Fast Batched Solution for Real-Time Optimal Power Flow With Penetration of Renewable Energy

James Peavy, Mert Akyurekli and Vishal Sivaraman

ARTICLE INFO

Keywords:
CUDA
OPF
PDIPM
CoDiPack
cuSolver
Sparse Systems
QP

ABSTRACT

Real Time Optimal Power Flow problems are increasing in relevance because of the rise in the use of renewable energy generators in power systems. Because of uncertainties in the renewable energy sources, with solar and wind being the focus of the paper, it becomes important to re-solve the power flow problem using the most recent data to predict the future in shorter horizons. This paper proposes a novel way to find a hot-start to the problem by running the OPF problem for multiple possible future scenarios so that we get close to the actual reality in one of them when the uncertainty is realised. The paper describes a parallelised Primal Dual Interior Point Method as a solution method for the problem. The parallelised method ensures the runs complete in a reasonable timescale as is required by RTOPF problems. We did an extensive literature review, attempted the CPU version of the algorithm and outlined our plans on how we are going to proceed further.

1. Introduction and Literature Review

The Optimal Power Flow problem was introduced in Carpentier (1962). It is defined as "the determination of the complete state of a power system corresponding to the best operation within security constraints" in Carpentier (1979). In simpler terms, it ensures that the electricity generation companies meet the forecasted demand with the least cost and at the same time ensure generation and transmission safety constraints are taken care of. For example (as we will see later in the problem formulation), there will be constraints on how much you can ramp up production and on how much you can generate to avoid overloading the power lines. These have been traditionally solved with derivative-based methods (including KKT conditions) as described in the review (Carpentier (1979)).

Real-Time Optimal Power Flow (RTOPF) problem was introduced in Tang, Dvijotham and Low (2017). It was established in the context to handle multiple energy resources where we need to handle lot of uncertainties - especially in the presence of renewable energy generators. The essential difference between optimal power flow and real time optimal powerflow is the fact that we are focused on ensuring the operating constraints are satisfied but we are willing to concede on the objective (which is the power generation cost) in order to ensure we quickly solve the problem. The work used quasi-Newton methods to compute sub-optimal updates to the solution so that it can operate under shorter timescales. Specifically it used the Limited memory BFGS algorithm developed by Liu and Nocedal (1989). The implementation was in *MATLAB*. This method of solving has been termed as "OPF-based energy management" in Mohagheghi, Alramlawi, Gabash and Li (2018). The article also talks about a classification of methods called "constraint satisfaction-based real-time energy management" and highlights the salient features of these two broad classes. As the name suggests the constraints satisfaction class focuses entirely on finding a basic feasible solution for the given constraints at hand. With the renewable energy generators being an important integral part of modern grid networks as the we move towards using greener sources of energy it becomes crucial to address this problem to ensure optimal operations (Huang and Dinavahi (2018a)). The natural questions that arise in one's mind now is,

- can we make the solution better (more optimal - in terms of the costs or in terms of set-point tracking)?
- can we compute the solution faster?

Researchers have attacked both these questions (often only one at a time). Woo, wu, Park and Roh (2020) use a Deep Reinforcement Learning method called TD3 which itself is an extension of Deep Deterministic Policy Gradient (DDPG) (Silver, Lever, Heess, Degris, Wierstra and Riedmiller (2014)). Here, they use a Gaussian noise on the loads to model uncertainties of the renewable energy generators. A safe Deep Reinforcement Learning method has been proposed in Wu, Chen, Lai and Zhong (2023) where by delinking the constraint violation penalty and the economic cost reward, they remediate the reward sparsity issue. Once the model is trained, the calculations for power flow are two orders of magnitude faster compared to solving the RT-OPF optimization problem. Solving of optimization problems can be accelerated by using Graphical Processing Units (GPUs). The amount of speed-up that can be done is constrained by by Amdahl's law (Amdahl (1967)):

$$S = \frac{1}{F_S + \frac{F_P}{P}} \tag{1}$$

where,

- S is the Speed-up
- P is the number of processors
- F_S is the fraction of serial work
- F_P is the fraction of parallel work

Interior-point methods are a class of solution methods used in solving the RT-OPF. They use barrier functions in order to ensure the feasibility of the inequality constraints (Mohagheghi et al. (2018)). Oliveira, Oliveira, Pereira, Honório, Silva and Marcato (2015) use a safety barrier parameter δ to enhance the convergence of such methods, and even if the problem is infeasible, they provide directions for operators to bring it back to feasible space. Predictor-corrector interior-point method is used by Rakai and Rosehart (2014) for solving the RTOPF problem where the most computationally expensive step of the algorithm - matrix factorization is parallelised. They use the parallel computing toolbox of MATLAB, so there will be some run time overheads since they did not use a low level language like C to program the logic. Su, He, Liu and Wu (2020) introduce a parallelised implementation of Newton-Raphson method for solving the problem. They analyze the sparsity of the problem and identify areas where specific algorithms which work well for sparsity and GPU vectorization can be applied. For larger systems ($>$ 10000 buses), Li, Li, Yuan, Cui and Hu (2017) achieved $\approx 3x$ speedups using GPUs. It integrates the inexact Newton methods and the preconditioned iterative conjugate gradient method to solve the OPF problem. The preconditioning logic is ported to GPUs. A meta-heuristics based method using Particle Swarm Optimization is proposed in Roberge, Tarbouchi and Okou (2016). Given that it is a highly parallelizable algorithm and the author's exploited the sparseness of the problem well to provide $\approx 17x$ speedups compared to CPU. Wang, Wende-von Berg and Braun (2021) presents a nice heterogeneous algorithm which uses both CPU and GPU for Newton-Raphson (specifically in LU factorization/re-factorization steps of the matrices).

The paper we are implementing - "Fast Batched Solution for Real-Time Optimal Power Flow With Penetration of Renewable Energy" (Huang and Dinavahi (2018a)) presents a two-pronged improvement of both reducing wait time and increasing accuracy of the solution which is not usually the case. This is achieved by parallelizing the solution procedure which is an interior-point method. The uncertainty of renewable energy is dealt with by generating multiple plausible future scenarios. Although this prediction aspect is not relevant to the course, we want to make a couple of brief mentions. Mohagheghi, Gabash and Li (2017) develop the method this work uses for generating scenarios for wind energy. The error is assumed to be Beta distributed and scenarios are sampled from that. The distribution is calibrated using historical data. However, they use the BONMIN solver in GAMS to solve the optimization problem. Solar energy is modelled by a bi-modal distribution in Reddy (2017) and they use Genetic Algorithm to solve the OPF problem. We solve each of these scenarios and maintain a look-up table. Once the actual scenario happens, we quickly resolve the problem by using the closest available scenario as a hot-start. Since all this is happening in real-time, the challenge is to solve multiple scenarios as quickly as possible. The exact procedure is described in the methods section.

We also found some more recent work that cite this work, improve some aspects of it - either extending the application to more general renewable energy systems or adapt the parallel implementation to specific systems. Mohagheghi, Alramlawi, Gabash, Blaabjerg and Li (2020) follows a similar scenario generation and maintains a look-up table for hot-starting the actual problem but sets up a Mixed Integer Non-Linear Program (MINLP) instead of a quadratic program. Example of integer variables include number of charge-discharge cycle of the battery. GAMS solvers are used to solve the problems. The authors of this work have also developed GPU accelerated Newton-Raphson methods in Huang and Dinavahi (2018b). There has also been a multi-objective problem formulated with a solution strategy in Chen, Qian, Zhang and Sun (2019) where other aspects of power generation like emissions are included as a part of the objective. Some other recent work which could be of interest include Shin, Pacaud and Anitescu (2024) which introduces a non-linear program with a completely parallelisable solution framework achieving 5x speedups. They are able to do this by porting sparse automatic differentiation (AD) and sparse linear solver routines to GPUs.
This final report has been divided into 8 sections. The second section elaborates on the problem formulation and solution strategy along with description of our code. The third section briefly touches upon the methodology prescribed in the paper and talks about our solution using existing Python and MATLAB libraries. The fourth section discusses our C implementation details at length. The fifth section shares the results we obtained and how we stand with respect to the original work. The sixth section presents our learnings and points of improvement. The last two sections share a sample case data and points to our GitHub link which has the instructions to setup and run our code.

2. Problem Formulation

The following is the optimization problem set up for the RTOPF.

Objective Function:

$$\min F = \sum\nolimits_{g\in\mathcal{N}_g} \left[a_g \left(P_g^G \right)^2 + b_g P_g^G + c_g \right]. \tag{2}$$

Nodal Power Balance Constraints

$$P_i^G - P_i^D = V_i \sum_{j\in\mathcal{N}_b} V_j (G_{ij}\cos\theta_{ij} + B_{ij}\sin\theta_{ij}), \tag{3}$$

$$Q_i^G - Q_i^D = V_i \sum_{j\in\mathcal{N}_b} V_j (G_{ij}\sin\theta_{ij} - B_{ij}\cos\theta_{ij}). \tag{4}$$

Generator Capacity Constraints

$$P_g^{G,min} \le P_g^G \le P_g^{G,max}, \tag{5}$$

$$Q_g^{G,min} \le Q_g^G \le Q_g^{G,max}. \tag{6}$$

Ramp Rate Constraints

$$P_g^{G0} - R_g^{G,down} \leq P_g^G \leq P_g^{G0} + R_g^{G,up}. \tag{7}$$

Voltage Deviation Constraints

$$V_i^{min} \leq V_i \leq V_i^{max}. \tag{8}$$

Line Security Constraints

$$f_{ij}^{min} \leq f_{ij} \leq f_{ij}^{max}. \tag{9}$$

which is effectively:

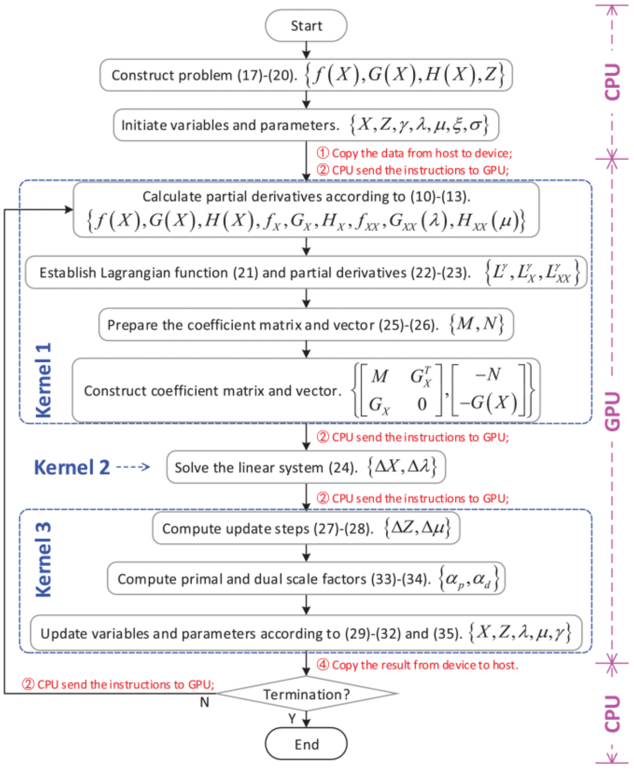
$$\theta_{ij}^{min} \leq \left(\theta_{ij} = \theta_i - \theta_j\right) \leq \theta_{ij}^{max}. \tag{10}$$

where,

- $\mathcal{N}_b, \mathcal{N}_l$ Set of buses and lines
- $\mathcal{N}_g, \mathcal{N}_r$ Set of buses where thermal and renewable generators are integrated.
- a_g, b_g, c_g Coefficients of the quadratic cost function of generator g.
- P_g^G, Q_g^G Active and reactive power output of thermal generator g.
- P_r^R, Q_r^R Active and reactive power output of REG r.
- P_i^D, Q_i^D Active and reactive power demand of bus i.
- G_{ij}, B_{ij} Transfer conductance and susceptance between buses i and j.
- V_i, V_j Voltages magnitude at node bus i and j.
- P_g^{G0} Active power output of generator g in the previous sub-interval.
- f_{ij} Power flow on line ij.
- θ_{ij} Voltage angle difference between buses i and j.
- θ_i, θ_j Voltage angle at node bus i and j.
- ${}^{down}, {}^{up}$ Ramp down and up limits of thermal generator.
- ${}^{min}, {}^{max}$ Lower and upper limits of specified variables

The entire formulation has been borrowed from the paper we are working on (Huang and Dinavahi (2018a)).
The decision variables here are P^G, Q^G, V_i and θ_i. It is important to note that some of these values must be computed using data not shown in this paper, specifically the G_{ij} and B_{ij} values. We opted to compute both of these values ourselves using the physical relations between G_{ij} and B_{ij} using the following formula, with X_{ij} being the Line Reactance and R_{ij} being the Line Resistance:

$$G_{ij} = \frac{R_{ij}}{\sqrt{R_{ij}^2 + X_{ij}^2}} \tag{11}$$



Start
Construct problem (17)-(20). $\{f(X), G(X), H(X), Z\}$
Initiate variables and parameters. $\{X, Z, \gamma, \lambda, \mu, \xi, \sigma\}$
① Copy the data from host to device;
② CPU send the instructions to GPU;
Calculate partial derivatives according to (10)-(13). $\{f(X), G(X), H(X), f_X, G_X, H_X, f_{XX}, G_{XX}(\lambda), H_{XX}(\mu)\}$
Establish Lagrangian function (21) and partial derivatives (22)-(23). $\{L^\gamma, L_X^\gamma, L_{XX}^\gamma\}$
Prepare the coefficient matrix and vector (25)-(26). $\{M, N\}$
Construct coefficient matrix and vector. $\left\{\begin{bmatrix} M & G_X^T \\ G_X & 0 \end{bmatrix}, \begin{bmatrix} -N \\ -G(X) \end{bmatrix}\right\}$
Kernel 1
② CPU send the instructions to GPU;
Kernel 2
Solve the linear system (24). $\{\Delta X, \Delta\lambda\}$
② CPU send the instructions to GPU;
Kernel 3
Compute update steps (27)-(28). $\{\Delta Z, \Delta\mu\}$
Compute primal and dual scale factors (33)-(34). $\{\alpha_p, \alpha_d\}$
Update variables and parameters according to (29)-(32) and (35). $\{X, Z, \lambda, \mu, \gamma\}$
④ Copy the result from device to host.
② CPU send the instructions to GPU;
Termination?
N
Y
End
CPU
GPU
CPU

Figure 1: Kernel Setup

$$B_{ij} = \frac{X_{ij}}{\sqrt{R_{ij}^2 + X_{ij}^2}} \tag{12}$$

These values were computed on python before the data was loaded onto C by using the following code:

```
R = 'Resistance'
X = 'Reactance'

Case14BranchDF['G'] = round(Case14BranchDF[R] / sqrt(
    Case14BranchDF[R]**2 + Case14BranchDF[X]**2), 3)
Case14BranchDF['B'] = round(Case14BranchDF[X] / sqrt(
    Case14BranchDF[R]**2 + Case14BranchDF[X]**2), 3)
Case14BranchDF.to_csv('Case 14 Branch.csv', index=False)
```

The above optimization problem is solved for multiple scenarios generated using the currently available renewable energy data. At a given state, the paper forecasts 1024 possible futures for generation of renewable energy: P_i^D. There are therefore 1024 different RTOPF problems to solve. This is done in a parallel fashion, as seen in figure 1. The algorithm used is the Primal-Dual Interior Point Method (PDIPM) as mentioend in Wang, Murillo-Sanchez, Zimmerman and Thomas (2007). The inequality constraints are converted to equality constraints by bringing in the slack vector Z.

3. Methodology in the Paper

- **Kernel-1**: Calculation of partial derivatives of the objective and constraints. The derivatives are analytically computed and implemented in the code to

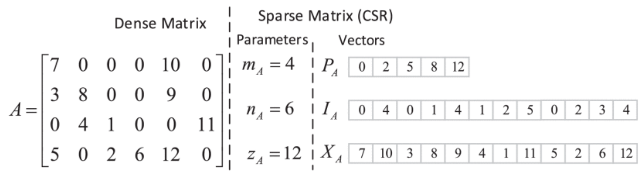

Dense Matrix
Sparse Matrix (CSR)
Parameters
Vectors
$A = \begin{bmatrix} 7 & 0 & 0 & 0 & 10 & 0 \\ 3 & 8 & 0 & 0 & 9 & 0 \\ 0 & 4 & 1 & 0 & 0 & 11 \\ 5 & 0 & 2 & 6 & 12 & 0 \end{bmatrix}$
$m_A = 4$
$n_A = 6$
$z_A = 12$
P_A 0 2 5 8 12
I_A 0 4 0 1 4 1 2 5 0 2 3 4
X_A 7 10 3 8 9 4 1 11 5 2 6 12

Figure 2: Showcase of CSR format. P_A being the Row Indexes, I_A being the Column Indexes, and X_A being the non-zero data, all in the Classic CSR format

do these computations. Further, the Lagrangians are computed and the matrix is generated. Additionally, as our matrices are stored in CSR format, matrix concatenation is a more complicated business. Pseudo-code in the original paper is to be translated into parallel code in kernel 1 as a part of the data preparation.

- **Kernel-2**: This is the most compute intensive set as it involves matrix inversion (solving of a linear system of equations). This is accomplished using the CuSolver.
- **Kernel-3**: The solved linear equation is to compute and evaluate the update steps for the decision variables for the next iteration

This process is repeated till the following occur:

- The (scaled) constraints are followed with a tolerance of ϵ
- The (scaled) Lagrangian function is close enough to zero with tolerance of ϵ
- The scaled change in objective is within ϵ

For this paper, we have tasked ourselves with the understanding of the basic problem and its implementation using cuSolver, therefore we will assume no REG data is yet added. Given the context of the course we believe the scenario generation itself is of no particular interest to us, therefore basic methods of scenario generation will be used and further discussed in the Conclusion. Our focus is going to be on implementing the optimization algorithm in parallel and comparing the GPU speed-ups.

We have solved the basic problem in cvxpy in order to get an example on both the proper form of the solution, which is not discussed in the original paper, and the time increase we would expect for each version of the basic problem (14, 57, 118, and 300 Bus cases). We also used the MATPOWER package in MATLAB developed by Zimmerman, Murillo-Sánchez and Thomas (2011a). Specifically we used two interior point methods present in the package - MIPS (MATPOWER Interior Point Solver - Wang et al. (2007)) and PDIPM (Primal Dual Interior Point Method - Zimmerman, Murillo-Sánchez and Thomas (2011b)). It is important to note that PDIPM is the algorithm is used in the paper. These results are shown in the next section.

4. Our GPU Implementaton

4.1. Derivatives Methods

We used CoDiPack: Fast gradient evaluation in C++ based on Expression Templates to compute the first and second derivatives (M. Sagebaum (2019)). The first derivative was numerically computed using the forward mode (algorithmic differentiation), while the second derivative of the Lagrange function was computed using the Hessian and Jacobian. When noting the usage of the CoDiPack, for all derivatives, we were required to pass in values of a custom type called RealForward (in the future we refer to it as Real). This type required us to perform extra translations between types, this is further discussed in section 6.3. Given this, we had to perform many of these non-trivial operations on our equivalent to the kernel 1 in figure 1, with the difference being that these were performed on the CPU rather than the GPU (the RealForward type not having a robust CUDA variant). We were tasked with finding the first and second partial derivatives of multiple functions, given a real vector $X = [x_1, x_2, \dots, x_n]^T$, $f(X) : \mathbb{R}^n \to \mathbb{R}$

$$f_X = \frac{\partial f}{\partial X} = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right], \tag{13}$$

$$f_{XX} = \frac{\partial^2 f}{\partial X^2} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}. \tag{14}$$

$G(X) : \mathbb{R}^n \to \mathbb{R}^p$

$$G_X = \frac{\partial G}{\partial X} = \begin{bmatrix} \frac{\partial G_1}{\partial x_1} & \cdots & \frac{\partial G_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial G_p}{\partial x_1} & \cdots & \frac{\partial G_p}{\partial x_n} \end{bmatrix}. \tag{15}$$

$$G_{XX}(\lambda) = \frac{\partial ({G_X}^T \lambda)}{\partial X}. \tag{16}$$

$H(X) : \mathbb{R}^n \to \mathbb{R}^q$

$$H_X = \frac{\partial H}{\partial X} = \begin{bmatrix} \frac{\partial H_1}{\partial x_1} & \cdots & \frac{\partial H_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial H_q}{\partial x_1} & \cdots & \frac{\partial H_q}{\partial x_n} \end{bmatrix}. \tag{17}$$

$$H_{XX}(\lambda) = \frac{\partial ({H_X}^T \mu)}{\partial X}. \tag{18}$$

Using these values, you can compute the Lagrangian Function:

$$L^\gamma = f(X) + \lambda^T G(X) + \mu^T (H(X) + Z) - \gamma \sum_{i=1}^{q} \ln(Z_i).$$

(19)

And we can compute its derivatives:

$$L_X^\gamma = f_X + G_X\lambda + H_X\mu, \tag{20}$$

$$L_{XX}^\gamma = f_{XX} + G_{XX}(\lambda) + H_{XX}(\mu). \tag{21}$$

4.2. Derivatives Implementation

Given the above equation's structures, we were able to alter the way in which one can compute these values. Given how one may use CoDiPack, we were able to compute the Lagrangian second derivative without the computation of f_{XX}, $G_{XX}(\lambda)$, or $H_{XX}(\mu)$. Given this, we were able to then compute the M and N values for the setup of the cuSolver problem:

$$\begin{bmatrix} M & {G_X}^T \\ G_X & 0 \end{bmatrix} \begin{bmatrix} \Delta X \\ \Delta\lambda \end{bmatrix} = \begin{bmatrix} -N \\ -G(X) \end{bmatrix}, \tag{22}$$

where

$$M = L_{XX}^\gamma + {H_X}^T[Z]^{-1}[\mu]H_X, \tag{23}$$

$$N = {L_X^\gamma}^T + {H_X}^T[Z]^{-1}(\gamma e + [\mu]H(X)). \tag{24}$$

The exact implementation details are covered in the next subsection.

4.3. Code Structure

This subsection will cover in-depth how we have structured our code. Apart from the standard libraries, and cuSolver, you will need the following open-source libraries:

- CoDiPack (as mentioned earlier)
- fast-csv-parser
- BLAS library

We have provided detailed installation steps in our GitHub repository.
Short description of important files:

- driver.cpp: the main function which solves the optimization problem
- solverfile.cu: computes solution for a given sparse matrix linear system
- read_data.h: loads data into global variables accessible across functions
- utils_derivs.h: contains the main utilities, mathematical functions (including Lagrangian) and their first derivatives
- second_derivs.h: setup to compute the second derivative of the Lagrangian

The `driver.cpp` file which is the driver file/entry-point for solving the RTOPF problem. It first calls the `load_data()` function from `read_data.h` library. We have opted to save all the parameter related to the scenario as global constants to avoid creating multiple copies when being passed around. And currently, since the first and third kernels are not parallelised, this will not cause affect run time because of slow global memory access speeds. We converted the MATLAB data of MATPOWER into multiple csv files using a simple Python code. Each of them correspond to a different aspect of the data - branch data, generator capacities, REG capacities and bus data. These files are read using the `load_data()` function and each of the columns are stored in a separate `vector<double>`.

4.3.1. f, G and H

All the required optimization formulation related functions are defined in `utils_derivs.h`. The implementation of the objective/cost function is relatively simple. The bound constraints on Powers and Voltages is again simple to implement. For the bound on voltage angle we were able to figure out θ_i and θ_j of each branch (ij) using the indices of the buses as present in FromBus and ToBus to access θ. The equality constraints requires us to use data along each branch. However, the admittance data is given with FromBus, ToBus and admittance in separate columns. So before we write the balances we first constructed a map from branch (ij) to their respective conductance, susceptance and voltage angle. Once we did that it is once again straightforward to construct the equalities.

4.3.2. First Derivative

To enable this computation we need to strictly use pointers (equivalently arrays) of type `codi::RealForward` (henceforth referenced to as `Real`). We believe the regular double operations like "*", "+", "/", "-" are overloaded in the `Real` class, which also has methods like gradient and setting seeding for the gradient. To compute the derivatives, we need to ensure all our inputs and outputs are of `Real` type. We have an intuition of the working as that, when you call the derivative function, it computes the finite differences somehow using the overloaded operators. And since all operations involve the `Real` class objects, the computations are kept track of and the derivative is computed. We convert this derivative to a BLAS vector or a double* as the case may need. We compute the derivative of f, G, H and the Lagrangian L.

4.3.3. Second Derivative

We had to shift our logic completely for the second derivative, though we might've been able to avoid rewriting functions using `<template typename>`. We shifted because the the methods of evalHessian was defined to use with different kind of functions, and not necessarily the Real type. A quirk was needing a wrapper which accepted the input and output as vectors, and then call the actual function which accepts pointers of any type `<template typename>`. Having the network parameters as global constants greatly simplified the coding in this part, and in fact it is the need to compute Lagrangian (which needs literally all of our data) and its second derivative that inspired us to set them as global constraints in the first place. But the good thing is we just needed the second derivative of the Lagrangian and not of f, G, or H. We directly computed its second derivative (and did not build it from derivatives of f, G, H). This helps us avoid lot of matrix multplications and we also believe it will reduce the estimation error because the Lagrangian has just one output while G and H have multiple outputs, resulting in computing 3D hessians for those functions and then projecting them down to 2D by multiplying them with the Lagrange coefficients.

4.3.4. Solving and Updating values

Once we setup the derivatives rest of the operations are quite straightforward albeit the frequent type changes. We used BLAS to perform matrix multiplications and so forth. We wrote our own code for matrix concatenation to setup the cusolver matrix. We also wrote our own code for converting matrix from a column major form to CSR format. Interestingly, NVIDIA discontinued the function which did that in the newer CUDA versions, we are not sure why. Explanation on using cuSolver to solve the sparse linear system is given in the subsection 4.4. After we get the solution from cuSolver as double pointers, we convert them to BLAS matrices and then perform the update operations. It is fairly straightforward to check the termination conditions.

4.4. GPU-Accelerated Computation Setup

The computational challenges presented by the RTOPF problem, particularly with the integration of renewable energy resources, necessitate highly efficient and scalable solutions. To address this, we utilized NVIDIA's cuSolverSp and cuSparse libraries to efficiently manage sparse matrix operations, which are crucial for the large, sparse systems typical in power flow calculations. This section details the specific steps taken, the structure of the implementation, and the observed performance benefits.

The cuSolverSp library offers robust methods for sparse matrix decompositions essential for the linear algebra operations involved in our approach, while cuSparse assists in handling sparse matrix formats and operations effectively.

4.4.1. Matrix Setup and Decomposition

The matrix A, representing the Jacobian in the Newton-Raphson iterations of the PDIPM approach, is sparse due to the network connectivity and typical sparsity patterns of power systems. The QR decomposition of this matrix, necessary for the step calculations in PDIPM, was efficiently performed using cuSolverSp's sparse QR decomposition functions:

4.4.2. Conversion to CSR Format

Initially, the dense matrix A is converted to Compressed Sparse Row (CSR) format, which is a storage-efficient way to maintain sparse matrices:

```
cusparseCreateMatDescr(&descrA);
cusparseSetMatType(descrA, CUSPARSE_MATRIX_TYPE_GENERAL);
cusparseSetMatIndexBase(descrA, CUSPARSE_INDEX_BASE_ONE);
```

4.4.3. Performing QR Decomposition

The QR decomposition is then executed using the following cuSolverSp functions:

```
cusolverSpCreate(&cusolverH);
cusolverSpXcsrqrAnalysisBatched(cusolverH, m, m, nnzA,
    descrA, d_csrRowPtrA, d_csrColIndA, info);
cusolverSpDcsrqrsvBatched(cusolverH, m, m, nnzA, descrA,
    d_csrValA, d_csrRowPtrA, d_csrColIndA,
                        d_b, d_x, batchSize, info,
                            buffer_qr);
```

Here, `d_csrValA`, `d_csrColIndA`, and `d_csrRowPtrA` are pointers to the matrix `A` in CSR format on the device. `batchSize` refers to the number of systems solved in parallel, enhancing throughput significantly.

4.4.4. System Solving Using cuSparse

Once we have Q and R from the QR decomposition, solving the linear system $Rx = Q^T b$ for x involves:

- Application of Q^T to b, performed within the function named `cusolverSpDcsrqrsvBatched`
- Direct solving of the triangular system, also handled by `cusolverSpDcsrqrsvBatched`.

These steps are integrated within the cuSolverSp library's batched sparse QR solver, which efficiently manages multiple decompositions and system solutions simultaneously.

4.4.5. Performance Optimization

To optimize the performance of the GPU-accelerated RTOPF solution, we intended to employ the following strategies:

- **Batch Processing:**
  - *Objective:* Handle multiple forecasted states simultaneously, reducing computational time and improving throughput.
  - *Implementation:* Utilized the batched versions of the QR decomposition and linear system solving routines provided by cuSolverSp to process multiple matrices in parallel.
- **Memory Management:**
  - *Objective:* Minimize the overhead associated with data transfers between the host (CPU) and the device (GPU), crucial for large-scale system matrices.
  - *Implementation:*
    - ∗ Used pinned (page-locked) memory to expedite the transfer of data.
    - ∗ Carefully managed the allocation and deallocation of GPU memory to prevent memory leaks and ensure efficient resource utilization.
- **Kernel Tuning:**

- *Objective:* Optimize parts of the algorithm not directly supported by cuSolverSp functionalities, specifically those involving non-standard data manipulations.
- *Implementation:*
  - Developed custom CUDA kernels for updating constraint matrices and operations involving CSR formats, commonly used in sparse matrix representations in power systems.
  - Tuned these kernels for optimal performance on the GPU, focusing on memory access patterns and thread block sizing.

We were able to employ only the memory management component of these ideas.
These enhancements are aimed at leveraging the full potential of GPU resources to achieve faster computation times and handle the increasing complexity and scale of modern power systems.

5. Results and Discussion

5.1. CPU implementation

Results using MIPS and PDIPM are present in table 2. Firstly, we observe that the optimal objective value attained is same in both cases, which is good. Secondly, we see PDIPM, the algorithm used in the paper, beats the other popularly used solver MIPS by an order of magnitude in some cases. This is the expected line as commented on the MATPOWER user manual. PDIPM becomes better as we increase the bus sizes and hence the complexity of the system under consideration. The MATPOWER package code provides us a good starting point for us to get started with some of the aspects of the C implementation, especially for implementing the constraints and their derivatives.

5.2. GPU implementation

For the C implementation, we can see the sparsity plot of the matrix to be inverted in figure3. The bottom right part of the matrix is set to zero by the setup. The top right and bottom left represent the transpose of and the original derivative of G matrix. The top left part is the M matrix. Since we start off with λ and μ vectors being zero, in the first iteration this part is empty. It is also interesting to note that the same scenario gives different number of zeros each time you run it. We think it might be because of rounding off errors, so we tried setting bounds to identify zero elements (eg. $|a_{ij}| < 10^{-8}$ is identified as a zero element) but while that kind of reduced the randomness, it did not solve the problem entirely.
Currently, PACE gives a seg fault in the matrix prep stage. We identified the function where the error occurs, but we are not able to zero in on which line is causing the error. More strangely, the code runs fine locally, but unfortunately, we can't use CuSolver locally.
All the modules have been extensively tested separately, including the CUDA code. You might be able to find codes that help you do that in either "./cusolver_trials/" or in "./old_code/".

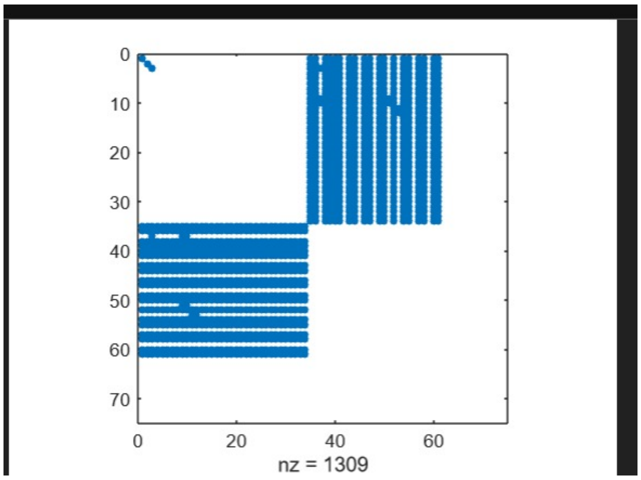


Figure 3: Sparsity Pattern

Table 1
Performance comparison of optimization algorithms SCS and OSQP

Case	Compile Time (s)		Solver Time (s)	
	SCS	OSQP	SCS	OSQP
14	0.268	0.443	0.017	0.0089
57	3.43	3.5	0.035	0.012
118	8.37	7.48	0.15	0.0322
300	77.3	67.57	0.43	0.22

6. Lessons Learned and Further Research

Lessons Learned

Given the nature of our problem, that being hard and requiring many extra computational steps in order to prepare the solution, we had to access many extra packages in order to ensure it was possible to achieve a solution. These were packages such as BLAS, CoDiPack, and FastCSVParser. In the course of our project, we encountered many type errors, given the nature of the packages we used, many of their types were incompatible. Given time constraints, we elected to solve these problems by the creation of many of our own functions via the `utils_derivs.h` header file. In addition, we created the `second_derivs` and `read_data.h` header files.

We learned quite a bit about the debugging and C coding processes as well.

6.1. Challenges Encountered

6.1.1. cuSolver and CUDA

The implementation of batched operations using cuSolver's batched QR decomposition functions posed several challenges, which are detailed below:

Table 2
Performance comparison of optimization algorithms MIPS and PDIPM

Case	Compute Time (s)		Optimal Objective ($/hr)	
	MIPS	PDIPM	MIPS	PDIPM
14	0.09	0.02	8081.53	8081.53
57	0.12	0.03	41 737.79	41 737.79
118	0.17	0.06	129 660.70	129 660.70
300	0.27	0.16	719 725.10	719 725.10

- **Complexity in Setup:** Properly setting up and managing arrays of device pointers was particularly challenging. This process required meticulous attention to detail to:
  - Avoid memory leaks which can cripple long-running computations.
  - Ensure correct execution across multiple scenarios simultaneously processed.
- **Debugging and Validation:** Debugging GPU code, especially that involving batched operations, proved to be significantly more complex than equivalent CPU-based code. Key issues included:
  - The non-intuitive nature of parallel and asynchronous executions typical in GPU computations.
  - Ensuring the correctness of all operations, particularly under edge cases for renewable scenarios, required extensive testing.
  - Validation against CPU-based implementations to verify accuracy and correctness, necessitating comprehensive comparative analysis and testing protocols.

These challenges highlighted the need for robust error handling and performance profiling strategies to effectively leverage GPU acceleration for RTOPF problems.

6.1.2. *Derivatives and Update Steps*

Designing the f, G and H functions needed a lot of intuition on using different datastructures (like maps) and how to implement them. Focus was on restricting the complexity to O(N^2) which is reasonable for matrices (but not necessarily optimal). We learnt a lot on using pointers, passing by value vs passing by reference, using global variables common across code files. We think understanding how the derivatives library works was a steep learning curve. We always started off small with simple mathematical functions whose values we can calculate by hand before we scaled it up for our massive problems having ≈ 70 variables at the least. We clearly understood from this, the need for code modularity. In fact, each sub-unit of our code has been well tested before being put in place. This helped save a lot of time as running the whole thing repeatedly when trying to debug will be a difficult and time-consuming task.

6.2. Previously Asked Questions in the Presentation

- "Quick walkthrough on scenario generation": Obtain data from NREL website and other sources, fit a distribution as mentioned in Mohagheghi et al. (2017), generate random future possibilities
- "The paper did the acceleration process on 3 kernels, do they have a reason for that number, do you agree and will be trying the same thing?": Yes, the 3 kernels represent most of the significant portions that can be parallelised.
- "what method do you plan to use to deal with those sparse matrices? Why use Cusolver for one of the kernels? What is the benefit of that?": CuSolver efficiently handles sparse matrices if we supply them in their CSR form.
- "How do you handle partial derivatives? Do you solve it numerically or use anylibraries?": We do it numerically using CoDiPack developed in TU Kaiserslautern.
- "What implications would the performance differences between MIPS and PDIPM have have for real-world applications?": We are not sure, but according to TSPOPF, PDIPM is significantly faster.

6.3. Further Research

In the process of doing our project, we found a few areas where further researchers may want to look at. We were unable to fully parallelized what the original paper showed as kernel 1 and kernel 3. This was due to a lack of understanding/testing with the RealForwardCUDA type of CoDiPack. When contacting the authors of the CoDiPack, they told us that there had been limited testing with regards to the first derivative using this data type, and no testing done with regards to the nested second derivative using this same type. Due to this, kernel 1 operations were unable to be fully parallelized. Now as for kernel 3, we believe that this could have been parallelized, but given the nature of the operations performed, and the size of data we were testing on (case 14 of the IEEE test cases), we deemed it of little importance to parallelize. With that being said, the parallelization of these type of operations (differentiation) with regards to optimization is a place of active and useful research. Below is a showcase for the first derivative of our Cost function f.

```
vector<vector<Real>> fX(Real* X, int XSize, Real* Y, int
     YSize, vector<double> a, vector<double> b, vector<
     double> c, vector<vector<Real>> result)
{
        for(int i = 0; i < XSize; ++i)
        {
                // Step 1: Set tangent seeding
                X[i].gradient() = 1.0;
                // Step 2: Evaluate function
                f(X, a, b, c, Y);
                for(int j = 0; j < YSize; ++j) {
                        // Step 3: Access gradients
```

$$\begin{bmatrix} M & {G_X}^T \\ G_X & 0 \end{bmatrix} \begin{bmatrix} \Delta X \\ \Delta \lambda \end{bmatrix} = \begin{bmatrix} -N \\ -G(X) \end{bmatrix},$$

Figure 4: Solution setup for cuSolver, note the matrix on the left

```
                result[j][i] = Y[j].getGradient();
            }
            // Step 4: Reset tangent seeding
            X[i].gradient() = 0.0;
        }
        return(result);
}
```

In addition to this, our group had to perform many operations that we found to be mathematically trivial, but (at least in C/C++) computationally non-trivial. These were operations such as matrix concatenation and diagonal matrix inversion. Given time constraints and a desire to keep our external package calls to a minimum, we did not pursue other linear algebra packages besides BLAS. With that being said, we do think that these operations have the ability to be easily implemented in BLAS or, if they are present, be documented in a better form for future research. Below is an example of the type of operations we had to perform to compute the matrix in Figure 3.

```
void CreateCuSolverMatrix(ublas::compressed_matrix<double>
     M, ublas::compressed_matrix<double> dGXdXMatrix,
    double* cuSolverMatrix) {
        int iterator = 0;
        for(int i = 0; i < (M.size2() + dGXdXMatrix.size1
            ()); ++i) { //cols
                for(int j = 0; j < (M.size1() +
                    dGXdXMatrix.size1()); ++j) { //rows
                        if(i < M.size2()) {
                                if(j < M.size1()) {
                                        cuSolverMatrix[
                                            iterator] = M
                                            (j,i); //
                                            upper left
                                            hand matrix
                                }
                                if(j >= M.size1()) {
                                        cuSolverMatrix[
                                            iterator] =
                                            dGXdXMatrix((
                                            j-M.size1()),
                                            i); // lower
                                            left hand
                                            matrix
                                }
                        }
                        if(i >= M.size2()) {
                                if(j < M.size1()) {
                                        cuSolverMatrix[
                                            iterator] =
                                            dGXdXMatrix((
                                            i-M.size2()),
                                            j); // upper
                                            right hand
                                            matrix NOTE:
                                            transposes
                                            the input
                                }
                                if(j >= M.size1()) {
                                        cuSolverMatrix[
                                            iterator] =
                                            0; // lower
                                            right hand
                                            matrix
                                }
                        }
                        iterator++;
                }
        }
}
```

Note that the above code results in a matrix in column-major form to ensure that cuSolver will ensure that the threads will achieve a coalesced access pattern. This is due to the nature of the $Ax = b$ form performing its multiplication and addition column-wise.

Due to the amount of libraries, with their own custom types, used in this project, we had to perform many type changing operations. We are certain that each of these operations brought about a large amount of inefficiencies in the equivalent kernel 1 and kernel 3 operations. Given more time and access to the source code of the original paper, we are certain that one could utilize the CoDiPack Automatic Differentiation functions to achieve some additional speedup and further code optimization.

Solving the linear systems with the CPU using Csparse would give a good benchmark to compare the cuSolver implementation with. We could not do it because of time constraints.

Lastly, we were unable to utilize the cuSolver Batched mode to solve multiple linear systems in parallel. This was due to both the time constraints, lack of source code to check against, and the inability to efficiently setup the multiple equations in our equivalent kernel 1 and kernel 3 operations.

A Note on Scenario Generation

In the original paper, the concept of scenario generation was brought up. As it was not the focus of the paper, it was only briefly mentioned. Due to this and our inability to access any source code associated with the original paper, our ability to generate scenarios was significantly hampered. With that being said, our solution was to generate the scenarios by utilizing the fact that all of the original thermal generators had some constraint on their upper and lower bound for both their Real and React power outputs. With that being said, we created scenarios by arbitrarily taking the bottom n number of generators (n being the number of REGs we would like to insert into our problem) and assigning them random Real and React power outputs constrained by their upper and lower bounds. The following code showcases how this was done in python.

```
NumCases = [2, 4, 10, 25]

Case14REGDF = Case14GeneratorDF[['Bus ID', 'Max Real Power
      Output', 'Min Real Power Output', 'Max React Power
     Output', 'Min React Power Output']]
Case14REGDF = Case14REGDF.tail(NumCases[0])
Case14REGDF = pd.concat([Case14REGDF]*1024, ignore_index=
     True)
```

```
Case14REGDF['Real Power Output'] = [np.random.randint(min,
      max) for min,max in zip(Case14REGDF['Min Real Power
      Output'], Case14REGDF['Max Real Power Output'])]
Case14REGDF['React Power Output'] = [np.random.randint(min
      , max) for min,max in zip(Case14REGDF['Min React
      Power Output'], Case14REGDF['Max React Power Output
      '])]
Case14GeneratorDFMinusREG = Case14GeneratorDF.head(len(
      Case14GeneratorDF)-NumCases[0])
Case14GeneratorCostDFMinusREG = Case14GeneratorCostDF.head
      (len(Case14GeneratorCostDF)-NumCases[0])

Case14REGDF.to_csv('Case 14 REG.csv', index=False)
Case14GeneratorDFMinusREG.to_csv('Case 14 Generator.csv',
      index=False)
Case14GeneratorCostDFMinusREG.to_csv('Case 14 Generator
      Cost.csv', index=False)
```

7. Data Sources

1. MATPOWER
   (a) Cases 14, 57, 118, 300

7.1. Case 14 Data

For illustrative purposes, we added the IEEE 14-bus test case values into the variables in our optimization expressions. All values are vectors, read from top to bottom, left to right.

$$P_g^{G0} = \begin{bmatrix} 232.4 & 40 & 0 & 0 & 0 \end{bmatrix}$$
$$Q_g^{G0} = \begin{bmatrix} -16.9 & 42.4 & 23.4 & 12.2 & 17.4 \end{bmatrix}$$
$$a_g = \begin{bmatrix} 0.0430292599 & 0.25 & 0.01 & 0.01 & 0.01 \end{bmatrix}$$
$$b_g = \begin{bmatrix} 20 & 20 & 40 & 40 & 40 \end{bmatrix}$$
$$c_g = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$
$$P_i^D = \begin{bmatrix} 0 & 21.7 & 94.2 & 47.8 & 7.6 & 11.2 & 0 \\ 0 & 29.5 & 9 & 3.5 & 6.1 & 13.5 & 14.9 \end{bmatrix}$$
$$Q_i^D = \begin{bmatrix} 0 & 12.7 & 19 & -3.9 & 1.6 & 7.5 & 0 \\ 0 & 16.6 & 5.8 & 1.8 & 1.6 & 5.8 & 5 \end{bmatrix}$$
$$V_i = \begin{bmatrix} 1.06 & 1.045 & 1.01 & 1.019 & 1.02 & 1.07 & 1.062 \\ 1.09 & 1.056 & 1.051 & 1.057 & 1.055 & 1.05 & 1.036 \end{bmatrix}$$
$$V_i^{min} = \begin{bmatrix} 0.94 \end{bmatrix}$$
$$V_i^{max} = \begin{bmatrix} 1.06 \end{bmatrix}$$
$$\theta_i = \begin{bmatrix} 0 & -4.98 \\ -12.72 & -10.33 \\ -8.78 & -14.22 \\ -13.37 & -13.36 \\ -14.94 & -15.1 \\ -14.79 & -15.07 \\ -15.16 & -16.04 \end{bmatrix}$$
$$\theta_{ij}^{min} = \begin{bmatrix} -360 \end{bmatrix}$$
$$\theta_{ij}^{max} = \begin{bmatrix} 360 \end{bmatrix}$$
$$G_{ij} = \begin{bmatrix} 0 \end{bmatrix}$$
$$B_{ij} = \begin{bmatrix} 0 \end{bmatrix}$$
$$P_g^{G,min} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$
$$P_g^{G,max} = \begin{bmatrix} 332.4 & 140 & 100 & 100 & 100 \end{bmatrix}$$
$$Q_g^{G,min} = \begin{bmatrix} 0 & -40 & 0 & -6 & -6 \end{bmatrix}$$
$$Q_g^{G,max} = \begin{bmatrix} 10 & 50 & 40 & 24 & 24 \end{bmatrix}$$
$$R_g^{G,down} = \begin{bmatrix} \infty \end{bmatrix}$$
$$R_g^{G,up} = \begin{bmatrix} \infty \end{bmatrix}$$

8. Project Source Code

The code used for this project was uploaded to the following github with a helpful README. GitHub - mew-two-github/ISyE6679-Project

References

Amdahl, G.M., 1967. Validity of the single processor approach to achieving large scale computing capabilities, in: Proceedings of the April 18-20, 1967, spring joint computer conference, pp. 483–485.

Carpentier, J., 1962. Contribution a l'etude du dispatching economique. Bull. Soc. Fr. Elec. Ser. 3, 431.

Carpentier, J., 1979. Optimal power flows. International Journal of Electrical Power & Energy Systems 1, 3–15. URL: https://www.sciencedirect.com/science/article/pii/0142061579900267, doi:https://doi.org/10.1016/0142-0615(79)90026-7.

Chen, G., Qian, J., Zhang, Z., Sun, Z., 2019. Applications of novel hybrid bat algorithm with constrained pareto fuzzy dominant rule on multi-objective optimal power flow problems. IEEE Access 7, 52060–52084. doi:10.1109/ACCESS.2019.2912643.

Huang, S., Dinavahi, V., 2018a. Fast batched solution for real-time optimal power flow with penetration of renewable energy. IEEE Access 6, 13898–13910. doi:10.1109/ACCESS.2018.2812084.

Huang, S., Dinavahi, V., 2018b. Gpu-based parallel real-time volt/var optimisation for distribution network considering distributed generators. IET Generation, Transmission & Distribution 12, 4472–4481. doi:https://doi.org/10.1049/iet-gtd.2017.1887.

Li, X., Li, F., Yuan, H., Cui, H., Hu, Q., 2017. Gpu-based fast decoupled power flow with preconditioned iterative solver and inexact newton method. IEEE Transactions on Power Systems 32, 2695–2703. doi:10.1109/TPWRS.2016.2618889.

Liu, D.C., Nocedal, J., 1989. On the limited memory bfgs method for large scale optimization. Mathematical Programming 45, 503–528. URL: https://doi.org/10.1007/BF01589116, doi:10.1007/BF01589116.

M. Sagebaum, T. Albring, N.G., 2019. High-performance derivative computations using codipack. ACM Transactions on Mathematical Software (TOMS) 45. URL: https://dl.acm.org/doi/abs/10.1145/3356900.

Mohagheghi, E., Alramlawi, M., Gabash, A., Blaabjerg, F., Li, P., 2020. Real-time active-reactive optimal power flow with flexible operation of battery storage systems. Energies 13. URL: https://www.mdpi.com/1996-1073/13/7/1697, doi:10.3390/en13071697.

Mohagheghi, E., Alramlawi, M., Gabash, A., Li, P., 2018. A survey of real-time optimal power flow. Energies 11. URL: https://www.mdpi.com/1996-1073/11/11/3142, doi:10.3390/en11113142.

Mohagheghi, E., Gabash, A., Li, P., 2017. A framework for real-time optimal power flow under wind energy penetration. Energies 10. URL: https://www.mdpi.com/1996-1073/10/4/535, doi:10.3390/en10040535.

Oliveira, E.J., Oliveira, L.W., Pereira, J., Honório, L.M., Silva, I.C., Marcato, A., 2015. An optimal power flow based on safety barrier interior point method. International Journal of Electrical Power & Energy Systems 64, 977–985. URL: https://www.sciencedirect.com/science/article/pii/S0142061514005419, doi:https://doi.org/10.1016/j.ijepes.2014.08.015.

Rakai, L., Rosehart, W., 2014. Gpu-accelerated solutions to optimal power flow problems, in: 2014 47th Hawaii International Conference on System Sciences, pp. 2511–2516. doi:10.1109/HICSS.2014.315.

Reddy, S.S., 2017. Optimal power flow with renewable energy resources including storage. Electrical Engineering 99, 685–695. URL: https://doi.org/10.1007/s00202-016-0402-5, doi:10.1007/s00202-016-0402-5.

Roberge, V., Tarbouchi, M., Okou, F., 2016. Optimal power flow based on parallel metaheuristics for graphics processing units. Electric Power Systems Research 140, 344–353. URL: https://www.sciencedirect.com/science/article/pii/S0378779616302140, doi:https://doi.org/10.1016/j.epsr.2016.06.006.

Shin, S., Pacaud, F., Anitescu, M., 2024. Accelerating optimal power flow with gpus: Simd abstraction of nonlinear programs and condensed-space interior-point methods arXiv:2307.16830.

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., Riedmiller, M., 2014. Deterministic policy gradient algorithms, in: Xing, E.P., Jebara, T. (Eds.), Proceedings of the 31st International Conference on Machine Learning, PMLR, Bejing, China. pp. 387–395. URL: https://proceedings.mlr.press/v32/silver14.html.

Su, X., He, C., Liu, T., Wu, L., 2020. Full parallel power flow solution: A gpu-cpu-based vectorization parallelization and sparse techniques for newton–raphson implementation. IEEE Transactions on Smart Grid 11, 1833–1844. doi:10.1109/TSG.2019.2943746.

Tang, Y., Dvijotham, K., Low, S., 2017. Real-time optimal power flow. IEEE Transactions on Smart Grid 8, 2963–2973. doi:10.1109/TSG.2017.2704922.

Wang, H., Murillo-Sanchez, C., Zimmerman, R., Thomas, R., 2007. On computational issues of market-based optimal power flow. Power Systems, IEEE Transactions on 22, 1185 – 1193. doi:10.1109/TPWRS.2007.901301.

Wang, Z., Wende-von Berg, S., Braun, M., 2021. Fast parallel newton–raphson power flow solver for large number of system calculations with cpu and gpu. Sustainable Energy, Grids and Networks 27, 100483. URL: https://www.sciencedirect.com/science/article/pii/S2352467721000540, doi:https://doi.org/10.1016/j.segan.2021.100483.

Woo, J., wu, L., Park, J.B., Roh, J., 2020. Real-time optimal power flow using twin delayed deep deterministic policy gradient algorithm. IEEE Access 8, 213611–213618. doi:10.1109/ACCESS.2020.3041007.

Wu, P., Chen, C., Lai, D., Zhong, J., 2023. A safe drl method for fast solution of real-time optimal power flow. arXiv:2308.03420.

Zimmerman, R.D., Murillo-Sánchez, C.E., Thomas, R.J., 2011a. Matpower: Steady-state operations, planning, and analysis tools for power systems research and education. IEEE Transactions on Power Systems 26, 12–19. doi:10.1109/TPWRS.2010.2051168.

Zimmerman, R.D., Murillo-Sánchez, C.E., Thomas, R.J., 2011b. Matpower: Steady-state operations, planning, and analysis tools for power systems research and education. IEEE Transactions on Power Systems 26, 12–19. doi:10.1109/TPWRS.2010.2051168.