Fast Parallel Newton-Raphson Power Flow Solver for Large Number of System Calculations with CPU and GPU

Zhenqi Wang\textsuperscript{a,*}, Sebastian Wende-von Berg\textsuperscript{a,b}, Martin Braun\textsuperscript{a,b}

\textsuperscript{a}Department of Energy Management and Power System Operation, University of Kassel, Wilhelmshöher Allee 73, Kassel, 34121, Germany
\textsuperscript{b}Fraunhofer Institute for Energy Economics and Energy System Technology, Königstor 59, 34119 Kassel, Germany

Abstract

To analyze large sets of grid states, e.g. in time series calculations or parameter studies, large number of power flow calculations have to be performed, as well as evaluating the impact from uncertainties of the renewable energy e.g. wind and PV of power systems with Monte-Carlo simulation. For the application in real-time grid operation and in cases when computational time is critical, a novel approach on parallelization of Newton-Raphson power flow for many calculations on CPU and with GPU-acceleration is proposed. The result shows a speed-up of over x100 comparing to the open-source tool pandapower, when performing repetitive power flows of system with admittance matrix of the same sparsity pattern on both CPU and GPU. The speed-up relies on the optimized algorithm and parallelization strategy, which can reduce the repetitive work and saturate the high hardware performance of modern CPUs and GPUs well. This is achieved with the proposed batched sparse matrix operation and batched linear solver based on LU-refactorization. The batched linear solver shows a large performance improvement comparing to the state-of-art linear system solver KLU library and a better saturation of the GPU performance with small problem scale. Finally, the method of integrating the proposed solver into pandapower is presented, thus the parallel power flow solver with outstanding performance can be easily applied in challenging real-life grid operation and innovative researches e.g. data-driven machine learning studies.

Keywords: Probabilistic Power Flow, Monte-Carlo Simulation, Contingency Analysis, GPU-acceleration, Newton-Raphson, Parallel Computing

1. Introduction

The penetration of Distributed Energy Resources (DER) e.g. Wind and PV causes high uncertainties in planning and operation of power systems. For both grid planning and operation, with the probability distribution of the infeed and load with uncertainties known, it can be formulated as Probabilistic Power Flow (PPF)-problem and evaluated with Monte-Carlo Simulation (MCS)-Power Flow (PF). The computational effort to solve the PPF problem varies according to the complexity of the unknown variable and the sampling methods. In the past decades, the researches have succeeded in reducing the required number of PFs [1, 2, 3]. Besides, the MCS-PF finds good application on a similar evaluation for the uncertainty with historical or synthetic injection and load profiles.

Static N-1 contingency analysis is required for the real-time and future grid state in the grid operation, with which grid congestion caused by the renewable energy needs to be properly handled with market and operational measurements [4]. Thus, it is essential to evaluate large number of PFs fast.

Furthermore, recent power system research trends show an increasing demand for a powerful PF solver for many PFs. Data-driven machine learning methods show great potential in power system state estimation and approximation of PF. Those approaches require large number of PFs of similar grid topology to be performed in the training phase [4, 5, 6]. The proposed method finds successful application in the study of using Artificial Neural Network for Medium-Voltage (MV) power system state estimation [7].

With the open source tools e.g. matpower[8] and pandapower[9], the grid can be easily modelled and single PF can be conveniently solved. The performance of these tools in solving many PFs with similar grid topology is unsatisfactory. Emerging works have shown great potential of the application of massive parallel hardware e.g. GPU in acceleration single PF[10, 11, 12, 13] as well as many PFs e.g. for static contingency analysis and online PPF [14, 15, 16, 17, 18, 19].
Motivated by the aforementioned challenge and demand, a general-purpose parallel PF solver for solving many PFs for grid with same sparse pattern of admittance matrix is presented in this work. PF with classical Newton-Raphson (NR) method is efficiently solved on both CPU and GPU with the proposed approach. Recent development [15, 20, 21] show great potential to exploit the power of parallelization for solving many PFs. Our work distinguished itself from the existing works with a new parallelization scheme on CPU with the Single Instruction Multiple Data (SIMD) parallelization [22]. By using the full floating-point arithmetic (FP) potential, a performance improvement on top of the already good performance multi-threaded version is achieved. An easy-to-implement row-level parallelization strategy is proposed for batched sparse matrix operation on GPU. Batched LU refactorization methods for CPU with SIMD and GPU is proposed, which shows better performance (20x-70x) than KLU [23] library. Additionally, the method of integrating the advanced parallel PF solver into the open-source power system analysis tool pandapower is presented, which is essential of the application into real-life grid planning, operation and researches.

This paper is formulated in 5 sections. Section 2 introduces the CPU and GPU architecture and how the performance of computation tasks can be optimized respectively. Section 3 introduces the proposed approach of implementation of the parallel PF solver. Section 4 introduces the proposed batched linear solver for CPU and GPU platform. Section 5 presents the benchmarking result with comprehensive analyze.

2. Review of parallelization on CPU and GPU

This section gives a brief introduction of the modern CPU and GPU regarding its specialties in order to solve many PFs efficiently. Fig. 1 shows a generalized CPU-GPU hardware structure. On modern CPU, multiple physical cores are usually available, with which multiple tasks can be performed independently. For computational tasks, a good utilization of multi-cores greatly improves the performance. Furthermore, with large size of on-chip cache available, the memory transaction with limited memory bandwidth can be avoided on cache-hits.

The SIMD instruction set is available on CPU on each physical core, which can execute the same operation on multiple data resides in the extended registers with the size of 128 bits of higher [24]. This corresponds to 2 and more double precision FPs, with which all the FP capability of CPU is greatly improved [25]. The SIMD instruction set e.g. SSE and AVX2 is available on most modern CPU, which can carry out 2 and 4 double-precision FPs simultaneously.

Comparing to CPU, modern GPU is designed not only for graphic rendering but also as a powerful highly parallel programmable processor [26]. Large number of parallel Stream Processor (SP)s are integrated on-chip, which result in a high FPs peak performance comparing to CPU. The SPs are clustered into Stream Multiprocessor (SM)s, which contains L1-cache, shared memory and control units [27]. Besides, GPU has high memory bandwidth as shown in Fig. 1, which brings advantage on the memory-bounding tasks.

2.1. Parallelization on CPU

Multi-threaded task-level parallelization can be realized with OpenMP [28], which is a popular library in scientific computing for parallelization on shared-memory. This parallelism paradigm is called Simultaneous multithreading (SMT) [29]. OpenMP uses compiler directives to allocate the parallelized regions in the serialized program during the compiling phase.

Since OpenMP 4.0, SIMD parallelization is directly supported within the framework [30]. On top of the SMT parallelization, mini-batch tasks can further profit from the usage of SIMD instruction set, this type of parallelism is denoted as SMT+SIMD.

2.2. Principle of SIMT parallelization on GPU

In order to perform large-scaling scientific computation tasks on massive parallelization hardware like GPU, Single Instruction Multiple Threads (SIMT) programming model is widely adapted. Using NVIDIA CUDA as an example,
the user is able to program functions called CUDA Kernel, which defines the behavior of massive simultaneous threads. A simplified 2D programming model of CUDA kernel is shown in Fig. 2. The massive threads are organized in a two-level structure, with each level supports up to 3-dimensions. The first level is called thread block, all the threads within a block are executed on a single SM and can be synchronized explicitly with synchronization barrier [27]. All the thread blocks are organized in a grid. The execution of grid is distributed on all the SMs on-chip. The thread is aware of its indexing in the two-level structure, which is essential to define the data-indexing relevant individual tasks. The size of thread block and grid can be configured upon kernel call.

During the execution of a CUDA kernel, 32 consecutive threads within one block are aggregated into the minimum execution unit called warp [31]. The threads within one warp execute the same instruction. When threads within a warp encounter different instructions, these are executed in a serial mode, thus lose the advantage of parallelism. To fully utilize the calculation capability for computing-bounding tasks, the first criterion for GPU performance optimization is to avoid thread divergence within one warp.

The second criterion is to optimize memory transaction with the device memory. Due to the high-latency of the graphics memory (multiple hundreds of cycles) [32], the instruction of memory transaction is queued in a First-In-First-Out way. It is possible to hide this latency and saturate the memory bandwidth by keeping the queue busy [31]. For each memory transaction, 32-bytes data of consecutive memory sections is accessed, to maximize the efficiency of memory transaction and reduce excessive memory transaction, it is important to enable the warp access for coalesced memory, especially for memory-bounding tasks [33]. Constants and texture can be accessed through specific cache, which improves the access with sparse matrix indexing.

Any tasks on GPU can only be executed, when the data is available on the device memory. Due to the high-latency of the graphics memory (multiple hundreds of cycles) [32], the instruction of memory transaction is queued in a First-In-First-Out way. It is possible to hide this latency and saturate the memory bandwidth by keeping the queue busy [31]. For each memory transaction, 32-bytes data of consecutive memory sections is accessed, to maximize the efficiency of memory transaction and reduce excessive memory transaction, it is important to enable the warp access for coalesced memory, especially for memory-bounding tasks [33]. Constants and texture can be accessed through specific cache, which improves the access with sparse matrix indexing.

3. Approach for a parallel NR-PF solver

In this work, an approach of the acceleration of many NR-PFs with same sparsity pattern admittance matrix $Y_{bus}$ (according to the matpower convention [8]) is presented. This special property leads to the same sparsity pattern of the update jacobian matrix (JM) required in the iterative solving process, which brings further advantages for speed-up by reducing repetitive works as following:

- Reuse of indexing of sparse matrix
- Reuse of static lookup for sparse matrix update
- Reuse of memory working space

Fig. 3 shows exemplary the indexing data and profile data required for the parallel PF solver. The $Y_{bus}$ is stored in Compressed Row Storage (CRS)-format with extended diagonal index. The standard CRS sparse matrix indexing consists of $RowPtr$ with the length of number of rows plus one and $ColIx$ equals the number of non-zero elements, which is efficient for the iteration over rows of the matrix. The extra $DiagPtr$ to represent the data index of the diagonal element of each row for the convenience of iterating and aggregating on the diagonal elements (required in update JM). Fig. 3b gives the aforementioned sparse indexing of the non-zero elements of $Y_{bus}$ in Fig. 3a. For different types of calculation, the variable profile data required is shown in Fig. 3c. For PPF, $Y_{bus}$ requires only once while the $PQ$ equals the size of the sampling. For static N-1 analysis, the number of $Y_{bus}$ equals the number of contingency cases (number of post-contingency matrices) while $PQ$ requires only once.

$$S_{bus} = V_{bus} \cdot (V_{bus} \cdot V_{bus})^*$$

As shown in Fig. 3a and given in Eq. (1), PF problem is defined as finding the unknown complex voltage $V_{bus}$, which minimized the power injection mismatch between the given complex bus injection $S_{bus}$ and the one calculated with $V_{bus}$ [34]. With different bus types, different values are given as input. For slack bus the complex voltage is given, while for PV bus the constant voltage magnitude and active power and for the PQ bus with active and reactive power is predefined (shown in Fig. 3a). The complex voltage is represented in polar notation (2V, |V|) and complex power injection $S_{bus}$ is given in cartesian form (P, Q). For NR algorithm, the following processes are performed iteratively until maximum allowed iteration number reached or the maximum of the power injection mismatch drops below the tolerance (typically 10^{-8}[8]).

1. Calculate Nodal Power Mismatch (NPM) $\Delta P_V, \Delta Q_V$ with $V_{bus}$
2. Create/Update JM $J$ with $V_{bus}$
3. Solve the linear system
4. Update $V_{bus}$ with $\Delta V_{pq}$, $\Delta V_{pq}$

In each iteration, the convergence (power injection mismatch) is checked with the NPM calculation. Section 4 explained the details of solving linear system step with LU-refactorization.

3.1. Batched operation on sparse matrix

The utilization of the sparse matrix is essential to reduce the required memory space and avoid useless arithmetic operations on zeros, due to the high sparsity rate in $Y_{bus}$, $J$. Since the sparse matrices $Y_{bus}$, $J$ share the same sparsity...
Figure 3: CRS sparse matrix for NR-PF and batched data.

Figure 4: System overview of parallel PF solver.

pattern, when iterating over the elements, it is possible to broadcast operations performed on these matrices among tasks within batch. The storage of the batched profile data in memory is shown in Fig. 3d). The batched profile data in memory is aligned on the same element within the batch (mini-batch) and stored in contiguous memory address instead of within each task vector. On GPU, this property guarantees automatically coalesced memory access and avoid the threads divergence within one warp [20], as long as the batch has a size of multiply of the default warp size (32 for CUDA). With SMT+SIMD on CPU with mini-batch, it makes full utilization of the FP capability of the instruction set. A mini-batch of size 4 is used for the efficient usage of AVX2-instruction set.

In order to achieve an early saturation of the GPU performance or memory bandwidth depends on the property of the kernel. Further actions to increase the saturation is required, as explained below.

3.1.1. Calculate nodal power injection mismatch

Calculation of NPM with $V_{bus}$ is given in Eq. (1). The process can be performed with iterating over the element in the $Y_{bus}$ once. Due to the task independency between rows in $Y_{bus}$, extra row-level parallelism can be achieved with one CUDA thread responsible for a row of one task. Algorithm 1 shows the proposed process on GPU, which is a computing-bounding task, the extra row-level parallelization improves the kernel saturation on GPU. On CPU platform, with SMT parallelization the $taskId$ is given as an input. For SMT+SIMD parallelization the arithmetic operation is automatically broadcasted to all the task within the mini-batch.

3.1.2. Update Jacobian Matrix

Similar to Algorithm 1, update of $J$ matrix is a computing-bounding task. Due to the independency between rows, the process can be extended with row-level parallelization for a better saturation.

In the first step, only the diagonal elements are calculated with help of the $DiagPtr$ as shown in Fig. 3b. On
the second step, the non-diagonal elements are calculated
by iterating over the $Y_{bus}$ once, during which the diago-
nal elements are updated. With the $d(P,Q)/d\{V,|V|\}$
matrix consisting of four sub-matrices available, it can be
subset into $J$ matrix (without $P,Q$ for slack and $Q$ for
PV bus) or direct into permuted form $A$ with static
lookup.

3.2. System overview of parallel PF solver and its integra-
tion with pandapower

Fig. 4 shows the method of integrating the aforementioned parallel PF solver with pandapower. The initialization
step is carried out with the pandapower in python envi-
ronment including the initialization of the sparse indexing
of $Y_{bus}$ and $J$ as well as the static lookup for updating
sparse matrix and the profile data. With unified initial-
ization step, the PFs can be solved with the three types
(SMT, SMT+SIMD, CPU SIMD) of parallel PF solver.
The SMT+SIMD requires the extra transpose to mini-
batch step, while the GPU version requires the memory
transaction between device and host and the transpose to
batch step on device. The resulted power injection mis-
mismatch is checked on CPU, if any PF task cannot converge
due to numerical instability during LU refactorization, this
task will be given a second chance to get fixed with the
KLU numerical factorization. The branch flow is calcu-
lated in parallel after the final convergence check with the
resulted $V_{bus}$. For quasi-static time-series simulation, the
control logic for DER reactive power mode and On-Load
Tap Changer (OLTC) can be integrated into the frame-
work, which updates the profile data in each time step.

The proposed approach is realized with C++/CUDA
with an interface to Python with Cython 3.0[35]. The pro-
gramming language Python has convenient tool for data
integration and processing e.g. Pandas (High-Level)[36]
and efficient matrix and array operation with Numpy [37].
The underlying data stored with Numpy array can be
shared and ported directly to the parallel PF solver.

4. Batched Linear System Solver

The solving of linear system is a time-consuming step
in solving NR PF, as it is in the actual pandapower
implementation[38]. The mathematical formulation of
solving linear system step in NR process is given in Eq. (2).
The $b$ is the active and reactive power mismatch with the
given $V$, the result $x$ is used to update the $V$.

$$J_{1..n} \cdot x_{1..n} = b_{1..n}$$
$$x = [\Delta V_{pv,pq}, \Delta |V_{pq}|]$$
$$b = [\Delta P_V, \Delta Q_V]$$

In this work, the existing process of solving linear system
is analyzed and an optimized algorithm for batched linear
system solver is proposed.

4.1. Process of solving with direct LU solver

In solving linear system with LU factorization, the origi-
\nal linear system is pre-ordered (pre-scaled), factorized
into lower-triangular matrix $L$ and upper-triangular ma-
trix $U$ and finally solved with Forward Substitution (FS)-
Backward Substitution (BS), as it is in the modern imple-
\mentations [23, 39, 40, 41]. KLU is considered one of the
fastest single-thread linear solver for JM[42, 12, 20, 16],
which is used as the reference for the proposed method
in this work. The KLU solver can be easily extended for
task-level parallelization with OpenMP. Following theore-
tical introduction focuses on the implementation of KLU
and gives hints on the optimization for solving many PFs
with same $Y_{bus}$ sparsity pattern.

In the pre-ordering phase, the original $J$ is permuted
in order to reduce the required FPs by reducing the fill-
ins. Multiple heuristic methods are available for the pre-
ordering, which gives different performance related to the
matrix pattern. In this work, AMD [43] is chosen as de-
fault pre-ordering algorithm, since [42] has reported its
good performance. With the permutation, the original
linear system $J$ is permuted into the $A$ as given in Eq. (3).
Perm col, Perm row are the correspondent column and row
permutation matrix. The numerical factorization is per-
formed on $A$.

$$A_{1..n} = Perm_{col} \cdot J_{1..n} \cdot Perm_{row}$$
$$A_{1..n} = L_{1..n} \cdot U_{1..n}$$

In the numerical factorization of KLU, the GILBERT-PEIERS
left-looking algorithm (G-P) algorithm is utilized. Ad-
ditionally, partial pivoting is performed to improve the

numerical stability, with which the permutation matrix $Perm_{row}$ is updated, in order to avoid the tiny pivot. This step has effect on the final sparsity pattern on $A$.

Refactorization is a much faster process, which reduced the computational overhead by presuming the numerical stability of the permutation matrix $Perm_{row}$, $Perm_{col}$. For NR iterations in solving PF and circuit simulation, refactorization is preferred [41]. The refactorization mode is also supported by KLU.

The pattern of $A$, $L$ and $U$ remains unchanged for all tasks when solving many PFs with same $Y_{bus}$ sparsity pattern. The pre-ordering only need to be performed once at the beginning [15]. Based on this property, a static lookup can be created with the final permutation matrices $Perm_{row}$, $Perm_{col}$ found. Since in G-P algorithm Compressed Column Storage (CCS) matrix is required, which is efficient for iterating over columns. A static lookup can direct convert the original CRS JM or $d\{P, Q\}/d\{V, |V|\}$ matrix into the permuted $A$ into CCS format.

4.2. CPU LU Refactorization

Algorithm 2 Sparse G-P refactorization algorithm with column working space

```
1: for tId in 0:N_{task} do
2:    # do in parallel on CPU
3:    for col in 0:N_{col} do
4:        # Copy column to working space
5:        x = A(:, col, tId)
6:        for row in URowIx(:, col) do
7:            # Sparse VMAD on column working space
8:            x(row + 1 :) = x(row) · L(:, row, tId)
9:        end for
10:       # Normalize L and update LU
11:      U(:, col, tId) = x(col)
12:      L(:, col, tId) = x(col + 1 :) / x(col)
13: end for
14: end for
```

Algorithm 2 gives the SMT LU refactorization algorithm. Sparse G-P[44] refactorization on CPU with column working space is implemented. The working space has the size of the dimension of $A$ for SMT. With the column working space $x$, only the copy of non-zero values of $A$ is required for the sparse vector multiply and add (VMAD).

For SMT+SIMD, working space with the width of the mini-batch size is required. By replacing $tId$ with the ID of mini-batch, the copy, VMAD and normalization can be extended with SIMD instructions, so that the operation is broadcasted within mini-batch.

Since task-level parallelization can fully saturate all physical cores of CPU with SMT and SMT+SIMD, column-level parallelization as proposed in [45] is not needed.

4.3. GPU batched LU Refactorization and FS-BS

Figure 6: Example LU matrix for LU Refactorization.

4.3.1. Theoretical analysis

Recent effort on LU factorization with GPU is presented in [41, 40], both works focus on accelerating single LU factorization on GPU, especially for large scaling matrix. For batched LU factorization, [20] presents a batched LU-factorization scheme with batch and column-level parallelization. It is essential to saturate the GPU memory bandwidth for LU refactorization and FS-BS to achieve good performance on GPU. Due to the underlying data dependency, not all columns can be factorized simultaneously. A directed acyclic graph (DAG) is used to describe
and schedule the columns which can be finished at the same time [46]. The vertices represent the columns and the edges represent the operation between two columns. An example matrix and its DAG is shown in Fig. 6 and Fig. 7 respectively.

Considering the G-P algorithm with column-level parallelization proposed in [16], as the example of two standard grids shown in Fig. 5, at the beginning with high number of available columns per level, the GPU resource can be easily saturated even with small batch size. However, significant amount of FPs are located in the serial region, where only few columns can be factorized at the same time.

Using NVIDIA GTX1080 graphics card as an example, the GPU has 20 SMs with each SM has 128 SPs. Assuming the batch size is 512, which equals to less than 1 active warps per SM, which means 75% of the SP remains idle when no extra parallelization is applied. Consequently, the memory-bandwidth cannot be efficiently utilized.

Inspired by the work [41], our work improves the batched LU-refactorization with fine-grained parallelization for serial levels, which improves the saturation of hardware significantly with small batch size. This improvement make greater sense on the future applications with even more computational resources and higher memory-bandwidth available on new generations of GPUs.

4.3.2. Proposed algorithm on GPU

Algorithm 3 Multi-stage sparse G-P algorithm on CUDA

```plaintext
1: # DEFINE VMAD parallelization width as N
2: # Do in parallel on CUDA
3: # Find task (tId) with thread indexing
4: batchId = ThreadBlock.Idx
5: tIdinBatch = Thread.Idx
6: tId = batchId \cdot ThreadBlock.DimX + tIdinBatch
7: # IF STAGE 1
8: col = AvailableColsinLevel(ThreadBlock.Idy)
9: # IF STAGE 2,3
10: col = LeftoverCols(ThreadBlock.Idy)
11: for row in URowIx(:, col) do
12: # IF STAGE 2,3
13: if row not in FinishedCols then
14: break
15: end if
16: # IF STAGE 2,3
17: if row in FinishedRowsOfCol(col) then
18: # Skip finished rows
19: continue
20: end if
21: # Sparse VMAD with direct indexing
22: # IF STAGE 2 element-wise iteration in vector
23: # IF STAGE 3 N-elements-wise iteration in vector
24: LU(row + 1 :, col, tId) =
25: LU(row, col, tId) \cdot LU(row + 1 :, row, tId)
26: end for
27: # Check column is finished
28: if row == URowIx(-1, col) then
29: # Set flag on finished columns
30: UpdateFinishedColStatus(col)
31: # Normalize L
32: # IF STAGE 2 element-wise iteration in vector
33: # IF STAGE 3 N-elements-wise iteration in vector
34: LU(row + 1 :, col, tId) = LU(row, col, tId)
35: else
36: # IF STAGE 2,3
37: # Memorize the processed row of the column
38: UpdateFinishedRowsOfCol(row, col)
39: end if
```

A three-stage batched G-P algorithm is proposed for the GPU refactorization, which is given in Algorithm 3. \(LU\) denotes the working space for LU refactorization \((L+U-I)\) including all the fill-ins predefined. The values in A needs to be copied into \(LU\) with the fill-ins initialized as 0.

In stage 1, with large number of columns available in each level, the columns are fully factorized in parallel. Each CUDA thread is responsible for one column in one task. In stage 2, when the available column of the level drops below e.g. 32, besides the columns, which can be fully factorized, the viable operations for all the unfinished columns can be executed simultaneously (shown in
Fig. 7 as green lines). In stage 3, when only few columns e.g. one column available in each level, the extra VMAD parallelization as shown in Fig. 6 could be applied with extra CUDA threads scheduled within the same CUDA threadBlock, since only threads within one threadBlock can be explicitly synchronized. The extra VMAD parallelization could use width of e.g. 4. In stage 2 and 3, on each level some columns can only be partially factorized, the already processed rows of each column are memorized, the finished columns of each level will be directly normalized.

For example matrix LU shown in Fig. 6, the level one in Fig. 7 corresponds to the stage 1 of Algorithm 3. After the first level is finished, tasks which belong to the columns in later levels can be executed in an earlier stage, which in turn increased the saturation rate and reduced the tasks in the serial levels (see Fig. 8). Level 2 corresponds to the stage 2 of the algorithm. In level 3, 4, when only one single column is available the extra VMAD parallelization is applied according to stage 3. In this case, the row 8, 9, 10 are assigned onto the Thread.Idy, thus VMAD can be performed more efficiently with suitable parallelization width instead of element-wise.

4.3.3. GPU Forward Substitution Backward Substitution

5. Case Studies and Performance Evaluation

The case study on CPU is performed on a Windows 10 PC with Intel i7-8700 CPU (6 physical cores and Hyper-Threading (HT) technology for 12 virtual cores) and 16 GB DDR4 RAM, with the prioritized Intel C++ Compiler V2020 and the Intel Math Kernel libraries, which is highly optimized for the vectorized computation. The code is compiled with O3 and forced using the AVX2 instruction set.

The case study on GPU is performed on a PC with Intel i7-8700k, 32 GB DDR4 RAM and 2x Nvidia GTX 1080-8GB GPUs. The PC is running Ubuntu 18.04.4 LTS. The proposed approach is programmed in C++ and compiled with GCC V7.4 and CUDA V10.1 with O3. The main focus of the benchmarking is the duration of the PF solving and each subprocess. Because the kernel execution in CUDA is initialized from CPU and executed on GPU asynchronously. To record the duration of each kernel, the test is performed with synchronization at the kernel (or multiple-kernels e.g. for LU-Refactorization) exit.

On both test platforms, double precision float number is used, due to the high numerical stability requirement during the LU refactorization. The examples are tested with small-size grid "IEEE case300" and mid-size grid "case2869pegase" available in pandapower. Both contain meshed Extra-High-Voltage (EHV) and High-Voltage (HV) voltage levels, IEEE case300 contains also MV and Low-Voltage (LV) buses.

5.1. Batched linear solver performance analysis

This section gives a performance evaluation of the aforementioned batched linear solver on CPU and GPU and the relative performance to KLU. The AMD pre-ordering method with the initial numerical factorization is performed with KLU.

Fig. 10 shows the performance on CPU platform. On both test grids, a performance improvement of the implemented G-P algorithm can be observed over KLU against both numerical factorization and refactorization modes. Because it works directly on the already permuted A matrix, as the KLU takes the original matrix as input. With SMT parallelization, a performance improvement of >x6 can be observed for all the methods tested when increasing the number of threads, especially when below the number of physical cores (6). With further increasing the thread number, the HT technology helps further improve the saturation of the physical core, thus improve the overall performance. With the SMT+SIMD parallelization, further speed-ups can be observed. However, a slight performance decreasing can be observed on the mid-size grid under the G-P SMT+SIMD version, when increasing the threads number from 6 to 12, this behavior is caused by the reduced cache hitting rate, due to the large problem scaling. Overall, with the proposed method, on CPU platform, we achieved a good performance improvement of x20 - x70.
The acceleration rate has a close relationship to the size of the problem (number of busses).

On the GPU platform, Fig. 11 shows the benchmarking result with different batch sizes, the best CPU results is used as baseline. It can be observed, that our approach of further fine-grained parallelization of the refactorization process leads to an earlier saturation of the GPU resource and achieved much better performance when the batch size is small for both grids. With the extra VMAD parallelization with the 4x threads in stage 3, it improves the performance when the batch size is very small (< 512) for LU refactorization and FS-BS, slight difference can be observed when the GPU is fully saturated. When dealing with a large grid, the improvement is more significant comparing to the CPU counterpart due to the higher cache misses on CPU.

5.2. Performance analysis on CPU

For the convenience of comparing the performance of each functions involved in NR PF, these can be categorized as following:

- **FP Dominant (Float):** Calculate NPM, Update \( d(P, Q)/d|V| \)
- **Memory and FP (LU Total):** LU Refactorization, FS-BS
- **Memory Dominant (Memory):** Permute JM(A), Update V

The performance on CPU is evaluated with timing and Intel Vtune profiling tool for the memory access pattern analysis. Fig. 12 shows the time of each process of SMT parallelization and SMT+SIMD parallelization with different number of threads. For both small and mid-size grid, similar to the observation in batched linear solver, the performance increased almost linear at the beginning and only slightly after 6 threads. For the mid-size grid, with the last-level cache missing rate of the SMT+SIMD with 12 threads increased from 0.1% in total to 5% for "LU Total" and 25% for "memory" comparing to with only 4 threads. The random memory access pattern of "LU Total" and "memory" requires a frequent exchange between the cache and system DRAM, which due to its limited bandwidth actually, drags the overall performance back by increasing thread number. The SMT+SIMD version still outperforms 12 threaded SMT parallelization scheme at the thread number of 5. For the small grid, with the data fits in the cache well, the increase of threads number improves the overall performance.

Concluding, on CPU platform, increasing number of threads brings only benefit, when solving PF of a small scale grid. While with SIMD+SMT, the cache should be considered and the number of threads should be carefully chosen given the grid size and available hardware. On the test CPU for instance, the number of the physical cores (in this case 6) is a good first trial.

5.3. Performance Analysis on GPU

The GPU version requires the extra memory transaction between host and device, which is labelled as "DH Memory". Fig. 13 shows the function performance on GPU with one stream for both grids with 10,000 calculations. It can be observed that except for the "LU Total", the other tasks can saturate the GPU resources with small batch size, thus it is insensitive to the change of the batch size for both grids. Comparing to the the best CPU performance, the "FP" function achieved the average improvement among x5-6.

Fig. 14 shows the improvement with the usage of CUDA concurrency with stream and multi GPUs on different
Figure 12: Function benchmark results on CPU for the performance comparison over SMT and SMT+SIMD (left: IEEE case300, right: case2869pegase).

Figure 13: Function benchmark results on GPU (left: IEEE case300, right: case2869pegase).

Table 1: Benchmark results profile simulation including initialization

| case name           | case118 | case300 | case1354 | case2869 | case9241 | sb mv-lv | sb hv-mv | sb ehv-hv |
|---------------------|---------|---------|----------|----------|----------|----------|----------|-----------|
|                     | Timing of the initialization (ms) |         | Timing of 10,000 PFs (ms) |         |         |         |         |           |
|                     | 24      | 67      | 267      | 608      | 2211     | 36       | 353      | 142       |
| pandapower          | 18,956  | 47,373  | 149,234  | 464,009  | 1,794,955| 12,366   | 126,062  | 101,279   |
| SMT best            | 175     | 778     | 2,666    | 8,323    | 40,835   | 382      | 8,126    | 5,576     |
| SMT+SIMD best       | 97      | 333     | 1,524    | 7,217    | 38,848   | 181      | 5,697    | 2,620     |
| 1 gpu best          | 50      | 201     | 639      | 1,941    | 10,658   | 113      | 2,100    | 1,611     |
| 2 gpu best          | 26      | 102     | 342      | 1,021    | 6,525    | 76       | 1,258    | 980       |

Figure 14: Duration and speedups with batches and number of streams. The test case with batch size 512 and 1 stream is used as base line scenario. With the multiple streams on one GPU, an improvement due to the hiding the memory transaction between host and device can be observed with all batch sizes. When the batch size is small, the improvement is higher which is related to the potential of kernel overlapping. When two GPUs are available, an acceleration of around factor x2 can be observed. However, due to the imbalanced computation load distribution among GPUs and streams. With some stream/GPU undertakes more tasks than the others, the improvement of the overall execution time varies. Using batch size 2048 as an example, when executing 10,000 PFs, 4x2048 tasks are equally distributed, while one GPU/Stream have to execute the 1808 leftovers.

5.4. Benchmarking Result of the integrated parallel PF-solver

The benchmarking result of running 10,000 PFs of multiple grids with same \( Y_{bus} \) is listed in Table 1. The parallel PF-solver is integrated in pandapower. Besides case2869pegase and IEEE case300, further standard grids "IEEE case118", "case1354pegase" and "case9241pegase" available in pandapower are utilized to evaluate the performance under different grid dimensions. As well as three grids from the SimBench open-source data set[47], which was recreated to represent the characteristics of real German grid and contains realistic time series data for loads.
and DERs, are used for benchmark. These are "sb mv-lv", "sb lv-mv", "sb ehv-lv" with 109, 1665 and 713 buses respectively. With multiple performance optimizations especially the creation of JM, pandapower gives already better performance comparing to matpower [9, 38], thus the baseline case is performed with PF-function of pandapower named "newtonpf", which is not parallelized and used SuperLU as linear system solver.

From Table 1 it can be observed, that the grid dimension has direct impact on the total calculation time. With the increment of grid dimension, the acceleration on the GPU is more significant as on CPU, due to the high FP capability and memory bandwidth of GPU and the aforementioned cache issue on GPU. On large grids, e.g., case9241pegase, the GPU version is x4 times faster than the best CPU performance. The effect of acceleration by using SMT+SIMD is also decreasing comparing to SMT due to the aforementioned cache issue. To be noticed by case9241pegase, due to the large requirements of memory for process data, batch size larger than 2048 failed in the benchmarking. On small size grid, because of the well usage of CPU cache, the performance of CPU is satisfying, urgent need for using GPU in this calculation does not exist. The same one-time initialization process according to Fig. 4 is applied on both CPU and GPU tests. It should be noticed that the one-time initialization takes significant amount of time, since most of the code is executed in Python environment, which can be further optimized e.g. with just-in time (JIT) or C/C++ integration for performance critical applications.

In the test considering the number of PF calculation, when increasing the number of calculation from 100 to 10,000, the number of calculation has few impact on the average time of solving single PF on CPU SMT and SMT+SIMD. The GPU version, due to the aforementioned saturation issue, the performance is improved with the increase of the number of calculation. After the saturation point of around 2000 calculations is reached, the average time of solving PF is almost constant.

6. Conclusion

In this paper, an approach for the parallel solving of many power flows with CPU/GPU is proposed. When the grid admittance matrices share the same sparsity pattern, the Newton-Raphson power flows can be efficiently solved in parallel. The performance is evaluated in detail with multiple standard test cases and open-source data sets SimBench.

Impressive acceleration (more than 100x over open-source tool pandapower) was already achieved with CPU alone, while GPU brings even more. The performance of the fast parallel power flow solver originated mainly from the following points:

- Avoidance of repetitive work (sparse matrix indexing initialization, pre-ordering, lookup creation, etc.),
- Reduction of computational overhead with LU-refactorization,
- Hardware-specific optimization and parallelization strategies.

In detail, on CPU platform, the power flows can be accelerated with less effort with SMT parallelization. With SMT+SIMD parallelization, the acceleration effect is highly dependent to the problem scaling. On small grids, a further speed-up of x2-3 comparing to SMT parallelization can be expected. On the GPU platform, with the batch operation on sparse matrix, the high FP capability and high memory bandwidth can be effectively saturated. Comparing to the CPU counterparts, the computing-bounding functions are accelerated significantly, while the memory-bounding functions depend highly on the problem scaling.

The outstanding performance of the proposed parallel power flow solver shows promising application in the real-time grid operation in order to allow the consideration of uncertainties. The innovative researches in the data-driven machine-learning methods in power systems can be great benefitted. Even more potential can be exploited with the application of the proposed solver on a high-performance computing clusters with multiple CPUs and GPUs.

Acknowledgements

The authors would like to thank Florian Schäfer and Dr. Alexander Scheidler for their suggestions to improve the quality of this paper. The work was supported by the European Union’s Horizon 2020 research and innovation programme within the project EU-SysFlex under grant agreement No 773505.

References

[1] H. Yu, C. Y. Chung, K. P. Wong, H. W. Lee, J. H. Zhang, Probabilistic load flow evaluation with hybrid latin hypercube sampling and cholesky decomposition, IEEE Transactions on Power Systems 24 (2) (2009) 661–667. doi:10.1109/TPWRS.2009.2015689.
[2] C.-L. Su, Probabilistic load-flow computation using point estimate method, IEEE Transactions on Power Systems 20 (4) (2005) 1843–1851. doi:10.1109/TPWRS.2005.857921.
[3] J. Usaola, Probabilistic load flow with correlated wind power injections, Electric Power Systems Research 80 (5) (2010) 528–536. doi:10.1016/j. EPSR.2009.10.023.
[4] F. Schafer, J.-H. Menke, M. Braun, Contingency analysis of power systems with artificial neural networks, in: 2018 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm), IEEE, 29.10.2018 - 31.10.2018, pp. 1–6. doi:10.1109/SmarGridComm.2018.8587482.
[5] K. R. Mestav, J. Luengo-Rozas, L. Tong, Bayesian state estimation for unobservable distribution systems via deep learning, IEEE Transactions on Power Systems 34 (6) (2019) 4910–4920. doi:10.1109/TPWRS.2019.2919157.
[6] Y. Liu, N. Zhang, Y. Wang, J. Yang, C. Kang, Data-driven power flow linearization: A regression approach, IEEE Transactions on Smart Grid 10 (3) (2019) 2569–2580. doi:10.1109/TSG.2018.2805169.
rithm 837: Amd, an approximate minimum degree ordering algorithm, ACM Transactions on Mathematical Software 30 (3) (2004) 381–388. doi:10.1145/1024074.1024081.

[44] John R. Gilbert, Timothy Peierls, Sparse partial pivoting in time proportional to arithmetic operations: Tr 86-783.

[45] X. Chen, Y. Wang, H. Yang, Niculu: An adaptive sparse matrix solver for parallel circuit simulation, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 32 (2) (2013) 261–274. doi:10.1109/TCAD.2012.2217964.

[46] T. Nechma, M. Zwolinski, Parallel sparse matrix solution for circuit simulation on fpgas, IEEE Transactions on Computers 64 (4) (2015) 1090–1103. doi:10.1109/TC.2014.2308202.

[47] C. Spalthoff, D. Sarajlic, C. Kittl, S. Drauz, T. Kneiske, C. Rehtanz, M. Braun, Simbench: Open source time series of power load, storage and generation for the simulation of electrical distribution grids, in: International ETG-Congress 2019; ETG Symposium, 2019, pp. 1–6.