial \delta_j} &= |V_i||V_j|(-G_{ij}\cos\delta_{ij} - B_{ij}\sin\delta_{ij}) = -P_{ij}(\mathbf{V}) \\ \frac{\partial Q_i(\mathbf{V})}{\partial \delta_i} &= \sum_{k\neq i} |V_i||V_k|(G_{ik}\cos\delta_{ik} + B_{ik}\sin\delta_{ik}) = P_i(\mathbf{V}) - |V_i|^2 G_{ii} \\ \frac{\partial P_i(\mathbf{V})}{\partial |V_j|} &= |V_i|(G_{ij}\cos\delta_{ij} + B_{ij}\sin\delta_{ij}) = \frac{P_{ij}(\mathbf{V})}{|V_i|} \end{aligned}$$

$$\begin{aligned}
\frac{\partial P_i(\mathbf{V})}{\partial |V_j|} &= 2|V_i|G_{ii} + \sum_{k\neq i} |V_k|(G_{ik}\cos\delta_{ik} + B_{ik}\sin\delta_{ik}) = \frac{P_i(\mathbf{V})}{|V_i|} + |V_i|B_{ii} \\
\frac{\partial Q_i(\mathbf{V})}{\partial |V_j|} &= |V_i|(G_{ij}\sin\delta_{ij} - B_{ij}\cos\delta_{ij}) = \frac{Q_{ij}(\mathbf{V})}{|V_j|} \\
\frac{\partial Q_i(\mathbf{V})}{\partial |V_i|} &= -2|V_i|B_{ii} + \sum_{k\neq i} |V_k|(G_{ik}\sin\delta_{ik} - B_{ik}\cos\delta_{ik}) = \frac{Q_i(\mathbf{V})}{|V_i|} - |V_i|B_{ii}
\end{aligned}$$

4 Linear solvers

In the previous chapter we have seen that

$$\mathbf{V^{k+1}} = \mathbf{V^k} - J^{-1}(\mathbf{V^k})\mathcal{F}(\mathbf{V^k}).$$

To compute the iteration we have to solve the linear equations as given in chapter 3. We can use the iterative methods Bi-CGSTAB or GMRES. For this, we use a preconditioner.

4.1 Preconditioners

A preconditioner is a matrix that transforms the linear system. The transformed system has the same solution as the original system. The transformed coefficient matrix (the preconditioner) has a more favorable spectrum. Assume that we have the linear system $Au = b$, then the transformed system is given by

$$M^{-1}Au = M^{-1}b.$$

These systems have the same solution. For matrix M it must hold that the eigenvalues of $M^{-1}A$ should be clustered around 1 and it should be possible to obtain $M^{-1}y$ at low cost.

When we change the linear system $Au = b$ such that the eigenvalue distribution becomes more favorable with respect to the CG convergence, we speak about the preconditioning of a linear system. The idea is to write the system as $\tilde{A}\tilde{u} = \tilde{b}$, where $\tilde{A} = P^{-1}AP^{-T}, u = P^{-T}\tilde{u}$ and $\tilde{b} = P^{-1}b$. The matrix P is non-singular and the preconditioner matrix M is given by $M = PP^T$. In the next subsections we describe three preconditioners.

4.1.1 Diagonal scaling

We can choose for P a diagonal matrix, where $p_{ii} = \sqrt{a_{ii}}$. It has been shown that this Matrix P is such that it minimizes the condition number of $P^{-1}AP^{-T}$ [2] within the class of diagonal matrices. An advantage is that $\tilde{A} = P^{-1}AP^{-T}$ is easily to calculate and $diag(P^{-1}AP^{-T}) = 1$, so this saves us n multiplications in the matrix vector product.

4.1.2 Basic iterative method

In the basic iterative method we compute the iterates by the following recursion:

$$u^{i+1} = u^i + B^{-1}r^i,$$

where $r^i = b - Au^i$. So it holds that $u^i \in u^0 + \mathsf{span}\{B^{-1}r^0, B^{-1}A(B^{-1}r^0), \ldots, (B^{-1}A)^{i-1}(B^{-1}r^0)\}$. We call subspace $K^i(A; r^0) := \mathsf{span}\{r^0, Ar^0, \ldots, A^{i-1}r^0\}$ the Krylov-space of dimension i corresponding to matrix A and initial residual r^0. When u^i is calculated bij a basic iterative method, then $u^i \in u^0 + K^i(B^{-1}A; B^{-1}r^0)$.

The basic iterative methods uses a splitting of the matrix $A = B - R$. So the i-th iteration y^i from the basic method is an element of $u^i \in u^0 + K^i(B^{-1}A; B^{-1}r^0)$. Two examples for a splitting are the Jacobi and the Gauss-Seidel methods.

Let $A \in \mathbb{R}^{N,N}$, with $A = (a_{mn}), m, n = 1, \ldots, N$ and $a_{mm} \neq 0 (m = 1, \ldots, N), b \in \mathbb{R}^N$.
In the Jacobi method we start with a guess u_m^0 and find recursively the vectors $u^1, \ldots, u^p$ by:

$$u_m^{i+1} = \frac{1}{a_{mm}}(b_m - \sum_{k=1(k\neq m)}^{N} a_{mk}u_k^i), \qquad m =, 1, \ldots, N.$$

The Jacobi Method
This is for one iteration

```
for $m = 1 : N$
    $q_m = (b_m - \sum_{k\neq m} a_{mk}u_k)/a_{mm}$
    $\underline{u} = \underline{q}$
end
```

With Jacobi' s method we find a splitting $A = B - R$, with a diagonal matrix B and a matrix R with zeros on the diagonal. The Jacobi method in matrix notation is $Bu^{i+1} = Ru^i$+b.

In Gauss-Seidel's method we use the iteration

$$u_m^{i+1} = \frac{1}{a_{mm}}(b_m - \sum_{k=1}^{m-1} a_{mk}u_k^{i+1} - \sum_{k=m+1}^{N} a_{mk}u_k^i), \qquad m =, 1, \ldots, N.$$

The Gauss-Seidel Method
This is for one iteration

```
for $m = 1 : N$
    $u_m = (b_m - \sum_{k\neq m} a_{mk}u_k)/a_{mm}$
end
```

4.1.3 Incomplete decomposition

Here we use a combination of an iterative method and an approximate direct method. Let $A \in \mathbb{R}^{N\times N}$ be the coefficient matrix of the problem. The matrix A has at most 5 non-zero elements per row. The matrix is symmetric and positive definite. The structue of the matrix is [4](pag. 69)

$$A = \begin{bmatrix} a_1 & b_1 & & c_1 & & & & & \\ b_1 & a_2 & b_2 & & c_2 & & & & \\ \vdots & \ddots & \ddots & & & \ddots & & & \oslash \\ c_1 & & b_m & a_{m+1} & b_{m+1} & & c_{m+1} & & \\ & \ddots & \oslash & \ddots & \ddots & \ddots & \oslash & \ddots & \\ & \oslash & & & & & & & \end{bmatrix}.$$

Take the lower triangular matrix L such that $A = L^T L$ and $P = L$. The matrix L is a good choice with respect to convergence. The zero elements in the band of A become non-zero elements in the band of L. It can be a lot of work to construct L. Make the non-zero elements of L on the positions where the elements of A are zero also zero. Now it is less work to find L.

Denote the set of all pairs of indices of the off-diagonal matrix by:

$$Q_N = \{(i,j) | i \neq j, 1 \leq i \leq N, 1 \leq j \leq N\}.$$

Let Q a subset of Q_N, where Q are the places (i,j) where L should be zero.

Theorem 1. *[8] if A is a symmetric M-matrix, there exist for each $Q \subset Q_N$ (with the property that $(i,j) \in Q$ implies $(j,i) \in Q$), a uniquely defined lower triangular matrix L and a symmetric nonnegative matrix R with $l_{ij} = 0$ if $(i,j) \in Q$ and $r_{ij} = 0$ if $(i,j) \notin Q$, such that the splitting $A = LL^T - R$ leads to convergent iterative process*

$$LL^T u^{i+1} = Ru^i + b \qquad \textit{for each choise } u^0,$$

where $u^i \to u = A^{-1}b$.

4.2 Bi-CGSTAB

We give the description of the method [5]:

Bi-CGSTAB method

```
$u^0$ is an initial guess; $r^0 = b - Au^0$;
$\overline{r}^0$ is an arbitrary vector, such that $(\overline{r}^0, r^0) \neq 0$, e.g., $\overline{r}^0 = r^0$;
$\rho_{-1} = \alpha_{-1} = \omega_{-1} = 1$;
$v^{-1} = p^{-1} = 0$;
for $i = 0, 1, 2, ...$ do
    $\rho_i = (\overline{r}^0, r^i)$; $\beta_{i-1} = (\rho_i/\rho_{i-1})(\alpha_{i-1}/\omega_{i-1})$;
    $p^i = r^i + \beta_{i-1}(p^{i-1} - \omega_{i-1}v^{i-1})$;
    $\hat{p} = M^{-1}p^i$;
    $v^i = A\hat{p}$;
    $\alpha_i = \rho_i/(\overline{r}^0, v^i)$;
    $s = r^i - \alpha_i v^i$;
    if $||s||$ small enough then
```

```
        $u^{i+1} = u^i + \alpha_i \hat{p}$; quit;
    $z = M^{-1}s$;
    $t = Az$;
    $\omega_i = (t,s)/(t,t)$;
    $u^{i+1} = u^i + \alpha_i \hat{p} + \omega_i z$;
    if $u^{i+1}$ is accurate enough then quit;
    $r^{i+1} = s - \omega_i t$;
end for
```

Here the matrix M is the preconditioning matrix.
This method uses short recurrences. But small changes in the algorithm can lead to instabilities.

4.3 GMRES

This is the second iterative method that we look into. Here we used Arnoldi's method for computing an orthonormal basis $v^1, ..., v^k$ of the Krylov subspace $K^k(A; r^0) := span\{r_0, Ar_0, \dots A^{k-1}r_0\}$. We give here the description [6].

GMRES method

```
Start: choose $u^0$ and compute $r0 = b - Au^0$ and $v^1 = r^0/||r^0||_2$.
Iterate: for $j = 1, ..., k$ do:
            $v^{j+1} = Av^j$
            for $i = 1, ..., j$ do:
                $h_{ij} := (v^{j+1})^T v^i, v^{j+1} := v^{j+1} - h_{ij}v^i$,
            end for
            $h_{j+1,j} := ||v^{j+1}||_2, v^{j+1} := v_{j+1}/h_{j+1,j}$
        end for
```

The entries of $(k+1) \times k$ the upper Hessenberg matrix $\overline{H}_k$ are the scalars h_{ij}.

The form of the Hessenberg matrix H_j is

$$H_j = \begin{bmatrix} h_{11} & \dots & \dots & h_{1j} \\ h_{21} & \ddots & & \vdots \\ & \ddots & \ddots & \vdots \\ O & & h_{j,j-1} & h_{jj} \end{bmatrix}.$$

The GMRES method is based on long recurrences, but it has optimality properties. The disadvantage is that the j^{th} iteration takes more time to compute than the $j-1^{th}$ iteration. So it is not possible to run the full algoritm for large number of iterations. For the GMRES method there exists a convergence proof and for the Bi-CGSTAB there does not.

5 GPU Parallel programming with MATLAB on the GPU

In this chapter we start with a brief explanation about the disign of a GPU. Then we will describe how we can program with MATLAB on the GPU. We want to solve a linear system $A\underline{x} = \underline{b}$. The admittance matrix is a sparse matrix (the matrix we use by solving the load flow problem). The matrix appears to be unstructured. So in this chapter we will focus on programming the Bi-CGSTAB method where matrix A is the Poisson matrix instead of the admittance matrix. Choosing the Poisson matrix makes programming easier, because of the known structure of the Poisson matrix. In the next chapter we will use the admittance matrix.
The iterative method uses a preconditioner, so we will describe two of them: diagonal scaling and incomplete decomposition. The reason that we will use the GPU is that we want speed up. So we shall look if we can achieve speed up.

5.1 The Design of the GPU

Historically, computer users have the expectation that programs run faster with each new generation of microprocessors. However, sequential programs will only run on one of the processor cores and will not become significantly faster than before if a multi-core machine is used.

By a parallel program, multiple threads of execution cooperate to complete the work faster. So application software can be faster when it is a parallel program.
We use a many-core trajectory which focuses more on the execution throughput of parallel applications. Here we have a large number of simple cores. The GPU (graphics processing unit) is an example of a many-core machine. A many-core machine has the advantage of more GFLOPS (giga floating-point operations per second) than a Dual-core or a Quad-core on a CPU (central processing unit).

The reason that there is a large performance gap between many-core GPU and a multicore CPU, is the design between the two types. The designs are illustrated in figure 2

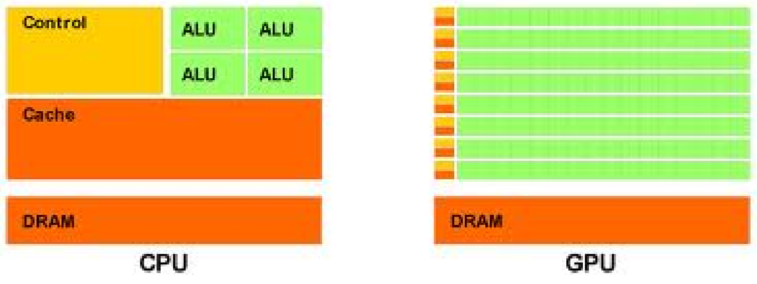

Figure 2: Disign of a CPU and a GPU [9].

The design of a CPU is optimized for sequential code performance. The large cache memories are provided to reduce the instructions. But there is no contribution to calculation speed.
Then we have also the memory bandwidth. The bandwidth of a Graphic chip is more (say 10 times) than the bandwidth of a presently available CPU chip. We speak here about moving data (GB/s) in and out of its main dynamic random access memory (DRAM). Small cache memories are provided to help control the bandwidth requirements of these applications, so

multiple threads that access the same memory data do not need to go all to the DRAM. So there is more space on the chip for floating-point calculations.

5.2 MATLAB on the GPU

We can use MATLAB for programming on the GPU. We create a matrix A with MATLAB in the standard way. If we use the code $class(A)$, then the output is *double*. Now we say $B = gpuArray(A)$ and then we use the code $class(B)$. The output is *parallel.gpu.GPUArray*. So we can set a matrix on the GPU. With the code $C = gather(B)$ we can copy the matrix to the CPU.
Say that the matrices A and B are both on the GPU. When we create a new matrix in MATLAB with the code $C = A + B$, then matrix C is also placed on the GPU.

With an example we can see that there are time savings when we use the GPU.

```
---------------------------------------
X=rand(n,1);
Y=rand(n,1);
Z=rand(n,1);

tic
for i=1:m
   X=plus(Y,X); %the sum of X and Y
   X=plus(Z,X);
   X=6*X;
end
timeCPU=toc;
X;
---------------------------------------
```

We do not put the vectors X, Y and Z on the GPU.

The next vectors P, G and Q are placed on the GPU.

```
---------------------------------------
G = gpuArray(X);
P = gpuArray(Y);
Q = gpuArray(Z);

tic
for i=1:m
   G=plus(P,G);
   G=plus(Q,G);
   G=6*G;
end
timeGPU=toc
k=timeCPU/timeGPU
---------------------------------------
```

Let k be the ratio between timeCPU and timeGPU. In table 1 we can see an approximation of $k = \frac{timeCPU}{timeGPU}$ with different values of n.

n ($\times 10^6$)	0.1	0.5	1	10	15	20	25	30
k	0.013	0.013	2.7	3.6	3.6	3.7	0.9	not enough memory

Table 1: $k = \frac{timeCPU}{timeGPU}$ en $m = 100$.

If the size of the vectors is larger, the factor k is larger. But there is a maximum size of the vectors in case there is not enough space on the GPU. The highest value of k is approximately $3,7$. If we change m, we see little difference.
We can also use the code $arrayfun$. This code ensures that the entire calculation takes place on the GPU. All the data are placed on the memory of the GPU and the calculations are performed on the GPU. First we make a single program.

```
---------------------------------------
function tell = testPlus(X0,Y0,Z0,m0)
tell=1;
while (tell <= m0)
    X0=X0+Y0;
    X0=X0+Z0;
    X0=6*X0;
    tell=tell+1;
end
---------------------------------------
```

Then the vectors G, P and Q placed on the GPU.

```
---------------------------------------
tic
tell = arrayfun(@testPlus, G, P, Q, m);
tijdGPU2 = toc;

k=timeCPU/timeGPU2;
---------------------------------------
```

Let k be the ratio between timeCPU and timeGPU2. In Table 2 we can see an approximation of $k = \frac{timeCPU}{timeGPU2}$ for different values of n.

n ($\times 10^6$)	0,1	0,5	1	10	15	20	25	30
k	0.21	0.21	26.4	29.4	29.5	29.4	29.6	not enough memory

Tabel 2: $k = \frac{timeCPU}{timeGPU2}$ en $m = 100$.

This time the factor k can be 30, see figure 3 . So when we program on the GPU using arrayfun, the program is much faster. However, the limited size of the memory can be a problem.

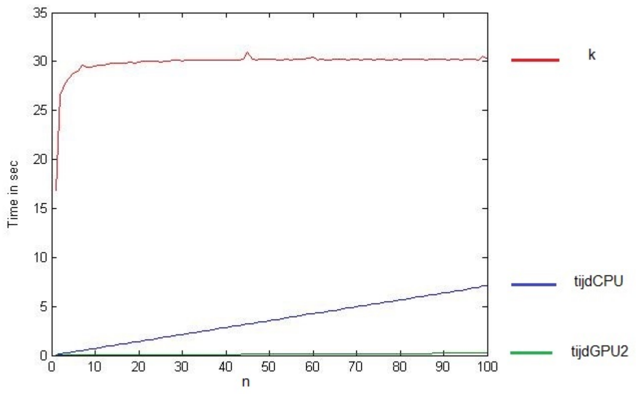

Figure 3: timeCPU, timeGPU2 and k with different n, where the size of the vector is $n \times 106$.

So the code is much faster when we can use *arrayfun*, because the operations supported by *arrayfun* are strictly element-wise. The problem is that only a restricted amount of functions and operators can be combined with arrayfun in MATLAB. For instance, when we want to multiply two vectors, MATLAB gives an error. Let vector $V = [v_1, v_2, v_3, ..., v_n]$. With arrayfun, MATLAB gives an error when we calculate $V^T V$. But when we use VV, we get $[v_1 v_1, v_2 v_2, v_3 v_3, ..., v_n v_n]$. This problem is easy to fix. But if we multiply two matrices A and B, then arrayfun gives a matrix where on row i and column j the value is $A(i, j) \times B(i, j)$. So we get a componentwise form of a matrix-matrix multiplication. When we want to multiply matrices in the normal way with arrayfun, it takes a lot of time and memory. Note that we do not need to multiply in our method. But we have to multiply a matrix and a vector, which gives the same problems.

5.3 Bi-CGSTAB with the Poisson matrix

The easiest way to compute the system $A\underline{x} = \underline{b}$ with Bi-CGSTAB is with the code *x=bicgstab(A,b)*. When we use this code, we do not use the GPU. If the matrix A and vector b are placed on the GPU, we get an error.

First we start make a Possion matrix. In MATLAB we can use the code *A=gallery('poisson',n)*. MATLAB gives us:

```
---------------------------------------
A =

   (1,1)        4
   (2,1)       -1
   (3,1)       -1
   (1,2)       -1
   (2,2)        4
   (4,2)       -1
   (1,3)       -1
   (3,3)        4
   (4,3)       -1
   (2,4)       -1
   (3,4)       -1
   (4,4)        4
---------------------------------------
```

We get an error if we enter $G = gpuArray(A)$. On the GPU, Matlab does not recognize the matrix stored in a sparse format. We can "fix" it if we use the next code, where the matrix is stored as a full matrix.

```
---------------------------------------
P = gallery('poisson',n)
m=n*n;
A=zeros(m);
for i=1:m
    for j=1:m
       A(i,j)=P(i,j);
    end
end
G = gpuArray(A)
---------------------------------------
```

The problem here, is that MATLAB recognizes the matrix as a full matrix with a lot of zeros.

5.4 Programming the Preconditioners

We gave a couple of options for preconditioners in section 4.3. We can choose for diagonal scaling, where the matrix $D = diag(A)$ with $a_{ii} \neq 0$. (When we use the Jacobi method, we find the same diagonal matrix). This matrix is easy to compute and it is parallel programmable.

Matlab code for diagonal scaling

```
-------------------
D=0*A;
for i=1:n
   D(i,i)=A(i,i) ;
end
-------------------
```

We do not use the Gauss-Seidel method here, because the preconditioner is not SPD.

5.4.1 Incomplete Decomposition Preconditioner

We can also use incomplete decomposition. When we want to use this method, matrix A must be sparse, symmetric and positive definite.

The next code is the Matlab code with which we compute the lower triangle matrix we want.

```
---------------------------------------
L=A*0;
for p=1:n
    A(p,p)=sqrt(A(p,p));
      for i=p+1:n
        if A(i,p)~=0
         A(i,p)=A(i,p)/A(p,p);
        end
      end
   for j=p+1:n
      for i=j:n
        if A(i,j)~=0
          A(i,j)=A(i,j)-A(i,p)*A(j,p);
        end
      end
   end
end

for i=1:n
   for j=i+1:n
       A(i,j)=0;
   end
end
L=A;
---------------------------------------
```

It is easy to construct an $n \times n$ Poisson matrix in Matlab by entering A = *gallery('poisson',n)*. With the code *L=ichol(A)*, Matlab gives the matrix L. In case matrix A is not sparse enough, Matlab gives an error message.

For an incomplete decomposition we saw that $A \approx LL^T$. If we want to approximate A^{-1}, we can use the next expression:

$$A^{-1} \approx (LL^T)^{-1} = (L^T)^{-1}L^{-1} = (L^{-1})^T L^{-1}.$$

So if we can compute L^{-1}, we have an approximation of A^{-1}.
Note: L^{-1} is not parallel.

Let D be the diagonal matrix of L, matrix N a lower triangle matrix with zeros on the diagonal and I be the identity matrix. Then we can write the matrix L as follows:

$$L = D(I + N).$$

Now we have also matrix N:

$$L = DI + DN$$
$$DN = L - DI$$
$$N = D^{-1}(L - DI)$$

Now we can write the inverse of L:

$$L^{-1} = (D(I + N))^{-1} = (I + N)^{-1}D^{-1}.$$

It is easy to compute D^{-1}, but $(I + N)^{-1}$ is difficult to calculate in a parallel way. Because matrix N has zeros on the diagonal and is a lower triangular matrix, it holds that $N^n = 0$ if $n \to \infty$. So $(I - N^n) = I$ if $n \to \infty$. Now we use the next rule:

$$(I - N^n) = (I + N)(I - N + N^2 - N^3 + \ldots + (-1)^{n-1}N^{n-1}) = I.$$

Now it follows that:

$$(I + N)^{-1} = (I - N + N^2 - N^3 + \ldots + (-1)^{n-1}N^{n-1}) = \sum_{k=0}^{n-1}(-1)^k N^k.$$

This is called a Neumann series. The rate of convergence is defined in the next theorem.

Theorem 2 (Neumann series)**.** *[10]*
Is A is a square matrix, $||A|| < 1$, then $I - A$ is nonsingular and $(I-A)^{-1} = I + A + A^2 + \cdots = \sum_{k=0}^{\infty} A^k$. This is the Neumann series. The speed of convergence depends on the size of $||A||$.

Now we can approximate L^{-1} by a small number of terms of the Neumann series. Since matrix vector products can be computed in parallel, we now have a parallel approximation of $(L^{-1})^T$, and so we find an approximation of matrix A^{-1}.

Note that N is a sparse matrix, but $(I - N + N^2)$ can become a less sparse matrix. So the multiplication $(I - N + N^2)$ directly with a vector $\underline{v}$ costs a lot of memory when we program it in parallel. So $(I - N + N^2)\underline{v} = I\underline{v} - N\underline{v} + N(N\underline{v})$. In the right part of the equation we can use the sparse property in each multiplication.
The following code represents the rest of the code.

```
---------------------------------------
D=0*L;
for i=1:n
    D(i,i)=L(i,i);
end
%We can also D=diag(L), but it can give problems on the GPU.
%MATLAB recognize only arrays/vectors on GPU.
invD=0*L;
for i=1:n
    invD(i,i)=1/L(i,i);
end

N=invD*(L-D);
I=eye(n);
invT=I-N+N*N;
invL=invT*invD;
invA=invL'*invL;
---------------------------------------
```

We put this code into a function called "IPofPoisson". When we put the matrices on the GPU with the code "gpuArray(...)", we must use a full matrix A. This costs a lot of time, so this is not a good option.

5.4.2 Bi-CGSTAB with Incomplete decomposition preconditioner

In appendix B you can read the code. We used the function "IPvanPoisson" that can be found in appendix A. This function does not use the GPU. We also used the function "IP-vanPoissonGPU". This code does practically the same, but in addition to the code *gpuArray* it copies the matrices and vectors to the GPU.

```
---------------------------------------
A = gallery('poisson',N);
n=N*N;
A=gpuArray(zeros(n)+A);
.....
I=gpuArray(eye(n));
.....
invT=gpuArray(zeros(n));
.....
---------------------------------------
```

When we use this code, we must make matrix A full with a lot of zeros. This costs memory and speedup. Off course we do not want a full matrix, however this is necessary otherwise we can not use the GPU (see section 5.2)

In the code of the Bi-CGSTAB we compute the residual with $||Au^i - b||_2$. First we solve the linear system with the MATLAB code *x=bicgstab(A,b)* (here matrix A is a sparse matrix) and *error=norm(b-A*x,2)/norm(b,2)*. We find the residual with this command, so it is fair to compare. In Table 3 we see the results.

$N \times Ngrid$	time code *x=bicgstab(A,b)*	time CPU	time GPU	iterations
2	0.0174	0.0050	0.150	20
3	0.0016	0.0033	0.0523	6
4	0.00189	0.0009	0.0277	4
5	0.0023	0.00114	0.0368	5
6	0.0026	0.0012	0.0428	5
7	0.0029	0.0014	0.0557	6
8	0.0045	0.0015	0.0654	6
9	0.0038	0.0017	0.0808	7
10	0.0044	0.0018	0.0928	7

Tabel 3: time Bi-CGSTAB with IDP (incomplete decomposition preconditioner)

We see that solving the linear system on the CPU is a lot faster than solving it on the GPU. The reason is that we must use a full matrix A when programming on the GPU. We see that 20 iterations are needed when we use a $2 \times 2 - grid$. I can not explain this large number of iterations.

5.4.3 Bi-CGSTAB with diagonal scaling preconditioner

The MATLAB code for Bi-CGSTAB with a diagonal scaling preconditioner can be found in appendix C. In table 4 we see the results of Bi-CGSTAB with a diagonal scaling preconditioner. The same results are also given in figure 4.

$N \times Ngrid$	time code *x=bicgstab(A,b)*	time CPU	time GPU	iterations
2	0.0815	0.0159	0.0426	3
3	0.0076	0.0031	0.0489	5
4	0.0019	0.0016	0.0722	7
5	0.0034	0.0028	0.1018	10
6	0.0026	0.0032	0.1371	13
7	0.0029	0.0032	0.1589	14
8	0.0033	0.0033	0.1328	18
9	0.0036	0.0033	0.1550	20
10	0.0036	0.0035	0.1670	20

Tabel 4: time Bi-CGSTAB with DSP (diagonal scaling preconditioner)

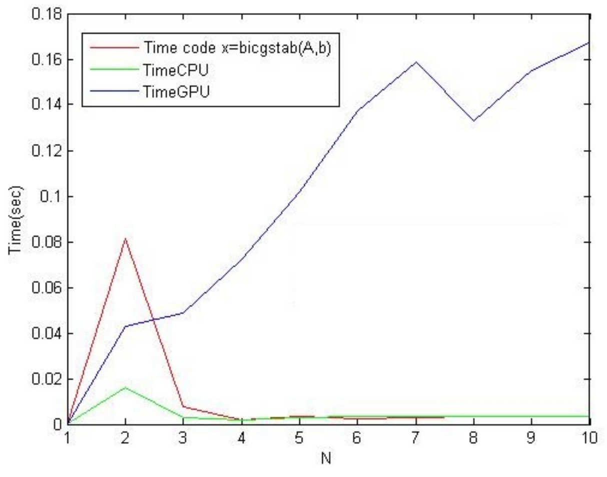

Figure 4: time MATLAB code, CPU and GPU

We see that when we use the GPU, the program is much slower. A reason for this result is that when we use the GPU, we have to use a full matrix A.

When we use a diagonal scaling preconditioner, the number of iterations increases. The maximum of iterations in MATLAB is 20 when we use the code *x=bicgstab(A,b)*, so that is also our maximum number of iterations.
There is an option using the code *x=bicgstab(A,b,tol,maxit)*. The value of *tol* specifies the tolerance of the method. The value of *maxit* specifies the maximum number of iterations. In this document we do not use these two options. We see that the program with the GPU is much slower, when we have a ”large” number of iterations. So in this document the maximum number of iterations is 20.
Notice the difference between the number of iterations when using the incomplete decomposition preconditioner (see Table (3)) or the diagonal scaling preconditioner (see Table (4)).

6 Results for Load Flow Problems

In chapter 2 and 3 we build the admittance matrix, but we can also use Matpower, a package in MATLAB. First we give a short description of Matpower. We shall see that using the GPU in this case do not give us any speed up.

6.1 Matpower

There is a package in MATLAB, named Matpower, to employ Newtons method for solving power flow problems. This package give us examples of networks including the matrices. There are a couple of case-files. These files contain admittance matrices, that we can use for computing. It is also possible to find the bus-matrix, the generator-matrix, the branch-matrix (the specifications of the transmission lines between the buses).

With the next code we get an admittance matrix of a case.

```
---------------------------------------
mpc = loadcase('case10ac');
[Y,~,~] = makeYbus(mpc.baseMVA, mpc.bus, mpc.branch);
---------------------------------------
```

With *loadcase* we can get a lot of cases belonging to different sizes of networks.

6.2 Matpower and Bi-CGSTAB

Instead of the Poisson matrix we used before, we use an admittance matrix Y from Matpower. We have seen that an incomplete decomposition preconditioner is a better option than a diagonal scaling preconditioner. So we use an incomplete decomposition preconditioner. We see the results of a couple of cases in table 5 .

From the results we see that the code of MATLAB, $x = bicgstab(A, b)$ is faster than our code on the CPU. The code is very slow by using the GPU, because we have to use a full matrix.

Note: There are other cases which do not give a solution with our code.

case	n	time MATLAB code	time CPU	time GPU	iterations
case10ac	10	0.0411	0.02337	0.51056	10
case30Q	30	0.04185	0.02428	5.07543	10
case39	39	0.04269	0.02566	9.06620	10
case118	118	0.0437	0.05071	188.9014	16
case3120sp	3120	0.90768	3.53407	too long	20

Tabel 5: time of a couple of cases

6.3 CUDA kernel

We want solving the load flow problem by only using MATLAB. We have seen that using the GPU in MATLAB does not give us any speed up. The choise for *arrayfun* was not correct, because *arrayfun* is usefull when there are only element-wise operations. The Bi-CGSTAB method we need uses matrix-vector multiplications, so *arrayfun* is not very usefull.

There is a command in MATLAB, *parallel.gpu.CUDAkernel(-,-)*, which can operate on MATLAB array or gpuArray variables. To use this code, you should still program in CUDA. In this thesis we only want programming in MATLAB without using knowledge of other programming languages. With the command *parallel.gpu.CUDAkernel(-,-)* MATLAB use a CUDA code, it means that we must first programming in CUDA. In this thesis we only use MATLAB, so the command *parallel.gpu.CUDAkernel(-,-)* is not an option.

The admittance is a sparse matrix, but the we do not see a structure in the matrix. What if the matrix have a clear structure? It is possible to use the GPU and only MATLAB to obtain speed up? This is the reason why we use in the next chapters the Poisson matrix. The Poisson matrix is a sparse matrix and have a simple structure.

7 Parallel Programming of a matrix vector multiplication with the Poisson matrix

We have seen in the previous chapters that computing becomes very slow when we use a full matrix instead of a sparse matrix. In this chapter we use a commonly used sparse matrix, the Poisson matrix. We use the structure of this matrix in our program to obtain speed up. First we look at the structure of the Poisson matrix. We want compute $A^k * \underline{b}$, where A is the Poisson matrix and $k \in \mathbb{N}$. We use arrayfun in our code, because we have seen that arrayfun can give us speed up. We make a couple of codes and compare them. Finally we have three codes: the MATLAB code *x=bicgstab(A,b)*, a code on the CPU and a code on the GPU. This codes we shall compare in chapter 8.

7.1 The Structure of the Poisson matrix

In this chapter we focus on the **scaled** Poisson matrix. We give an example of a Poisson matrix:

$$A = \begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{bmatrix}$$

We can see the Poisson matrix as a block tridiagonal matrix:

$$A = \begin{bmatrix} T & -I & 0 \\ -I & T & -I \\ 0 & -I & T \end{bmatrix}, \tag{8}$$

where matrix I is the identity matrix and matrix T has the form:

$$T = \begin{bmatrix} 4 & -1 & 0 & 0 \\ -1 & 4 & \ddots & 0 \\ 0 & \ddots & \ddots & -1 \\ 0 & 0 & -1 & 4 \end{bmatrix}. \tag{9}$$

First we try a multiplication of matrix A and a vector $\underline{b}$. In case we want to program on the GPU with MATLAB and use the Poisson matrix, it is not meaningful to only use the *gpuArray* code. The code can be faster if we use the *arrayfun* code, but we have seen that there is a problem when there is a multiplication of a matrix and a vector. So we do not use the code *A*b*.

The first idea is to calculate each component separately by only using the CPU. The next code is not an explicit matrix vector product, but the result is the same as $A * b$.

```
Name code: CPU1
---------------------------------------
%NxN-matrix and n=NxN

ans=zeros(n,1);
ans(1,1)=-b(2,1)-b(N+1,1);
ans(n,1)=-b(n-1,1)-b(n-N,1);
ans(N,1)=-b(N-1,1)-b(N+N,1);
ans(n-(N-1),1)=-b(n-(N-1)+1,1)-b(n-(N-1)-N,1);

for i=2:N-1     %first blok
   ans(i,1)=-b(i-1,1)-b(i+1,1)-b(i+N,1);
end

for i=N*N-(N-2):n-1;    %last blok
   ans(i,1)=-b(i-1,1)-b(i+1,1)-b(i-N,1);
end

for i=1:N-2  %first component others bloks
    f=i*N+1;
    ans(f,1)=-b(f+1,1)-b(f+N,1)-b(f-N,1);
end

for i=1:N-2 %last component others bloks
   f=(i+1)*N;
   ans(f,1)=-b(f-1,1)-b(f-N,1)-b(f+N,1);
end

for i=1:N-2
   f=i*N;
   for j=1:N-2
      p=f+(j+1);
      ans(p,1)=-b(p+1,1)-b(p+N,1)-b(p-1,1)-b(p-N,1);
   end
end

ans=ans+4*b; %the diagonal
---------------------------------------
```

7.2 Multiplication Poisson matrix with arrayfun

We want to use the code *arrayfun*, so the code above is not useful because *arrayfun* cannot work with the command *b(n,m)*. First start with a simple version of the Poisson matrix. Let matrix B:

$$B = \begin{bmatrix} 4 & -1 & 0 & 0 \\ -1 & 4 & \ddots & 0 \\ 0 & \ddots & \ddots & -1 \\ 0 & 0 & -1 & 4 \end{bmatrix}.$$

NOTE: there are no zeros on $B(i,i+1)$ and $B(i+1,i)$.
Now we make the multiplication $B * b$.

$$B * b = \begin{bmatrix} 4 & -1 & 0 & 0 \\ -1 & 4 & \ddots & 0 \\ 0 & \ddots & \ddots & -1 \\ 0 & 0 & -1 & 4 \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix} = 4 * \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ b_n \end{bmatrix} - \begin{bmatrix} b_2 \\ b_3 \\ \vdots \\ b_n \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-1} \end{bmatrix}. \quad (10)$$

We make the function *VermenigPoison.*

```
---------------------------------------
function UitPois1 = VermenigPoison(b0,b10,b20)

        UitPois1=4*b0-b10-b20;

end
---------------------------------------
```

Now we can use *arrayfun*:

```
---------------------------------------
N=.......;   n=N*N;

b=rand(n,1);   b1=zeros(n,1);   b2=zeros(n,1);

b1(2:n)=b(1:n-1);
b2(1:n-1)=b(2:n);

b0=gpuArray(b);
b10=gpuArray(b1);
b20=gpuArray(b2);
UitPois1=arrayfun(@VermenigPoison,b0,b10,b20);
---------------------------------------
```

We give also the code without using the GPU:

```
---------------------------------------
N=.......;   n=N*N;

b=rand(n,1);   b1=zeros(n,1);   b2=zeros(n,1);

b1(2:n)=b(1:n-1);
b2(1:n-1)=b(2:n);
AnsCPU=4*b-b1-b2;
---------------------------------------
```

We also use the code when we use the GPU, but not using *arrayfun*:

```
---------------------------------------
.......................
b0=gpuArray(b);
b10=gpuArray(b1);
b20=gpuArray(b2);
AnsCPU=4*b0-b10-b20;
---------------------------------------
```

We compare the computing time for running these codes for different values of n. We give the results in table 6.

n	10^3	10^4	10^5	10^6	10^7
Time CPU	0.000007	0.00004	0.00036	0.0037	0.034
Time GPU without arrayfun	0.0028	0.0039	0.0056	0.016	0.12
Time GPU with arrayfun	0.0015	0.0018	0.0040	0.011	0.075

Tabel 6: Time of computing the multiplication of the $n \times n$ matrix B with vector b.

Note: The time corresponding to the GPU is always include the time of copy data.

With $n = 10^8$ we get the comment *"Out of memory on device. You requested: 762.94Mb, device has 53.71Mb free"*.

In this case the code on the CPU is faster. But we see that if n is a larger number, the difference between Time CPU and Time GPU with arrayfun decreases. So when we have a GPU with much more memory, it can be that the code on the GPU is faster. But maybe we can save more time if we extend the calculation.

We will now change matrix B, so that it is more similar to the Poisson matrix. We call the new matrix C.

$$C = \begin{bmatrix} T & -\tilde{I} & 0 \\ -\tilde{I}^T & T & -\tilde{I} \\ 0 & -\tilde{I}^T & T \end{bmatrix}. \tag{11}$$

Here matrix T is the same matrix of (9) and matrix $\tilde{I}$:

$$\tilde{I} = \begin{bmatrix} 1 & 0 & \dots & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ \vdots & 0 & \ddots & \ddots & \vdots \\ 0 & & \ddots & \ddots & 0 \\ 1 & 0 & \dots & 0 & 1 \end{bmatrix}. \tag{12}$$

NOTE: matrix $\tilde{I}$ is not a diagonal matrix.
Matrix C is almost the same matrix as the Poisson matrix A (8), but there are no zeros on $C(i, i+1)$ and $C(i+1, i)$. An example of matrix C:

$$C = \begin{bmatrix} 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 \\ -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 & 0 \\ 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 0 & 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{bmatrix}.$$

Let C be an $n \times n$-matrix, where $n = N \times N$, $N \in \mathbb{N}$.

$$C * b = 4 * \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ b_n \end{bmatrix} - \begin{bmatrix} b_2 \\ b_3 \\ \vdots \\ \vdots \\ \vdots \\ b_n \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ b_1 \\ b_2 \\ \vdots \\ \vdots \\ \vdots \\ b_{n-1} \end{bmatrix} - \begin{bmatrix} b_{N+1} \\ b_{N+2} \\ \vdots \\ b_n \\ 0 \\ \vdots \\ 0 \end{bmatrix} - \begin{bmatrix} 0 \\ \vdots \\ 0 \\ b_1 \\ b_2 \\ \vdots \\ b_{n-N} \end{bmatrix} \tag{13}$$

The code for the CPU:

```
-------------------------------------
n=N*N
b=rand(n,1);
b1=zeros(n,1);   b2=zeros(n,1);   b3=zeros(n,1);   b4=zeros(n,1);

b1(2:n)=b(1:n-1);
b2(1:n-1)=b(2:n);
b3(1:n-N)=b(N+1:n);
b4(N+1:n)=b(1:n-N);

AnsCPU=4*b-b1-b2-b3-b4;
-------------------------------------
```

n	10^2	10^4	10^6	4×10^6
Time CPU	0.0000038	0.000066	0.0049	0.020
Time GPU without arrayfun	0.0035	0.0039	0.020	0.059
Time GPU with arrayfun	0.0022	0.0023	0.018	0.053

Tabel 7: Time of multiplication of the $n \times n$ matrix C with vector b..

The codes for the GPU with and without *arrayfun* are almost the same. We can read the results in table 7. If we look at the results with matrix B, we do not see much difference. MATLAB needs much time to copy the vectors on the GPU.

We will construct the code such that it is the same as a multiplication with the Poisson matrix. Let matrix A be the same matrix as the Poisson matrix A (8). If we change matrix C such that it is the same as matrix A, then we have $C(k{*}N+1, k{*}N) = 0$ and $C(k{*}N, k{*}N+1) = 0$ where $k < n$ and $k \in \mathbb{N}$. If we change two vectors in (13), we obtain the multiplication that we want to have, namely $A * b$.
In the second vector of (13), say vector $\underline{b}_1$, we want that every $(k * N + 1)^{th}$ element is equal to 0, where $k < n$ and $k \in \mathbb{N}$. We also want that in the third vector, say vector $\underline{b}_2$, every $(k * N)^{th}$ element is equal to 0. When we write this in the code:

```
Name code: GPU1
---------------------------------------
..................
b1(2:n)=b(1:n-1);
b2(1:n-1)=b(2:n);
b3(1:n-N)=b(N+1:n);
b4(N+1:n)=b(1:n-N);

for i=1:N-1
    b1(i*N+1,1)=0;
    b2(i*N,1)=0;
end
..................
---------------------------------------
```

For calculating $A * b$, we only need one for loop. When we have multiple matrix vector multiplications, it is convenient to use *arrayfun*. Say,

```
---------------------------------------
for i=1:p
    b=A * b
end
---------------------------------------
```

Vector b is different in every loop, so in every loop we must use the for loop in the code above. Note: we give the notation $A^p * \underline{b}$ for this code.

In *arrayfun* it holds that:

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} * \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} = \begin{bmatrix} a_1 * b_1 \\ a_2 * b_2 \\ \vdots \\ a_n * b_n \end{bmatrix}. \tag{14}$$

We can use this equality to change vectors b_1 and b_2 (see code GPU1). To do this, we first write the next function:

```
---------------------------------------
function UitPois3 = VermenigPoison3(b0,b10,b20,b30,b40,e10,e20)

        UitPois3=4*b0-b10*e10-b20*e20-b30-b40;

end
---------------------------------------
```

Now we have the next code:

```
Name code: GPU2
---------------------------------------
n=N*N;
b0=gpuArray(b);
b10=gpuArray(b1);   b20=gpuArray(b2);   b30=gpuArray(b3);   b40=gpuArray(b4);

e1=ones(n,1);
e2=e1;
for i=1:N-1
    e1(i*N+1,1)=0;
    e2(i*N,1)=0;
end
e1=gpuArray(e1);
e2=gpuArray(e2);

UitPois3=arrayfun(@VermenigPoison3,b0,b10,b20,b30,b40,e1,e2);
---------------------------------------
```

If we multiplicate matrix A with vector b with this code, we do not save time. In case the amount of multiplications is higher, we could get better performance.

We gave two codes a name: GPU1 and GPU2. We are going to compare the speed of these codes. We choose to compute $A^{20} * b$. In the chapters before, we see that we have used 20 as the maximum number of iterations for the Bi-CGSTAB.
We know that Bi-CGSTAB has two matrix-vector multiplications using matrix A in each iteration.

We use a for loop in both codes. In code GPU1, we must change the vectors $b1, b2, b3, b4$ in every loop and also a new vector b. But changing the vectors $b1$ and $b2$ cost more time because of the extra for loop. In the code GPU2 we must also change the vectors $b1, b2, b3, b4$ and b, but we have to construct the vectors $e1$ and $e2$ just once.
Let $timeGPU1$ be the time to run code GPU1 for computing $A^{20} * b$ and let $timeGPU2$ be the time to run code GPU2. Now we have factor $f = \frac{timeGPU1}{timeGPU2}$. We see the results in table (8) of factor f for different sizes of matrix A.

$n(*10^4)$	1	4	9	16	25	36	49	64	81	100
$f = \frac{timeGPU1}{timeGPU2}$	57	62	64	62	50	38	32	28	26	24

Tabel 8: $f = \frac{timeGPU1}{timeGPU2}$ of compute $A^{20} * b$, where A is a $n \times n$ matrix.

We conclude that code GPU2 is much faster than code GPU1. From table (8) we could conclude that the time difference between running GPU1 and GPU2 decreases, because the factor decreases. Also for large values of n the code GPU2 is much faster.

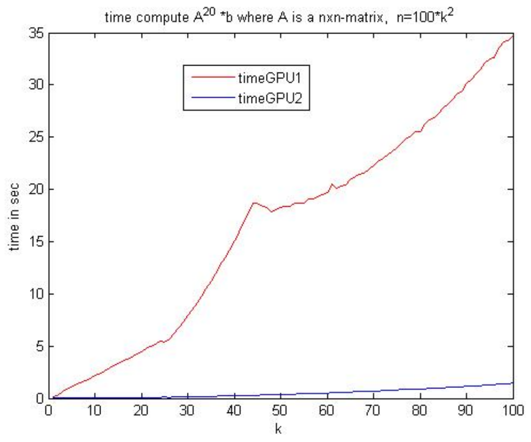

Figure 5: Time code GPU1 and GPU2 with different sizes of matrix A.

Now we are also going to look at the difference between runtimes of code GPU1 and code GPU2 when size of matrix A remains the same, but the number of iterations is changed. Let k be the number of iterations and factor $f = \frac{timeGPU1}{timeGPU2}$. Let A be a $10^4 \times 10^4$-matrix. In table (9) we can see the results. We see that factor f stays more or less constant and remains at about 60. In figure (6) we see the time of code GPU1 and code GPU2 and we see that code GPU2 is much faster.

k	10	20	30	40	50	60	70	80	90	100
$f = \frac{timeGPU1}{timeGPU2}$	54	56	57	58	58	59	59	57	59	59

Tabel 9: $f = \frac{timeGPU1}{timeGPU2}$ of compute $A^k * b$, where A is a $10^4 \times 10^4$ matrix.

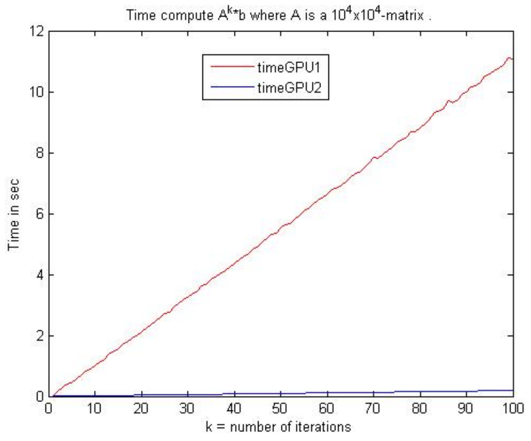

Figure 6: Time code GPU1 and GPU2 for different numbers of iterations.

We can conclude that code GPU2 is faster than code GPU1. So from now on we use code GPU2.

7.3 Computing matrix-vector multiplication without using the GPU

We want to compute a matrix-vector multiplication $A^k * b$, where A is a Poisson matrix and $k \in \mathbb{N}$. Of course we use the for loop instead of the code $A^k * b$, because of the number of operations.

```
name code: type2
---------------------------------------
A = gallery('poisson',N);
for i=1:k
    b=A*b;
end
---------------------------------------
```

Note: we give the notation $A^k * \underline{b}$ for this code.

We are going to compare the iteration time of a couple of codes. The first code we are going to use is a code that looks like the code GPU1, but without using the GPU.

```
name code: CPU2
---------------------------------------
n=N*N; %size of the matrix
k %number of iterations

b=rand(n,1);   b1=zeros(n,1);   b2=zeros(n,1);   b3=zeros(n,1);   b4=zeros(n,1);

b1(2:n)=b(1:n-1);
b2(1:n-1)=b(2:n);
b3(1:n-N)=b(N+1:n);
b4(N+1:n)=b(1:n-N);

for i=1:N-1
    b1(i*N+1,1)=0;
    b2(i*N,1)=0;
end

for i=1:k
    b=4*b-b1-b2-b3-b4;
    if i<k
        b1(2:n)=b(1:n-1);
        b2(1:n-1)=b(2:n);
        b3(1:n-N)=b(N+1:n);
        b4(N+1:n)=b(1:n-N);
           for i=1:N-1
               b1(i*N+1,1)=0;
               b2(i*N,1)=0;
           end
    end
end
---------------------------------------
```

At the beginning of this chapter we gave a code with a lot of for loops, named code CPU1. Before we compare the time iteration of different types of codes, we must know which of these two codes is the fastest. First we look at the time difference between running code CPU1 and code CPU2 when the size of matrix A remains the same, but the number of iterations is changed. We see the results in table (7).

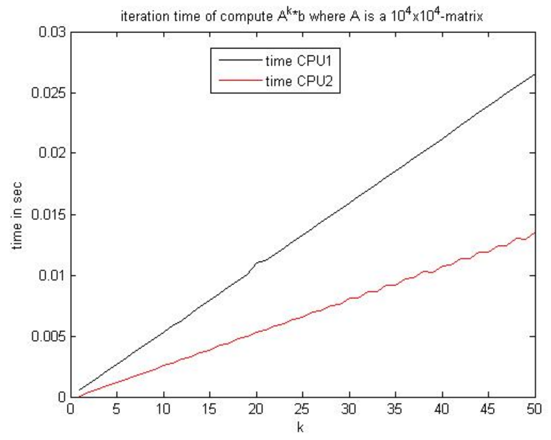

Figure 7: Time code CPU1 and CPU2 for different numbers of iterations.

These results give us the idea that code CPU2 is a better choice. However, it is more important to look at different sizes of matrix A than the high number of iterations. So we are now going to look at the runtimes, with the number of iterations fixed on $k = 20$ and an increasing size of matrix A. In figure (8) we see the results.

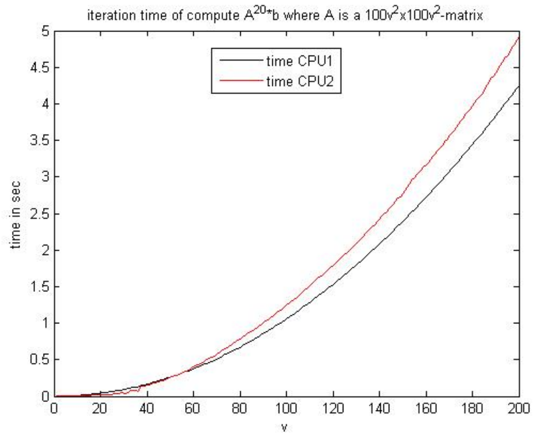

Figure 8: Time code CPU1 and CPU2 for different sizes of matrix A.

We conclude from this figure, that code CPU1 is faster when the size of matrix A increases. We want use this codes for large matrices, so we choose to use code CPU1 instead of CPU2.

7.4 Compare different ways of a matrix-vector multiplication with a Poisson matrix

The three codes we compare to compute $A^k * b$ are: *type2, CPU1* and *GPU2*. In figure (9), (10), (11), (12) en (13) we can see the results in case we compute 1, 5, 10, 20 or 30 iterations respectively. Poisson matrix A is a $100v^2 \times 100v^2$-matrix.

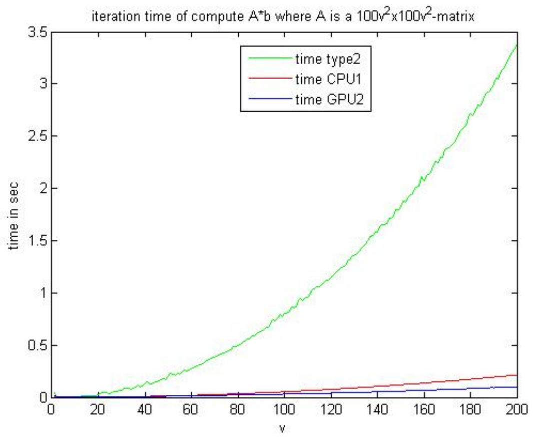

Figure 9: Time code type2, CPU1 and GPU2 for different sizes of matrix A, computing $A * b$

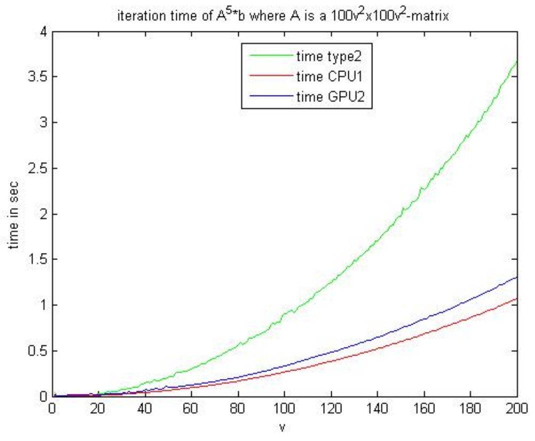

Figure 10: Time code type2, CPU1 and GPU2 for different sizes of matrix A, computing $A^5 * b$

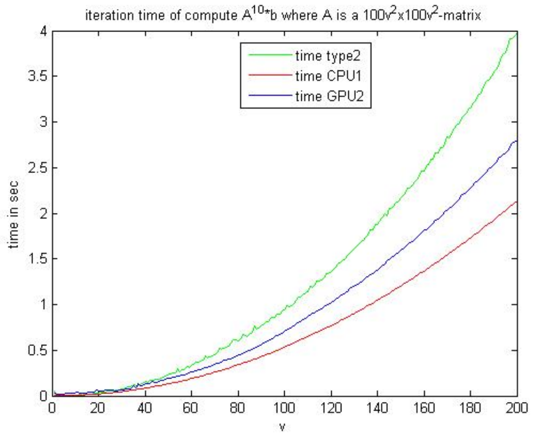

Figure 11: Time code type2, CPU1 and GPU2 for different sizes of matrix A, computing $A^{10} * b$

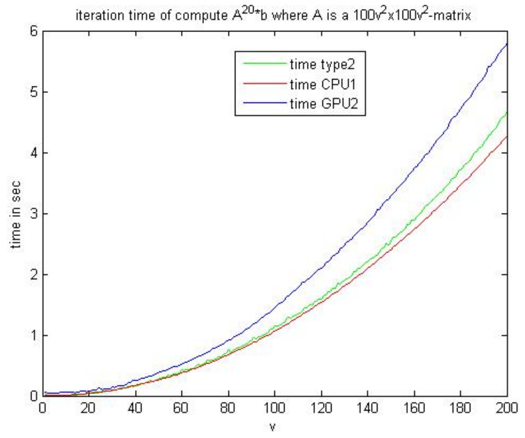

Figure 12: Time code type2, CPU1 and GPU2 for different sizes of matrix A, computing $A^{20} * b$

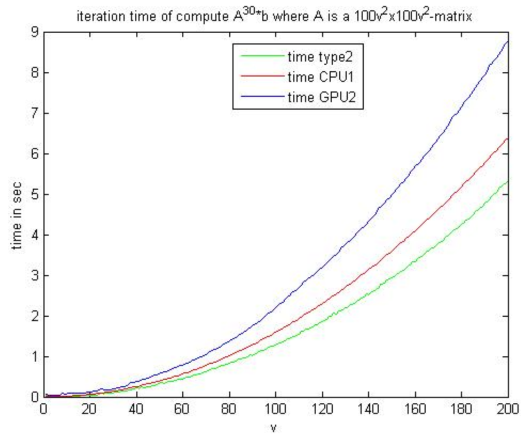

Figure 13: Time code type2, CPU1 and GPU2 for different sizes of matrix A, computing $A^{30} * b$

Code GPU2 gives the best results for low amounts of iterations. Code CPU1 is the fastest after 20 iterations, but it does not differ much with code type2. When we have 30 iterations, code type2 is the fastest.

In figure (14) we give the results when we let the Poisson matrix A be a $10^6 \times 10^6$-matrix and use diffent number of iterations. Code GPU2 is slower than code CPU1 after 4 or 5 iterations. We also see that code type2 is faster than code CPU1 after 20 iterations.

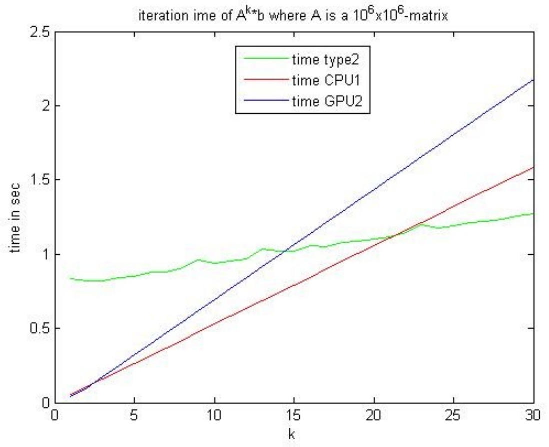

Figure 14: Time code type2, CPU1 and GPU2 for different number of iterations, where Poisson matrix A is a $10^6 \times 10^6$-matrix, computing $A^{30} * b$

In the next chapter we use the codes in a Bi-CGSTAB code. It seems reasonable that we use the code CPU1 in case we have more than 5 iterations. However, when we integrate it into a different code, the results can be different.

8 Parallel Programming of the Bi-CGSTAB with the Poisson matrix

We want to compute the system $A\underline{x} = \underline{b}$ with the Bi-CGSTAB method. In Chapter 5 matrix A is the Poisson matrix. We have seen that the MATLAB code *x=bicgstab(A,b)* is faster than our code, whether we use CPU or GPU. In this chapter we are going to use the structure of the Poisson matrix to speed up our code on the CPU or GPU. We know that there are better and faster algorithms for computing the system $A\underline{x} = \underline{b}$ if A is a Poisson matrix, but for a power flow problem we used the Bi-CGSTAB algorithm. For the moment we do not know if there is a structure in the admittance matrix Y. This is the reason that we use the algorithm Bi-CGSTAB and also use the preconditioners: diagonal scaling predonditioner (DSP) and parallel incomplete decomposition preconditioner (IDP). As in Chapter 7 the Poisson matrix A is scaled, so we do not write the factor $\frac{1}{h^2}$.

8.1 Bi-CGSTAB with the Poisson matrix and the diagonal scaling predonditioner

We give the description of the Bi-CGSTAB method in Chapter 4, but for convenience we state it again:

Bi-CGSTAB method

```
$u^0$ is an initial guess; $r^0 = b - Au^0$;
$\overline{r}^0$ is an arbitrary vector, such that $(\overline{r}^0, r^0) \neq 0$, e.g., $\overline{r}^0 = r^0$;
$\rho_{-1} = \alpha_{-1} = \omega_{-1} = 1$;
$v^{-1} = p^{-1} = 0$;
for $i = 0, 1, 2, ...$ do
    $\rho_i = (\overline{r}^0, r^i)$; $\beta_{i-1} = (\rho_i/\rho_{i-1})(\alpha_{i-1}/\omega_{i-1})$;
    $p^i = r^i + \beta_{i-1}(p^{i-1} - \omega_{i-1}v^{i-1})$;
    $\hat{p} = M^{-1}p^i$;
    $v^i = A\hat{p}$;
    $\alpha_i = \rho_i/(\overline{r}^0, v^i)$;
    $s = r^i - \alpha_i v^i$;
    if $||s||$ small enough then
        $u^{i+1} = u^i + \alpha_i\hat{p}$; quit;
    $z = M^{-1}s$;
    $t = Az$;
    $\omega_i = (t, s)/(t, t)$;
    $u^{i+1} = u^i + \alpha_i\hat{p} + \omega_i z$;
    if $u^{i+1}$ is accurate enough then quit;
    $r^{i+1} = s - \omega_i t$;
end for
```

This method uses a matrix-vector multiplication three times, where A is a Poisson matrix. For this multiplication we can use the results of Chapter 7.

8.1.1 Substantiation code

In the description we see also two matrix-vector multiplications with matrix M^{-1}. We know that matrix M is the preconditioner. Here, matrix M is a DSP and has the form:

$$M = \begin{bmatrix} 4 & & \Theta \\ & \ddots & \\ \Theta & & 4 \end{bmatrix} \tag{15}$$

So matrix M^{-1} has the form:

$$M^{-1} = \begin{bmatrix} \frac{1}{4} & & \Theta \\ & \ddots & \\ \Theta & & \frac{1}{4} \end{bmatrix} \tag{16}$$

Seeing the structure of matrix M^{-1}, we can say:

$$M^{-1} * \underline{a} = \frac{1}{4} * \underline{a}.$$

The parts of the code where we do not use GPU, should now be changed such that it can be used by the GPU.

We must also compute dotproducts, like (t, s). In MATLAB we can use the code c=dot(t,s), but we can also compute this with the code *c=t'*s*. We will first find out which command computes the dotproduct faster. We see the results in figure (15).

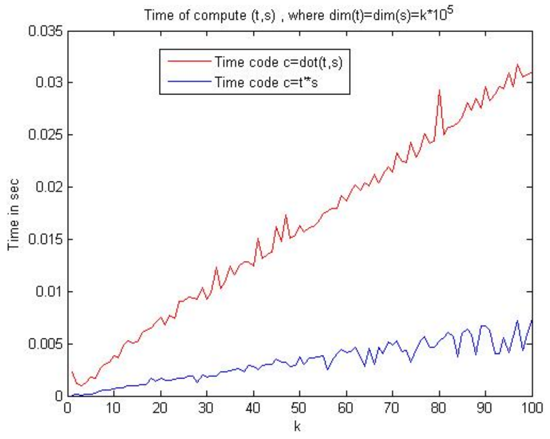

Figure 15: Time code by using type2, CPU1 and GPU2 for different sizes of matrix A, computing $A^{30} * b$

We see that the command *c=t'*s* is faster than the MATLAB function *c=dot(t,s).* So we use the command *c=t'*s* to compute the dotproduct.

Then we make function *PoissonAwithCPU*. In Appendix E can we read the whole code and also the associated functions.

Now we have everyting for the code without using the GPU and using the codes *type2* and *CPU1*. For the part with the GPU we have other functions. We make function PoissonAwithGPU, where we use the code GPU2. This function can be found in Appendix E. We use functions *Vector1, Vector2* and *Vector3*. These are small codes where we compute simple vector calculations and where we use arrayfun. These codes can also be found in Appendix E.

Let b' the transpose vector of vector b. Now we are going to look what is the fastest code to compute the dotproduct with the GPU. We can use three codes: using command $c = b' * d$, using an arrayfun code or using *c=dot(b,d)*. Let vetor $\underline{b}$ and $\underline{d}$ be two vecors on the GPU, *b=gpuArray(b); d=gpuArray(d);*. The codes $b' * d$ and *dot(b,d)* are straightforward. We give the function DotPro, which can be used with arrayfun code:

```
Function DotPro
---------------------------------------
function DotPro = DotProduct(b0,d0)
        DotPro=b0*d0;
end
---------------------------------------
```

Note: in this function we write $b0 * d0$ and not $b0' * d0$, because we want to use *arrayfun*. The explanation can be found in section 7.2, see equation (14).

```
code dotproduct with arrayfun
---------------------------------------
dotBD=arrayfun(@DotProduct,b,d);
x=sum(dotBD);
---------------------------------------
```

In Figure (16) can we see the results. It is clear that code $b' * d$ is the slowest. Figure (17) gives the results without the code $b' * d$.

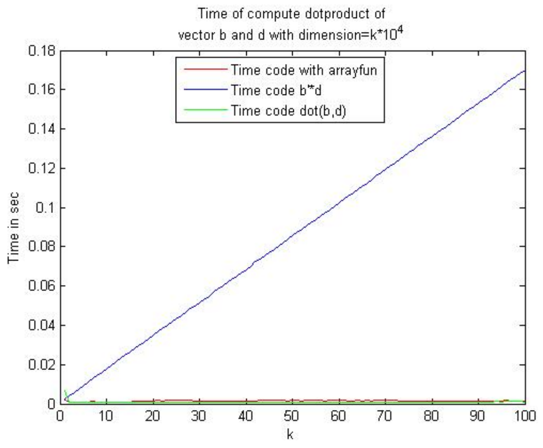

Figure 16: Time computing, on the GPU, dotproduct of vectors $\underline{b}$ and $\underline{d}$ with different dimensions.

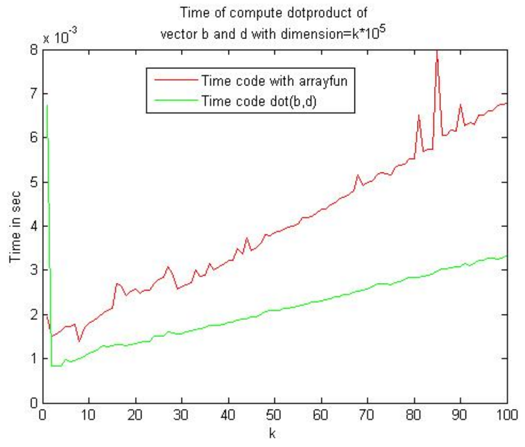

Figure 17: Time computing, on the GPU, dotproduct of vectors b and d with different dimensions.

We conclude that the code *dot(b,d)* is the fastest.

8.1.2 Results

We compare four codes:

- the MATLAB code *x=bicgstab(A,b)*,
- the Bi-CGSTAB code where we use code CPU1,
- the Bi-CGSTAB code where we use code type2 (CPU),
- the Bi-CGSTAB code where we use the GPU.

We want to make a fair comparison, so every code has the same number of iterations. We start with the code *x=bicgstab(A,b)*. For computing the number of iterations, we write the next code:

```
---------------------------------------
[x,flag,relres,iter]=bicgstab(A,b);
k=iter
---------------------------------------
```

Here, k is the number of iterations of the code *x=bicgstab(A,b)*. For the other codes we will use the same number of iterations.

Now we look which code is faster for different sizes of Poisson matrix A. Figure (18) shows the results.

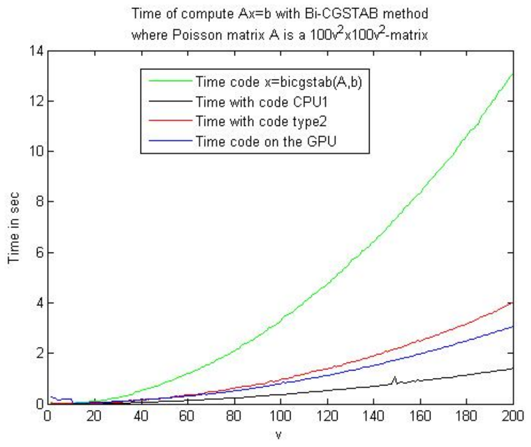

Figure 18: Time of computing the system $A\underline{x} = b$ with the Bi-CGSTAB with different sizes of Poisson matrix A.

We see that every code is faster than the MATLAB code *x=bicgstab(A,b)* when the size of the Poisson matrix A increases. However, we can not yet make this conclusion. Note that v says something about the size of the matix A (see Figure (18)). If we look at the number of iterations, we see that the code *x=bicgstab(A,b)* needs 20 iterations when $v \leq 10$, but when $v > 10$ the code uses 1 iteration (A is a $100v^2 \times 100v^2$-matrix). I do not know the reason of this choice. We let the number of iterations of the other codes equal to the number of iterations of code *x=bicgstab(A,b)*. We can use the relative residual to say it is a fair comparison.

8.1.3 Relative Residual

Suppose $\underline{\widetilde{x}}$ is an approximation of the solution of the linear system defined by $A\underline{x} = \underline{b}$. The relative residual is:

$$\epsilon = \frac{||\underline{b} - A\underline{\widetilde{x}}||}{||\underline{b}||}$$

Except for the code *x=bicgstab(A,b)*, the ϵ of the codes are the same. In figure (19) we can see the difference of the values of the relative residual ϵ.

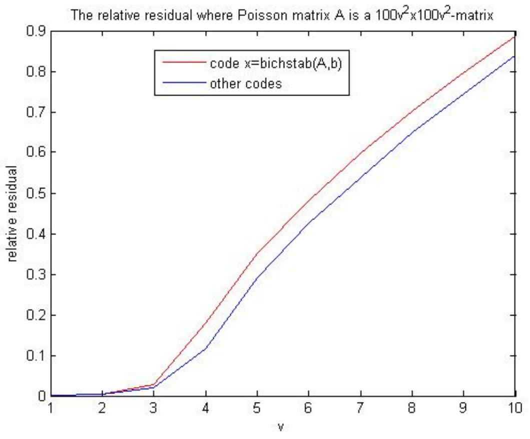

Figure 19: The relative residual of different sizes of Poisson matrix A.

We see that the relative residual of the other codes is lower than the relative residual of the MATLAB code *x=bicgstab(A,b)*. With this result and figure (18) we can conclude that our code is faster than the MATLAB code *x=bicgstab(A,b)*. We see also that the relative residual increase till 1. This means that $\widetilde{\underline{x}}$ is not a good approximation of the linear system $A\underline{x} = \underline{b}$ when the size of Poisson matrix A increase. In figure (20) we see what is happend if v increase.

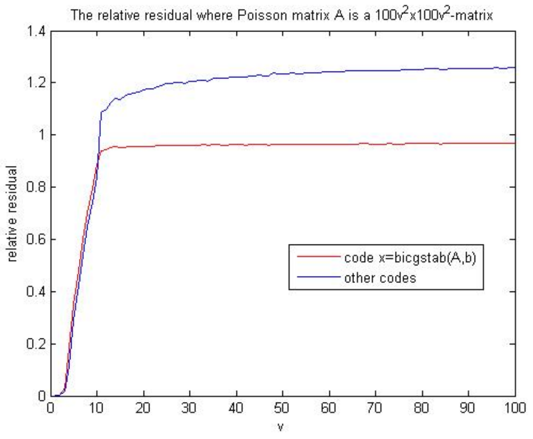

Figure 20: The relative residual of different sizes of Poisson matrix A.

The relative residual of code *x=bicgstab(A,b)* is close to 1 when v increase, but the relative residual of the other codes is higher than 1. This is a strange and undesirable result. We do not know why both methods gives this bad results for large sizes of Poisson matrix A.

8.2 Bi-CGSTAB with the Poisson matrix and the incomplete decomposition preconditioner

The code that we use in this section is almost the same as in the previous section, but now we use another preconditioner M. In section 8.1, M (the DSP) has a convenient structure. We can make a scalar vector multiplication for computing $M^{-1}\underline{v}$. In this section, where we use an IDP, the structure of matrix M is not like that. Section 4.3.3 describes how we find the IDP. In the Bi-CGSTAB method we need matrix M^{-1}. In case we use the ISP, the matrix M^{-1} is a full matrix. So we can not compute $M^{-1}\underline{v}$ as a scalar vector multiplication, but we must use the whole matrix M^{-1}. The problem is that a matrix-vector multiplication costs a lot of time. Appendix F describes the function *IPvanPoissonGPU* where we compute the matrix M^{-1}. In the first part of the code of the function we do not use the GPU, because it is not a parallel computation. After the matrices L, D and $invD$ have been computed, we use the GPU.

In figure (21) can we see the results.The time of the code on the GPU is much larger then the other codes. We can conclude that it is not a good idea use the GPU when we use the IDP.

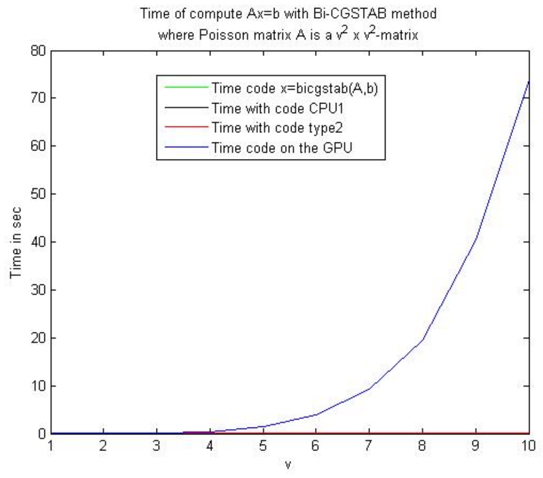

Figure 21: Time of compute the system $A\underline{x} = b$ with the Bi-CGSTAB with different sizes of Poisson matrix A, use the IDP.

Figure (22) shows all the results, except the results for the code on the GPU.

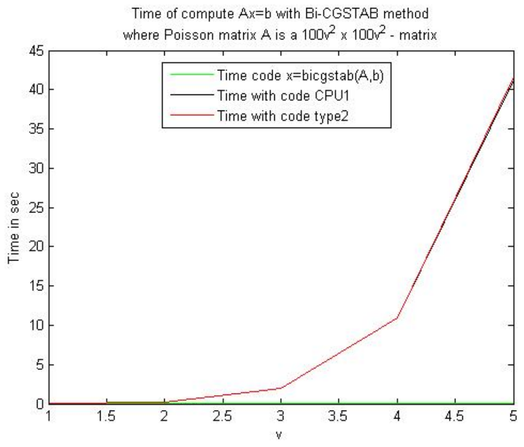

Figure 22: Time of compute the system $A\underline{x} = b$ with the Bi-CGSTAB with different sizes of Poisson matrix A, use the IDP.

We see that code *x=bicgstab(A,b)* is much faster. The other codes have also one iteration and the relative residual is not lower. So using the IDP is not an option when we work in MATLAB.

9 Conclusion and Discussion

This document can be separated into two topics: solve a load flow problem with MATLAB by using the GPU and speed up the code for solve the linear system $A\underline{x} = \underline{b}$ with Bi-CGSTAB in MATLAB where A is the Poisson matrix. First we discuss the results of the part about the load flow problem. After that we discuss the second part.

In chapter 2 and 3 we describe what is a load flow problem and how it can be solved. This subject was part of my literature review, where an important source was the document of R. Idema [3]. These chapters were especially background information, because there is a program Matpower that gives us the admittance matrices. In chapter 4 we give two iterative methods and three preconditioners, but we made the choice to choose the Bi-CGSTAB method and use two preconditioners. In chapter 5 we start programming on the GPU with MATLAB, first with the Poisson matrix. We saw that the code be slower when we use full matrices, especially when we use the GPU. In chapter 5 can one read that MATLAB works fine with sparse matrices, but when we put this sparse matrix on the GPU, MATLAB can only work with it when it is a full matrix (with a lot of zeros). The other problem was that the admittance matrix does not have a convenient structure, like the Poisson matrix. So we conclude in chapter 6 that it gives us not the speed up we want. We have not found a fast code in MATLAB by using the GPU, because of the full matrices. This was the reason of the second part: can we speed up the code if matrix A has a convenient structure.

In chapter 7 we start with the structure of a Poisson matrix A. We have seen that the tool *arrayfun* can give us a lot of speed up, dependent on the type of the calculation. The problem is that *arrayfun* does not work well with matrices and the standard vector multiplication is not supported. So we must find tricks to remedy this. By these tricks the code has not been faster. Nevertheless, we succeeded to make the Bi-CGSTAB code faster than the MATLAB code *x=bicgstab(A,b)*, but only when we use a DSP. This preconditioner can be translated to a vector seeing the structure of the matrix. When we use the IDP, then we have again the problem that we must use a full matrix. But when we use an IDP, our codes were faster, but the code with the CPU was faster than the code with the GPU. So we have to find a faster code, but using the GPU in MATLAB does not give us any speed up.

We can not conlude that using the GPU in MATLAB is senseless. In our method we have the situation that we work with matrices and multiple vectors. This costs memory and time because of the placing of vectors to the GPU. In other problems and methods it is possible that using the GPU gives time savings, see the easy example in chapter 5. But for solving the linear system $A\underline{x} = \underline{b}$ we think that using the GPU in MATLAB in general is not a good idea. The disadvantage of using MATLAB is that we do not exactly know how de code *x=bicgstab(A,b)* in MATLAB works, and how MATLAB moved vectors on the GPU.

There is a command in MATLAB, *parallel.gpu.CUDAkernel(-,-)*, which can operate on MATLAB array or gpuArray variables. To use this code, you should still program in CUDA. In this case, using MATLAB is then unnecessary.
If you really want more speed up, you can read it in a lot of literature, program in CUDA (programming in C) and not using MATLAB.

A function IPvanPoisson

```
---------------------------------------
function [A,L,invL,invM,n]=IPvanPoisson(N);
A = gallery('poisson',N);
L=A;    n=N*N;
for p=1:n
   L(p,p)=sqrt(L(p,p));
   for i=p+1:n
    if L(i,p)~=0
        L(i,p)=L(i,p)/L(p,p);
    end
   end
   for j=p+1:n
      for i=j:n
          if L(i,j)~=0
              L(i,j)=L(i,j)-L(i,p)*L(j,p);
          end
      end
   end
end

for i=1:n
   for j=i+1:n
       L(i,j)=0;
    end
end
D=0*L;
invD=0*L;
for i=1:n
    D(i,i)=L(i,i);
    invD(i,i)=1/L(i,i);
end

N=invD*(L-D);
I=eye(n);
L=D*(I+N);
T=I+N;
invT=zeros(n);
for i=0:3
     invT=invT + (-1)^i*(N)^i;
end
invL=invT*invD;
invM=invL'*invL;
---------------------------------------
```

B
Bi-CGSTAB with IP on the CPU and GPU

```
--------------------------------------
s=rng('default');
format long;
aantaltell=7

TijdDirect=zeros(aantaltell,1);
TijdmetIP=zeros(aantaltell,1);
TijdmetIPGPU=zeros(aantaltell,1);

for tell=1:aantaltell
N=tell;
%With MATLAB code bicgstab
tic
A = gallery('poisson',N);
n=N*N;
k=3;
b=ones(n,1);
for i=1:n
    b(i,1)=b(i,1)*rand;
end
x=bicgstab(A,b);
tijddirect=toc;
error=norm(A*x-b,2)/norm(x,2);
if error == Inf
   error = 10^-8
end
%code without GPU

[A,L,invL,invM,n]=IPvanPoisson(N);
tic
u0=rand*ones(n,1);
r0=b-A*u0;
r_0=r0;
c=0;
for i=1:n
   c=c+r_0(i,1)*r0(i,1); %inner product (r_0,r0)
end

RowiMin1=1;
AlphaiMin1=1;
OhmiMin1=1;
viMin1=zeros(n,1);
piMin1=zeros(n,1);
```

```
rowi=1;
alphai=1;
ohmi=1;
vi=zeros(n,1);
pi=zeros(n,1);
ri=r0;
ui=u0;
errorCPU=1000000;
k=0;

while errorCPU>=error && k<20
    k=k+1;
    rowi=dot(r_0,ri);
    betaiMin1=(rowi/RowiMin1)*(AlphaiMin1/OhmiMin1);
    pi=ri+betaiMin1*(piMin1-OhmiMin1*viMin1);
    pdak=invM*pi;
    vi=A*pdak;
    alphai=rowi/dot(r_0,vi);
    s=ri-alphai*vi;
    z=invM*s;
    t=A*z;
    ohmi=dot(t,s)/dot(t,t);
    ui=ui+alphai*pdak+ohmi*z;
    ri=s-ohmi*t;
    RowiMin1=rowi;
    AlphaiMin1=alphai;
    OhmiMin1=ohmi;
    viMin1=vi;
    piMin1=pi;
    errorCPU=norm(A*ui-b,2)/norm(b,2);
end
CPUui=ui;
tijdmetIP=toc;


%code with GPU
[A,L,invL,invM,n]=IPvanPoissonGPU(N);
tic
u0=rand*ones(n,1);
r0=b-A*u0;
r_0=r0;
c=0;
for i=1:n
   c=c+r_0(i,1)*r0(i,1); %inner product (r_0,r0)
end
```

```
RowiMin1=1;
AlphaiMin1=1;
OhmiMin1=1;
viMin1=zeros(n,1);
piMin1=zeros(n,1);
rowi=1;
alphai=1;
ohmi=1;
vi=zeros(n,1);
pi=zeros(n,1);
ri=r0;
ui=u0;
errorGPU=1000000;
k=0;

while errorGPU>=error && k<20
    k=k+1;
    rowi=dot(r_0,ri);
    betaiMin1=(rowi/RowiMin1)*(AlphaiMin1/OhmiMin1);
    pi=ri+betaiMin1*(piMin1-OhmiMin1*viMin1);
    pdak=invM*pi;
    vi=A*pdak;
    alphai=rowi/dot(r_0,vi);
    s=ri-alphai*vi;
    z=invM*s;
    t=A*z;
    ohmi=dot(t,s)/dot(t,t);
    ui=ui+alphai*pdak+ohmi*z;
    ri=s-ohmi*t;
    RowiMin1=rowi;
    AlphaiMin1=alphai;
    OhmiMin1=ohmi;
    viMin1=vi;
    piMin1=pi;
    errorGPU=norm(A*ui-b,2)/norm(b,2);
end
GPUui=ui;
tijdmetIPGPU=toc;
end

plot(TijdDirect,'r')
hold on
plot(TijdmetIP,'g')
hold on
plot(TijdmetIPGPU,'b')
---------------------------------------
```

C Bi-CGSTAB with diagonal scaling on the CPU and GPU

```
----------------------------------------
s=rng('default');
format long;
aantaltell=7;
TijdDirect=zeros(aantaltell,1);
TijdmetDP=zeros(aantaltell,1);
TijdmetDPGPU=zeros(aantaltell,1);
for tell=2:aantaltell
N=tell;
%With MATLAB code bicgstab
tic
A = gallery('poisson',N);
n=N*N;
k=3;
b=ones(n,1);
for i=1:n
    b(i,1)=b(i,1)*rand;
end
x=bicgstab(A,b);
tijddirect=toc;
error=norm(A*x-b,2)/norm(x,2);
if error == Inf
   error = 10^-8
end
%code without GPU
invM=A;
for i=1:n
     invM(i,i)=1/A(i,i);
end
tic
u0=rand*ones(n,1);
r0=b-A*u0;
r_0=r0;
c=0;
for i=1:n
   c=c+r_0(i,1)*r0(i,1);
end
RowiMin1=1;
AlphaiMin1=1;
OhmiMin1=1;
viMin1=zeros(n,1);
piMin1=zeros(n,1);
rowi=1;
```

```
alphai=1;
ohmi=1;
vi=zeros(n,1);
pi=zeros(n,1);
ri=r0;
ui=u0;
errorCPU=1000000;
k=0;
while errorCPU>=error && k<20
    k=k+1;
    rowi=dot(r_0,ri);
    betaiMin1=(rowi/RowiMin1)*(AlphaiMin1/OhmiMin1);
    pi=ri+betaiMin1*(piMin1-OhmiMin1*viMin1);
    pdak=invM*pi;
    vi=A*pdak;
    alphai=rowi/dot(r_0,vi);
    s=ri-alphai*vi;
    z=invM*s;
    t=A*z;
    ohmi=dot(t,s)/dot(t,t);
    ui=ui+alphai*pdak+ohmi*z;
    ri=s-ohmi*t;
    RowiMin1=rowi;
    AlphaiMin1=alphai;
    OhmiMin1=ohmi;
    viMin1=vi;
    piMin1=pi;
    errorCPU=norm(A*ui-b,2)/norm(b,2);
end
CPUui=ui;
tijdmetDP=toc;
%code with GPU
A=zeros(n)+A;   %here we make a full matrix
A=gpuArray(A);
invM=gpuArray(zeros(n));
for i=1:n
     invM(i,i)=1/A(i,i);
end
b=gpuArray(b); %place the vector on the GPU
tic
u0=gpuArray(rand*ones(n,1));
r0=b-A*u0;
r_0=r0;
c=0;
for i=1:n
   c=c+r_0(i,1)*r0(i,1);
end
```

```
RowiMin1=1;
AlphaiMin1=1;
OhmiMin1=1;
viMin1=gpuArray(zeros(n,1));
piMin1=gpuArray(zeros(n,1));
rowi=1;
alphai=1;
ohmi=1;
vi=gpuArray(zeros(n,1));
pi=gpuArray(zeros(n,1));
ri=r0;
ui=u0;
errorGPU=1000000;
k=0;
while errorGPU>=error && k<20
    k=k+1;
    rowi=dot(r_0,ri);
    betaiMin1=(rowi/RowiMin1)*(AlphaiMin1/OhmiMin1);
    pi=ri+betaiMin1*(piMin1-OhmiMin1*viMin1);
    pdak=invM*pi;
    vi=A*pdak;
    alphai=rowi/dot(r_0,vi);
    s=ri-alphai*vi;
    z=invM*s;
    t=A*z;
    ohmi=dot(t,s)/dot(t,t);
    ui=ui+alphai*pdak+ohmi*z;
    ri=s-ohmi*t;
    RowiMin1=rowi;
    AlphaiMin1=alphai;
    OhmiMin1=ohmi;
    viMin1=vi;
    piMin1=pi;
    errorGPU=norm(A*ui-b,2)/norm(b,2);
end
GPUui=ui;
tijdmetDPGPU=toc;
TijdDirect(tell,1)=tijddirect;
TijdmetDP(tell,1)=tijdmetDP;
TijdmetDPGPU(tell,1)=tijdmetDPGPU;
end
plot(TijdDirect,'r')
hold on
plot(TijdmetIP,'g')
hold on
plot(TijdmetIPGPU,'b')
---------------------------------------
```

D Matpower en Bi-CGSTAB

```
--------------------------------------
clear all;
close all;
clc
format long;
% Functions below come from "runpf()"
n=10;
mpc = loadcase('case10ac');  %mpc = loadcase('case3120sp');  %mpc = loadcase('case30Q');
%mpc = loadcase('case39'); %mpc = loadcase('case118');

% Make admittance matrix
[Y,~,~] = makeYbus(mpc.baseMVA, mpc.bus, mpc.branch);
A=zeros(n);
for i=1:n
   for j=1:n
        A(i,j)=Y(i,j);
   end
end
%With MATLAB code bicgstab
tic
b=rand(n,1);
x=bicgstab(A,b);
tijddirect=toc;
%code without GPU
[A,L,invL,invM]=aaaIPvanCPU(A,n);
tic
u0=rand*ones(n,1);
U0=u0;
r0=b-A*u0;
r_0=r0;
c=0;
for i=1:n
   c=c+r_0(i,1)*r0(i,1);
end
RowiMin1=1;
AlphaiMin1=1;
OhmiMin1=1;
viMin1=zeros(n,1);
piMin1=zeros(n,1);
rowi=1;
alphai=1;
ohmi=1;
vi=zeros(n,1);
```

```
pi=zeros(n,1);
ri=r0;
ui=u0;
k=0;
while k<20
    k=k+1;
    rowi=dot(r_0,ri);
    betaiMin1=(rowi/RowiMin1)*(AlphaiMin1/OhmiMin1);
    pi=ri+betaiMin1*(piMin1-OhmiMin1*viMin1);
    pdak=invM*pi;
    vi=A*pdak;
    alphai=rowi/dot(r_0,vi);
    s=ri-alphai*vi;
    z=invM*s;
    t=A*z;
    ohmi=dot(t,s)/dot(t,t);
    ui=ui+alphai*pdak+ohmi*z;
    ri=s-ohmi*t;
    RowiMin1=rowi;
    AlphaiMin1=alphai;
    OhmiMin1=ohmi;
    viMin1=vi;
    piMin1=pi;
end
tijdmetIP=toc;
%code with GPU
tic
u0=U0;
r0=b-A*u0;
r_0=r0;
c=0;
for i=1:n
   c=c+r_0(i,1)*r0(i,1);
end
RowiMin1=1;
AlphaiMin1=1;
OhmiMin1=1;
viMin1=zeros(n,1);
piMin1=zeros(n,1);
rowi=1;
alphai=1;
ohmi=1;
vi=zeros(n,1);
pi=zeros(n,1);
ri=r0;
ui=u0;
k=0;
```

```
while k<20
    k=k+1;
    rowi=dot(r_0,ri);
    betaiMin1=(rowi/RowiMin1)*(AlphaiMin1/OhmiMin1);
    pi=ri+betaiMin1*(piMin1-OhmiMin1*viMin1);
    pdak=invM*pi;
    vi=A*pdak;
    alphai=rowi/dot(r_0,vi);
    s=ri-alphai*vi;
    z=invM*s;
    t=A*z;
    ohmi=dot(t,s)/dot(t,t);
    ui=ui+alphai*pdak+ohmi*z;
    ri=s-ohmi*t;
    RowiMin1=rowi;
    AlphaiMin1=alphai;
    OhmiMin1=ohmi;
    viMin1=vi;
    piMin1=pi;
end
tijdmetIPGPU=toc;
---------------------------------------
```

E
Poisson matrix en Bi-CGSTAB with DSP, with functions

```
Function PoissonAwithCPU
---------------------------------------
function AkeerZcpu = PoissonAwithCPU(N,c)
    d=c;
    n=N*N;
    c(1,1)=-d(2,1)-d(N+1,1);
    c(n,1)=-d(n-1,1)-d(n-N,1);
    c(N,1)=-d(N-1,1)-d(N+N,1);
    c(n-(N-1),1)=-d(n-(N-1)+1,1)-d(n-(N-1)-N,1);

    for i=2:N-1
       c(i,1)=-d(i-1,1)-d(i+1,1)-d(i+N,1);
    end

    for i=N*N-(N-2):n-1;
       c(i,1)=-d(i-1,1)-d(i+1,1)-d(i-N,1);
    end

    for i=1:N-2
        f=i*N+1;
        c(f,1)=-d(f+1,1)-d(f+N,1)-d(f-N,1);
    end

    for i=1:N-2
       f=(i+1)*N;
       c(f,1)=-d(f-1,1)-d(f-N,1)-d(f+N,1);
    end

    for i=1:N-2
       f=i*N;
       for j=1:N-2
          p=f+(j+1);
          c(p,1)=-d(p+1,1)-d(p+N,1)-d(p-1,1)-d(p-N,1);
       end
    end

    AkeerZcpu=c+4*d;
end
---------------------------------------

Function PoissonAwithGPU
---------------------------------------
function AkeerZgpu = PoissonAwithGPU(N,b00,e1,e2)
    n=N*N;
    b10=0*b00;    b20=0*b00;    b30=0*b00;    b40=0*b00;

    b10(2:n)=b00(1:n-1);
    b20(1:n-1)=b00(2:n);
    b30(1:n-N)=b00(N+1:n);
    b40(N+1:n)=b00(1:n-N);

    AkeerZgpu=arrayfun(@VermenigPoison3,b00,b10,b20,b30,b40,e1,e2);
 end
---------------------------------------
```

Function Vector1

```
---------------------------------------
function V1 = Vector1(a0,g,b0)

        V1=a0-g*b0;
end
---------------------------------------
```

Function Vector2

```
---------------------------------------
function V2 = Vector2(a0,g1,b0,g2,c0)

        v1=g1*b0;
        v2=g2*c0;
        V2=a0+v1+v2;
end
---------------------------------------
```

Function Vector3

```
---------------------------------------
function V3 = Vector3(a0,g1,b0,g2,c0)

        v=b0-g2*c0
        V3=a0+g1*v;
end
---------------------------------------
```

Code Bi-CGSTAB with the Poisson matrix and DSP

```
---------------------------------------
clc;
clear all;
s=rng('default');

format long;

N=1000;
n=N*N;
```

```
b=rand(n,1);

%code MATLAB bicgstab
tic
A = gallery('poisson',N);
[x,flag,relres,iter]=bicgstab(A,b);
k=iter                  %number of iteraties
matlabcode=toc

% code cpu with CPU1
tic
u0=ones(n,1);
Au=PoissonAwithCPU(N,u0);  %Poissonmatrix * u0
r0=b-Au;
r_0=r0;
RowiMin1=1;
AlphaiMin1=1;
OhmiMin1=1;
viMin1=zeros(n,1);
piMin1=zeros(n,1);
rowi=1;
alphai=1;
ohmi=1;
vi=zeros(n,1);
pi=zeros(n,1);
ri=r0;
ui=u0;
for i=0:k
    rowi=r_0'*ri;
    betaiMin1=(rowi/RowiMin1)*(AlphaiMin1/OhmiMin1);
    pi=ri+betaiMin1*(piMin1-OhmiMin1*viMin1);
    pdak=1/4*pi;  %invM*pi
    vi=PoissonAwithCPU(N,pdak);
    Dotr_0vi=r_0'*vi;
    alphai=rowi/Dotr_0vi;
    s=ri-alphai*vi;
    z= 1/4*s;   %invM*pi
    t=PoissonAwithCPU(N,z);
    Dotts=t'*s;
    Dottt=t'*t;
    ohmi=Dotts/Dottt;
    ui=ui+alphai*pdak+ohmi*z;
    ri=s-ohmi*t;
    RowiMin1=rowi;
    AlphaiMin1=alphai;
    OhmiMin1=ohmi;
    viMin1=vi;
    piMin1=pi;
end
CPU1code=toc

%code cpu with type2
tic
A = gallery('poisson',N);
u0=ones(n,1);
Au=A*u0;  %Poissonmatrix * u0
r0=b-Au;
r_0=r0;
RowiMin1=1;
AlphaiMin1=1;
OhmiMin1=1;
viMin1=zeros(n,1);
piMin1=zeros(n,1);
rowi=1;
alphai=1;
ohmi=1;
vi=zeros(n,1);
pi=zeros(n,1);
ri=r0;
ui=u0;
for i=0:k
    rowi=r_0'*ri;
    betaiMin1=(rowi/RowiMin1)*(AlphaiMin1/OhmiMin1);
    pi=ri+betaiMin1*(piMin1-OhmiMin1*viMin1);
    pdak=1/4*pi;  % invM*pi
    vi=A*pdak;
    Dotr_0vi=r_0'*vi;
    alphai=rowi/Dotr_0vi;
    s=ri-alphai*vi;
    z= 1/4*s;   %invM*pi
    t=A*z;
    Dotts=t'*s;
    Dottt=t'*t;
    ohmi=Dotts/Dottt;
    ui=ui+alphai*pdak+ohmi*z;
    ri=s-ohmi*t;
    RowiMin1=rowi;
    AlphaiMin1=alphai;
    OhmiMin1=ohmi;
    viMin1=vi;
    piMin1=pi;
end
type2code=toc
```

```
%code gpu with GPU1
tic
u0=gpuArray(ones(n,1));
%we make one time e1 and e2
    e1=ones(n,1);
    e2=e1;
    for i=1:N-1
        e1(i*N+1,1)=0;
        e2(i*N,1)=0;
    end
    e1=gpuArray(e1);
    e2=gpuArray(e2);

Au=PoissonAwithGPU(N,u0,e1,e2);  %Poissonmatrix * u0

r0=gpuArray(b)-Au;
r_0=r0; %ones(n,1);
RowiMin1=gpuArray(1);
AlphaiMin1=gpuArray(1);
OhmiMin1=gpuArray(1);
viMin1=gpuArray(zeros(n,1));
piMin1=gpuArray(zeros(n,1));
rowi=gpuArray(1);
alphai=rowi;
ohmi=rowi;
vi=viMin1;
pi=piMin1;
ri=r0;
ui=u0;

for i=0:k
    rowi=r_0'*ri;    %dot(r_0,ri);
    betaiMin1=(rowi/RowiMin1)*(AlphaiMin1/OhmiMin1);
    pi=arrayfun(@Vector3,ri,betaiMin1,piMin1,OhmiMin1,viMin1);
    pdak=1/4*pi;  % invM*pi
    vi=PoissonAwithGPU(N,pdak,e1,e2);
    Dotr_0vi=dot(r_0,vi);
    alphai=rowi/Dotr_0vi;
    s=arrayfun(@Vector1,ri,alphai,vi);
    z=1/4*s;   %invM*pi
    t=PoissonAwithGPU(N,z,e1,e2);
    Dotts=dot(t,s);
    Dottt=dot(t,t);
    ohmi=Dotts/Dottt;
    ui=arrayfun(@Vector2,ui,alphai,pdak,ohmi,z);
    ri=arrayfun(@Vector1,s,ohmi,t);
    RowiMin1=rowi;
    AlphaiMin1=alphai;
    OhmiMin1=ohmi;
    viMin1=vi;
    piMin1=pi;
end
GPU2code=toc
---------------------------------------
```

F
Function ISP on the GPU

```
Function IPvanPoissonGPU
---------------------------------------
function [L,invL,invM,n]=IPvanPoissonGPU(N);
n=N*N;
A = gallery('poisson',N);
L=A;
for p=1:n
   L(p,p)=sqrt(L(p,p));
   for i=p+1:n
    if L(i,p)~=0
        L(i,p)=L(i,p)/L(p,p);
    end
   end
   for j=p+1:n
      for i=j:n
          if L(i,j)~=0
              L(i,j)=L(i,j)-L(i,p)*L(j,p);
          end
      end
   end
end
for i=1:n
   for j=i+1:n
       L(i,j)=0;
    end
end
D=0*L;
for i=1:n
    D(i,i)=L(i,i);
end
invD=0*L;
for i=1:n
    invD(i,i)=1/L(i,i);
end
D=gpuArray(zeros(n)+D);
L=gpuArray(zeros(n)+L);
invD=gpuArray(zeros(n)+invD);
N=invD*(L-D);
I=gpuArray(eye(n));
invT=I-N+N*N;
invL=invT*invD;
invM=invL'*invL;
---------------------------------------
```

Reference

[1] Shiming Xur: *Power Grid Simulation with GPU-Accelerated Iterative Solvers and Preconditioners*,Delft University of Technology (2011). ISBN: 978-94-6186-006-4.

[2] Pieter Schavemaker en Lou van der Sluis: *electrical power system essentials* (first edition), Delft University of Technology, the Netherlands (2008), ISBN 978-0470-51027-8.

[3] R. Idema, D.J.P. Lahaye, and C. Vuik: *Load Flow Literature Survey*, Delft University of Technology (2009)

[4] C.W.Oosterlee & C. Vuik: *Lecture notes: Scientific Computing*, TU Delft (2008)

[5] H.A. van der Vorst. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for solution of non-symmetric linear systems. *SIAM J. Sci. Stat. Comp.* , 13:631-644, 1992.

[6] Y. Saad and M.H. Schultz. GMRES: a generalized minimal residual algoritm for solving non-symmetric linear systems. *SIAM J. Sci. Stat. Comp.* , 7:856-869, 1986.

[7] A. van der Sluis. Conditioning, equilibration, and pivoting in linear algebraic systems. *Numer. Math.*, 15:74-86, 1970.

[8] J.A. Meijerink and H.A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Math. Comp.*, 31:148-162, 1977.

[9] D. Kirk and W.Hwu: *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, San Fransisco, 20120. ISBN: 978-0-12-381472-2.

[10] C.D. Meyer: *Matrix Analysis and Applied Linear Algebra Book*, siam, Society for Industrial and Applied Mathematic, 2000. ISBN: 9780898714548.