Performance Analysis of GPU-Accelerated Fast Decoupled Power Flow Using Direct Linear Solver

Shengjun Huang[(1)(2)], *Student Member*, *IEEE* and Venkata Dinavahi[(1)], *Senior Member*, *IEEE*
(1) Department of Electrical and Computer Engineering, University of Alberta, Edmonton, Alberta, Canada
(2) College of Information System and Management, National University of Defense Technology, Changsha, Hunan, China
shengjun@ualberta.ca, dinavahi@ualberta.ca

***Abstract*—Achieving high solution efficiency for alternating current power flow (ACPF) analysis from high-performance computing (HPC) architecture is a leading and important challenge in power system analytics and computation. This paper investigates the performance of the fast decoupled (FD) method, which is based on the direct linear solver and implemented on the graphics processing unit (GPU), for the solution of ACPF. Implementation platforms, linear equations solution strategies, data storage formats, and fill-in reduction algorithms are compared and discussed on five benchmark systems ranging from 300 to 13,659 buses. Within the GPU's compute unified device architecture (CUDA) environment, the shortest ACPF solution time for the largest test case is 0.313s, which is 4.16× faster than its Matlab counterpart.**

***Index Terms*—Fast decoupled power flow, graphics processing unit, linear solver, LU decomposition, parallel computing.**

I. Introduction

Alternating current power flow (ACPF) analysis is one of the most fundamental tasks for the power system operation and optimization [1], which dominates the essential steps of many practical problems, such as contingency analysis, economic dispatch, optimal power flow, etc. The challenge of quick solution techniques always exists for the ACPF, since the shorter calculation time means better situational awareness, faster response, and less adverse impact on the system. Except for the time-critical features, ACPF is also confronted with great challenges from the increasing system size [2]. In order to alleviate the solution pressure of ACPF, different proposals developed by combining advanced algorithms and modern computation facilities are investigated and evaluated in this paper.

Historically, a lot of promising algorithms are developed for ACPF analysis, of which the Newton-Raphson (NR) [3] and fast decoupled (FD) [4] method received extensive attention due to their favorable convergence characteristics. According to their philosophy, the nonlinear ACPF problem is addressed by a successive solution of linear equation systems (LESs), e.g., (1) and (2) for NR and FD respectively.

$$\begin{bmatrix} \boldsymbol{H} & \boldsymbol{N} \\ \boldsymbol{J} & \boldsymbol{L} \end{bmatrix} \begin{bmatrix} \Delta\boldsymbol{\theta} \\ \Delta\boldsymbol{V}/\boldsymbol{V} \end{bmatrix} = \begin{bmatrix} \Delta\boldsymbol{P} \\ \Delta\boldsymbol{Q} \end{bmatrix}, \tag{1}$$

$$\begin{Bmatrix} \boldsymbol{B}'\boldsymbol{V}\Delta\boldsymbol{\theta} = \Delta\boldsymbol{P}/\boldsymbol{V} \\ \boldsymbol{B}''\Delta\boldsymbol{V} = \Delta\boldsymbol{Q}/\boldsymbol{V} \end{Bmatrix}. \tag{2}$$

Although derived from the NR, the FD is much simpler and more efficient algorithmically [1]. One of the main reason is that the coefficient matrices $\boldsymbol{B}'$ and $\boldsymbol{B}''$ are fixed during the solution process, i.e., the factorization results are reusable, while the decomposition process should be carried out iteration by iteration due to the varied coefficient matrices for the NR method.

Despite the fact that the iterative solver is more desirable for the solution of large-scale LESs in the context of parallel computing [2], we intend to prove in this work that the direct solver is more preferable for the FD. The procedure of the direct solver usually consists of factorization and substitution, which matches the aforementioned property of the FD. In addition, the direct solver is more robust for ill-conditioned problems.

Apart from the numerical algorithms, the computation facility also affects the solution efficiency. In the literature, the ACPF has been fully investigated on CPU architecture, including shared memory computers [5], distributed systems [6], and clusters [7]. Although the reported performance is solid, the computation infrastructure is unaccessible for the majority of researchers. On the other hand, with the advances made in hardware, the graphics processing unit (GPU) has gained a lot of popularity for the scientific computation [8]. GPUs have been reported to offer significant acceleration in several critical power system simulation problems, such as dynamic security assessment [9], electromagnetic transient simulation [10], and dynamic state estimation [11]. The GPU is first introduced for the solution of DC power flow by [12]. In terms of GPU accelerated ACPF solution, the NR method is very popular [13]–[15], while the FD method has not been utilized until recently [2]. In [2], the preconditioned iterative solver is employed to address LESs and the whole solution process is implemented on the GPU; nevertheless, the achieved speedup is limited.

Developed by Nvidia® in the late 2006, the compute unified device architecture (CUDA) [16] programming environment enables researches to extract acceleration possibilities for general purpose computing from Nvidia® GPUs. The CUDA platform is designed for low-level languages, such as C, C++, and Fortran, with the capability of controlling every single thread of a specified block and grid abstractions of the physical GPU cores. As a high-level language, Matlab started to support GPU computing in 2010. Although many built-in functions and toolboxes are enhanced for implementation on GPU, which is beneficial to relieve the low-level programming

978-1-5386-0817-3/17/$31.00 ©2017 IEEE

effort, the performance might be limited since the parallel programming details are hidden and unaccessible.

In this work, the potential of GPU for the solution of ACPF with the FD method based on the direct solver is investigated. Both Matlab and CUDA are selected for implementation. The comparison has been carried out between CPU and GPU platforms, as well as dense and sparse matrix techniques. In addition, the performances of different linear solution strategies and fill-in reduction algorithms are investigated. Numerical experiments are conducted on five benchmark systems ranging from 300 to 13,659 buses. The fastest version of ACPF with Matlab and CUDA for the largest case is 1.303s and 0.313s respectively.

The rest of this paper is organized as follows. A brief introduction of the FD method framework for ACPF is given in Section II. Section III is devoted to the introduction of directive linear solver. GPU implementation details and numerical experiments are presented in Section IV, as well as discussions on the results. Finally, Section V concludes this paper.

II. Fast Decoupled Power Flow

Given a specified network configuration and generator power output, the ACPF determines node voltages and branch power flows such that the system operates under steady-state, i.e., the power imbalance at each bus is less or equal to a predefined tolerance ϵ. The polar form of the nodal power equations is employed by the FD, where the voltage angle and magnitude are separately updated according to $\Delta\boldsymbol{\theta}$ and $\Delta\boldsymbol{V}$, which are obtained from LES (2). In order to solve (2) efficiently, their coefficient matrices $\boldsymbol{B}'$ and $\boldsymbol{B}''$ are factorized at the very beginning, and then at each iteration, the modification step length $\Delta\boldsymbol{\theta}$ and $\Delta\boldsymbol{V}$ can be quickly identified by backward and forward (B/F) substitutions. For a LES, the B/F substitutions dependent on the factorization result of coefficient matrix and the right hand side (RHS) vector. In terms of LES (2), the RHSs are the active and reactive power mismatches, which can be quickly generated by the nodal power equations at each iteration. Fig. 1 depicts a general framework of the FD for the ACPF.

III. Direct Linear Solver

For simplicity, the LES (2) is represented by a standard form in this section,

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}, \tag{3}$$

where the coefficient matrix $\boldsymbol{A}$ is sparse due to the nature of the power system structure.

Generally, after factorization, the lower and upper triangular matrices of a sparse matrix are still sparse [17]. Nevertheless, the fill-ins (matrix entries modified from zero to non-zero by the factorization) are usually inevitable as shown in Fig. 2, which demands extra memory space and more arithmetic operations. Fortunately, it can be greatly reduced by simple row and column switching, whose performance is demonstrated in Fig. 2 by shifting $\boldsymbol{A}$ to $\boldsymbol{B}$. The transformation is commonly described as,

$$\boldsymbol{B} = \boldsymbol{Q}\boldsymbol{A}\boldsymbol{Q}^T, \tag{4}$$

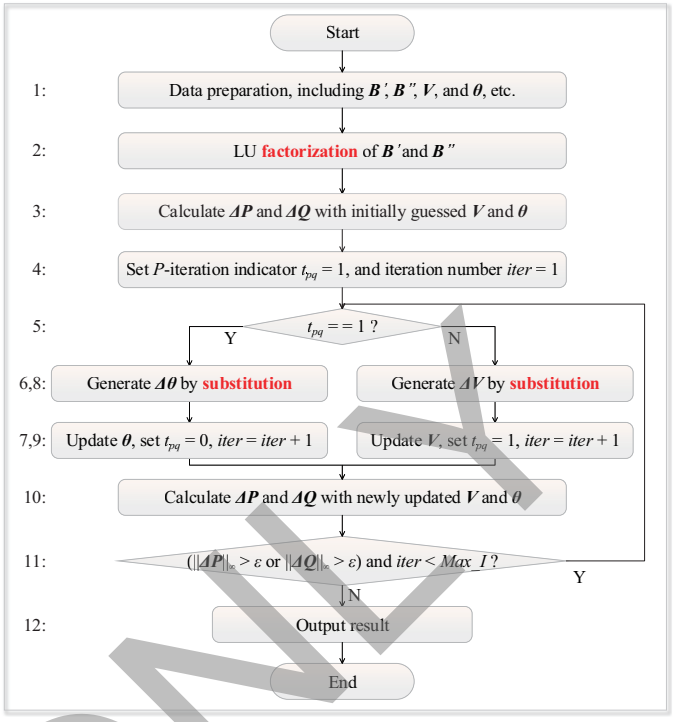

Fig. 1. General framework of the fast decoupled method for power flow analysis.

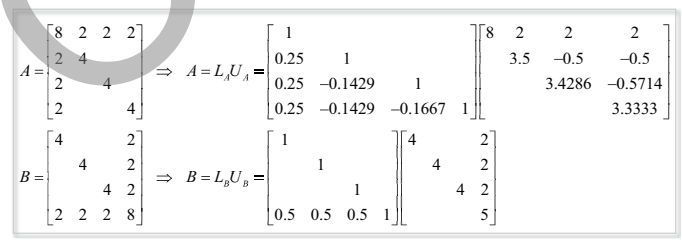

Fig. 2. Difference on the number of fill-ins by row and column switching.

where $\boldsymbol{Q}$ is the permutation matrix derived from permutation array $\boldsymbol{q}$. In terms of Fig. 2, $\boldsymbol{Q}$ and $\boldsymbol{q}$ are given as,

$$\boldsymbol{Q} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \qquad \boldsymbol{q} = \begin{bmatrix} 4 \\ 3 \\ 2 \\ 1 \end{bmatrix}.$$

It should be noted that there is only one entry with value 1 for each row and column in $\boldsymbol{Q}$, while all the other elements are 0. In addition, $\boldsymbol{Q}$ has the following property,

$$\boldsymbol{Q}\boldsymbol{Q}^T = \boldsymbol{Q}^T\boldsymbol{Q} = \boldsymbol{I}. \tag{5}$$

Based on the introduction of $\boldsymbol{Q}$, the following equations can be deduced,

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \;\Rightarrow\; \boldsymbol{A}\boldsymbol{Q}^T\boldsymbol{Q}\boldsymbol{x} = \boldsymbol{b} \;\Rightarrow\; \boldsymbol{Q}\boldsymbol{A}\boldsymbol{Q}^T\boldsymbol{Q}\boldsymbol{x} = \boldsymbol{Q}\boldsymbol{b}. \tag{6}$$

Remark $\boldsymbol{Q}\boldsymbol{x} = \hat{\boldsymbol{x}}$ and $\boldsymbol{Q}\boldsymbol{b} = \hat{\boldsymbol{b}}$, then equation (6) can be rewritten as,

$$\boldsymbol{B}\hat{\boldsymbol{x}} = \hat{\boldsymbol{b}}. \tag{7}$$

TABLE I
GENERAL INFORMATION OF BENCHMARK SYSTEMS

Cases	System scales		B'		B''	
	Bus	Branch	Size	Sparsity	Size	Sparsity
A	300	411	299	0.9875	231	0.9851
B	1,354	1,991	1,353	0.9974	1,094	0.9972
C	2,746	3,279	2,745	0.9988	2,382	0.9988
D	9,241	16,049	9,240	0.9996	7,796	0.9995
E	13,659	20,467	13,658	0.9997	9,567	0.9996

On the basis of the above analysis, the solution process of (3) can be summarized as follows:

- **Step 1**: Generate permutation array $\boldsymbol{q}$ and matrix $\boldsymbol{Q}$.
- **Step 2**: Construct $\boldsymbol{B}$ according to (4) and then factorize it into $\boldsymbol{L}_B$ and $\boldsymbol{U}_B$.
- **Step 3**: Establish $\hat{\boldsymbol{b}}$. Except for the matrix-vector multiplication $\hat{\boldsymbol{b}} = \boldsymbol{Q}\boldsymbol{b}$, the vector $\hat{\boldsymbol{b}}$ can also be quickly generated with,
$$\hat{b}_i = b_{q_i}. \tag{8}$$
- **Step 4**: Deduce $\hat{\boldsymbol{x}}$ with B/F substitution,
$$\boldsymbol{L}_B\hat{\boldsymbol{y}} = \hat{\boldsymbol{b}} \quad \Rightarrow \quad \hat{\boldsymbol{y}} = \boldsymbol{L}_B^{-1}\hat{\boldsymbol{b}}, \tag{9}$$
$$\boldsymbol{U}_B\hat{\boldsymbol{x}} = \hat{\boldsymbol{y}} \quad \Rightarrow \quad \hat{\boldsymbol{x}} = \boldsymbol{U}_B^{-1}\hat{\boldsymbol{y}}. \tag{10}$$
- **Step 5**: Retrieve the final result $\boldsymbol{x}$ by any of the following methods,
$$\boldsymbol{x} = \boldsymbol{Q}^T\hat{\boldsymbol{x}}, \tag{11}$$
$$x_{q_i} = \hat{x}_i. \tag{12}$$

IV. NUMERICAL EXPERIMENTS WITH GPU

A. Benchmark Systems

Five benchmark systems retrieved from [18] are utilized for numerical experiments. Table I summarizes basic information on the power system scale, matrix size, and sparsity. The implementation platform includes: Intel Xeon E5-2620 CPU with 32GB RAM, Nvidia® GeForce Titan Black GPU, Matlab version 2015b, CUDA version 8.0, and Visual Studio 2015.

B. GPU Implementation with Matlab

1) GPU Programming Features in Matlab: Without user intervention, the Matlab code will run on the CPU and all data will be stored in the workspace allocated by Matlab in CPU. On the other hand, the GPU also provides a few Gigabytes of space called device memory. All the data stored in the device memory should be in the type of `gpuArray`. The data transformation from CPU to GPU is explicitly fulfilled by the function `gpuArray()`, or it can also be performed implicitly by any GPU-enabled built-in functions (GEBFs), such as `mtimes()`. A full list of the latest GEBFs is posted in [19]. In contrast, retrieving data from GPU to CPU can be achieved by the function `gather()`.

The type of the input data determines where the GEBF will be executed. If any input arguments are with the type of `gpuArray`, the GEBF will be executed on the GPU; otherwise, the CPU will be utilized for calculation. Therefore, the simplest way to employ GPU for computation in Matlab is to employ two steps: 1) convert all the input data into `gpuArray` type; and 2) fetch results from device memory after the algorithm termination. Intermediate data generated from GEBFs running on GPU will be automatically stored in device memory in the type of `gpuArray`. The data transfer rate between CPU and GPU is limited by the PCIe interface bandwidth.

TABLE II
EXECUTION TIME OF DIFFERENT TYPES OF FD WITH DENSE MATRICES USING MATLAB (S)

Cases	`lu()`		`mldivide()`	
	CPU	GPU	CPU	GPU
A	**0.018**	0.257	0.063	0.160
B	**0.282**	1.402	0.751	0.762
C	**1.420**	8.048	6.606	4.203
D	**17.413**	out of memory	141.371	out of memory
E	**37.847**	out of memory	327.209	out of memory

2) Implementation Strategies: In order to explore the performance of the FD for ACPF in detail, different data storage formats, LES solution techniques, and implementation platforms are investigated and compared, which can be divided into the following three pairs:

- **CPU versus GPU**: As two different architectures, CPU and GPU have distinctive area of expertise. Generally, CPU is suitable for randomly accessed computing, while GPU is skillful for intensively regulated calculation.
- **`lu()` versus `mldivide()`**: Except for the factorization strategy introduced in Section III, which is based on the GEBF `lu()`, Matlab also provides another powerful LES solution technique `mldivide()`. The former gains profits from the iterative process of ACPF, where the coefficient matrices of LESs are fixed. Based on the detection of the coefficient matrix property, the latter dispatches an appropriate solver from its formidable arsenal to minimize the computation time.
- **Dense versus Sparse**: As shown in Table I, the coefficient matrices $\boldsymbol{B}'$ and $\boldsymbol{B}''$ are extremely sparse and suitable for sparse technique application. Nevertheless, the two GEBFs `lu()` and `mldivide()` do not support sparse `gpuArray` at present, i.e., the sparse version of FD on GPU is not feasible on the basis of GEBFs. Therefore, only the dense FD is implemented on the GPU.

3) Experimental Results and Discussion: All the results are grouped into Table II and Table III according to dense and sparse storage types respectively. Fig. 3 and Fig. 4 give the visualization for Table II and Table III for the purpose of identifying the increasing trend of execution time along with system size. The following observations can be collected corresponding to the above comparison categories:

- **CPU versus GPU**: Since there is no GPU result in Table

TABLE III
EXECUTION TIME OF DIFFERENT TYPES OF FD WITH SPARSE MATRICES USING MATLAB (S)

Cases	`lu()`		`mldivide()`	
	CPU	GPU	CPU	GPU
A	**0.008**	not supported	0.015	not supported
B	0.062	not supported	**0.059**	not supported
C	0.198	not supported	**0.173**	not supported
D	4.143	not supported	**0.964**	not supported
E	7.986	not supported	**1.303**	not supported

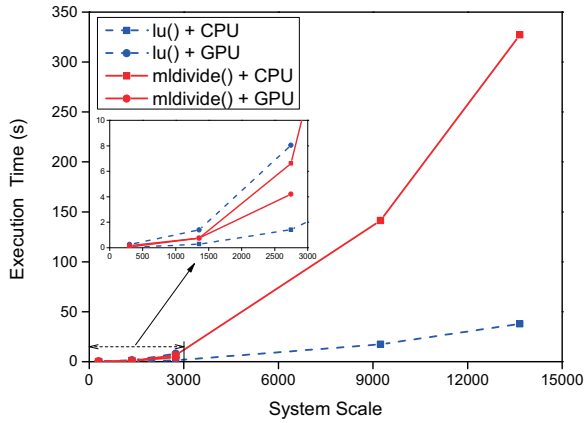

Fig. 3. Execution time of different types of FD with dense matrices.

III, the finding is drawn from Table II and Fig. 3. If `lu()` is employed, the CPU is always faster than GPU, but the speedup decreases from 14.5× in `CaseA` to 5.7× in `CaseC`. As highlighted in Table II, GPU outperforms CPU in `CaseC` where `mldivide()` is utilized. Overall, two remarks should be given: 1) GPU performs better for larger systems; and 2) the limited device memory space restricts its utilization for large-scale systems with dense matrices.

- **`lu()` versus `mldivide()`**: To evaluate the performance of `lu()` and `mldivide()`, the implementation platform should be separated. On the GPU, the superiority of `mldivide()` has been validated by all successive cases. On the other hand, if run on CPU, `lu()` outperforms `mldivide()` with dense matrices in Table II; nevertheless, the circumstance is totally reversed for sparse matrices in Table III. Therefore, the superiority depends on which architecture is utilized.
- **Dense versus Sparse**: According to Table II and Table III, it is obvious that the sparse techniques benefit both `lu()` and `mldivier()` in CPU. Although dense matrix is fully supported with GPU, the performance is only mediocre. On the contrary, the support for GPU with sparse matrices requires further investigation.

In addition to the above findings corresponding to implementation, more observations related with computation complexity and scalability are accessible. Without rigorous

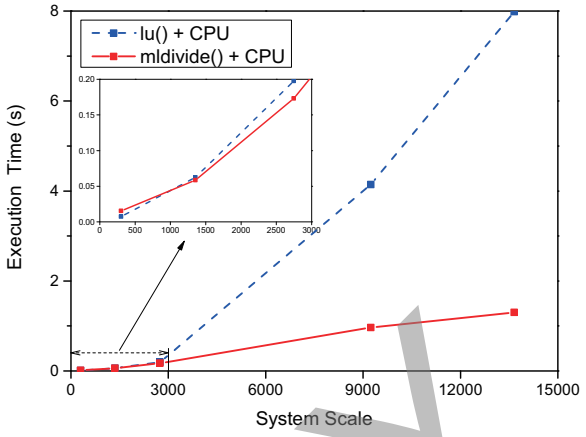

Fig. 4. Execution time of different types of FD with sparse matrices.

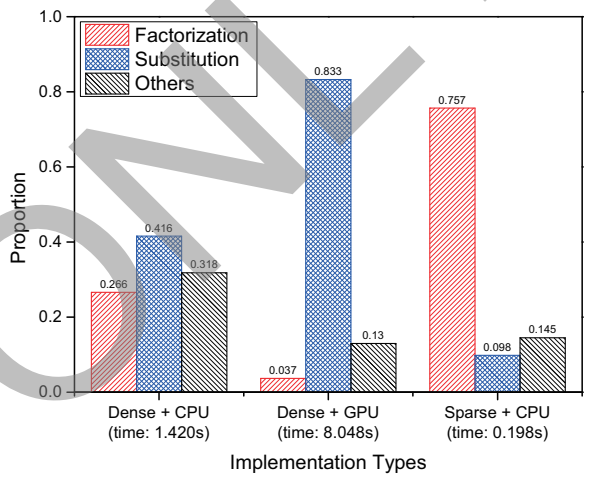

Fig. 5. Execution time proportions of different steps for the FD with `lu()`.

mathematical analysis, the execution time in the same platform can be regarded as an indication of computation burden. For each line in Fig. 3 and Fig. 4, the computing environment of all five cases is the same; therefore, the line increasing trend represents the computation complexity. It can be seen in Fig. 3 that all lines are steep, which means the execute time increases faster than the system scales. On the other hand, the solid line in Fig. 4 is the mostly flat, i.e., the scalability of sparse `mldivide()` is more favorable.

The main steps of FD shown in Fig. 1 and Section II are also analyzed. Fig. 5 illustrates the execution time proportion of main steps for `CaseC`. It can be seen that the substitution process, which is highly sequential, heavily drags the performance in GPU with dense matrices. The improvement on sparse matrices should be put on the factorization process in the future since it consumes the largest amount of time.

C. GPU Implementation with CUDA

1) GPU Programming Features with CUDA: Different with C functions running on the CPU only once with one call, the `kernels` are CUDA C extended functions that can be executed N times simultaneously with N different threads.

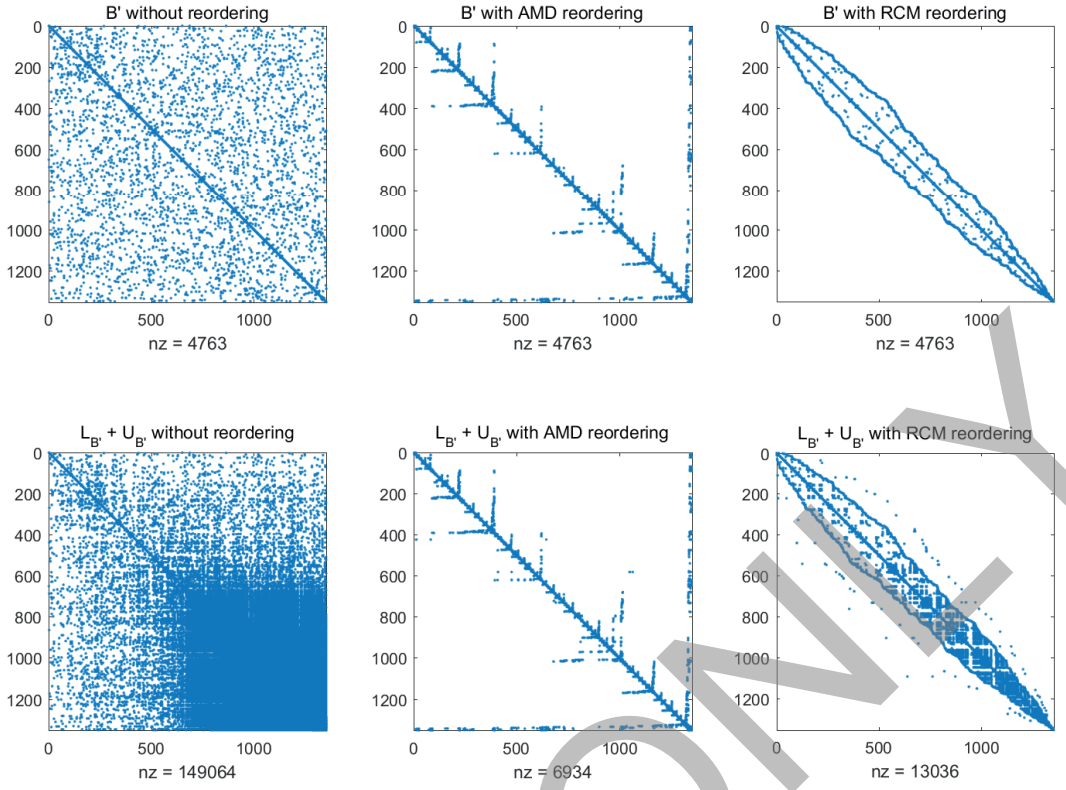

Fig. 6. Sparsity structure of $\boldsymbol{B}'$ and $\boldsymbol{L}'_B + \boldsymbol{U}'_B$ for `caseB`.

`Kernels` access the input data from device memory spaces, including global, constant, texture, shared, and local memories [16]. The memory throughput and multiprocessor occupancy achieved by the `kernels` greatly determine the parallel efficiency of the whole application, which demands careful code tuning and proper algorithm structure design. Fortunately, a lot of GPU-accelerated libraries containing highly-optimized algorithms and functions are provided by CUDA [20], such as cuBLAS, cuSPARSE, and cuSOLVER.

2) Implementation Schemes: Although execution on GPU with single data type is much faster, it cannot meet the precision requirement of $\epsilon = 10^{-8}$; therefore, the double precision data is utilized in this work. Except for the data preparation and condition judgments, the majority of FD steps shown in Fig. 1 are fulfilled with the refined `kernels` contained in cuSOLVER, such as LU factorization and substitution. As indicated in Section III, for sparse coefficient matrices, the permutation is of key importance for the reduction of the fill-ins. Two strategies for reordering provided by cuSOLVER are implemented, i.e., reverse Cuthill-Mckee (RCM) and approximate minimum degree (AMD) algorithms. The intuitive performance of RCM and AMD is illustrated in Fig. 6, where $\boldsymbol{B}'$ is generated from `CaseB`. It can be seen that both AMD and RCM gain excellent performance by curtailing the number of fill-ins from 149,064 to 6,934 and 13,036 respectively, with the reduction rate reaching 95.35% and 91.25% respectively. The behavior of AMD and RCM for other cases are summarized in Table IV.

TABLE IV
FILL-IN REDUCTIONS ACHIEVED BY THE AMD AND RCM ALGORITHMS

Cases	Default	AMD reordering		RCM reordering	
	Size	Size	Reduction	Size	Reduction
A	7,889	1,640	79.21%	2,515	68.12%
B	149,064	6,934	95.35%	13,036	91.25%
C	451,657	17,328	96.16%	58,326	87.09%
D	3,709,484	65,876	98.22%	200,921	94.58%
E	4,078,641	79,751	98.04%	228,221	94.40%

3) Experimental Results and Discussion: It is noticeable in Table IV that the AMD outperforms RCM in the fill-in reduction; nevertheless, the performance is reversed when they are integrated in the FD, which is shown in Fig. 7. The speedup of RCM over AMD is also demonstrated in Fig. 7, which indicates that the difference is even higher for large-scale systems. One of the explanation for this reversal is that the AMD pursues more powerful algorithmic performance with the sacrifice of longer execution time, which means the AMD is more preferable for memory-restricted circumstances.

Several types of FD coded with Matlab are implemented in Section IV.B; however, the performance of GPU-enabled ones is unsatisfactory. Therefore, the most efficient CPU version (sparse matrix and `mldivide()` with Matlab running on CPU, the fourth column of Table III) is utilized in this subsection for comparison with AMD and RCM, whose

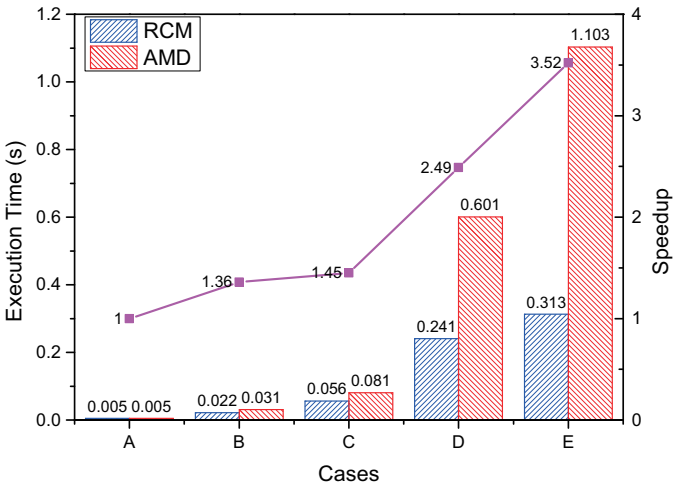

Fig. 7. Execution time of FD with cuSOLVER based on AMD and RCM.

TABLE V
SPEEDUPS GAINED BY THE AMD AND RCM ALGORITHMS IMPLEMENTED WITH CUDA OVER THE FASTEST MATLAB IMPLEMENTATION

Algorithms	CaseA	CaseB	CaseC	CaseD	CaseE
AMD	3.05	1.95	2.14	1.60	1.18
RCM	3.05	2.66	3.09	4.00	4.16

execution time is given in Fig. 7. Table V summarizes the results. Although AMD is much slower than RCM, it is still more efficient than the Matlab implementation. RCM gains a maximum speedup of 4.16× over the fastest Matlab execution in `CaseE` with only 0.313s.

V. CONCLUSION

To investigate the GPU-accelerated ACPF solution performance, the FD method based on the direct linear solver is implemented and analyzed. The comparison is conducted on different architectures, data storage formats, and fill-in reduction algorithms with five benchmark systems ranging from 300 to 13,659 buses. The GPU implementation with Matlab is restricted to dense matrices and the performance is unsatisfactory. Although sparse matrices running on CPU is acceptable with Matlab, it is slower than the CUDA version with both AMD and RCM. The CUDA with RCM is the most promising of all investigated implementations. Furthermore, the obtained speedup for a single power flow solution may not appear significant; however, mulitple power flow simulations for contingency analysis exploring the GPU's massive data parallelism can be expected to provide higher speedups. For future work, the promising FD method with sparse matrices based on direct linear solver with Matlab and CUDA is reserved for investigation on newer GPU architectures.

VI. ACKNOWLEDGE

S. Huang was sponsored by the China Scholarship Council (CSC) under Grant No. 201403170337. This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

REFERENCES

[1] X. Wang, Y. Song, and M. Irving, *Modern power systems analysis*. New York, NY, USA: Springer, 2008.
[2] X. Li, F. Li, H. Yuan, H. Cui, and Q. Hu, "GPU-based fast decoupled power flow with preconditioned iterative solver and inexact Newton method," *IEEE Trans. Power Syst.*, vol. PP, no. 99, pp. 1–9, 2017.
[3] W. F. Tinney and C. E. Hart, "Power flow solution by Newton's method," *IEEE Trans. Power Appar. Syst.*, vol. PAS-86, no. 11, pp. 1449–1460, Nov. 1967.
[4] B. Stott and O. Alsac, "Fast decoupled load flow," *IEEE Trans. Power Appar. Syst.*, vol. PAS-93, no. 3, pp. 859–869, May 1974.
[5] T. Cui, R. Yang, G. Hug, and F. Franchetti, "Accelerated AC contingency calculation on commodity multi-core SIMD CPUs," in *Proc. IEEE Power Energy Soc. Gen. Meeting,*, MD, USA, Jul. 2014, pp. 1–5.
[6] X. Yang, C. Liu, and J. Wang, "Large-scale branch contingency analysis through master/slave parallel computing," *J. Mod. Power Syst. Clean Energy*, vol. 1, no. 2, pp. 159–166, Sept. 2013.
[7] Z. Huang, Y. Chen, and J. Nieplocha, "Massive contingency analysis with high performance computing," in *Proc. IEEE Power Energy Soc. Gen. Meeting,*, Calgary, AB, Canada, Jul. 2009, pp. 1–8.
[8] N. Ploskas and N. Samaras, *GPU programming in MATLAB*, 1st ed. Cambridge, MA, USA: Elsevier, 2016.
[9] V. Jalili-Marandi and V. Dinavahi, "Simd-based large-scale transient stability simulation on the graphics processing unit," *IEEE Trans. Power Syst.*, vol. 25, no. 3, pp. 1589–1599, Aug. 2010.
[10] Z. Zhou and V. Dinavahi, "Parallel massive-thread electromagnetic transient simulation on GPU," *IEEE Trans. Power Delivery*, vol. 29, no. 3, pp. 1045–1053, Jun. 2014.
[11] H. Karimipour and V. Dinavahi, "Extended kalman filter-based parallel dynamic state estimation," *IEEE Trans. Smart Grid*, vol. 6, no. 3, pp. 1539–1549, May 2015.
[12] A. Gopal, D. Niebur, and S. Venkatasubramanian, "DC power flow based contingency analysis using graphics processing units," in *Proc. IEEE Power Tech.*, Lausanne, Switzerland, Jul. 2007, pp. 731–736.
[13] V. Roberge, M. Tarbouchi, and F. Okou, "Parallel power flow on graphics processing units for concurrent evaluation of many networks," *IEEE Trans. Smart Grid*, vol. PP, no. 99, pp. 1–10, Nov. 2015.
[14] D. Chen, H. Jiang, Y. Li, and D. Xu, "A two-layered parallel static security assessment for large-scale grids based on GPU," *IEEE Trans. Smart Grid*, vol. PP, no. 99, pp. 1–10, Aug. 2016.
[15] G. Zhou, Y. Feng, R. Bo, L. Chien, X. Zhang, Y. Lang, Y. Jia, and Z. Chen, "GPU-accelerated batch-ACPF solution for N-1 static security analysis," *IEEE Trans. Smart Grid*, vol. PP, no. 99, pp. 1–11, Aug. 2016.
[16] NVIDIA, "CUDA C programming guide 8.0," Santa Clara, CA, USA, 2017.
[17] T. A. Davis, *Direct methods for sparse linear systems*. Philadelphia, PA, USA: SIAM, 2006.
[18] R. D. Zimmerman, C. E. Murillo-Sanchez, and R. J. Thomas, "MATPOWER: Steady-state operations, planning, and analysis tools for power systems research and education," *IEEE Trans. Power Syst.*, vol. 26, no. 1, pp. 12–19, Feb. 2011.
[19] MathWorks, "Run built-in functions on a GPU," [Online], available: https://www.mathworks.com/help/distcomp/run-built-in-functions-on-a-gpu.html.
[20] NVIDIA, "GPU-accelerated libraries," [Online], available: https://developer.nvidia.com/gpu-accelerated-libraries.