IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 22, 2021, accepted April 4, 2021, date of publication April 9, 2021, date of current version April 19, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3072338

Comparing Different Approaches for Solving Large Scale Power-Flow Problems With the Newton-Raphson Method

MANOLO D'ORTO[1], SVANTE SJÖBLOM[2], LUNG SHENG CHIEN[3], LILIT AXNER[4], AND JING GONG[1]

[1]PDC Center for High Performance Computer, School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, 10044 Stockholm, Sweden
[2]Svenska Kraftnät, 172 24 Sundbyberg, Sweden
[3]NVIDIA Corporation, Santa Clara, CA 95050, USA
[4]ENCCS, Department of Information Technology, Uppsala University, 751 05 Uppsala, Sweden

Corresponding author: Jing Gong (gongjing@kth.se)

This work was supported in part by the Swedish e-Science Research Center (SeRC) and the EuroCC Project which has received funding from the European Union's Horizon 2020 research and innovation programme under Grant 951732, and in part by the computations are enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at HPC2N through the Swedish Research Council under Grant 2018-05973.

ABSTRACT This paper focuses on using the Newton-Raphson method to solve the power-flow problems. Since the most computationally demanding part of the Newton-Raphson method is to solve the linear equations at each iteration, this study investigates different approaches to solve the linear equations on both central processing unit (CPU) and graphical processing unit (GPU). Six different approaches have been developed and evaluated in this paper: two approaches of these run entirely on CPU while other two of these run entirely on GPU, and the remaining two are hybrid approaches that run on both CPU and GPU. All six direct linear solvers use either *LU* or *QR* factorization to solve the linear equations. Two different hardware platforms have been used to conduct the experiments. The performance results show that the CPU version with *LU* factorization gives better performance compared to the GPU version using standard library called cuSOLVER even for the larger power-flow problems. Moreover, it has been proven that the best performance is achieved using a hybrid method where the Jacobian matrix is assembled on GPU, the preprocessing with a sparse high performance linear solver called KLU is performed on the CPU in the first iteration, and the linear equation is factorized on the GPU and solved on the CPU. Maximum speed up in this study is obtained on the largest case with 25000 buses. The hybrid version shows a speedup factor of 9.6 with a NVIDIA P100 GPU while 13.1 with a NVIDIA V100 GPU in comparison with baseline CPU version on an Intel Xeon Gold 6132 CPU.

INDEX TERMS High performance computing, Newton method, parallel algorithms, power engineering computing, power-flow, direct solver.

I. INTRODUCTION

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

High performance computing (HPC) techniques have lately started to take advantage of the computing power of GPU to overcome the limitations on CPU leading to hybrid CPU/GPU implementations. GPU-accelerated computing has been the most common accelerators for the power-flow problems in the last few years.

The associate editor coordinating the review of this manuscript and approving it for publication was Weipeng Jing.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

Here we address some related studies on performance and implementation efforts by others based on CPU/GPU accelerations. In [1] the authors presented a new approach of parallel ant colony optimization for GPU. A maximum speed-up of 44 can be achieved for the traveling salesman problem in comparison with the CPU counter-part. In [2] the authors proposed a parallel ant colony optimization on multi-core CPU base on the single-instruction, multiple-data (SIMD), achieving a maximum speed up of 57.8 compared to the standard CPU sequential version. In [3] the authors proposed an efficient parallel tabu search algorithm using co-design strategies for hardware/software and CPU/GPU. Their implementation demonstrates how to reduce transfer data between CPU and GPU and an optimized transfer strategy for GPU.

In [4] the authors studied various methods namely Gauss-Seidel method (G-S), the Newton-Raphson method (N-R), and P-Q decoupled method to solve the power-flow problem with CUDA parallel computing platform and application programming interface model created by NVIDIA, and C, a general-purpose procedural computer programming language. However, the sparsity of both Jacobian matrix and admittance matrix were not exploited. Paper [5] compared the performance of N-R applied to the power-flow problem. The CPU code was written in C++ while the GPU code in CUDA. The CPU version was accelerated with Intel Math Kernel Library (Intel MKL) which contains a set of optimized, thread-parallel mathematical functions for solving the linear equations and the matrix operations. The CUDA platform [6] with dense Jacobian matrix was used to solve the linear equation for the GPU version. In [7] a study was performed to find if the Fast Decoupled method with an iterative conjugate gradient (CG) linear solver performed better on GPU compared to CPU. The GPU version used CUDA Basic Linear Algebra Subprogram (cuBLAS) and the CUDA sparse matrix library (cuSPARSE) to execute the code on the GPU. The iterative linear solver used was the CG. The results showed that a speedup factor of 2.86 can be achieved using a single GPU. A comparative analysis of three different LU decomposition methods (i.e. KLU, NICSLU and GLU2) applied to power system simulations was presented in [8]. The study showed that KLU performed best overall when it came to both preprocessing and factorization time due to the factor that preprocessing step were KLU outperformed both NICSLU and GLU2. When the study was published in 2016 it showed great speedup compared to existing approaches [9]. Since then, several updates of the GLU algorithm have been released to further increase the performance [10], [11].

In this paper we focus on N-R for large scale power-flow problems. The main contributions of this work are as follows:

- Develop six different approaches to solve the power-flow problems. Two versions are executed on the CPU as a way to benchmark the other versions. The four remaining versions included two hybrid versions that are partly executed on the CPU and on the GPU and two versions that are executed almost entirely on the GPU.
- Provide a simple CUDA kernel to assemble sparse Jacobian matrix on GPU and explain why the simple kernel is efficiency.
- Illustrate there exists linear collaborations between filled in non-zeros elements and the execution time for the direct solver
- Evaluate every part of direct linear solvers and identify which parts of the program consume most of the execution times and measure the GPU and memory usages using profiling tools.
- Based on the evaluations, propose the two hybrid methods that combines analysis and factorization phases on GPU and solve phase on CPU, which are the best suited versions for the N-R method applied to large scale power flow problems. Maximum speed-up of 13.1 can be achieved by comparing with the baseline on CPU.

The rest of the paper is organized as follows. In Section II we introduce the power-flow problems and N-R applied to these power-flow problems. In Section III we present the methods used in this paper and detail the information about the different implementations. In Section IV we perform a complete experimental evaluation using these methods developed in Section III. Finally, in Section IV we conclude this paper and present ideas for possible future works.

II. POWER-FLOW PROBLEM

A. POWER-FLOW EQUATIONS

Power-flow is the analysis of the steady-state flow of electrical power in an interconnected system [12]. These power systems consist of buses (nodes) and lines (edges). The solution to the power-flow problem is usually obtained by solving nodal power balance equations. These types of equations are non-linear and therefore iterative techniques such as N-R are commonly used. The power-flow problem aims to determine the voltages at each bus and the active and reactive power in each line. The active and reactive powers can be calculated once voltage magnitudes and angles are known for all buses. The buses are divided into three types [13]:

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

The net injected power at any bus i can be calculated using the bus voltage V_i, its neighbouring bus voltages V_j and the admittances between the neighbouring buses Y_{ij} [13]. The power equation at any bus can be written as Equation (1).

$$S_i = p_i + jq_i = V_i I_i^* \tag{1}$$

with

p_i = The active power at bus i
q_i = The reactive power at bus i
j = The imaginary unit

By using the Kirchhoff's current law, $I_i = \sum_{j=1}^{N} Y_{ij}V_j$, we finally get the power flow equations (2).

$$p_i = v_i \sum_{j=1}^{N}(g_{ij}v_j\cos(\theta_{ij}) + b_{ij}v_j\sin(\theta_{ij}))$$
$$q_i = v_i \sum_{j=1}^{N}(g_{ij}v_j\sin(\theta_{ij}) - b_{ij}v_j\cos(\theta_{ij})) \quad (2)$$

where:

$\theta_{ij} = \theta_i - \theta_j$, the difference in phase angle between bus i and j,
g_{ij} = The real part of Y_{ij}
b_{ij} = The imaginary part of Y_{ij}

B. THE NEWTON-RAPHSON METHOD

The power-flow problem is often solved with N-R, G-S, and the Fast-Decoupled method (F-D) [14]. For N-R and F-D, the number of iterations needed to find a solution is not dependent on the size of the system, however this is not true for the G-S which scales poorly for larger systems. The N-R is in general more robust compared to both the F-D and G-S, when it comes to heavily loaded systems. This paper focuses on the N-R since it is well suited for large systems and systems that are stressed due to high active power transfer.

We assume that there is a network of total n buses consisting of n_v PV and n_q PQ buses. By applying the N-R to Equation (2), we obtain

$$\begin{bmatrix}\Delta P\\ \Delta Q\end{bmatrix} = \begin{bmatrix}\mathbf{J}_1 & \mathbf{J}_2\\ \mathbf{J}_3 & \mathbf{J}_4\end{bmatrix}\begin{bmatrix}\Delta\theta\\ \Delta V\end{bmatrix} = \mathbf{J}\begin{bmatrix}\Delta\theta\\ \Delta V\end{bmatrix} \quad (3)$$

where:

$$\mathbf{J} = \begin{bmatrix}\mathbf{J}_1 & \mathbf{J}_2\\ \mathbf{J}_3 & \mathbf{J}_4\end{bmatrix} \in \mathbb{R}^{(n+n_q-1)\times(n+n_q-1)},$$
$$\mathbf{J}_1 = \frac{\partial P}{\partial\theta} \in \mathbb{R}^{(n_v-1)\times(n_v-1)},$$
$$\mathbf{J}_2 = \frac{\partial P}{\partial V} \in \mathbb{R}^{(n_v-1)\times 2n_q},$$
$$\mathbf{J}_3 = \frac{\partial Q}{\partial\theta} \in \mathbb{R}^{2n_q\times(n_v-1)},$$
$$\mathbf{J}_4 = \frac{\partial Q}{\partial V} \in \mathbb{R}^{2n_q\times 2n_q},$$
$$\Delta P = \begin{bmatrix}p_2^{spec} - p_2^{calc}\\ \vdots\\ p_n^{spec} - p_n^{calc}\end{bmatrix}, \quad \Delta Q = \begin{bmatrix}q_2^{spec} - q_2^{calc}\\ \vdots\\ q_n^{spec} - q_n^{calc}\end{bmatrix}$$

Jacobian matrix $\mathbf{J}$ consists of four submatrices. Jacobian matrix $\mathbf{J}$ is a square matrix with the number of rows and columns of $n_v + 2n_q - 1 = n + n_q - 1$ as the slack bus is not included and the voltage magnitudes of the PV buses are known.

$$j_1(i,i) = \frac{\partial p_i}{\partial\theta_i} = v_i\sum_{\substack{j=1\\ j\neq i}}^{N-1} v_j(-g_{ij}\sin(\theta_{ij}) + b_{ij}\cos(\theta_{ij}))$$
$$j_1(i,j) = \frac{\partial p_i}{\partial\theta_j} = v_iv_j(g_{ij}\sin(\theta_{ij}) - b_{ij}\cos(\theta_{ij})), \quad i\neq j$$
$$j_2(i,i) = \frac{\partial p_i}{\partial v_i} = 2g_{ij} + \sum_{\substack{j=1\\ j\neq i}}^{N_q} v_j(g_{ij}\cos(\theta_{ij}) + b_{ij}\sin(\theta_{ij}))$$
$$j_2(i,j) = \frac{\partial p_i}{\partial v_j} = v_j(g_{ij}\cos(\theta_{ij}) + b_{ij}\sin(\theta_{ij})), \quad i\neq j$$
$$j_3(i,i) = \frac{\partial q_i}{\partial\theta_i} = -v_i\sum_{\substack{j=1\\ j\neq i}}^{N-1} v_j(g_{ij}\cos(\theta_{ij}) + b_{ij}\sin(\theta_{ij}))$$
$$j_3(i,j) = \frac{\partial q_i}{\partial\theta_j} = v_iv_j(-g_{ij}\cos(\theta_{ij}) - b_{ij}\sin(\theta_{ij})), \quad i\neq j$$
$$j_4(i,i) = \frac{\partial q_i}{\partial v_i} = -2v_i + \sum_{\substack{j=1\\ j\neq i}}^{N_q} v_j(g_{ij}\sin(\theta_{ij}) - b_{ij}\cos(\theta_{ij}))$$
$$j_4(i,j) = \frac{\partial q_i}{\partial v_j} = v_j(g_{ij}\sin(\theta_{ij}) - b_{ij}\cos(\theta_{ij})), \quad i\neq j$$

When calculating the diagonal elements, P and Q can be reused to minimize the complexity. The simplification is done by negating q_i in Equation (2) and subtracting $v_i^2b_{ij}$ to formula $j_1(i,i)$,

$$j_1(i,i) = \frac{\partial p_i}{\partial\theta_i} = -q_i - v_i^2b_{ii} \quad (4)$$

Similarly, the diagonal elements of the submatrices $\mathbf{J}_2 - \mathbf{J}_4$ can be calculated as

$$j_2(i,i) = \frac{\partial p_i}{\partial v_i} = \frac{p_i}{v_i} + v_ig_{ii} \quad (5)$$
$$j_3(i,i) = \frac{\partial q_i}{\partial\theta_i} = p_i - v_i^2g_{ii} \quad (6)$$
$$j_4(i,i) = \frac{\partial q_i}{\partial v_i} = \frac{q_i}{v_i} - v_ib_{ii} \quad (7)$$

The voltage angles and magnitudes are updated at each iteration k in Equation (8). The iterations are repeated until each element in $[\Delta P^k, \Delta Q^k]^T$ is less than a specified tolerance ε.

$$\begin{bmatrix}\Delta\theta^k\\ \Delta V^k\end{bmatrix} = \begin{bmatrix}\mathbf{J}_1 & \mathbf{J}_2\\ \mathbf{J}_3 & \mathbf{J}_4\end{bmatrix}^{-1}\begin{bmatrix}\Delta P^k\\ \Delta Q^k\end{bmatrix}$$
$$\begin{bmatrix}\theta^k\\ V^k\end{bmatrix} = \begin{bmatrix}\theta^{k-1}\\ V^{k-1}\end{bmatrix} + \begin{bmatrix}\Delta\theta^k\\ \Delta V^k\end{bmatrix} \quad (8)$$

C. LINEAR EQUATION SOLVERS

For each iteration of N-R, the linear equation (8) has to be solved. There are two ways of obtaining the solution, either by using so-called iterative linear solvers or by using direct linear solvers. Iterative linear solvers start with an estimation and then iterate until they converge. Iterative linear solvers

do not guarantee that a solution will be found. On the other hand, direct linear solvers do not start with an estimation and converge toward the solution but rather solve the system straight away. When it comes to the large systems, iterative linear solvers might give better performance, however for highly sparse matrices direct linear solvers are better suited. Since the Jacobian matrix is highly sparse this paper focuses on the direct linear solvers.

Traditionally, the linear system has been solved with an *LU* factorization of the Jacobian matrix for the power-flow problem. The most expensive parts of N-R are the linear equation solvers at each iteration [15]. Experiments have shown that solving the linear equations with LU factorization took about 85% of the total execution time of a power system with 3493 buses on CPU [15].

When solving sparse linear systems using direct linear solvers, reordering schemes can be applied to minimize the fill-in. The fill-in means that the zero entries of a matrix turn into a non-zero value during the execution of an algorithm [16]. The reordering schemes used in this paper are shown below.

- *METIS Reordering:* METIS is a software package for computing fill-reducing orderings for sparse symmetric matrices [17]. This function `METIS_NodeND` computes the orderings to reduce the fill-in based on the multilevel nested dissection paradigm. The nested dissection paradigm is based on computing the vertex separator of the graph corresponding to the sparse matrix.
- *AMD Reordering:* The algorithm is based on the observation that when a variable is eliminated, a clique is formed by its neighbours for sparse symmetric matrices [18]. Each of the edges within the clique contributes to the fill-in. Thus, the AMD reordering aims at minimizing the fill-in by forming the smallest possible clique at each step.
- *COLAMD Reordering:* One of the main differences between COLAMD and AMD is that COLAMD does not need the sparsity pattern of the input matrix to be symmetrical [18]. COLAMD computes the column permutations without calculating the normal equations. This makes it a good choice for QR factorization since the QR factorization does not calculate the normal equations.
- *SYMRCM Reordering:* Similar to COLAMD reordering and METIS reordering, the sparsity pattern of the input matrix needs to be symmetrical [19]. The SYMRCM is based on a breadth-first search algorithm. The aim of SYMRCM is to minimize the bandwidth of the matrix.

By combining with these reordering schemes, special techniques such as Gibert's algorithm [20] as well as KLU in [21] and GLU (for GPU LU) direct solvers are employed to solve the linear system raised from the power-flow problems. The basic idea of the Gilbert's algorithm is to solve the numerical factors L and U column by column without explicit reordering of the row entries during the pivoting process. The symbolic sparsity of each column is pre-processed by topological ordering from sparse triangular solve. The size of LU is adjusted by this estimation, then numerical factorization and collection of non-zero entries of that column is performed. The main advantage of Gilbert algorithm is that it doesn't involve row swapping, thus spares us from memory management.

KLU is another algorithm presented in [21] that exploits the fact that the sparsity pattern of the Jacobian matrix applied to most general power-flow problems is exactly the same at each iteration of N-R. The algorithm is based on *LU* factorization and has a total of four steps which are listed below.

1) The matrix is permuted into block triangular form
2) A reordering scheme is applied to each created block to reduce fill-in. This is done to reduce the amount of memory needed and the total number of arithmetic operation.
3) The diagonal blocks are scaled and factorized according to Gilbert and Peierls' left looking algorithm with partial pivoting [20]
4) Finally, the linear system is solved using block back substitution

Since the sparsity pattern of the Jacobian matrix is the same at each iteration, the future iterations disregard the first two steps [21]. Furthermore, the third step implements a simplification of the left looking algorithm which does not perform partial pivoting. With this, the depth-first search used in Gilbert and Peierls' algorithm can be omitted. This is because the non-zero patterns of L and U are already known from the first iteration.

GPU accelerated *LU* factorization (GLU) solver is based on a hybrid right-looking *LU* factorization for sparse matrices [9]. The GLU algorithm performs a total of 4 steps [8]:

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

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

III. METHODS

Six different versions of solving the power-flow problem are developed and evaluated. Two of these versions are executed exclusively on the CPU as baselines, two are executed on the GPU and the remaining two are hybrids CPU/GPU. The general outline for all the versions is listed below.

1) Assembly of the **Y**-matrix from an input file
2) Calculation of the power-flow equations
3) Assembly of the Jacobian matrix
4) Application of a linear solver
5) Update of voltage angle and voltage magnitude

Both Jacobian **J** and admittance **Y** are sparse matrices for large networks. These can be stored in compressed storage formats. The format used for Jacobian matrix is Compressed Sparse Row (CSR). The CSR uses three vectors to store the information to specify a sparse matrix [22]: the row pointer vector contains the offset of the first value on each row, and the last element in the row pointer vector contains the total number of non-zeroes in the matrix; the column indices vector contains the column of each of the non-zero elements of the matrix; and the vector with values contains all the non-zero values of the original matrix. The admittance matrix is used in Coordinate Format (COO) since it can be directly assembled from the input data in [23].

```
__global__ void powerEqn(double *P, double *Q,
    int* yRow, int* yCol,
    cuDoubleComplex* Ymatrix,
    double *theta, double *Voltage, int nnzY) {
  int ix = blockIdx.x*blockDim.x+threadIdx.x;
  if (ix < nnz) {
     int i = yRow[ix], j = yCol[ix];
     cuDoubleComplex yij = Ymatrix[ix];
     double gij = cuCreal(yij);
     double bij = cuCimag(yij);
     double vi = Voltage[i], vj = Voltage[j];
     double theta_ij = theta[i]-theta[j];
     atomicAdd(&P[i],vi*vj*(gij*cos(theta_ij)
        +  bij*sin(theta_ij)) ;
     atomicAdd(&Q[i],vi*vj*(gij*sin(theta_ij)
        - bij*cos(theta_ij));
  }
}
```

Listing 1. The kernel that calculates the power-flow equations.

Listing 1 presents the kernel to calculate the power-flow equations. In contrast to how the C++ implementation is executed with being iterated over the number of non-zero (`nnz`) elements of the **Y**-matrix, the CUDA version performed this through launching a kernel running as many threads as there are `nnz` elements in the **Y**-matrix. As a precaution to avoid data races and similar problems, CUDA atomic addition is used when the power equations are calculated, see Listing 1.

In the GPU versions the elements in Jacobian matrix are calculated in the same manner as in the CPU C++ versions, i.e., updated P and Q values from previous step. The main difference is that four kernels are used to calculate Jacobian matrix. Each kernel calculated one of the four sub-matrices $\mathbf{J}_1-\mathbf{J}_4$ and is launched with as many threads as the number of non-zero element. As an example, the kernel for the assembly of sub-matrix $\mathbf{J}_1$ of Jacobian matrix can be seen in Listing 2.

In Lisiting 2 the sparse admittance matrix **Y** stores in the three arrays `yRow`, `yCol`, and `yMatrix` in COO format. Jacobian matrix **J** in CSR format is assembled using two integer arrays `jacRow`, `jacCol` and one float array `jac`.

```
dim3 numBlocksJ1((nnzYJ+threads-1)/threads);
getJ1<<<numBlocksJ1,numThreads,0,streams[2]>>>
 (d_jacRow,d_jacCol,d_jac,d_yJRow,d_yJCol,
  d_yJ,d_voltage,d_angle,d_qCalc,d_nnzJ2,nnzYJ);
__global__ void getJ1(int *jacRow, int *jacCol,
  double *jac, int *yRow, int *yCol,
  cuDoubleComplex* yMatrix,
  double *Voltage, double *theta,
  double *q, int *nnzJ2, int nnzJ1) {
  int ix = blockIdx.x*blockDim.x+threadIdx.x;
  if (ix < nnzJ1) {
    int row = yRow[ix], col = yCol[ix];
    int global_ix = ix + nnzJ2[row];
    if (row != 0 && col != 0) {
       jacRow[global_ix] = row;
       jacCol[global_ix] = col;
       double v_row = Voltage[row];
       double v_col = Voltage[col];
       cuDoubleComplex y_ix = Ymatrix[ix];
       double g_ix = cuCreal(y_ix);
       double b_ix = cuCimag(y_ix);
       if (row == col) {
         jacVal[ix+offset] =
           -q[row]-v_row*v_row*b_ix;
       } else {
         double theta_ix = theta[row]-theta[col];
         jacVal[global_ix] = v_row*v_col*(
           g_ix*sin(theta_ix)-b_ix*cos(theta_ix));
       }
    }
  }
}
```

Listing 2. The kernel that assembles the submatrix $\mathbf{J}_1$.

The variable `global_ix` indicates the global position of one element of submatrix $\mathbf{J}_1$ in Jacobian matrix **J**.

The linear solvers used for the C++ version were sparse *LU* and sparse *QR* factorization. Both types of factorization were implemented using `Eigen` library [24]. This library provides three different reordering schemes to minimize the complexity of the factorization schemes. The reordering schemes were column approximate minimum degree ordering, approximate minimum degree ordering, and natural ordering. The library used in GPU versions to solve the linear equations is cuSOLVER [25] which provides several dense and sparse linear solvers. The linear solvers used in this paper are the sparse *QR* and dense *LU* factorization solvers. As opposed to the C++ implementation, the sparse LU factorization solver is not used for the CUDA version since the sparse LU factorization provided by cuSOLVER does not run on the GPU. The version with dense *LU* factorization is mostly evaluated to better compare the different versions since one of the CPU versions and the hybrid versions use sparse *LU* factorization.

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

Two different hybrid versions are developed and evaluated. They are similar with each other and the difference being

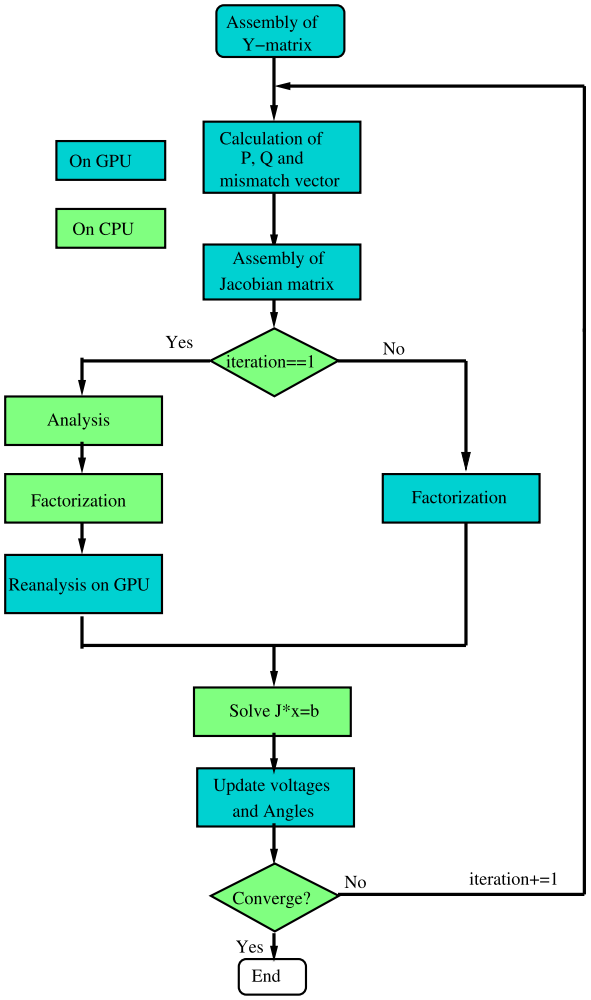

FIGURE 1. The general workflow of hybrid version with Gilbert's algorithm.

how the preprocessing of the *LU* factorization is approached. In the first hybrid version, the preprocessing step is based on Gilbert's algorithm [20] and is executed on the CPU. In the second hybrid version, the preprocessing step is based on KLU [21] and is executed on the CPU. Both the hybrid versions exploited the fact that the sparsity pattern of the Jacobian matrix, the L matrix and the U matrix are exactly the same at each iteration and therefore the preprocessing is only needed in the first iteration. The workflow diagram for the second hybrid version is shown in Figures 1.

The under developed cuSOLVER low-lever API is used for the linear equations in the hybrid versions, see Listing 3. The process is split into standard steps but with different APIs.

Once the linear equations are solved, the voltage angle is updated for all the buses except for the slack bus. The voltage magnitude is updated for all the PQ buses. This is done with one kernel running $N - 1$ threads. The kernel used for updating voltage angles and magnitudes can be seen in Listing 4.

```
// (GLU) analyze the parallelization
cusolverSpDgluAnalysis( );

// (GLU) reset the matrix value
cusolverSpDgluReset( );

// (GLU) refactorization M = L*U
cusolverSpDgluFactor( );

// (GLU) solve A*b with iterative refinement
cusolverSpDgluSolve( )
```

Listing 3. The kernel that solves the linear equations.

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

Listing 4. The kernel that updated the voltage angles and magnitudes.

IV. EXPERIMENTAL EVALUATION

In this section, first we address the filled in non-zero elements using various reordering schemes and the sparse pattern of Jacobian matrices and the following is execution times on the assembly of Jacobian matrices. And then two CPU versions and two pure GPU version for direct solvers have been carried out. The best performances in different phases (i.e. analysis, factorization, and solve) have been identified. Finally, two main hybrid methods combining with CPU and GPU have been performed based on previous detailed analysis.

Two different hardware platforms are used to conduct the experiments. One platform is the Kebnekaise system at the High Performance Computing Center North (HPC2N). On the system each GPU node consists of an Intel Xeon Gold 6132 CPU and an NVIDIA V100 with 32GB of HBM2. The version of the GCC compiler is 8.3.0 and the CUDA version is 10.1.243. The other hardware platform consists of an Intel Xeon Broadwell CPU with 128GB DDR4 RAM and a NVIDIA Tesla P100 GPU with 16GB HBM2 Stacked Memory.

All the input data is taken from the Github of MATPOWER [23]. The input data is divided into four matrices but only the first three of these are of interest. The first two matrices contained data about each bus in the network with each row corresponded to one bus. The third matrix of the input data contained data for each branch in the network. The branch data contained all data needed to assemble the admittance matrix $\mathbf{Y}$.

Table 1 presents the sizes and number of non-zero of Jacobian matrices corresponding to various buses. By comparing with other algorithms, the KLU algorithm with AMD reordering requests minimum filled in non-zero elements that

TABLE 1. The fill-in with different reordering schemes.

#Buses (N)	500	1354	1888	2000	2383	2869	3120	9241	25000
mJ	909	2447	3498	3514	4438	5227	5890	17036	46764
nnzJ	6037	15803	23958	25652	27874	36591	37040	129412	314922
nnzM (SparseQR+METIS)	40356	159910	327877	628431	483232	445697	730269	2525732	11830604
nnzM (SparseQR+RCM)	44065	135945	243872	944683	576591	319379	748453	2048815	18827548
nnzM (SparseQR+AMD)	37491	115264	206082	1023083	552317	669104	790103	4862728	15089202
nnzM (Gilert+METIS)	9269	24915	38298	87478	50212	62297	68506	241992	817906
nnzM (KLU+AMD)	**8549**	**22011**	**33270**	**76080**	**44726**	**54969**	**61648**	**214926**	**696576**
nnzM (KLU+COLAMD)	12145	30107	47288	129388	69788	82131	94200	350022	1210501

mJ is the number of columns of Jacobian matrix; **nnzJ** is number of non-zero elements of Jacobian matrix;
nnzM is filled in non-zero elements; highlighted black numbers are minimum filled in non-zero elements

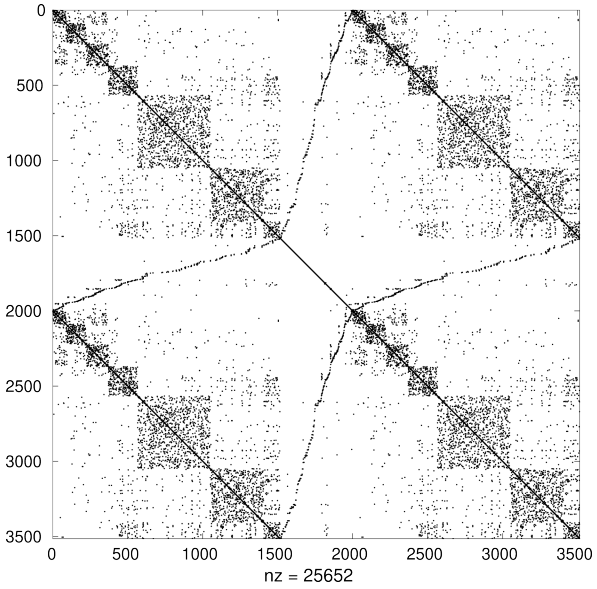

FIGURE 2. The Sparsity pattern for Jacobian matrix with 2000 buses.

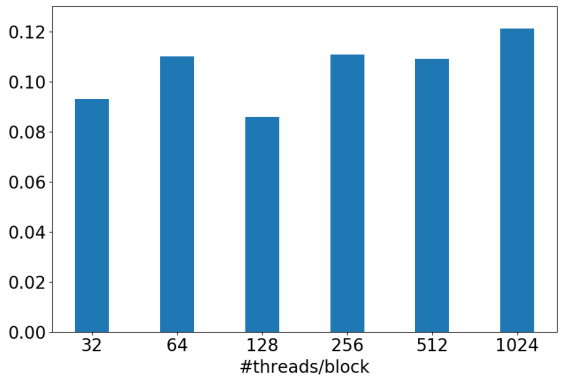

FIGURE 3. The execution times (ms) on the assembly of Jacobian matrix with 9241 buses using different numbers of threads per block.

highlighted in Table 1. The sparsity pattern of Jacobian matrix with 2000 buses is shown in Figure 2.

Figure 3 shows the execution times on the assembly of Jacobian matrix with 9241 buses using different numbers of threads per block on a single P100 GPU. The assembly of Jacobian matrix is evaluated since it is the second most time-consuming kernel. Timing the entire program would lead to excessive overhead as the factorization methods are the most time consuming part of the calculation. From Figure 3 the best performance can be obtained using 128 threads per block. This is due to the streaming multiprocessors keeping busy but not being overloaded with work. For smaller networks the optimized threads per block can vary from 128. However, since the execution time on the assembly the Jacobian matrix is relatively short compared to the execution time for solving the linear equations, further experiments for finding the optimal number of threads per block for various cases are not performed. Consequently, the GPU kernels are launched with 128 threads per block by default in these experiments.

All CUDA operations run in a stream either implicitly or explicitly, this is true for both kernels and data transactions. If nothing is stated the default stream is used, also known as the NULL stream. If the programmer wants to overlap different CUDA operations, streams have to be declared explicitly. Streams can be used to overlap kernels and data transfers. It is important to note that pinned memory needs to be used if one wants to overlap data transfers. When a stream is created it is a so-called blocking stream, this means that those streams can be blocked waiting for earlier operations in the NULL stream. The execution times on the assembly of Jacobian matrices are presented using the three implements in Table 2. The execution times on GPU are rather consistence for all buses while the execution times on CPU increases significantly with the number of buses. The best performances for the assembly for all buses can be achieved using CUDA version with stream and the maximum speed-up on GPU is 127 for buses 25000.

Four streams are used with each stream calculating a sub-matrix of the Jacobian matrix. For cases $500-9241$ buses the CUDA version takes around $0.07-0.08$ms with 4 streams while it takes around $0.11-0.14$ms without stream, i.e. the execution time reduces 50% with streams. For the case of 25000 buses, the difference between with and without stream is small (0.04ms). In comparison with CPU C++ version, maximum speed-up of 126 can be obtained. The Jacobian update belongs to data parallelism, all nonzeros can be computed independently. Naïve translation from C++ code (CPU) to CUDA kernel can reach decent performance because the collective GPU threads with the same index i can share θ_i and v_i in cache, access of y_{ij} is coalesced, the only non-coalesced access is θ_j and v_j. Although GPU is not saturated when launching four Jacobians with four streams

TABLE 2. The execution times (ms) on the assembly of the Jacobian matrices.

#Buses (N)	500	1354	1888	2000	2383	2869	3120	9241	25000
C++	0.369	0.544	0.756	1.248	0.899	1.863	0.969	4.187	16.924
CUDA w/o stream	0.133	0.143	0.134	0.135	0.135	0.115	0.123	0.129	0.175
CUDA with stream	**0.078**	**0.076**	**0.078**	**0.075**	**0.077**	**0.076**	**0.083**	**0.085**	**0.133**

Highlighted black numbers are best performances on the assembly.

TABLE 3. The execution time (ms) per iteration using dense *LU* and sparse *QR* on a single V100 GPU.

	#Buses (N)	500	1354	1888	2000	2383	2869	3120	9241	25000
Dense LU	DGETRF(piovt)	6.28	22.37	35.68	36.00	51.30	67.17	82.69	779.87	-
	DGETRF(no pivot)	1.71	7.14	12.64	12.64	20.12	28.53	37.09	589.31	-
	DGETRS	0.36	0.87	1.22	1.24	1.53	1.81	2.06	7.62	-
	Total	6.64	23.25	36.90	**37.24**	**52.82**	68.98	**84.75**	787.49	-
Sparse QR (RCM)	analysis	1.807	5.256	8.658	21.554	15.934	11.722	19.473	57.680	517.696
	factor+solve	5.349	13.643	21.732	75.142	48.668	35.333	57.552	180.444	1213.771
	Total	7.156	18.899	**30.390**	96.696	64.602	**47.055**	77.025	**238.124**	**1731.467**
Sparse QR (METIS)	analysis	2.607	9.633	16.343	22.029	24.192	25.876	30.232	129.32	577.219
	factor+solve	3.637	15.868	26.955	56.610	49.846	47.004	82.097	347.641	1230.006
	Total	**6.244**	25.501	43.298	78.639	74.038	72.880	112.329	476.961	1807.225
Sparse QR (AMD)	analysis	2.609	10.214	16.938	33.686	31.067	40.989	47.303	408.158	2104.309
	factor+solve	4.820	12.140	22.683	59.949	50.953	63.626	122.989	484.306	1353.003
	Total	7.429	**22.354**	39.621	93.635	82.020	104.615	170.292	892.464	3457.312

highlighted black numbers are the best performances in total times

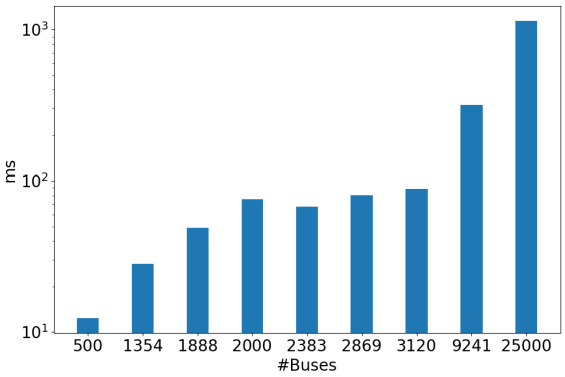

FIGURE 4. The execution time (ms) of the C++ version of *LU* factorization with COLAMD reordering on CPU.

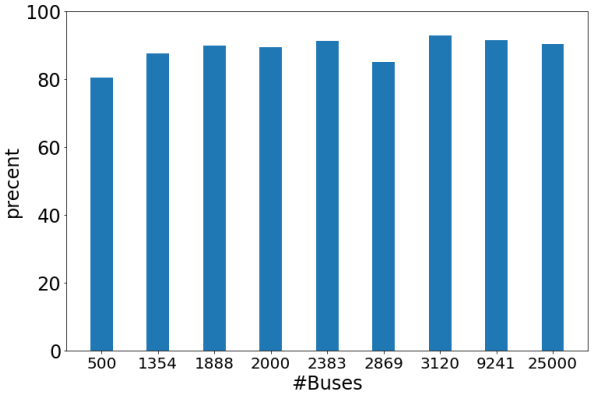

FIGURE 5. The percentage of total execution time on the baseline linear solver with *LU* factorization on CPU.

simultaneously, we don't expect 4× speedup compared to sequential launches because the kernel runtime is so tiny that extra kernel launch overhead ($5-10\mu$s) counts.

In the CPU C++ version, the execution times for the *QR* factorization increase drastically with the larger case regardless of which reordering schemes used. It takes more than 20 hours to solve the equations with 9241 buses. The reason is that the fill-in of non-zero for the *QR* factorization is too large. As shown in Table 1, The filled in non-zero elements are $2.0-4.8$M (nnzM/nnZ = 16-37) with various reordering schemes. Consequently, the performance results for the CPU version of *QR* factorization will not be presented. AMD reordering for the sparse *LU* factorization outperforms the other reordering schemes significantly since AMD reordering produces less fill-in compared to COLAMD and natural orderings. The *LU* factorization with AMD on the CPU will be used as baseline in the follows. We run all cases in fixed iterations (6) to obtain better comparison. Figure 4 shows the execution time of the LU factorization on the CPU.

Figure 5 shows the percentage of total execution times on the baseline linear solver with *LU* factorization on CPU. To solve the linear equation takes $81-93\%$ of the total execution times for various number of buses on CPU, which agree with the conclusion in [15].

Table 3 presents the execution time for the two GPU versions using standard dense *LU* and sparse *QR* within cuSOLVER on a single V100 GPU. Three reordering schemes namely METIS, RCM, and AMD for the sparse *QR* have been employed. The corresponding numbers of fill-in non-zero entries are presented in Table 1. The dense *LU* solver is out of memory for the case of buses 25000. The execution times on the phases of factorization and solve have high consistency with the number of fill-in, see Figure 6. In comparison with the performance results of CPU baseline version shown in Figure 4, the performances cannot be accelerated using both these GPU versions.

TABLE 4. The execution time (ms) per iteration using different reorder schemes on a P100 GPU node.

Reordering		#Buses (N)	500	1354	1888	2000	2383	2869	3120	9241	25000
Gilbert METIS	CPU LU	analysis	1.778	6.687	9.424	11.806	15.098	17.641	18.795	79.390	260.376
		factor	0.377	1.174	1.7381	4.971	2.467	2.943	3.259	12.716	49.570
		solver	0.039	0.083	0.112	0.165	0.138	0.159	0.167	0.598	2.015
	GLU	analysis	0.439	**0.4977**	**0.507**	**0.661**	**0.545**	**0.543**	**0.601**	**0.845**	**0.7183**
		factor	**0.229**	**0.374**	**0.437**	**0.747**	**0.569**	**0.629**	**0.644**	**1.766**	**6.472**
		solve	1.097	2.418	1.832	5.212	2.092	3.714	2.467	4.659	11.166
KLU AMD	CPU LU	analysis	**0.248**	0.865	1.139	1.520	1.545	2.107	2.1251	6.993	16.511
		factor	0.343	1.065	1.459	3.499	2.160	2.758	2.815	10.457	34.917
		solve	**0.020**	**0.075**	**0.080**	**0.148**	**0.120**	**0.149**	**0.150**	**0.460**	**1.658**
	GLU	analysis	0.454	0.510	0.552	0.806	0.566	0.656	0.687	1.061	6.405
		factor	0.251	0.381	0.495	0.958	0.660	0.817	0.873	2.403	6.944
		solve	1.1189	1.687	1.732	3.007	2.178	2.698	2.901	6.339	20.06
KLU COLAMD	CPU LU	analysis	0.473	1.365	2.039	2.425	2.764	3.384	3.526	12.084	26.798
		factor	0.530	1.441	2.161	6.222	3.162	3.948	4.373	17.812	75.744
		solve	0.027	0.087	0.112	0.216	0.169	0.197	0.213	0.712	2.857
	GLU	analysis	0.594	0.668	0.755	1.528	0.805	1.107	1.273	1.747	6.318
		factor	0.363	0.462	0.636	1.627	0.832	1.289	1.224	3.313	13.712
		solve	1.363	2.005	2.614	4.920	3.191	4.837	4.855	14.265	30.500

highlighted black numbers are the bast performances in analysis phase; highlighted red numbers are the bast performances in factorization phase; highlighted green numbers are the bast performance in solve phase.

TABLE 5. The execution time (ms) per iteration using different reorder schemes on a V100 GPU node.

Reordering		#Buses (N)	500	1354	1888	2000	2383	2869	3120	9241	25000
Gilbert METIS	CPU LU	analysis	1.226	4.774	6.386	7.832	10.16	12.04	13.166	56.186	184.529
		factor	0.289	0.896	1.261	3.439	1.783	2.204	2.401	8.41	34.848
		solve	**0.018**	0.066	**0.06**	**0.122**	**0.105**	**0.134**	**0.136**	**0.488**	**1.755**
	GLU	analysis	0.391	0.477	**0.480**	**0.605**	**0.495**	**0.529**	**0.551**	**0.755**	**6.240**
		factor	**0.167**	0.277	**0.325**	**0.530**	**0.404**	**0.417**	**0.448**	**1.003**	**4.419**
		solve	0.387	0.570	0.716	1.177	0.735	0.807	0.803	1.527	7.436
KLU AMD	CPU LU	analysis	**0.207**	0.602	0.785	1.009	1.091	1.339	1.366	4.729	10.792
		factor	0.267	0.746	1.065	2.322	1.458	1.898	1.992	8.016	25.094
		solve	0.018	**0.041**	0.097	0.13	0.088	0.106	0.122	0.361	1.277
	GLU	analysis	0.404	**0.465**	0.500	0.746	0.56	0.657	0.623	1.072	9.033
		factor	0.195	**0.272**	0.339	0.679	0.458	0.561	0.549	1.435	4.745
		solve	0.432	0.631	0.781	2.29	0.93	1.291	2.224	4.289	8.688
KLU COLAMD	CPU LU	analysis	0.299	0.97	1.197	1.753	1.86	2.299	2.42	8.442	19.149
		factor	0.346	0.982	1.46	4.437	2.13	2.806	2.889	13.202	59.544
		solve	0.035	0.052	0.095	0.183	0.117	0.136	0.146	0.545	2.737
	GLU	analysis	0.55	0.608	0.663	1.214	0.735	0.997	1.088	1.498	5.622
		factor	0.293	0.338	0.443	1.142	0.542	0.882	0.831	2.009	9.704
		solve	0.828	1.232	1.793	3.732	1.912	3.859	3.45	8.98	20.14

highlighted black numbers are the bast performances in analysis phase; highlighted red numbers are the bast performances in factorization phase; highlighted green numbers are the bast performance in solve phase.

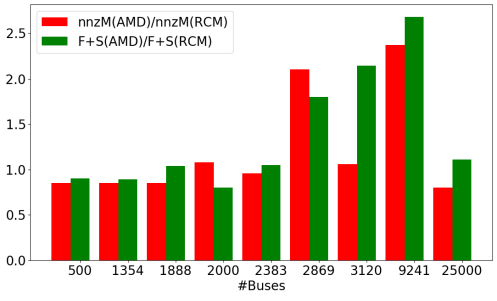

FIGURE 6. The factors of *nnzM* and *LU* factorization between AMD and RCM ordering schemes on CPU.

In order to exploit the capabilities of both CPU and GPU, the three phases of a direct solver (`analysis`, `factorization`, and `solve`) should be identified. Tables 4 and 5 show the execution times using different reordering schemes. The best performance on each phase has been highlighted. Even the type of Intel CPUs are slight different, The GLU solver on GPU has best best performances on `analysis` and `factorization` while LU solver on CPU has best performance on `solve`. Moreover, GLU using Gilbert's algorithm with METIS reordering scheme has fastest analysis and factorization configuration on P100 for all cases except case of bus 500. In the case of bus 500, the KLU with AMD reordering on CPU has fastest `analysis` phase with 0.248, see Table 4. KLU with AMD reordering on CPU has also fastest `solve` for all cases. The best performance still can be achieved on the V100 GPU the GLU solver using the Gilbert's algorithm with METIS reordering schemes for all cases except two smallest cases. On the CPU the best performance has been obtained using LU solver based on Gilbert's algorithm with METIS reordering scheme.

When analyzing Tables 4 and 5 it can be seen that the execution time for the case with 2000 buses is actually longer than the case with 3120 buses for some of the versions. This is

TABLE 6. The peak performance (GFlop/s) and maximum memory (MB) of the GLU algorithm with Gilbert METIS reordering scheme.

#Buses (N)	500	1354	1888	2000	2383	2869	3120	9241	25000
peak performance (GFlop/s)	31.0	73.3	103.7	121.1	146.4	144.7	192.2	276.7	687.1
maximum memory (MB)	566	587	608	612	636	662	685	1448	2464

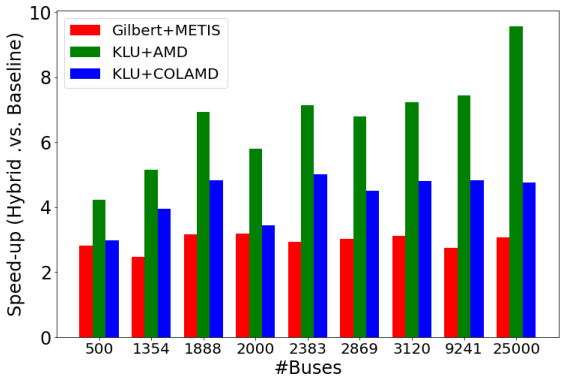

FIGURE 7. The speed-up of hybrid versions in comparison with the baseline CPU version on P100 GPU node.

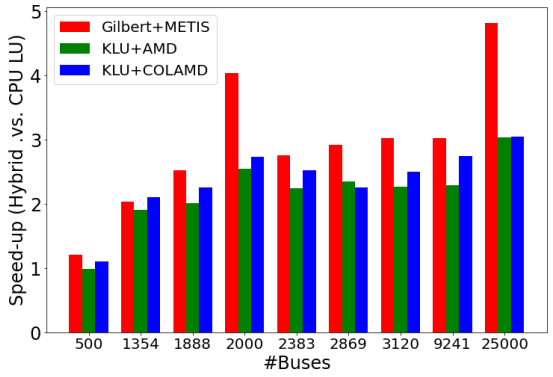

FIGURE 8. The speed-up of hybrid versions in comparison with KLU CPU version on P100 GPU node.

a surprising outcome but a possible explanation of this could be that the sparsity pattern of the Jacobian matrix for the two cases differed in how dense they are or in the way that they are assembled. As shown in Table 1, The fill-in of case 2000 are large than that of case 3120 for all reordering methods. Thus the fill-in has an impact not only on the memory usages but also on the execution time.

We demonstrate results on a single V100 GPU to illustrate the GPU version has better performances on the analysis and factorization phases. With the NVIDIA profiling tool `nvprof` the performances in Flop/s for the double precision can be measured by the metric `flop_count_dp` and runtime. Table 6 presents the peak performance in GFlop/s and maximum memory usage in MB bases on GLU algorithm with Gilbert METIS reordering scheme for various buses on a V100 GPU. The peak performance occurs when the kernels for the sparse matrix vector multiplication are called in the analysis and factorization phases. The performance depends highly on the computational workload of the GPU. As shown in Table 6, the performance increases with the number of buses and maximum of 687.1 GFlop/s can be achieved when run the case of 25000 buses.

Based on the performance results presented in Tables 4 and 5, two hybrid versions that `analysis` and `factorization` phases are performed on GPU while `solve` phase is performed on CPU have been developed, see the workflow in Figure 1. One hybrid version use the Gibert's algorithm with METIS order and the other use the KLU algorithm with AMD and COLAMD reordering schemes.

As shown in Figure 1, the `analysis` phase only run in the first iteration for the hybrid versions. The total execution time with the hybrid methods can be calculated as

$$T^{\text{total}} = T^{\text{analysis}}_{\text{CPU}} + T^{\text{factor}}_{\text{CPU}} + T^{\text{solve}}_{\text{CPU}} + T^{\text{analysis}}_{\text{GPU}} + (n-1) \times (T^{\text{factor}}_{\text{GPU}} + T^{\text{solve}}_{CPU} + T^{\text{transfer\_data}})$$

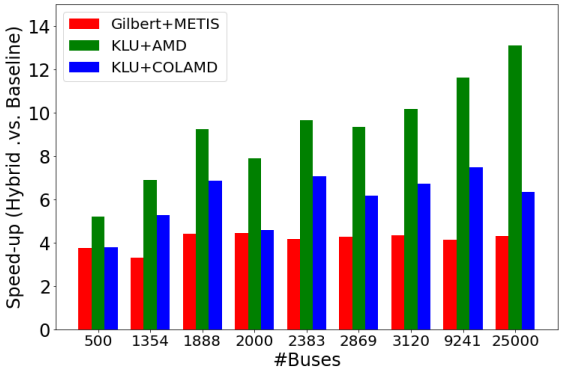

FIGURE 9. The speed-up of hybrid versions in comparison with the baseline CPU version on V100 GPU node.

where n is the number of iterations and $T^{\text{transfer\_data}}$ is the time to transfer data between host and device. N-R converges with different number of iterations for various cases. In this study we run fixed iterations of 6 in order to obtain better comparison for different solvers.

The speed-up of hybrid versions in comparison with the baseline CPU version on P100 GPU node is shown in Figure 7. Maximum speed-up of 9.6 can be obtained using the Gilbert's algorithm with METIS reordering for case of bus 25000. Figure 8 presents the comparison between the hybrid versions with CPU KLU solver. Maximum speed-up of 4.8 can be obtained using the KLU with AMD reordering for case of bus 25000. Figures 9 and 10 show the same comparison but on the V100 GPU node. Maximum speed-up of 13.1 can be obtained using the KLU algorithm with AMD reordering for case of bus 25000. Though the GLU with Gilbert's algorithm has best performances with `analysis` and factor phases, the speed-up of the hybrid version using the method does not achieve maximum. The reason is that the Gilbert's algorithm with METIS takes much times on the `analysis` phase

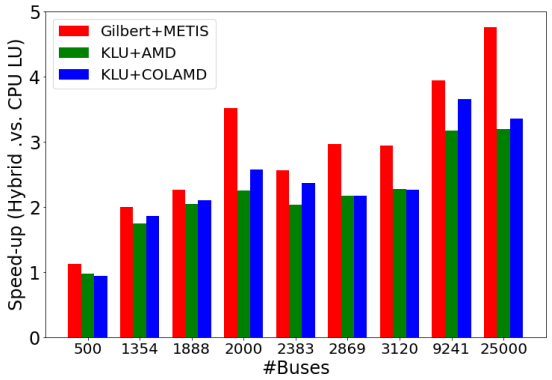

FIGURE 10. **The speedup of hybrid versions in comparison with the KLU CPU version on V100 GPU node.**

running on CPU in first iteration. However, the hybrid version with Gilbert's algorithm might more suit for power-flow problems with slower convergence rate since the performance increases with the number of iterations.

V. CONCLUSION AND FURTHER WORK

This study shows that the hybrid versions, which combine both the CPU and GPU for the computation, performed better than all the other versions developed and evaluated in this paper. When comparing the best performing hybrid version with the baseline CPU version, both running the network with 25000 buses, a speedup factor of 13.1 is achieved. When comparing the best performing GPU version with the KLU CPU version, a speedup factor of 4.8 with 25000 buses is achieved.

Based on the results, the conclusion can be drawn that the hybrid versions are the best suited version for N-R applied to large scale power-flow problems. The best performances can be achieved using the hybrid versions for all cases.

The different versions presented in this study could be evaluated further by executing them on a larger number of hardware platforms and with larger problems. It would be interesting to have a comparison between the versions proposed in this paper, especially the hybrid versions and with other existed algorithms for direct solvers to see which linear solver has the best performance when it comes to the power-flow problem. To further evaluate the performance of the hybrid versions, the hybrid versions could be compared to different iterative techniques to solve the linear equations.

ACKNOWLEDGMENT

S. S. Author would like to thank Magnus Johansson, Tobias Fendin, and Joonas Pesonen for their valuable contributions to this article.

REFERENCES

[1] Y. Zhou, F. He, and Y. Qiu, "Dynamic strategy based parallel ant colony optimization on GPUs for TSPs," *Sci. China Inf. Sci.*, vol. 60, no. 6, Jun. 2017, Art. no. 068102.

[2] Y. Zhou, F. He, N. Hou, and Y. Qiu, "Parallel ant colony optimization on multi-core SIMD CPUs," *Future Gener. Comput. Syst.*, vol. 79, no. 2, pp. 473–487, Feb. 2018.

[3] N. Hou, F. He, Y. Zhou, and Y. Chen, "An efficient GPU-based parallel tabu search algorithm for hardware/software co-design," *Frontiers Comput. Sci.*, vol. 14, no. 5, Oct. 2020, Art. no. 145316 .

[4] C. Guo, B. Jiang, H. Yuan, Z. Yang, L. Wang, and S. Ren, "Performance comparisons of parallel power flow solvers on GPU system," in *Proc. IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, Aug. 2012, pp. 232–239.

[5] J. Singh and I. Aruni, "Accelerating power flow studies on graphics processing unit," in *Proc. Annu. IEEE India Conf. (INDICON)*, Dec. 2010, pp. 1–5.

[6] *CUDA (Compute Unified Device Architecture)*. Accessed: Mar. 20, 2021. [Online]. Available: https://developer.nvidia.com/cuda-zone

[7] X. Li, F. Li, H. Yuan, H. Cui, and Q. Hu, "GPU-based fast decoupled power flow with preconditioned iterative solver and inexact Newton method," *IEEE Trans. Power Syst.*, vol. 32, no. 4, pp. 2695–2703, Jul. 2017.

[8] L. Razik, L. Schumacher, A. Monti, A. Guironnet, and G. Bureau, "A comparative analysis of LU decomposition methods for power system simulations," in *Proc. IEEE Milan PowerTech*, Jun. 2019, pp. 1–6.

[9] K. He, S. X. D. Tan, H. Wang, and G. Shi, "GPU-accelerated parallel sparse LU factorization method for fast circuit analysis," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 3, pp. 1140–1150, Mar. 2016.

[10] W.-K. Lee, R. Achar, and M. S. Nakhla, "Dynamic GPU parallel sparse LU factorization for fast circuit simulation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 11, pp. 2518–2529, Nov. 2018.

[11] S. Peng and S. X. D. Tan. (2019). *GLU3.0: Fast GPU-Based Parallel Sparse LU Factorization for Circuit Simulation*. [Online]. Available: http://arxiv.org/licenses/nonexclusive-distrib/1.0

[12] P. Schavemaker and L. van der Sluis, *Electrical Power System Essentials*. Hoboken, NJ, USA: Wiley, 2008.

[13] M. Albadi, "Power flow analysis," in *Computational Models in Engineering*, K. Volkov, Ed. Rijeka, Croatia: IntechOpen, 2020, ch. 5, doi: 10.5772/intechopen.83374.

[14] A. Vijayvargia, S. Jain, S. Meena, V. Gupta, and M. Lalwani, "Comparison between different load flow methodologies by analyzing various bus systems," *Int. J. Electr. Eng.*, vol. 9, no. 2, pp. 127–138, 2016.

[15] A. J. Flueck and H.-D. Chiang, "Solving the nonlinear power flow equations with a Newton process and GMRES," in *Proc. IEEE ISCAS*, May 1996, pp. 657–660.

[16] T. Ohtsuki, L. K. Cheung, and T. Fujisawa, "Minimal triangulation of a graph and optimal pivoting order in a sparse matrix," *J. Math. Anal. Appl.*, vol. 54, no. 3, pp. 622–633, Jun. 1976.

[17] G. Karypis and V. Kumar, "METIS: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices," Dept. Comput. Sci., Army HPC Res. Center Minneapolis, Univ. Minnesota, Tech. Rep., Sep. 1998. [Online]. Available: https://www.bibsonomy.org/bibtex/239641dbce7e631fddff1d1250939300a/dhruvbansal

[18] P. Agarwal and E. Olson, "Variable reordering strategies for SLAM," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Oct. 2012, pp. 3844–3850.

[19] W. M. Chan and A. George, "A linear time implementation of the reverse Cuthill–McKee algorithm," *BIT*, vol. 20, no. 1, pp. 8–14, Mar. 1980, doi: 10.1007/BF01933580.

[20] J. R. Gilbert and T. Peierls, "Sparse partial pivoting in time proportional to arithmetic operations," *SIAM J. Sci. Stat. Comput.*, vol. 9, no. 5, pp. 862–874, Sep. 1988.

[21] T. A. Davis and E. P. Natarajan, "Algorithm 907: KLU, a direct sparse solver for circuit simulation problems," *ACM Trans. Math. Softw.*, vol. 37, no. 3, pp. 1–17, Sep. 2010, doi: 10.1145/1824801.1824814.

[22] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format," in *Proc. SC*, Nov. 2014, pp. 769–780.

[23] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas, "MATPOWER: Steady-state operations, planning, and analysis tools for power systems research and education," *IEEE Trans. Power Syst.*, vol. 26, no. 1, pp. 12–19, Feb. 2011.

[24] G. Guennebaud and B. Jacob. (2010). *Eigen V3*. [Online]. Available: http://eigen.tuxfamily.org

[25] J. Cheng, M. Grossman, and T. McKercher, *Professional CUDA C Programming* (EBL-Schweitzer). Hoboken, NJ, USA: Wiley, 2014. [Online]. Available: https://books.google.se/books?id=q3DvBQAAQBAJ

MANOLO D'ORTO received the M.S. degree in computer science from the KTH Royal Institute of Technology, Sweden. He is currently working with SiATM as a Software Engineer. He is also developing air traffic management systems.

SVANTE SJÖBLOM received the M.S. degree in electrical engineering from Uppsala University, Sweden. He joined the Swedish Transmission System Operator Svenska Kraftnät, in 2016, where he is currently working in the development of software for power system analysis.

LUNG SHENG CHIEN received the B.S. and M.S. degrees from the Department of Computer Science, National Tsing Hua University, in 2003 and 2005, respectively, and the Ph.D. degree from the Department of Mathematics, National Tsing Hua University. He is currently a Software Engineer with NVIDIA, working on CUSOLVER library. His current research interests include sparse linear solver and dense symmetric eigenvalue solver.

LILIT AXNER received the M.B.A. and Ph.D. degrees in computer science. She was a HPC Advisor with the SURFsara the National Supercomputing and e-Science Support Center, Netherlands. She was a Project Manager with the PDC Center for High Performance Computing, Stockholm, Sweden. She was also co-leading the PDC Business Unit. She has been coordinating Swedish National Infrastructure centers within PRACE infrastructure, since 2010. She is currently the Director of the EuroCC National Competence Center Sweden (ENCCS). She have been engaged in projects, such as ScalaLife (EU FP7) as a Manager, different work packages and tasks leader within projects like HPC Eurpa3 (H2020), FocuCoE (H2020), ETP4HPC, DEISA, and DEISA2.

JING GONG received the M.Sc. degree in scientific computing from the KTH Royal Institute of Technology, Sweden, in 2003, and the Ph.D. degree in scientific computing from Uppsala University, in 2007. He is currently working as a Researcher with the PDC Center for High Performance Computing, KTH. He has many years of experience in high performance computing. He has been involved in several European exascale and e-infrastructure projects. His current research interests include programming environments and modeling for parallel and distribute computing with a focus on applications of computational fluid dynamics.