IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received May 27, 2018, accepted August 6, 2018, date of publication August 10, 2018, date of current version September 5, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2864757

Real-Time Contingency Analysis on Massively Parallel Architectures With Compensation Method

SHENGJUN HUANG[1,2] AND VENKATA DINAVAHI[1], (Senior Member, IEEE)
[1]Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB T6G 2V4, Canada
[2]College of Systems Engineering, National University of Defense Technology, Changsha 410073, China

Corresponding author: Shengjun Huang (shengjun@ualberta.ca)

This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC). The work of S. Huang was supported by the China Scholarship Council (CSC) under Grant 201403170337.

ABSTRACT Real-time contingency analysis (RTCA) is paramount for modern power systems as it forms the basis for important operator actions that help to improve system stability, optimize generator dispatch, manage disparate resources, prevent cascading outages, and enhance market operations. With increasing system size and the number of contingency scenarios, RTCA is faced with computational challenges. To alleviate this situation, massively parallel graphics processing units (GPUs) are introduced for the acceleration of RTCA solution in this paper, where the compensation method (CM) is utilized for the concurrent AC power flow solution. Strategies and principles on the data structure, kernel function, and memory management are provided. Five benchmark systems (ranging from 300- to 13,659-bus) are employed for case studies. Based on the sequential CM implemented on single-thread CPU, the performance analysis related to execution time and speedup is carried out for parallel CMs running on other architectures, including multi-thread CPU, single-GPU, and multi-GPUs. Results indicate that the parallel CM with multi-GPUs has sufficient accuracy, convergence, and scalability. Finally, the potential of the proposal for practical RTCA has been discussed with the reviewing of other state-of-the-art parallel computing methods reported in the literature.

INDEX TERMS Compensation method, fast decoupled power flow, graphics processing unit, parallel computing, real-time contingency analysis.

I. INTRODUCTION

Real-Time Contingency Analysis (RTCA), which seeks to assess the ability of a grid to withstand cascading component failures/contingencies within a specified time span (a few minutes) [1], is paramount for the reliable operation of modern power systems, underpinning system stability and economic prosperity [2]. In RTCA, each grid configuration with component outages is simulated as a scenario, whose feasibility is evaluated by the solution of nonlinear Alternating Current Power Flow (ACPF). If collapse consequence is detected in any scenario, preventive and corrective operations should be immediately implemented based on the analysis results. Therefore, fast solution of multiple ACPFs/scenarios is of great significance for RTCA. Nevertheless, the number of possible contingency scenarios for $N-k$ security criteria is $\sum_{n=1}^{k} \frac{N!}{n!(N-n)!}$, which holds an exponential relationship with N and k. During the last few decades, the system scale has expanded and more stringent criteria have been proposed by the North American Electric Reliability Corporation (NERC) [3], i.e., both N and k are increased, which introduces great challenges for RTCA in the context of smart grid as well as real-time operation.

Conventionally, two directions have been exploited to alleviate the difficult compromise between a large number of contingency scenarios and the limited solution time:

- *Robust selection procedure:* Since the evaluation of all scenarios is impractical, a subset is usually generated for analysis according to appropriate selection rules, such as the Performance Index (PI) contingency ranking method [4]. The size of the subset has a considerable impact on the solution process of RTCA: if it is very large (the conservative principle), the subsequent evaluation burden would be heavy; if it is relatively small

(the progressive principle), the critical scenarios might be skipped.

- *Fast evaluation methodology:* The cardinality of the contingency set can still be very large for practical systems even if progressive selection strategy is utilized, therefore the High-Performance Computing (HPC) platform should be resorted for acceleration. If the fast evaluation methodology can process thousands of scenarios with acceptable accuracy and efficiency, it may dominate the solution process of RTCA and relieve the pressure of selection procedure, which formulates the original motivation for this paper.

Investigations on RTCA with HPC were implemented on the multi-core Central Processing Unit (CPU) architecture [1], [5]–[10], such as shared memory computers, distributed systems, and CPU clusters. Although the performance reported is acceptable, their application scope is restricted since most engineers and researchers have limited access to supercomputers and large-scale CPU clusters. On the other hand, the many-core Graphics Processing Unit (GPU) architecture is accessible for the common researcher with PC or workstation. It has received significant attention in HPC [11]–[20] as a result of its higher floating-point performance, massive numbers of threads, lower power consumption, and lower thread launch time in comparison to its CPU counterpart. Therefore, GPU is determined as the implementation platform in this work.

On the GPU platform, both the Direct Current Power Flow (DCPF) and ACPF have been implemented to formulate the RTCA by [11]–[13] and [14]–[20] respectively. The DCPF is a linear simplification of real systems, with great advantages on the computation efficiency and convergence; however, the accuracy and capability of the solution are insufficient, e.g., inability to check voltage limit violations. On the contrary, the nonlinear ACPF is more accurate but complicated. In the literature, several sophisticated methods have been implemented on GPU to address ACPF-based RTCA, such as the Newton-Raphson (NR) method [14]–[19] and the Fast Decoupled (FD) method [14], [20]. Compared with NR, the FD is more efficient algorithmically [4] due to: 1) The Jacobian matrices $\boldsymbol{B}'$ and $\boldsymbol{B}''$ need to be factorized only once for the whole process of FD, while in NR the Jacobian matrix $\boldsymbol{J}$ should be decomposed in each iteration; 2) The dimension of $\boldsymbol{B}'$ and $\boldsymbol{B}''$ is less than or equal to half of the dimension of the original $\boldsymbol{J}$, thus the total factorization time of $\boldsymbol{B}'$ and $\boldsymbol{B}''$ is less than that of $\boldsymbol{J}$ as the time complexity of the fastest factorization algorithm is $O(\frac{2}{3}n^3)$ [21].

The process of FD for single-scenario RTCA problem can be concluded as the solution of a series of $\boldsymbol{A}\boldsymbol{x}_i = \boldsymbol{b}_i$ (i is the iteration index), for which the direct method [13]–[19] seems to be more attractive than the iterative method [11], [20] since $\boldsymbol{A}$ is fixed. Taking the LU decomposition as an example, $\boldsymbol{A}$ is factorized into $\boldsymbol{L}$ and $\boldsymbol{U}$ matrices, and $\boldsymbol{y}_i$ is generated from $\boldsymbol{L}\boldsymbol{y}_i = \boldsymbol{b}_i$ by forward substitution, then $\boldsymbol{x}_i$ is deduced from $\boldsymbol{U}\boldsymbol{x}_i = \boldsymbol{y}_i$ by backward substitution. During the whole process, the factorization takes the majority of the time even though it is executed only once while Forward and Backward (F/B) substitutions are performed several times. In order to accelerate the solution, efforts have been put forward on the sparse LU decomposition, such as reducing the number of fill-ins [21].

Nevertheless, in reality, RTCA comes with large numbers of scenarios, which can be represented as $\boldsymbol{A}_k\boldsymbol{x}_i = \boldsymbol{b}_i$ (k distinguishes different scenarios). The solution methodology depicted above for single-scenario RTCA can be easily extended into multi-scenario RTCA since each scenario is spontaneously independent. According to the classical FD method, each $\boldsymbol{A}_k$ should be factorized to perform the F/B substitutions. In order to boost the efficiency to a higher extent, the concept of Compensation Method (CM) is proposed [4], [22], which factorizes the matrix $\boldsymbol{A}$ of base case for only once, and deduces compensation factors for each scenario during the F/B substitutions, thus the time and effort corresponding to LU decomposition for all the other scenarios can be saved. In addition to the superb solution efficiency, the CM possesses satisfactory accuracy. Theoretically, the CM is integrated within the framework of FD, and FD derives from the same mathematical formulation with NR, therefore, the CM, FD, and NR should generate the same accuracy of results if the same convergence criterion is adopted. Different from the FD and NR with high popularity, the CM has not been reported with GPU architecture to the best of our knowledge, thus we intend to fill this gap.

Based on the sparse matrix techniques and double-precision floating-point format, the CM is implemented on GPU parallel architecture. Instead of the trivial integrated kernel design strategy, the decoupled kernels are developed to achieve high performance. Principles for memory management and multi-GPUs implementation are also presented. The accuracy and convergence properties of the proposed parallel implementation scheme are validated by the open source package Matpower [23], where five test systems ranging from 300 to 13,659 buses are utilized. In order to evaluate the performance of CM on parallel architectures, the sequential CM executed on single-thread CPU is determined as the benchmark. Results for full $N-1$ RTCA with parallel CM on multi-thread CPU, single-GPU, and multi-GPUs are collected, regulated, and compared, showing that the GPU is superior on the speedup and scalability over CPU for RTCA with CM. Finally, discussions on the potential of parallel CM for practical RTCA are provided based on the reviewing of other state-of-the-art parallel computing methods reported in the literature.

The remainder of this paper is organized as follows. Section II derives the solution framework of CM with detailed formulations. Parallel implementation strategies and principles with GPU are illustrated in Section III. Case studies and discussions are provided in Section IV. Finally, Section V concludes this paper.

II. COMPENSATION METHOD

Given a specified ACPF, the efficient FD method can be summarized as the iterative solution of correction equations:

$$\Delta \boldsymbol{P}/\boldsymbol{V} = \boldsymbol{B}'\Delta\boldsymbol{\theta}, \tag{1}$$

$$\Delta \boldsymbol{Q}/\boldsymbol{V} = \boldsymbol{B}''\Delta\boldsymbol{V}, \tag{2}$$

where $\Delta\boldsymbol{\theta}$ and $\Delta\boldsymbol{V}$ are decision vectors; $\boldsymbol{P}$, $\boldsymbol{Q}$, $\boldsymbol{V}$, and $\boldsymbol{\theta}$ are nodal active power, reactive power, voltage magnitude, and phase angle, respectively; Δ represents the error; $\boldsymbol{B}'$ and $\boldsymbol{B}''$ are symmetric square matrices with the dimension of $n-1$ and $n-r-1$, whose off-diagonal and diagonal elements are given in (3) and (4); n is the total number of buses and r is the number of PV nodes.

$$\boldsymbol{B}'_{ij} = -\frac{x_{ij}}{r_{ij}^2+x_{ij}^2}, \quad \boldsymbol{B}'_{ii} = \boldsymbol{B}'_{jj} = \sum_{j\in i}\frac{x_{ij}}{r_{ij}^2+x_{ij}^2}, \tag{3}$$

$$\boldsymbol{B}''_{ij} = \frac{-1}{k_{ij}x_{ij}}, \quad \boldsymbol{B}''_{jj} = \sum_{j\in i}\left(\frac{1}{x_{ij}}+0.5b_{ij}\right), \ \boldsymbol{B}''_{ii} = \frac{\boldsymbol{B}''_{jj}}{k_{ij}^2}, \tag{4}$$

where r_{ij}, x_{ij}, b_{ij}, and k_{ij} are the resistance, reactance, total line charging susceptance, and transformer off nominal turns ratio of branch ij, respectively.

Instead of single ACPF, $n_c \gg 1$ scenarios/ACPFs need to be evaluated for the solution of practical RTCA. Based on the aforementioned solution process, there will be n_c times of LU decomposition for $\boldsymbol{B}'$ and $\boldsymbol{B}''$, which is a heavy computational workload. Fortunately, the CM [22] is capable to reduce the number of LU factorization from n_c to 1 since the coefficient matrices for different scenarios are similar.

Suppose $\boldsymbol{B}' = \boldsymbol{L}'\boldsymbol{U}'$ and $\boldsymbol{B}'' = \boldsymbol{L}''\boldsymbol{U}''$ are matrices generated from the base case without branch outage, the following process is devoted to derive $\Delta\boldsymbol{\theta}$ and $\Delta\boldsymbol{V}$ from (1) and (2) for a new scenario (where the branch ij is out of service) without LU factorization operations. For simplicity, only the solution steps of (1) are given; (2) can be addressed accordingly.

1) Based on the topology variation induced by the outage of branch ij, $\boldsymbol{B}'$ can be updated as:

$$\boldsymbol{B}'_* = \boldsymbol{B}' + \Delta\boldsymbol{B}'_* = \boldsymbol{B}' + \boldsymbol{M}'_*\delta\boldsymbol{b}'_*\boldsymbol{M}'^T_*, \tag{5}$$

where subscript $_*$ is a stamp for the specified scenario, i.e., outage of branch ij; $\delta\boldsymbol{b}'_*$ is a $m\times m$ matrix containing correction information, $m = \{1, 2\}$; $\boldsymbol{M}'_*$ is a $n\times m$ incidence matrix relates to i and j. The details on generating $\delta\boldsymbol{b}'_*$, $\boldsymbol{M}'_*$, and m will be demonstrated later.

2) Calculate intermediate matrix:

$$\boldsymbol{c}'_* = \left[\boldsymbol{I} + \delta\boldsymbol{b}'_*\boldsymbol{M}'^T_*(\boldsymbol{U}'^{-1}(\boldsymbol{L}'^{-1}\boldsymbol{M}'_*))\right]^{-1}\delta\boldsymbol{b}'_*, \tag{6}$$

where $\boldsymbol{I}$ is a $m\times m$ identical matrix.

3) Calculate voltage angle variation vector:

$$\Delta\boldsymbol{\theta}_1 = \boldsymbol{L}'^{-1}(\Delta\boldsymbol{P}_*/\boldsymbol{V}_*), \tag{7}$$

$$\Delta\boldsymbol{\theta}_2 = \Delta\boldsymbol{\theta}_1 - \boldsymbol{L}'^{-1}\left(\boldsymbol{M}'_*\boldsymbol{c}'_*\boldsymbol{M}'^T_*(\boldsymbol{U}'^{-1}\Delta\boldsymbol{\theta}_1)\right), \tag{8}$$

$$\Delta\boldsymbol{\theta}_* = \boldsymbol{U}'^{-1}\Delta\boldsymbol{\theta}_2. \tag{9}$$

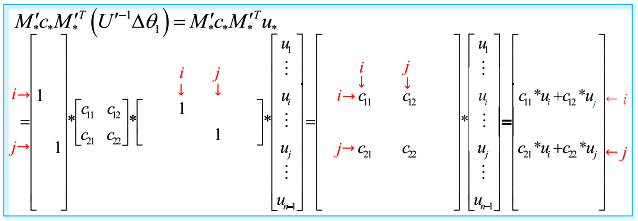

FIGURE 1. Demonstration of fixed pattern calculation.

Algorithm 1 Data Preparation of $\delta\boldsymbol{b}'_*$ and $\boldsymbol{M}'_*$

1: **if** Node i is a slack bus **then**
2: Let $\boldsymbol{M}'_* = \boldsymbol{e}'_j$ and $\delta\boldsymbol{b}'_* = -\frac{x_{ij}}{r_{ij}^2+x_{ij}^2}$.
3: **else**
4: **if** Node j is a slack bus **then**
5: Let $\boldsymbol{M}'_* = \boldsymbol{e}'_i$ and $\delta\boldsymbol{b}'_* = -\frac{x_{ij}}{r_{ij}^2+x_{ij}^2}$.
6: **else**
7: Set $\rho' = -\frac{x_{ij}}{r_{ij}^2+x_{ij}^2}$,
8: Let $\boldsymbol{M}'_* = \begin{bmatrix}\boldsymbol{e}'_i & \boldsymbol{e}'_j\end{bmatrix}$ and $\delta\boldsymbol{b}'_* = \begin{bmatrix}\rho' & -\rho'\\ -\rho' & \rho'\end{bmatrix}$.
9: **end if**
10: **end if**
11: Output $\delta\boldsymbol{b}'_*$ and $\boldsymbol{M}'_*$.

The detailed derivation process of the above steps based on inverse matrix modification lemma [24] is given in Appendices A and B.

It should be noted that all the inverse operations of $\boldsymbol{L}'$ and $\boldsymbol{U}'$ can be performed by F/B substitutions to reduce the computation burden, which are marked in the above by parentheses. Although the inverse operation indicated by brackets in (6) is inevitable, fortunately, the matrix size ($m\times m$) is limited. The matrix inverse operations for $m=1$ and $m=2$ are trivial, i.e.,

$$[a]^{-1} = \left[\tfrac{1}{a}\right] \quad \text{and} \quad \begin{bmatrix}a & b\\ c & d\end{bmatrix}^{-1} = \begin{bmatrix}\frac{d}{ad-bc} & \frac{-b}{ad-bc}\\ \frac{-c}{ad-bc} & \frac{a}{ad-bc}\end{bmatrix}. \tag{10}$$

More notes are given for the calculations related with $\boldsymbol{M}'_*$. Due to its highly sparse and fixed pattern, the calculation can be predefined to save time and effort, i.e., only few fixed points of the final result should be calculated and filled. Take $\left(\boldsymbol{M}'_*\boldsymbol{c}'_*\boldsymbol{M}'^T_*(\boldsymbol{U}'^{-1}\Delta\boldsymbol{\theta}_1)\right)$ in (8) as an example, suppose $m=2$ and rewrite the dense column vector $(\boldsymbol{U}'^{-1}\Delta\boldsymbol{\theta}_1)$ as $\boldsymbol{u}_*$, the solution process is illustrated by Fig. 1, where the final result can be directly derived and filled.

The critical step of utilizing CM is constructing $\delta\boldsymbol{b}'_*$ and $\boldsymbol{M}'_*$, which dominates the solution steps (6) – (9). The pattern and value of $\delta\boldsymbol{b}'_*$ and $\boldsymbol{M}'_*$ are dependent on the parameters of branch ij and the node type of bus i and j. The pseudo code of generating $\delta\boldsymbol{b}'_*$ and $\boldsymbol{M}'_*$ is summarized in **Algorithm 1**, where $\boldsymbol{e}'_i$ and $\boldsymbol{e}'_j$ are basis vectors with size $n-1$.

Algorithm 2 Data Preparation of $\delta \boldsymbol{b}''_*$ and $\boldsymbol{M}''_*$

if Node i is a PQ bus **then**
  if Node j is a PQ bus **then**
    Set $\sigma = \frac{1}{k_{ij} x_{ij}}$ and $\rho'' = -\frac{1}{x_{ij}} - 0.5 b_{ij}$,
    Let $\boldsymbol{M}''_* = \begin{bmatrix} \boldsymbol{e}''_i & \boldsymbol{e}''_j \end{bmatrix}$ and $\delta \boldsymbol{b}''_* = \begin{bmatrix} \rho''/k_{ij}^2 & \sigma \\ \sigma & \rho'' \end{bmatrix}$.
  else
    Let $\boldsymbol{M}''_* = \boldsymbol{e}''_i$ and $\delta \boldsymbol{b}''_* = -\frac{1}{x_{ij}} - 0.5 b_{ij}$.
  end if
else
  if Node j is a PQ bus **then**
    Let $\boldsymbol{M}''_* = \boldsymbol{e}''_j$ and $\delta \boldsymbol{b}''_* = -\frac{1}{x_{ij}} - 0.5 b_{ij}$.
  else
    Let $\boldsymbol{M}''_* = \boldsymbol{0}$ and $\delta \boldsymbol{b}''_* = 0$.
  end if
end if
Output $\delta \boldsymbol{b}''_*$ and $\boldsymbol{M}''_*$.

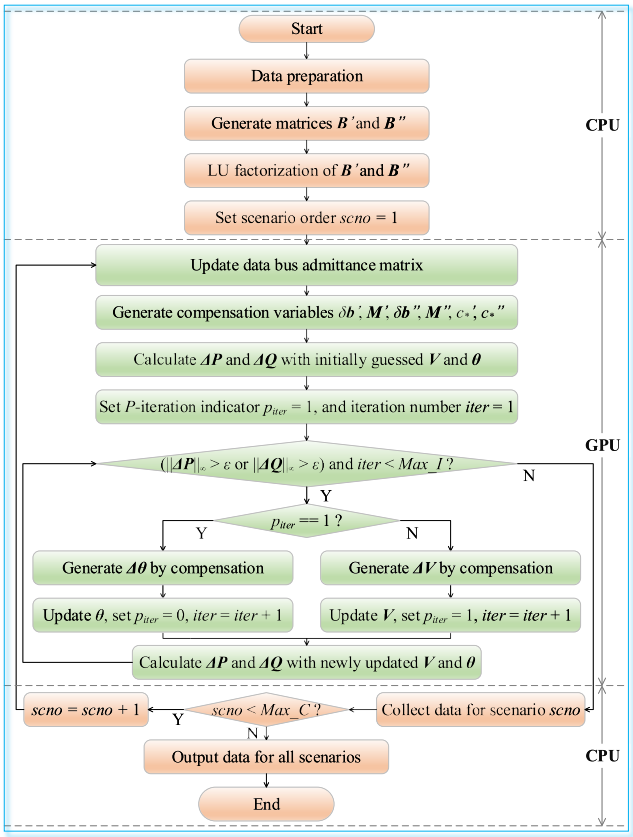

FIGURE 2. Flowchart of the compensation method.

Accordingly, the generation process for $\delta \boldsymbol{b}''_*$ and $\boldsymbol{M}''_*$ is given by **Algorithm 2**, where $\boldsymbol{e}''_i$ and $\boldsymbol{e}''_j$ are basis vectors with size $n-r-1$. A comprehensive implementation flowchart of CM corresponding to the above steps is illustrated in Fig. 2.

In order to validate the solution efficiency, the CM is implemented on a 2746-bus test system. It takes 40.31ms to analyze each scenario, which means 1,488 contingencies can be evaluated every minute. Nevertheless, in industry application, this performance is far from satisfactory. For example, Midcontinent Independent System Operator (MISO) [25], Electric Reliability Council of Texas (ERCOT) [7], and Pennsylvania-New Jersey-Maryland Interconnection (PJM) [26] simulate 2,875, 3938, and 6,000 contingency scenarios of large-scale power system in one minute, respectively.

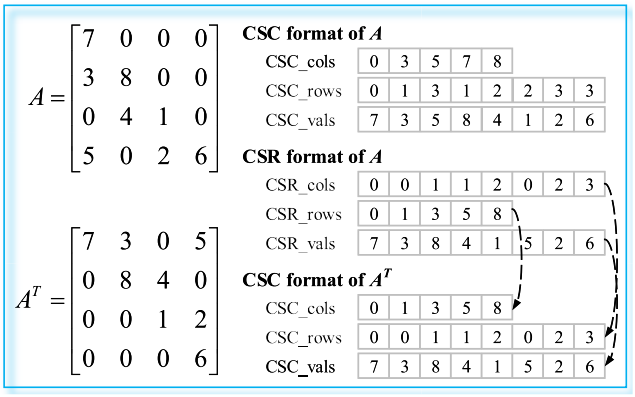

FIGURE 3. Transformation from CSC to CSR by matrix transposition.

III. PARALLEL IMPLEMENTATION ON GPUs

Since the sequential CM is not sufficient for industry application, advanced parallel hardware GPU is resorted for acceleration in this section, where CUDA [27] version 8.0 is employed for programming.

A. DATA STRUCTURE AND PRECISION

It is widely accepted that sparse matrix techniques should be adopted for the solution of large-scale power systems due to their high sparsity ratio of bus admittance matrix $\boldsymbol{Y}$. In order to reduce the amount of data transfer and the number of atomic operations, the sparse storage formats are commonly employed. In accordance with [21] and [23], the Compressed Sparse Column (CSC) format is utilized for the storage of $\boldsymbol{Y}$, $\boldsymbol{B}'$, $\boldsymbol{L}'$, $\boldsymbol{U}'$, $\boldsymbol{B}''$, $\boldsymbol{L}''$ and $\boldsymbol{U}''$. Unfortunately, part of `cuSparse` library [28] operations do not support the CSC format, such as matrix-vector multiplication (related with $\boldsymbol{Y}$) and F/B substitutions (implemented on $\boldsymbol{L}'$, $\boldsymbol{U}'$, $\boldsymbol{L}''$ and $\boldsymbol{U}''$). Therefore, $\boldsymbol{L}'$, $\boldsymbol{U}'$, $\boldsymbol{L}''$ and $\boldsymbol{U}''$ are translated into the Compressed Sparse Row (CSR) format from CSC. The transformation can be performed by a matrix transposition, which is demonstrated in Fig. 3. On the other hand, the matrix $\boldsymbol{Y}$ is free from transformation due to its symmetry, i.e., its CSC is the same with CSR. Except for the matrices, all the vectors are stored in dense format.

In this work, the NVIDIA GeForce GTX 1080 GPU is utilized, whose compute capability is 6.1 with 8.876TFLOP/s and 277.36GFLOP/s on single and double precision. Although the single precision is much faster, the accuracy is limited, which is illustrated by Fig. 4, therefore the double precision is finally applied.

B. SPARSE LINEAR SOLVER

The linear solver is a basic component of key importance for the whole performance of NR, FD, and CM, especially for NR, where it may take 80% of the total execution time [16].

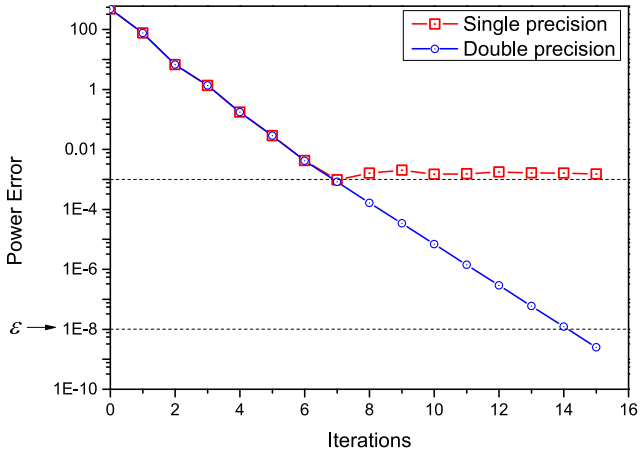

FIGURE 4. Convergence properties of CM for a 2746-bus test system with single and double precision.

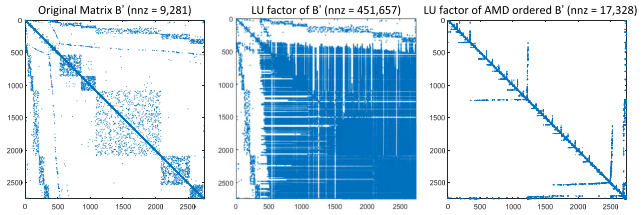

FIGURE 5. Performance illustration of AMD on a 2746-bus test system with sparsity pattern (nnz is the number of nonzero elements).

Two types of sparse linear solver are intensively discussed: iterative methods and direct methods. Although the former demands less memory and the solution process is controllable based on absolute or relative error [11], [20], the latter is more popular in the community [13]–[19].

Decomposition or factorization of the coefficient matrix $\boldsymbol{A}$ is ubiquitous in direct methods. Although the sparsity of $\boldsymbol{A}$ is inherited by $\boldsymbol{L}$ and $\boldsymbol{U}$, there still exists a lot of fill-ins (entries in $\boldsymbol{L}$ and $\boldsymbol{U}$ that do not appear in $\boldsymbol{A}$). Generally, fewer fill-ins means less requirement on the system memory and atomic operations; therefore, the approximate minimum degree (AMD) algorithm [21] is utilized, which is a heuristic to find the permutation P such that PAP^T has fewer fill-ins than $\boldsymbol{A}$ after the factorization. Fig. 5 illustrates the performance of AMD on the Jacobian matrix $\boldsymbol{B}'$ of a 2746-bus system. After AMD ordering, the $\boldsymbol{B}'$ is more concise, and the number of fill-ins after LU factorization is reduced from 451,657 to 17,328, which means the reduction rate reaches 96.16%.

C. SINGLE GPU ARCHITECTURE

1) KERNEL DESIGN STRATEGIES

One kernel function can be executed N times in parallel if N different CUDA threads are launched. Since all scenarios are independent, a naive parallel implementation strategy is that integrating all the CM steps into one whole kernel function and running in a fixed thread, whose execution pattern is

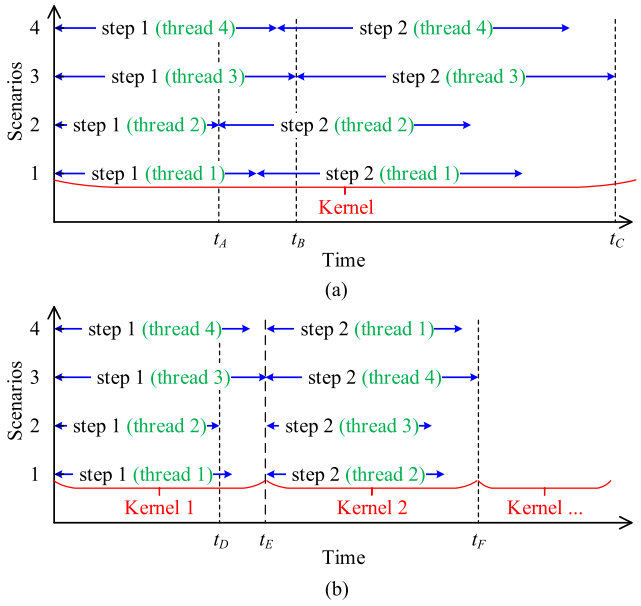

FIGURE 6. Illustration of kernel design strategies. (a) Integrated kernel: all steps are included in one whole kernel. (b) Decoupled kernel: different steps are performed by various kernels.

demonstrated by Fig. 6 (a). Although the integrated kernel strategy is straightforward and the waiting time can be minimized by static/dynamic load balancing [8], it is not suitable for CUDA due to its random data access feature. In CUDA, the parallel threads are managed, scheduled, and executed in groups of 32 called warps. A warp executes one common instruction at a time, and the full efficiency is achieved when all 32 threads within the warp agree on their execution path (coalesced access) [27]. Faced with path diversity, the warp serially executes each branch path by disabling threads that are not on that path. When all paths complete, the warp converges back to the same execution path. In Fig. 6 (a), threads 1 – 4 in the same warp should access successive addresses and do the same operations to achieve growth in performance. However, at t_A, thread 2 starts to execute step 2 whereas the other threads are still doing step 1, thus the coalesced access is declined. In addition, the path diversity increases as time goes on.

In order to achieve coalesced access, the whole kernel function is decoupled in Fig. 6 (b), i.e., each step is realized with one kernel function. It is observable that all threads do the same operations (go to the same path) within various intervals, e.g., execute step 1 from 0 to t_E and step 2 from t_E to t_F. Therefore, there is no branch diversity within warps. On the other hand, successive addresses are easily accessed by threads since scenario data is commonly stored in regulation. Detailed access pattern will be exemplified in the following subsection. Compared with Fig. 6 (a) and (b), the execution efficiencies of thread 2 for step 1 should be similar, i.e., $t_A \approx t_D$. The reason is that coalesced execution pattern is also achieved by Fig. 6 (a) from 0 to t_A. However, due to path diversity, deterioration will subsequently emerge in Fig. 6 (a), thus $t_B > t_E$ and $t_C > t_F$. It should be noted that

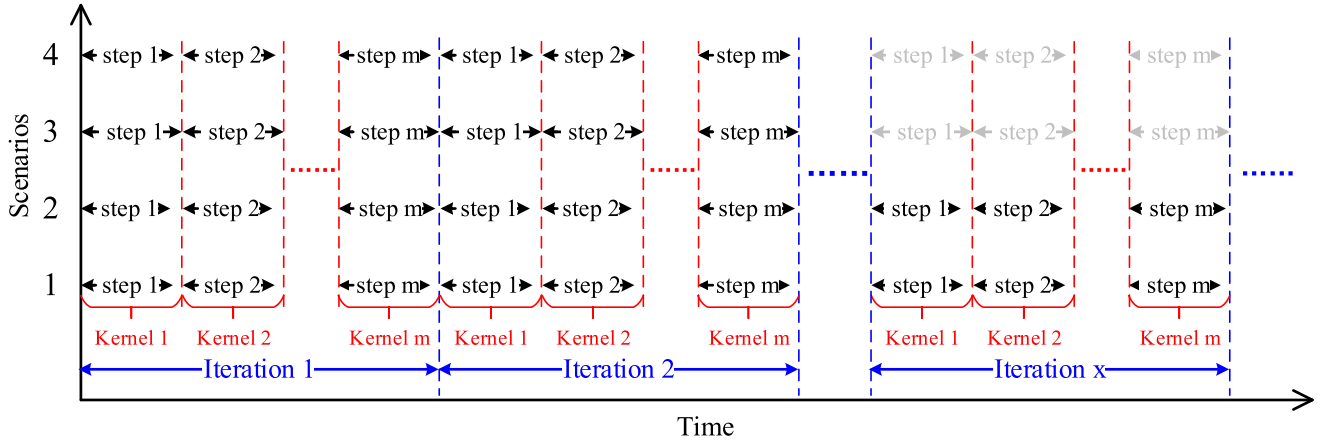

FIGURE 7. Solution process of multiple ACPFs/scenarios.

there is an implicit barrier between the successive kernels, which guarantees the logical sequence between iterations. Theoretically, wait time always exists before the barrier, but it is relatively small since the work load for each scenario on each kernel is even. Due to the multiple kernels execution, different steps of one specified scenario may not be executed on the same thread, which is illustrated in Fig. 6 (b).

The solution process of multiple ACPFs/scenarios is given in Fig. 7, wherein each scenario consists of several iterations, and each iteration comprises of various steps. Since a series of steps are executed in a fixed order across different iterations, and coalesced access within each step is achieved by the aforementioned decoupled kernel design strategy, it can be concluded that coalesced access within each iteration is also guaranteed. In Fig. 7, iterations 1 and 2 will be executed with coalesced access, but iteration x may not be. In Fig. 7, scenarios 3 and 4 achieve convergence at iteration $x-1$, resulting in 2 threads (may not be thread 3 and 4) being idle at iteration x. If new tasks (e.g., iteration 1 of scenarios 33 and 34) are assigned to these idle threads, coalesced access might be destroyed since the scenario number is not successive. In order to keep consistency and coalesced access, a group of scenarios will terminate only if all of them finished the calculation, i.e., the iteration x of scenarios 3 and 4 will be executed although they have achieved the convergence. It should be noted that there is a waste of computational resources on scenarios 3 and 4, but it is needed to preserve the efficiency. If computational resources are fully utilized for net computation, i.e., idle threads are assigned with new scenarios, and all threads can achieve coalesced access, the efficiency can be improved to a rate of up to 40% (supposing the longest scenario takes 80% more iterations than the shortest one). Nevertheless, if the coalesced access is not guaranteed, the efficiency can be tens of times slower since a maximum of 32 branches will be executed in each warp. Therefore, in this work, a part of the computational resource is sacrificed for coalesced access. In future work, full utilization of computational resource with coalesced access will be investigated.

2) KERNEL FUNCTIONS

In this work, n_c scenarios are solved simultaneously in a step-by-step pattern, with each step corresponding to one kernel. Fig. 8 demonstrates the implementation scheme of key steps of CM, where both self-built and `cusparse` library provided kernels are utilized. It should be noted that, every operation shown in Fig. 8 will be executed for n_c times with one for each scenario. As indicated in Section II, all scenarios derive the solutions from the base case, i.e., triangle matrices $\boldsymbol{L}'$ and $\boldsymbol{U}'$ are the same for all scenarios. Therefore, equations $\boldsymbol{T}_a = \boldsymbol{L}'^{-1}\boldsymbol{M}'_*$ for all scenarios can be jointly considered as a sparse triangular linear system with multiple right-hand sides. Library `cusparse` provides efficient solution routine for this kind of systems. Kernel `cusparseDcsrsm_analysis()` performs the matrix analysis of $\boldsymbol{L}'$ and $\boldsymbol{U}'$. This function needs to be executed only once since its result is reusable. Kernel `cusparseDcsrsm_solve()` is utilized to generate the result based on matrix analysis information.

For all `cusparse` kernels, the thread utilization mechanism is concealed. On the other hand, the thread organization of self-built kernel is tuned based on the target data storage pattern to achieve coalesced access and high efficiency. Take the `kernel_Update()` in Fig. 9 as an example, where $\boldsymbol{\theta}$ is stored scenario after scenario. If each scenario is intuitively performed with one thread, diversity will occur since threads 1 and 2 will access discrete addresses $\boldsymbol{\theta}_0$ and $\boldsymbol{\theta}_n$ at the same time. Therefore, each scenario is attributed to one warp in Fig. 9, where lanes $0-31$ will access a series of successive addresses $\boldsymbol{\theta}_0 - \boldsymbol{\theta}_{31}$. Different warps are independent of each other, and the branch divergence over warps does not affect the performance.

3) MEMORY MANAGEMENT

CUDA threads may access data from on-chip (register, shared memory, and L2 cache) and off-chip memories (global memory). The former is very fast but the size is limited, while the latter is large but more latency is required for accessing.

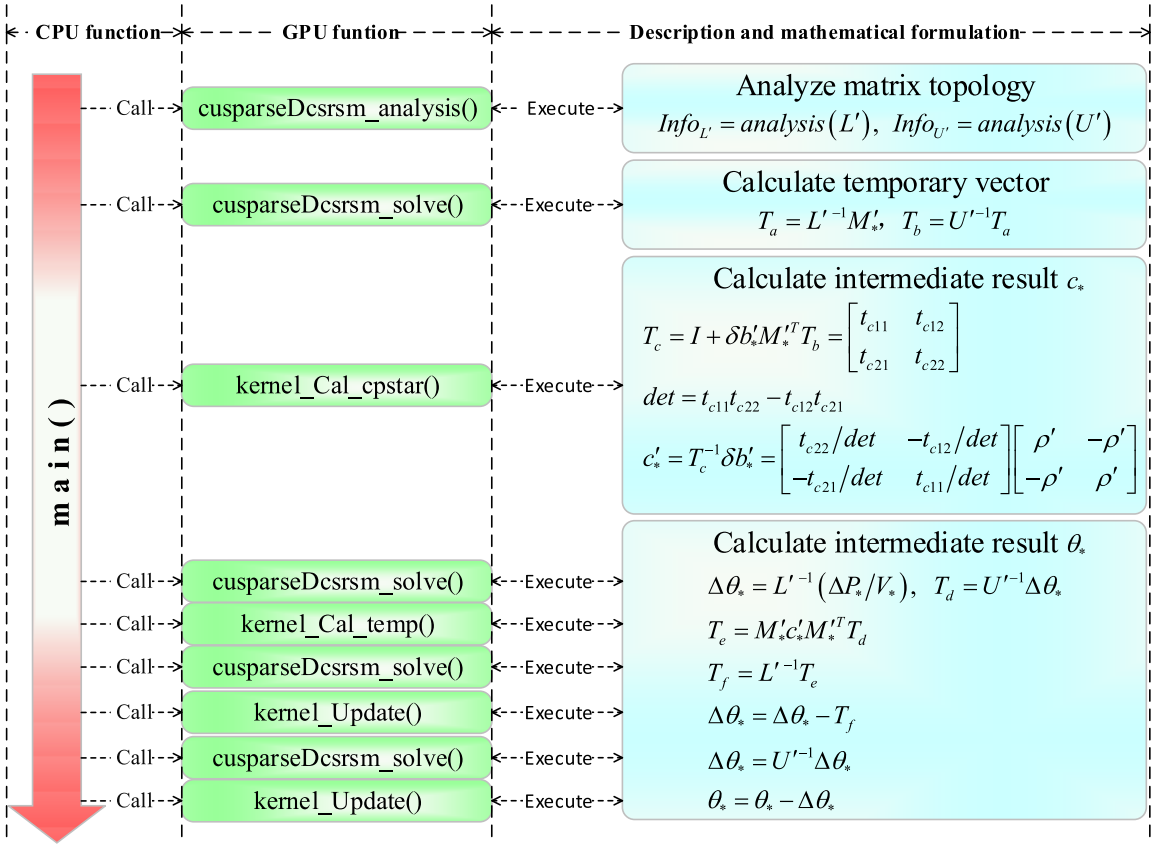

FIGURE 8. Decoupled kernels for the solution of equations (6)–(9).

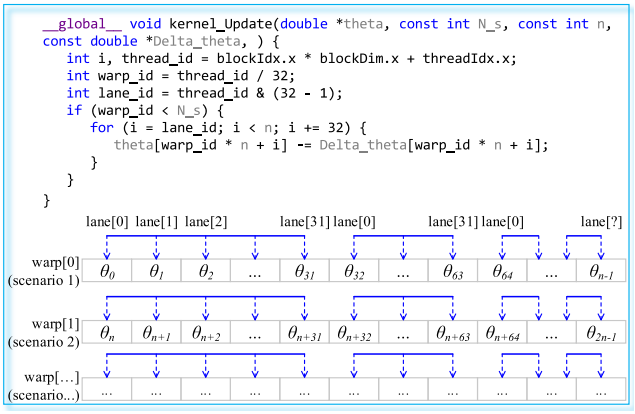

FIGURE 9. Coalesced access of θ in `kernel_Update()`.

To pursue a balance between the performance and feasibility, both are utilized with principles:

- To achieve the best performance of on-chip memory, all the small size intermediate vectors are stored as scalar variables. Take T_c in Fig. 8 as an example, instead of storing a matrix or vector, all the elements are declared as independent variables.
- All the long vectors and large matrices are stored in global memory, which can be accessed by every kernel for reading and writing. The maintenance of data in device memory is also beneficial to minimize the data exchange between host and device memories.
- Shared memory is widely utilized due to its low access latency, especially for the reduction operations, including the calculation of summation, minimum, and maximum values of a vector. For example, vector $\Delta \boldsymbol{P}$ is copied into the shared memory before calculating $||\Delta \boldsymbol{P}||_\infty$.

As indicated in Fig. 8, the CPU main function calls the GPU kernels but cannot get any feedback from them. The common strategy of controlling is data exchange between CPU and GPU via `cudaMemcpy()`. However, due to the limited bandwidth of PCIe, fewer data transformations are better. In this work, except for the data to start CM, which is copied from CPU to GPU at the beginning, only one boolean variable (showing that the termination condition has been met or not) is transferred from GPU to CPU at the end of each iteration.

D. MULTIPLE-GPU ARCHITECTURE

In order to further accelerate the computation, multiple-GPU architecture is also explored. As the scenarios are solidly independent, there is no communication between different devices; therefore, the implementation is relatively straightforward, i.e., evenly distribute the workload to all the

TABLE 1. Solution differences between GPU-based parallel CM, Matpower NR, and Matpower FD.

Cases	Names in [23]	$\Delta = \max\{\|\Delta P\|, \|\Delta Q\|\}$ CM vs. NR	CM vs. FD
CaseA	case**300**	6.16×10^{-9}	2.16×10^{-14}
CaseB	case**1354**pegase	6.79×10^{-9}	8.33×10^{-13}
CaseC	case**2746**wp	2.85×10^{-9}	9.99×10^{-13}
CaseD	case**9241**pegase	4.93×10^{-9}	7.61×10^{-13}
CaseE	case**13659**pegase	2.38×10^{-9}	1.09×10^{-13}

devices. Each device is administrated by one CPU thread, which is launched by OpenMP in this work.

IV. EXPERIMENTAL RESULTS

Five benchmark cases reported in [23] with scales ranging from 300 to 13,659 buses are employed in this section to validate the performance of the parallel CM with respect to accuracy and execution time. Based on the open source package Matpower 6.0b2 [23], the first test is provided to validate the accuracy and convergence properties of CM. The second test measures the execution time and speedup of CM in different platforms (CPU and GPU) with various implementation schemes (sequential and parallel). Finally, three types of state-of-the-art GPU-based parallel computing methods reported in the literature are included for discussion. Both CPU- and GPU-based CM are implemented with Visual Studio 2015 on a PC equipped with 12 physical Intel Xeon E5-2620 2.10GHz CPU cores and 2 NVIDIA GTX 1080 GPUs, running on Windows 8.1 operating system. Matlab version 2015b is utilized to execute Matpower. For all experiments, the convergence criteria ϵ is set as 10^{-8} p.u.

A. ACCURACY AND CONVERGENCE OF GPU-BASED PARALLEL CM

For any RTCA solution method, the credibility is of higher priority over efficiency. Thus the accuracy and convergence properties of GPU-based parallel CM are evaluated in this subsection. Matpower is introduced as a reference, where both NR and FD algorithms are utilized. For each test system, the same scenario is solved with three different methods. Table 1 summarizes the maximum differences on the node active/reactive powers $\max\{|\Delta P|, |\Delta Q|\}$. The data of column 3 is determined by the tolerance $\epsilon = 10^{-8}$, while column 4 shows that the parallel CM is highly identical with FD. Therefore, it can be concluded that the proposed parallel implementation of CM based on GPU is credible.

Fig. 10 illustrates the convergence properties of parallel CM and NR on different cases, where parabola and straight lines can be approximated for the NR and CM on the logarithmic coordinate due to their quadratic and geometric convergent properties respectively. It should be noted that only the maximum error after Q iteration is collected for the lines of CM. If maximum errors after both P and Q iterations are recorded, a wavy line with the same trend will appear. Although the CM takes more iterations to meet the convergence criteria, its computing requirement and execution time is far less than that of the NR. The comparison on the convergence process between CM and FD is omitted since they always take the same number of iterations before termination, which can also be accessed from the theoretical analysis in Section II.

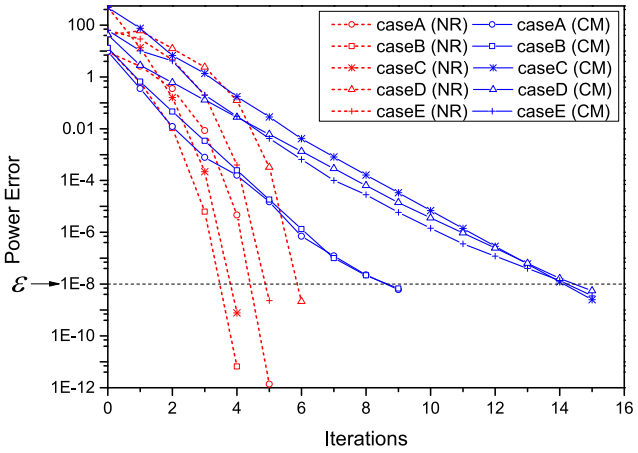

FIGURE 10. Convergence properties of FD and parallel CM on different cases.

B. PERFORMANCE OF CM ON VARIOUS PARALLEL ARCHITECTURES

In order to fully explore the potential of CM and parallel architectures, many implementation schemes are tested and compared. For each case, n_c different scenarios generated by the withdrawing of a single transmission line are considered and solved, i.e., the $N-1$ contingency criterion is addressed without any scenario reduction strategy. Table 2 illustrates the values of n_c for various cases. Due to the large variance of n_c, the total execution time for RTCA with the solution of all scenarios presents great differences across cases. Therefore, it is regulated by the division of n_c in the following, i.e., the henceforth reported time is the average execution time for the full iterative solution of single scenario without specification.

All CM implementation schemes to be presented below follow the flowchart shown in Fig. 2, where the LU decomposition for $\boldsymbol{B}'$ and $\boldsymbol{B}''$ at the very beginning is performed based on algorithms developed in [21]. The main differences between various implementation schemes are: 1) all considered scenarios will be evaluated in either sequential or parallel, which will be specified subsequently; 2) on GPU platform, the F/B substitution is performed with `cusparse` kernel functions presented in Section III.C; 3) on CPU platform, the F/B substitution is executed based on the routines provided by [21]. It should be noted that the basic mechanism and program logic of other steps are the same for both CPU and GPU versions, but the realized codes are different due to special GPU implementation strategies. Take the updating of $\boldsymbol{\theta}$ as an example, the code for GPU execution is shown in Fig. 9, where a lot of indices are included for threads, lanes, and warps.

TABLE 2. Number of scenarios considered for different cases.

Cases	CaseA	CaseB	CaseC	CaseD	CaseE
n_c	411	1,991	3,514	16,049	20,467

TABLE 3. Execution time (ms) of sequential CM with single-thread CPU.

Cases	CaseA	CaseB	CaseC	CaseD	CaseE
T_0	2.514	12.127	40.382	157.671	220.976

1) SEQUENTIAL CM WITH SINGLE-THREAD CPU

In this test, the single-thread CPU computing architecture is utilized, where all scenarios are evaluated in series with CPU. The execution time T_0 is reported in Table 3 with the coverage of all steps after the initial data reading, including admittance matrix generation, LU decomposition, compensation matrices construction, F/B substitution, and power mismatch calculation, etc. This test is determined as the reference for speedup analysis of other tests since it is the basic version of CPU implementation. The speedup S is defined as follows:

$$S_x = \frac{T_0}{T_x}, \tag{11}$$

where subscript $_x$ represents the following tests.

2) PARALLEL CM WITH MULTI-THREAD CPU

In order to accelerate the solution process of RTCA, multi-thread CPU is employed in this test, where all scenarios are assessed in parallel with CPU based on OpenMP. The CPU part shown in Fig. 2 is performed with the main thread, and the GPU part is fulfilled by y threads for concurrent execution. All scenarios are evenly distributed to y threads. In this test, y takes the values of 2, 4, 8, and 12 with the PC specified above. The execution time $T_{1,y}$ and speedup $S_{1,y}$ are summarized and illustrated in Table 4 and Fig. 11. Compared with T_0, a speedup is achieved with parallel computing. Two observations can be obtained: 1) For every y, $S_{1,y}$ is higher with larger cases. The parallel efficiency depends on both task workload and thread launch latency. If the latency is fixed, heavier work for each launch brings higher parallel efficiency. Thus, larger cases have higher speedups. 2) For each case, $S_{1,y}$ rises with y, but the increase rate is decreasing. The biggest performance gain is obtained by the thread number increasing from 1 to 2. More threads bring complex coordination challenge, thus the parallel efficiency is lower although the absolute speedup value is higher.

3) PARALLEL CM WITH SINGLE GPU

Due to the limited memory bandwidth, the saturation point of multi-thread CPU implementation is approaching, which means adding more CPU cores does not result in remarkable enhancement on the computational performance. Therefore, the GPU is introduced as an alternative. Based on the flowchart shown in Fig. 2 and details revealed in Section III.C, two tests are implemented, i.e., single GPU with integrated and decoupled kernels, whose execution time are marked as $T_{2,I}$ and $T_{2,D}$ respectively. In addition to all steps covered by T_0 and $T_{1,y}$, the device memory allocation and data transmission between CPU and GPU are included in $T_{2,I}$ and $T_{2,D}$. As shown in Fig. 2, all the iterative processes within each scenario are executed on GPU without the data exchange with CPU. The data copy from CPU to GPU is mainly done before the launch of GPU, where grid configuration and decomposed matrices ($\boldsymbol{L}'$, $\boldsymbol{U}'$, $\boldsymbol{L}''$, and $\boldsymbol{U}''$) are transferred to device memory. The feedback from GPU to CPU is only one binary vector with the length of n_c, where '1' represents the corresponding scenario is sufficient, and vice versa. Therefore, the communication time T_c mainly depends on the system scale (which dominates the size of matrices), whereas n_c has a limited influence on T_c. In our tests, T_c is always less than 5% of the total execution time. Table 5 summarizes the main results. It is observable from $S_{2,I}$ and $S_{2,D}$ that the proposed decoupled kernel performs better than integrated kernel for all cases. In addition, $S_{2,D}$ has a better scalability from CaseA to CaseE since the increase rates are higher, such as $\frac{34.67}{2.80} > \frac{21.15}{2.51}$ and $\frac{38.43}{2.80} > \frac{22.54}{2.51}$. The fastest multi-thread CPU execution is compared with GPU implementations in Fig. 12, which validates the superiority of GPU parallel architecture for RTCA.

TABLE 4. Execution time (ms) and speedup of parallel CM with multi-thread CPU.

Cases	CaseA	CaseB	CaseC	CaseD	CaseE
$T_{1,2}$	1.386	6.627	21.971	82.897	115.573
$T_{1,4}$	0.768	3.631	11.976	43.604	60.508
$T_{1,8}$	0.453	2.102	6.831	23.689	32.382
$T_{1,12}$	0.391	1.739	5.571	17.877	23.977
$S_{1,2}$	1.81	1.83	1.84	1.90	1.91
$S_{1,4}$	3.27	3.34	3.37	3.62	3.65
$S_{1,8}$	5.54	5.77	5.91	6.66	6.82
$S_{1,12}$	6.43	6.97	7.25	8.82	9.22

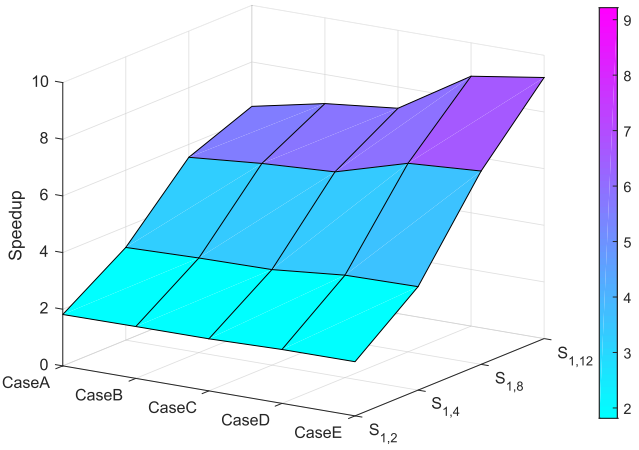

FIGURE 11. Speedup of parallel CM with multi-thread CPU.

TABLE 5. Execution time (ms) and speedup of CM with single GPU.

Cases	CaseA	CaseB	CaseC	CaseD	CaseE
$T_{2,I}$	1.002	1.358	2.749	7.455	9.804
$T_{2,D}$	0.899	1.118	2.069	4.548	5.750
$S_{2,I}$	2.51	8.93	14.69	21.15	22.54
$S_{2,D}$	2.80	10.85	19.52	34.67	38.43

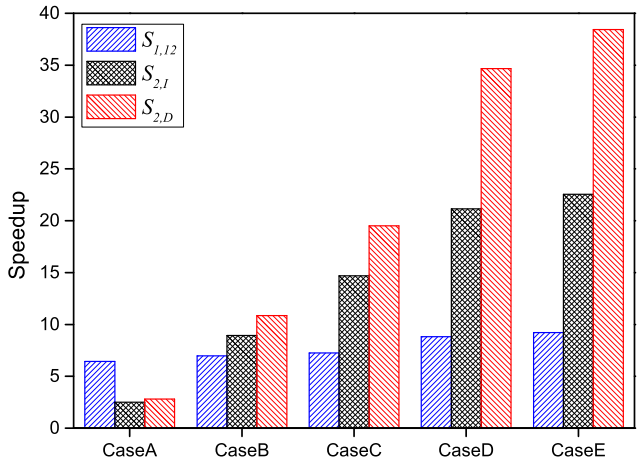

FIGURE 12. Speedup comparison between multi-thread CPU and GPU.

4) PARALLEL CM WITH MULTIPLE GPUs

Based on $T_{2,D}$, the total execution time of $N-1$ RTCA for CaseD and CaseE is 72.99s and 117.69s, respectively. In order to finish the solution within one minute, two-GPUs architecture is employed for acceleration. The implementation scheme is similar to single GPU, except that all scenarios are evenly distributed to two GPUs. Each GPU is managed with one CPU thread, and the CPU threads are separated with OpenMP. Since the CPU threads are launched for only once, the latency is insignificant when compared with the total execution time. Table 6 reports the results, where a maximum of 75.70× speedup is gained by the decoupled kernel strategy, which facilitates the RTCA to complete within one minute. In order to investigate the parallel efficiency of multiple GPUs, comparison with single GPU is demonstrated in Fig. 13. It is noticeable that both $S_{3,I}/S_{2,I}$ and $S_{3,D}/S_{2,D}$ are close to 2.0, which means the scalability of multiple GPUs is satisfactory. The main reasons are: 1) all scenarios are independent, such that there is no communication and synchronization between two GPUs; 2) the iterative process of ACPF is fully executed on GPU without the data exchange between CPU and GPU.

C. COMPARISON WITH OTHER PARALLEL COMPUTING METHODS

The superiority of parallel CM running on GPUs is established in the above subsection with the comparison of CM running on CPU. This subsection is devoted to the comparison against three types of state-of-the-art parallel computing methods running on GPU. It should be noted that all

TABLE 6. Execution time (ms) and speedup of CM with multiple GPUs.

Cases	CaseA	CaseB	CaseC	CaseD	CaseE
$T_{3,I}$	0.553	0.733	1.468	3.895	5.077
$T_{3,D}$	0.473	0.582	1.072	2.320	2.919
$S_{3,I}$	4.55	16.54	27.51	40.48	43.52
$S_{3,D}$	5.32	20.84	37.67	67.96	75.70

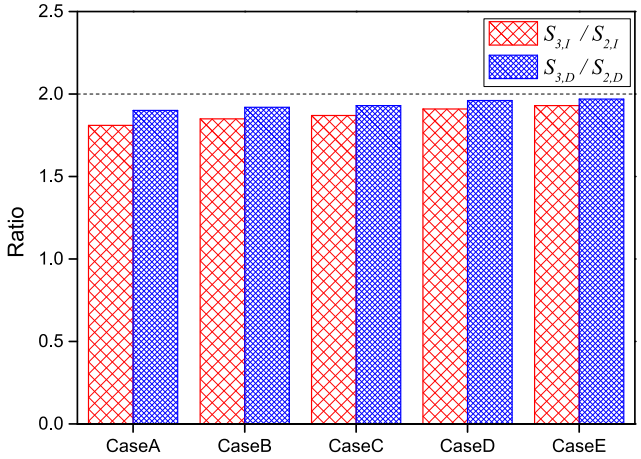

FIGURE 13. Speedup ratio of multiple GPUs over single GPU.

TABLE 7. Runtime reported in the literature with GS running on GPU (ms).

System Scales	Ref. [18]	Ref. [14]	Ref. [15]
4-bus			0.012
9-bus	22.397	327.600	
30-bus	22.551	705.100	0.511
118-bus	50.719	3,296.300	2.346
300-bus	118.519	7,299.200	56.293
678-bus	154.513		
974-bus		19,603.000	
2,383-bus	826.617		
3,061-bus	3,061.353		

results reported in this subsection are strictly retrieved from references. Due to differences on computation platforms and programming skills, the comparison can only be regarded as qualitative rather than quantitative.

1) PARALLEL GAUSS-SEIDEL (GS) METHOD

In [14], [15], and [18], the GS was implemented on GPU for power flow analysis, whose solution time is summarized in Table 7. Although larger cases have been reported in [14] and [18], the performance is weaker than [15] due to the utilization of dense matrix. The maximum case given in [15] is the 300-bus system, which consumes 56.293ms for each scenario, while that number is 0.473ms for parallel CM in this paper. Therefore, the advantage of CM over GS is identified.

2) PARALLEL NR METHOD

Although the combination of NR with GPU has been introduced in [14], [18], and [19], only three better results reported by [15]–[17] are included in Table 8 for comparison. The data

TABLE 8. Runtime reported in the literature with NR running on GPU (ms).

System Scales	Ref. [15]	Ref. [17]	Ref. [16]	This paper
4-bus	0.012			
30-bus	0.052			
118-bus	0.142			
300-bus	0.632			0.473
1,354-bus				0.582
1,586-bus		0.614*		
2,228-bus		1.104*		
2,383-bus	9.010			
2,746-bus				1.072
8,503-bus			5.471*	
9,241-bus				2.320
12,404-bus		4.026*		
13,659-bus				2.919
21,801-bus		4.767*		

*: Execution time for single scenario was calculated by dividing the total time with the number of scenarios considered.

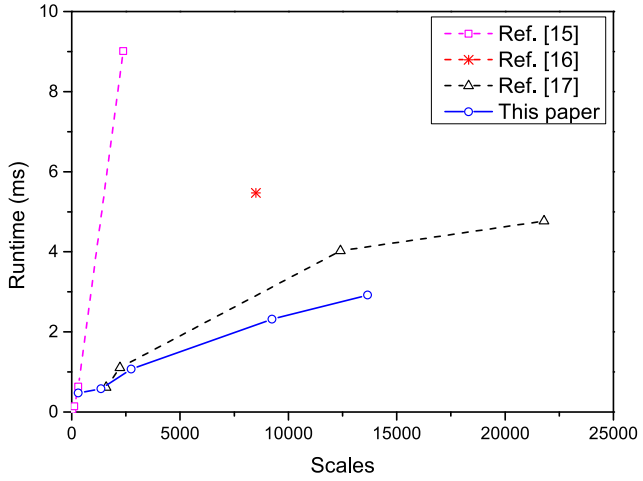

FIGURE 14. Runtime reported in the literature with NR running on GPU.

in Table 8 has been illustrated in Fig. 14. It is observable that the parallel CM consumes less time for all scales of cases, including small, medium, and large.

3) PARALLEL FD METHOD

In [14], the performance of FD on GPU is better than GS and NR; however, it is limited due to the utilization of dense matrix. [20] implements the FD on GPU with preconditioned conjugate gradient iterative solver and inexact Newton method. Although the reported execution time is 260∼1,210ms for cases with scales 1,354∼13,173, it is a worthwhile endeavor (especially for large-scale systems) since the majority of works rely on the direct solver.

V. CONCLUSION

While the real-time contingency analysis (RTCA) is an important energy management system functionality in many electric utilities, it is faced with the challenge of high computational burden. To address the requirement of fast solution for RTCA, the parallel implementation of compensation method (CM) on multi-GPU architecture is presented in this paper. Accuracy and efficiency have been validated with case studies on five benchmark systems ranging from 300 to 13,659 buses. The generated speedups are significant when compared with other platforms, including single-thread CPU, multi-thread CPU, and single-GPU. In addition, superiority over other state-of-the-art parallel computing methods is established. In conclusion, the parallel CM is promising for industrial application since it is capable to finish the whole $N-1$ RTCA analysis for the 13,659-bus system within one minute. A potential application of this method could be for post-contingency corrective transmission switching, toward which future efforts will be directed. On the other hand, the extension of parallel CM to GPU cluster will be investigated in the future.

APPENDIX A
DERIVATION OF THE CM IMPLEMENTATION STEPS

The process can be described as generating $\Delta\boldsymbol{\theta}_*$ from,

$$\Delta\boldsymbol{P}_*/\boldsymbol{V}_* = \boldsymbol{B}'_*\Delta\boldsymbol{\theta}_*, \tag{A.1}$$

where $\boldsymbol{B}'_* = \boldsymbol{B}' + \boldsymbol{M}'_*\delta\boldsymbol{b}'_*\boldsymbol{M}'^T_*$ and $\boldsymbol{B}' = \boldsymbol{L}'\boldsymbol{U}'$.

Based on $\boldsymbol{B}'_* = \boldsymbol{B}' + \boldsymbol{M}'_*\delta\boldsymbol{b}'_*\boldsymbol{M}'^T_*$ and the inverse matrix modification lemma (given in Appendix B), the inverse of $\boldsymbol{B}'_*$ can be obtained as,

$$\boldsymbol{B}'^{-1}_* = \boldsymbol{B}'^{-1} - \boldsymbol{B}'^{-1}\boldsymbol{M}'_*\boldsymbol{c}_*\boldsymbol{M}'^T_*\boldsymbol{B}'^{-1}, \tag{A.2}$$

$$\boldsymbol{c}'_* = \left[\boldsymbol{I} + \delta\boldsymbol{b}'_*\boldsymbol{M}'^T_*(\boldsymbol{U}'^{-1}(\boldsymbol{L}'^{-1}\boldsymbol{M}'_*))\right]^{-1}\delta\boldsymbol{b}'_*. \tag{A.3}$$

Therefore, based on (A.1) and (A.2), we have,

$$\begin{aligned}
\Delta\boldsymbol{\theta}_* &= \boldsymbol{B}'^{-1}_*\left(\Delta\boldsymbol{P}_*/\boldsymbol{V}_*\right) && \text{(A.4)}\\
&= \left(\boldsymbol{B}'^{-1} - \boldsymbol{B}'^{-1}\boldsymbol{M}'_*\boldsymbol{c}_*\boldsymbol{M}'^T_*\boldsymbol{B}'^{-1}\right)\left(\Delta\boldsymbol{P}_*/\boldsymbol{V}_*\right) && \text{(A.5)}\\
&= \Big(\boldsymbol{U}'^{-1}\boldsymbol{L}'^{-1}\left(\Delta\boldsymbol{P}_*/\boldsymbol{V}_*\right)\\
&\quad - \boldsymbol{U}'^{-1}\boldsymbol{L}'^{-1}\boldsymbol{M}'_*\boldsymbol{c}_*\boldsymbol{M}'^T_*\boldsymbol{U}'^{-1}\boldsymbol{L}'^{-1}\left(\Delta\boldsymbol{P}_*/\boldsymbol{V}_*\right)\Big) && \text{(A.6)}\\
&= \boldsymbol{U}'^{-1}\left(\Delta\boldsymbol{\theta}_1 - \boldsymbol{L}'^{-1}\boldsymbol{M}'_*\boldsymbol{c}_*\boldsymbol{M}'^T_*\boldsymbol{U}'^{-1}\Delta\boldsymbol{\theta}_1\right) && \text{(A.7)}\\
&= \boldsymbol{U}'^{-1}\Delta\boldsymbol{\theta}_2. && \text{(A.8)}
\end{aligned}$$

The transformation from (A.6) to (A.7) – (A.8) is based on the definition of (7) – (8). Finally, the identity between (A.7) – (A.8) and (8) – (9) concludes the derivation process. ■

APPENDIX B
PROOF OF INVERSE MATRIX MODIFICATION LEMMA

This lemma is utilized to derive (A.2). We start from $\boldsymbol{B}'^{-1}$,

$$\begin{aligned}
\boldsymbol{B}'^{-1} &= \left(\boldsymbol{B}' + \boldsymbol{M}'_*\delta\boldsymbol{b}'_*\boldsymbol{M}'^T_*\right)^{-1}\left(\boldsymbol{B}' + \boldsymbol{M}'_*\delta\boldsymbol{b}'_*\boldsymbol{M}'^T_*\right)\boldsymbol{B}'^{-1}\\
&= \left(\boldsymbol{B}' + \boldsymbol{M}'_*\delta\boldsymbol{b}'_*\boldsymbol{M}'^T_*\right)^{-1}\left(\boldsymbol{I} + \boldsymbol{M}'_*\delta\boldsymbol{b}'_*\boldsymbol{M}'^T_*\boldsymbol{B}'^{-1}\right)\\
&= \left(\boldsymbol{B}' + \boldsymbol{M}'_*\delta\boldsymbol{b}'_*\boldsymbol{M}'^T_*\right)^{-1} + \left(\boldsymbol{B}' + \boldsymbol{M}'_*\delta\boldsymbol{b}'_*\boldsymbol{M}'^T_*\right)^{-1}\\
&\quad \boldsymbol{M}'_*\delta\boldsymbol{b}'_*\boldsymbol{M}'^T_*\boldsymbol{B}'^{-1} && \text{(B.1)}\\
&= \left(\boldsymbol{B}' + \boldsymbol{M}'_*\delta\boldsymbol{b}'_*\boldsymbol{M}'^T_*\right)^{-1} + \boldsymbol{B}'^{-1}\boldsymbol{M}'_*\left(\delta\boldsymbol{b}'^{-1}_* + \boldsymbol{M}'^T_*\right.
\end{aligned}$$

$$\boldsymbol{B}'^{-1}\boldsymbol{M}'_*\Big)^{-1}\boldsymbol{M}'^{T}_*\boldsymbol{B}'^{-1}. \tag{B.2}$$

Since $\boldsymbol{B}'_* = \boldsymbol{B}' + \boldsymbol{M}'_*\delta\boldsymbol{b}'_*\boldsymbol{M}'^{T}_*$, we have,

$$\boldsymbol{B}'^{-1}_* = \boldsymbol{B}'^{-1} - \boldsymbol{B}'^{-1}\boldsymbol{M}'_*\boldsymbol{c}'_*\boldsymbol{M}'^{T}_*\boldsymbol{B}'^{-1}, \tag{B.3}$$

where

$$\boldsymbol{c}'_* = \left(\delta\boldsymbol{b}'^{-1}_* + \boldsymbol{M}'^{T}_*\boldsymbol{B}'^{-1}\boldsymbol{M}'_*\right)^{-1} \tag{B.4}$$

$$= \left(\boldsymbol{I} + \delta\boldsymbol{b}'_*\boldsymbol{M}'^{T}_*\boldsymbol{B}'^{-1}\boldsymbol{M}'_*\right)^{-1}\delta\boldsymbol{b}'_* \tag{B.5}$$

$$= \left[\boldsymbol{I} + \delta\boldsymbol{b}'_*\boldsymbol{M}'^{T}_*(\boldsymbol{U}'^{-1}(\boldsymbol{L}'^{-1}\boldsymbol{M}'_*))\right]^{-1}\delta\boldsymbol{b}'_*. \tag{B.6}$$

Thus, equations (B.3) and (B.6) are identical with (A.2) and (A.3). ■

REFERENCES

[1] A. Mittal, J. Hazra, N. Jain, V. Goyal, D. P. Seetharam, and Y. Sabharwal, "Real time contingency analysis for power grids," in *Proc. Eur. Conf. Euro-Par Parallel Process.*, Bordeaux, France, Sep. 2011, pp. 303–315.
[2] X. Li, P. Balasubramanian, M. Sahraei-Ardakani, M. Abdi-Khorsand, K. W. Hedman, and R. Podmore, "Real-time contingency analysis with corrective transmission switching," *IEEE Trans. Power Syst.*, vol. 32, no. 4, pp. 2604–2617, Jul. 2017.
[3] NERC. *Standard TPL-002-0b—System Performance Following Loss of a Single Bulk Electric System Element*. [Online]. Available: http://www.nerc.com/files/TPL-002-0b.pdf
[4] X. Wang, Y. Song, and M. Irving, *Modern Power Systems Analysis*. New York, NY, USA: Springer, 2008.
[5] C. Thompson, K. McIntyre, S. Nuthalapati, A. Garcia, and E. A. Villanueva, "Real-time contingency analysis methods to mitigate congestion in the ERCOT region," in *Proc. IEEE Power Energy Soc. Gen. Meeting*, Calgary, AB, Canada, Jul. 2009, pp. 1–7.
[6] Y. Chen, Z. Huang, and M. Rice, "Evaluation of counter-based dynamic load balancing schemes for massive contingency analysis on over 10,000 cores," in *Proc. SC Companion High Perform. Comput., Netw., Storage Anal. (SCC)*, Nov. 2012, pp. 341–346.
[7] F. Garcia *et al.*, "ERCOT control center experience in using real-time contingency analysis in the new nodal market," in *Proc. IEEE Power Energy Soc. Gen. Meeting*, San Diego, CA, USA, Jul. 2012, pp. 1–8.
[8] 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.
[9] 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, Sep. 2013.
[10] T. Cui, R. Yang, G. Hug, and F. Franchetti, "Accelerated AC contingency calculation on commodity multi-core SIMD CPUs," in *Proc. IEEE PES Gen. Meeting Conf. Expo.*, Jul. 2014, pp. 1–5.
[11] A. Gopal, D. Niebur, and S. Venkatasubramanian, "DC power flow based contingency analysis using graphics processing units," in *Proc. IEEE Lausanne Power Tech*, Jul. 2007, pp. 731–736.
[12] X. Li and F. Li, "GPU-based power flow analysis with chebyshev preconditioner and conjugate gradient method," *Electr. Power Syst. Res.*, vol. 116, pp. 87–93, Nov. 2014.
[13] G. Zhou *et al.*, "A novel GPU-accelerated strategy for contingency screening of static security analysis," *Int. J. Elect. Power Energy Syst.*, vol. 83, pp. 33–39, Dec. 2016.
[14] C. Guo, B. Jiang, H. Yuan, Z. Yang, L. Wang, and S. Ren, "Performance comparisons of parallel power flow solvers on GPU system," in *Proc. IEEE 18th Int. Conf. Embedded Real-Time Comput. Syst. Appl. (RTCSA)*, Aug. 2012, pp. 232–239.
[15] 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. 8, no. 4, pp. 1639–1648, Jul. 2017.
[16] G. Zhou *et al.*, "GPU-accelerated batch-ACPF solution for N-1 static security analysis," *IEEE Trans. Smart Grid*, vol. 8, no. 3, pp. 1406–1416, May 2017.
[17] 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. 8, no. 5, pp. 1396–1405, May 2017.
[18] J. Singh and I. Aruni, "Accelerating power flow studies on graphics processing unit," in *Proc. Annu. IEEE India Conf.*, Kolkata, India, Dec. 2010, pp. 1–5.
[19] M. Marin, D. Defour, and F. Milano, "Asynchronous power flow on graphic processing units," in *Proc. 25th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process.*, St. Petersburg, Russia, Mar. 2017, pp. 255–261.
[20] X. Li, F. Li, H. Yuan, H. Cui, and Q. Hu, "GPU-based fast decoupled power flow with preconditioned iterative solver and inexact Newton method," *IEEE Trans. Power Syst.*, vol. 32, no. 4, pp. 2695–2703, Jul. 2017.
[21] T. A. Davis, *Direct Methods for Sparse Linear Systems*. Philadelphia, PA, USA: SIAM, 2006.
[22] O. Alsac, B. Stott, and W. F. Tinney, "Sparsity-oriented compensation methods for modified network solutions," *IEEE Trans. Power App. Syst.*, vol. PAS-102, no. 5, pp. 1050–1060, May 1983.
[23] R. D. Zimmerman, C. E. Murillo-Sánchez, and R. J. Thomas, "MATPOWER: Steady-state operations, planning, and analysis tools for power systems research and education," *IEEE Trans. Power Syst.*, vol. 26, no. 1, pp. 12–19, Feb. 2011.
[24] M. A. Woodbury, "Inverting modified matrices," Stat. Res. Group, Princeton Univ., Princeton, NJ, USA, Tech. Memo., 1950, vol. 42, no. 106, p. 336.
[25] MISO. *MISO Reliability Assurance*. [Online]. Available: https://www.misoenergy.org/WhatWeDo/Pages/Reliability.aspx
[26] J. Baranowski and D. J. French, "Operational use of contingency analysis at PJM," in *Proc. IEEE Power Energy Soc. Gen. Meeting*, San Diego, CA, USA, Jul. 2012, pp. 1–4.
[27] *CUDA C Programming Guide 8.0*, NVIDIA, Santa Clara, CA, USA, 2017.
[28] *CUSPARSE Library 8.0*, NVIDIA, Santa Clara, CA, USA, 2017.

SHENGJUN HUANG received the B.Eng. and M.Eng. degrees in management science and engineering from the National University of Defense Technology (NUDT), Changsha, China, in 2011 and 2013, respectively, and the Ph.D. degree in energy systems from the University of Alberta, Edmonton, Canada, in 2018. He is currently a Lecturer with the College of Systems Engineering, NUDT. His research interests include mixed-integer linear programming, decomposition algorithms, large-scale power systems, optimization techniques, and parallel computing.

VENKATA DINAVAHI (S'94–M'00–SM'08) received the B.Eng. degree in electrical engineering from the National Institute of Technology, Nagpur, India, in 1993, the M.Tech. degree in electrical engineering from IIT Kanpur, India, in 1996, and the Ph.D. degree in electrical and computer engineering from the University of Toronto, Toronto, ON, Canada, in 2000. He is currently a Professor with the Department of Electrical and Computer engineering, University of Alberta, Edmonton, AB, Canada. His research interests include the real-time simulation of power systems and power electronic systems, electromagnetic transients, device-level modeling, large-scale systems, and parallel and distributed computing.