Research Article

Implementation and Optimization of GPU-Based Static State Security Analysis in Power Systems

Yong Chen,^{1,2} Hai Jin,¹ Han Jiang,³ Dechao Xu,² Ran Zheng,¹ and Haocheng Liu¹

¹Services Computing Technology and System Lab, Cluster and Grid Computing Lab, Big Data Technology and System Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

²China Electric Power Research Institute, Beijing 100192, China

³Global Energy Interconnection Development and Cooperation Organization, Beijing 100031, China

Correspondence should be addressed to Ran Zheng; zhraner@hust.edu.cn

Received 23 September 2016; Accepted 5 January 2017; Published 15 March 2017

Academic Editor: Beniamino Di Martino

Copyright © 2017 Yong Chen et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

Static state security analysis (SSSA) is one of the most important computations to check whether a power system is in normal and secure operating state. It is a challenge to satisfy real-time requirements with CPU-based concurrent methods due to the intensive computations. A sensitivity analysis-based method with *Graphics processing unit* (GPU) is proposed for power systems, which can reduce calculation time by 40% compared to the execution on a 4-core CPU. The proposed method involves load flow analysis and sensitivity analysis. In load flow analysis, a multifrontal method for sparse LU factorization is explored on GPU through dynamic frontal task scheduling between CPU and GPU. The varying matrix operations during sensitivity analysis on GPU are highly optimized in this study. The results of performance evaluations show that the proposed GPU-based SSSA with optimized matrix operations can achieve a significant reduction in computation time.

1. Introduction

An electric power system includes a network of connected electrical components which are used to supply, transfer, distribute, and use electric power. It is necessary to know the physical properties of the power system for safety. Different experiments can be conducted to study the properties of power system. Sometimes, however, it is impossible to perform such experiments on large-scale, complex power systems because the experiments are too expensive or difficult to take measurements on those power systems. Therefore, power system analysis [1] becomes another popular choice and plays more and more important role in planning, design, and operation of electrical power system. Power system analysis aims at building an equivalent model for a given power system, running load flow calculation, evaluating the effects when subjected to disturbances, and generating the ways to improve the stability performance of the power system. Power system analysis can be classified into *steady state analysis* and *transient stability analysis*. The former

is further categorized into *Load flow analysis* and *Static state security analysis* (SSSA). Load flow analysis involves determining voltages and power flow through a stable system, where any transient disturbances are assumed to have settled down. SSSA involves power flow calculation when a chosen device is connected or disconnected. Transient stability is analyzed to estimate the system's stability under dynamic conditions during disturbances.

Load flow analysis is one of the most significant computations in power system planning and operations. In load flow analysis, we need to model the entire network with all generators, loads, transmission lines, transformers, and reactors. Following the modeling, since power system is a large-scale and highly nonlinear dynamic system, power flow calculation involves solving higher-dimensional sparse nonlinear algorithmic equations based on nodal admittance form. Load flow equations allow us to compute bus voltage magnitudes and phase angles as well as branch current magnitudes. Solving nonlinear equations is related to iterative computations, which are data- and computation-intensive.

SSSA can be replaced by a series of load flow analyses. However, the computation would be more intensive if a rigorous load flow calculation is used. Some studies are focusing on SSSA to improve the performance [2, 3]. Many highly simplified algorithms are worked out to save iterative computations, and a few approaches with parallel load flow analysis are proposed. *Graphics processing unit* (GPU) is becoming an attractive accelerator for parallel computation [4] and can achieve high performance for general-purpose computation. Load flow analysis is intended to solve high-dimensional sparse nonlinear equations, and sensitivity analysis solves changes in network state based on the load flow results, which uses a number of sparse matrix operations, such as matrix multiplication, addition, and inversion. These features make GPU a suitable and viable solution for SSSA. In this study, while we have provided a GPU-based SSSA solution [5], in practice, a considerable amount of work is needed to continue improving its speed and performance.

Many operations in SSSA can be parallelized on GPU to improve the performance. However a hybrid approach is necessary to combine CPU and GPU computations in GPU-based SSSA. Branch prediction or speculative execution is not supported on GPU, which is not good at iterative operations and judgements of convergence in solving nonlinear equations. For a large sparse matrix, the nonzero storage and computation mechanism should be adopted. The proposed method can combine various small matrices into one matrix in multiplication to use the GPU threads better.

The reminder of this paper is organized as follows: the background and related work are described in Section 2. Section 3 describes the workflow of static state security analysis and its modules. Section 4 addresses the optimization of matrix operations. Section 5 evaluates the performance of the proposed system and addresses optimization issues. Section 6 contains the conclusions of this study as well as a discussion of future improvements in the proposed system.

2. Related Work

SSSA is an important computation tool in the design and operation of large, interconnected power systems. It determines whether a system is operating in a secure state at any given time with respect to unforeseen outages m in N buses, lines, transformers, or generators. $N - 1$ branch outage simulation is a basic validation of a power grid safety [6]. SSSA can be implemented by running one load flow calculation for each outage case. If a large number of cases are provided, the time needed for calculation and analysis can be very long. Hence, other methods, such as DC (*Direct Current*) power method and sensitivity analysis method, have been proposed to speed up static state security analysis.

The DC power method [7] can solve nonlinear equations in power systems by transforming them into linear equations. This reduces computational complexity to make calculations more efficient but can lead to poor precision. The DC power method can only check for the overload in practical application. If the voltage is raised above the upper limit, it will not be checked. The DC power method is commonly used

to design power systems, for which the active power flows are an important concern.

In sensitivity analysis, the offline of one transmission line is regarded as a perturbation under normal circumstances [8]. The method involves calculating a sensitivity matrix from the Taylor series expansion of load flow equations. The impact of line outage can be simulated by net injection and withdraw changes. Many analyses of outages on the same network conduct only one load flow calculation and perform their sensitivity analysis on the load flow results. This bypasses many intermediate, iterative steps of the calculation and significantly increases the efficiency of line-off analysis. This method can not only improve performance and precision, but also obtain active power, reactive power, voltage magnitude, and angle at the system node. Therefore, it is commonly used in general practice.

Static state security analysis pays more attention to the calculation of the linear algebraic operation. Reference [9] presents a systematic approach for a large set of frequently encountered dense linear algebra operations, but it is shown to yield new high-performance algorithms. There are many researches based on GPU, in which cuBLAS [10] is the most commonly used. cuBLAS is a linear algebraic library on GPU, which is an implementation of BLAS (*Basic Linear Algebra Subprograms*) on NVIDIA's CUDA (*Compute Unified Device Architecture*) runtime. When programming on cuBLAS, we use it like BLAS and do not care about the details of thread modeling or the storage model of CUDA programming.

Reference [11] proposed a hybrid CPU-GPU implementation of sparse Cholesky factorization based on the multi-frontal method. Reference [12] introduced an efficient GPU-based sparse solver for circuit problems and developed a hybrid parallel LU factorization approach combining task-level and data-level parallelism on GPUs. Reference [13] proposed an implementation of the *Newton-Raphson* (NR) load flow algorithm, as it pertained to parallelizing and implementing in CUDA. However, there is no simple and efficient static state security analysis method based on GPU to conduct $N - 1$ branch outage simulations.

3. Workflow and Modules of Static State Security Analysis

3.1. Overview of Static State Security Analysis. Sensitivity analysis is a method to be highly parallelized, which is convenient to be used with GPU to accelerate calculations in static state security analysis. The topology of the power grid can be simply modeled as a graph with m nodes and n edges, in which the nodes represent buses (power stations or transformers) and the edges represent transmission lines [14]. The graph network can be converted into a nodal admittance matrix or a nodal impedance matrix. The two matrices are highly sparse for a large power system. According to the conservation of complex power theorem, the load flow equations can be written as follows:

$$\Delta P_i = P_{is} - V_i \sum_{j \in i} V_j (G_{ij} \cos \theta_{ij} + B_{ij} \sin \theta_{ij}) = 0,$$

$$\Delta Q_i = Q_{is} - V_i \sum_{j \in i} V_j (G_{ij} \sin \theta_{ij} - B_{ij} \cos \theta_{ij}) = 0, \quad (i = 1, 2, \dots, n), \quad (1)$$

where P_{is} and Q_{is} are the injected active and the reactive powers, respectively, at node i . G_{ij} , B_{ij} are the elements of nodal admittance matrix. Load flow calculations can be roughly considered as the problem of solving node voltages V_i , θ_i ($i = 1, 2, \dots, n$) at each node when the injecting complex powers P_{is} , Q_{is} ($i = 1, 2, \dots, n$) are given. This nonlinear equation can be solved by the *Newton-Raphson* (NR) method, which is based on a shortened Taylor series. NR needs to iteratively solve (2) until it converges within a given tolerance of ΔP_i , ΔQ_i ($i = 1, 2, \dots, n$):

$$\begin{bmatrix} \Delta P \\ \Delta Q \end{bmatrix} = J \begin{bmatrix} \Delta \theta \\ \frac{\Delta V}{V} \end{bmatrix}. \quad (2)$$

For a large system, the load flow calculation may require significant computational resources to calculate, store, and factorize the Jacobian matrix J . Following load flow analysis, the normal state of the system is determined. If there is a randomly injected power disturbance ΔW or network change ΔY , the state vector is correspondingly changed by ΔX . The equation can be expressed as follows:

$$W_0 + \Delta W = f(X_0 + \Delta X, Y_0 + \Delta Y) \quad (3)$$

in which W_0 is the active and reactive power of nodes in normal state, X_0 is the voltage vector of the nodes, and Y_0 is the normal network parameter. When power disturbance is ignored, $\Delta W = 0$, and Taylor series expansion [15] is used for (3). Finally, we obtain the following equations:

$$\begin{aligned} \Delta x &= S_0 \Delta W_y, \\ \Delta W_y &= [I + L_0 S_0]^{-1} \cdot (-f'_y(X_0, Y_0) \Delta Y). \end{aligned} \quad (4)$$

In the above, S_0 is the sensitivity matrix, which is equal to J^{-1} . If only one line-off is considered in SSSA, the injected power change ΔW_y of nodes along the offline can be obtained by

$$\begin{bmatrix} \Delta P_i \\ \Delta Q_i \\ \Delta P_j \\ \Delta Q_j \end{bmatrix} = H^{-1} \begin{bmatrix} P_{ij} \\ Q_{ij} \\ P_{ji} \\ Q_{ji} \end{bmatrix}, \quad (5)$$

where

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$+ \begin{bmatrix} -H_{ij} & 2P_{ij} - N_{ij} & H_{ij} & N_{ij} \\ -J_{ij} & 2Q_{ij} - L_{ij} & J_{ij} & L_{ij} \\ H_{ji} & N_{ji} & -H_{ji} & 2P_{ji} - N_{ji} \\ J_{ji} & L_{ji} & -J_{ji} & 2Q_{ji} - L_{ji} \end{bmatrix} \times \begin{bmatrix} S_{ii}^{(1)} & S_{ii}^{(2)} & S_{ij}^{(1)} & S_{ij}^{(2)} \\ S_{ii}^{(3)} & S_{ii}^{(4)} & S_{ij}^{(3)} & S_{ij}^{(4)} \\ S_{ji}^{(1)} & S_{ji}^{(2)} & S_{jj}^{(1)} & S_{jj}^{(2)} \\ S_{ji}^{(3)} & S_{ji}^{(4)} & S_{jj}^{(3)} & S_{jj}^{(4)} \end{bmatrix}, \quad (6)$$

$$H = I + L \cdot S. \quad (7)$$

In the above equations, H_{ij} , N_{ij} , J_{ij} , and L_{ij} , the elements of the Jacobian matrix J , are calculated as follows:

$$\begin{aligned} H_{ij} &= \frac{\partial P_i}{\partial \theta_j} = V_i V_j (G_{ij} \sin \theta_{ij} - B_{ij} \cos \theta_{ij}), \\ N_{ij} &= V_j \frac{\partial P_i}{\partial V_j} = V_i V_j (G_{ij} \cos \theta_{ij} + B_{ij} \sin \theta_{ij}), \\ J_{ij} &= \frac{\partial Q_i}{\partial \theta_j} = -V_i V_j (G_{ij} \cos \theta_{ij} + B_{ij} \sin \theta_{ij}), \\ L_{ij} &= V_j \frac{\partial Q_i}{\partial V_j} = V_i V_j (G_{ij} \sin \theta_{ij} - B_{ij} \cos \theta_{ij}), \end{aligned} \quad (8)$$

($j \neq i$)

and $S_{ij}^{(1)}$, $S_{ij}^{(2)}$, $S_{ij}^{(3)}$, and $S_{ij}^{(4)}$, the elements related to the offline node in the sensitivity matrix, are calculated as follows:

$$\begin{aligned} S_{ij}^{(1)} &= \frac{\partial \theta_i}{\partial P_j}, \\ S_{ij}^{(2)} &= \frac{\partial \theta_i}{\partial Q_j}, \\ S_{ij}^{(3)} &= \frac{1}{V_i} \frac{\partial V_i}{\partial P_j}, \\ S_{ij}^{(4)} &= \frac{1}{V_i} \frac{\partial V_i}{\partial Q_j}. \end{aligned} \quad (9)$$

We find that the items on the right-hand side of (6) come from load flow calculation, and only two 4×4 matrix operations occur in the equation.

The overall workflow of the static state security analysis system is shown in Figure 1. It consists of four main modules: I/O, preprocessing, power flow calculation, and static state security analysis.

When the original data is input, it is preprocessed first for power flow calculations. The system then conducts a full power flow calculation and determines the Jacobian matrix J , which is prepared for static state security analysis. Finally, the

FIGURE 1: The overall workflow of static state security analysis.

changing network parameters are assumed by a line-off and new system state variables identified. Once a set of critical elements are simulated to be tripped, we can determine whether there are equipment-limit violations through SSSA. Hence, the state of power system (secure or unsecured) can be easily determined.

3.2. Preprocessing. Data preprocessing is a major and essential stage in power flow calculation, which can speed up static state security analysis. There are three processings: sparse matrix storage, node numbering optimization, and result storage optimization, which aim at reducing storage space for matrices and speeding up the calculation of nonzero elements.

Following raw data input, data in large sparse matrix is compressed in CSR (*Compressed Row Storage*) format [16]. CSR is very efficient due to an indirect addressing step for every single scalar operation in matrix vector. The subsequent nonzero elements in matrix rows are placed in contiguous locations in memory and traversed in a row-wise fashion.

By matrix sorting with the minimum-degree minimum-length algorithm [17] or the minimum-degree minimum-number algorithm [18], the rows and columns of sparse matrix are moved to avoid creating new nonzero elements in the matrix during factorization, and the forward and backward substitutions can save many nonzero computations. Such node numbering optimizations are conducted with an elimination tree, which can determine the position of injection elements in factorization.

Static state security analysis is performed on the results of power flow calculations. Since three read operations are necessary when accessing CSR-formatted data every time, it incurs a considerable costs for I/O operations. Hence, the matrix is decompressed prior to calculation. It may take up a lot of space to assemble several matrices, and the memory may not be large enough to store the matrices. Therefore, preprocessing involves the following three steps:

- (1) Analyze the result of power flow calculation J , where sensitivity matrix is obtained by the inverse S_0 . Allocate memory to store matrices assembled from S_0 .
- (2) Generate submatrices and vectors from S_0 with L and S in (6) and (7), respectively.
- (3) Check matrix size. If it does not exceed the capacity, copy it directly into GPU device memory; otherwise, partition the matrix and copy submatrices into device.

3.3. Power Flow Calculation. Power flow calculation is executed with the *Newton-Raphson* (NR) method to solve large sparse nonlinear equations. The NR method repeatedly solves large, sparse, linear systems of equations. It is a significant challenge of computational and memory capacity to factorize the Jacobian matrix at each iteration; hence, it is crucial to design a fast and efficient solution for the factorization. The multifrontal algorithm is a particularly useful method for factoring large sparse linear systems and is adopted in the proposed solution. The workflow of the method is shown in Figure 2.

FIGURE 2: The workflow of power flow calculation.

The multifrontal algorithm [19] is an effective method to solve large sparse matrices operation. A frontal matrix is a small, dense matrix. The term “*multi*” here refers to the fact that multiple frontal matrices are used during the factorization. The multifrontal method turns the factorization of sparse matrix into a sequence of factorizations of smaller, dense matrices organized as an elimination tree or an assembly tree. The method can get satisfactory data locality and great potential for parallelization. For example, [20] investigates an automatic tuning of SpMV (Sparse

Matrix Vector) multiplication kernel in a partitioned global address space language, which supports a hybrid thread- and process-based communication layer for multicore systems. The multifrontal method proposed here is associated with super-nodal implementation, so that multiple columns with the same nonzero patterns can be grouped together as a dense kernel for concurrent factorization.

The method is formulated in terms of frontal matrices and update matrices. The processing order of frontal matrices is determined by the elimination tree from the preprocessing.

A_{11}	A_{12}	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}
A_{41}	A_{42}	A_{43}	A_{44}

 $\times$

B_{11}	B_{12}	B_{13}	B_{14}
B_{21}	B_{22}	B_{23}	B_{24}
B_{31}	B_{32}	B_{33}	B_{34}
B_{41}	B_{42}	B_{43}	B_{44}

 $=$

C_{11}	C_{12}	C_{13}	C_{14}
C_{21}	C_{22}	C_{23}	C_{24}
C_{31}	C_{32}	C_{33}	C_{34}
C_{41}	C_{42}	C_{43}	C_{44}

Thread 1: $C_{11} = A_{11}B_{11} + A_{12}B_{21} + A_{13}B_{31} + A_{14}B_{41}$
 Thread 2: $C_{12} = A_{11}B_{12} + A_{12}B_{22} + A_{13}B_{32} + A_{14}B_{42}$
 $\vdots$
 Thread 16: $C_{44} = A_{41}B_{14} + A_{42}B_{24} + A_{43}B_{34} + A_{44}B_{44}$

FIGURE 3: Submatrix multiplication with 16 threads.

The order is expressed by a frontal matrix chain, in which each frontal matrix is designated as a leaf and processed by a CPU thread. For node j , subtree update matrix $\bar{U}_j$ is the sum of all subtree update matrices; frontal matrix F_j is formed by assembly over $\bar{U}_j$ and can be partitioned into four submatrices A , B , C , and D with (10). Then the Cholesky factor vector and the update matrix of j can be solved with following Cholesky factorization:

$$\begin{pmatrix} B & A \\ D & C \end{pmatrix} = \begin{pmatrix} L_1 & 0 \\ L_2 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & C - L_2 U_2 \end{pmatrix} \begin{pmatrix} U_1 & U_2 \\ 0 & 1 \end{pmatrix}. \quad (10)$$

It will take a considerable amount of time for matrix operations in the multifrontal method. A threshold is defined to distinguish CPU from GPU tasks and is derived from matrix calculations. If the number of the computations is greater than the threshold, the main thread will allocate tasks to GPU. Otherwise, a small-scale matrix is computed on CPU. Nodes in frontal matrix chain are calculated one by one until all of them are computed. ΔX is then computed backward through substitute, decomposed matrices L and U to update X . If the absolute value of ΔX is less than 10^{-8} , which is regarded as convergence, the calculation is concluded; otherwise, the iterations will continue to get a convergent result.

3.4. Static State Security Analysis. In this module, the GPU-based sensitivity method is used for static state security analysis. The elements in the matrices of (6) are prepared from the previous load flow analysis. It consists of four steps.

Step 1. Calculate $T = L \cdot S$ in (6) by matrix multiplication.

Step 2. Calculate $H = T + I$ in (6) by matrix addition.

Step 3. Calculate H^{-1} by matrix inversion from H .

Step 4. Calculate (5) by using a matrix multiplying vector.

Two matrix multiplication and one matrix inversion operations are involved during the analysis. Following these

steps, the changes in the node state variables can be obtained, so that the power flow on each branch can be acquired following line outage. Moreover, many cases of SSSA can be combined in parallel on a GPU to improve the efficiency.

4. Optimization of Matrix Operations

4.1. Small Matrix Multiplication. In 4×4 matrix multiplication on GPU, 16 threads are executed for concurrent element multiplication in one block, shown in Figure 3. The 16 threads are completely independent and communicate with one another in the block. However, the same instruction is executed in a warp, which is allowed to launch 32 threads at once. Whether there are 32 or 16 threads in a task, they are launched by a warp with the same time cost. Hence if only one submatrix multiplication occurs in a block with only a warp, 16 threads are launched and the other 16 are idle.

It is a best practice to allocate more than one multiplication operation to a block. A block can contain a maximum of 1024 threads, which can store 64 submatrix multiplication operations. Host threads can combine their data into a large matrix and copy it to share the memory of the block. Since the instructions are the same in a block, it is necessary to access own data of the large matrix in each thread. The distribution of logical memory is shown in Figure 4. We assume that every 16 threads process a matrix multiplication operation and 64 matrix multiplications can be concurrently handled in a block.

4.2. Small Matrix Inversion. Between J and S_0 , there is a matrix inversion operation. In general, there are two methods for matrix inversion. One is the adjoint matrix method, which executes the calculation of $A^{-1} = A^* / |A|$. The adjoint matrix A^* of A must be calculated first for this. It is too complicated to derive the expression of A^* for matrix A of order greater than 3. The other matrix inversion method is the elementary transformation method, which is simpler than the adjoint matrix method for high-order matrices, but more iterations are required and inefficient in terms of GPU

FIGURE 4: The distribution of logical memory.

resource consumption. In order to enhance efficiency, (11) is used to partition matrix inversion:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} A^{-1} + A^{-1}B(D - CA^{-1}B)^{-1}CA^{-1} & -A^{-1}B(D - CA^{-1}B)^{-1} \\ D(D - CA^{-1}B)^{-1}CA^{-1} & (D - CA^{-1}B)^{-1} \end{bmatrix}. \quad (11)$$

If a 4×4 matrix is provided, it can be divided into four 2×2 matrices. In this manner, the inversion task is divided and assigned to four threads, which can complete their tasks almost simultaneously. In (11), only the operations of matrix multiplication, addition, and 2×2 matrix inversion are given. 2×2 matrix inversion can be expressed as follows:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \begin{bmatrix} \frac{d}{ad-bc} & \frac{-b}{ad-bc} \\ \frac{-c}{ad-bc} & \frac{a}{ad-bc} \end{bmatrix}. \quad (12)$$

In this way, four threads can be used with (12) to calculate the inversion of A , B , C , and D in formula (11). The results are substituted into (11) to work out the inversion of the 4×4 matrix.

4.3. Vector Multiplication by Cross-Combining Storage. When solving (5), each change in power flow is obtained by multiplying 4×1 row vector by 1×4 column vector. A set of vector-vector multiplication can be combined into a matrix vector-vector multiplication as Figure 5.

In sensitivity analysis, $32n$ (n is a positive integer) row vectors are merged into one long row vector placed in global memory. Global memory allows a warp of 32 threads to access it concurrently. As Figure 6 shows, there are $32 \times 4 \times 1$ row

FIGURE 5: Matrix vector multiplication.

vectors; one long row vector x is created in global memory. The first, second, third, and fourth column elements of the 32 row vectors are set to $x[0 : 31]$, $x[32 : 63]$, $x[64 : 95]$, and $x[96 : 127]$, respectively. When row vector x is multiplied by column vector y , $x[0 : 31]$, $x[32 : 63]$, $x[64 : 95]$, and $x[96 : 127]$ are multiplied by $y[a]$, $y[b]$, $y[c]$, and $y[d]$, respectively. Since vector y is accessed many times, y is stored in the shared memory.

A number of row vectors are stored in one row vector by cross-combination. It is necessary to execute a reduce operation after vector multiplication. The result of vector multiplication is stored back into x . The four result elements (i , $i + 32$, $i + 64$, $i + 96$) are grouped, summed up, and stored at i . Eventually, $x[0 : 32]$ is the result of 32 multiplications of a 1×4 vector by another 4×1 vector.

4.4. RC-MM Storage Method. In general practice, cublasSgemm function in cuBLAS is called for GPU matrix multiplication. The matrices in C/C++ are in row-major order, but cuBLAS assumes that the matrices are stored in column-major order in the devices. The order exchange is a time-consuming operation. Hence the RC-MM (Row Column-Matrix Multiplication) method is adopted, and A , B , and C are stored in column-major, row-major, and column-major order, respectively, in $C = AB$.

For the multiplication in the multifrontal method, one matrix is stored in row-major order and the another matrix in column-major order. The matrices of the same frontal chain share an array, because of which the entire row or column data is copied to global memory. Each computation only uses part of the data, which avoids having to move large amounts of data on demand.

5. Performance Evaluation

5.1. Dataset and Experimental Environment. The experiments are conducted on a server with an Intel i7 950 (3.07 GHz) CPU, 16 GB memory, and NVIDIA GeForce GTX460. The CentOS 5.9 Linux operation system with CUDA 4.0 is used. From MATPOWER [21], a professional software in power calculation derived from a real power grid and some IEEE standard testing datasets are chosen, shown in Table 1.

FIGURE 6: Matrix vector multiplication by cross-combining storage.

TABLE 1: Test datasets for SSSA experiments.

Dataset name	CA300	CA3012wp	CA3120sp	CA5472	CA5492wp	CA6024	CA6240
Number of nodes	300	3012	3120	5472	5492	6024	6240
Number of lines	411	3572	3693	9794	9824	10706	11069

CA300 is the power flow data for the IEEE 300 bus test case. CA3012wp and CA3120sp are the power flow data cases for the Polish system winter 2007-08 evening peak and summer 2008 morning peak, respectively. In order to evaluate the computation involved in a large power system, two identical Polish power systems are connected to form the large symmetric power system CA5472, CA5492wp, CA6024, and CA6240.

The speedup ratio S is defined in (13), where $T_{\text{CPU_time}}$ and $T_{\text{GPU_time}}$ are the execution times on CPU and GPU, respectively. The program executed on CPU is a multithread program on multicore CPU.

$$S = \frac{T_{\text{CPU_time}}}{T_{\text{GPU_time}}} \times 100\%. \quad (13)$$

5.2. Evaluation of Static State Security Analysis System. Executed on GPU and CPU platforms, the datasets in Table 1 are used for testing. The results are shown in Table 2.

In Table 2, there is a 4-core processor in the CPU, with which the program can almost achieve best performance. But with increasing of the number of active processor cores in the compute node, the program on the CPU platform suffers serious performance degradation. In dataset CA3012wp, the calculation time on an 8-core CPU increased by 13% compared with a 4-core CPU.

The experimental results show that the execution time on GPU is shorter than on CPU, except on a scale of 300 nodes. It takes much time to transfer data between host and device in GPU. When the scale is 300 nodes, the transfer time cannot be ignored, and it is occupied by a significant proportion of the execution time on GPU. On the other hand, for only 300 submatrices, every two submatrices are processed in a warp on average. However for 336 cores on GTX 460, it means that over half the cores are idle throughout the processing, which is a serious waste of GPU resources to hinder overall performance.

Excluding the dataset of the 300 nodes, the accelerating effect of GPU computation is reflected, and significant

FIGURE 7: Comparison in terms of execution time between CPU and GPU speedup.

speedup is revealed with increasing data scale. GPU computing can reduce calculation time by 40% compared with the execution on 4-core CPU. The increasing trend of GPU speedup is shown in Figure 7.

5.3. Evaluation of Small Matrix Multiplication. Equation (6) is chosen to run the performance experiments. The formula consists of 4×4 matrix multiplication and 4×4 matrix addition. Two experiments are done. In Experiment 1 (Exp1), two combined submatrices are stored in one block after optimization. In Experiment 2 (Exp2), one submatrix is stored in one block. For different matrix storages, the experimental results for matrix multiplication and addition in (6) are shown in Figure 8.

The optimization method can achieve a speedup of approximately 1.7. A number of parallel threads are launched by a warp. If the matrices are not merged into a block, half the threads will be idle in a warp, which can launch 32 threads.

TABLE 2: Comparison in terms of execution time between CPU and GPU.

Scale	200		3000		5000		6000	
Dataset name	CA300	CA3012wp	CA3120sp	CA5472	CA5492wp	CA6024	CA6240	
CPU (ms)	10.0	446.6	476.7	2170.0	2186.7	2603.3	2793.3	
GPU (ms)	19.7	269.3	269.6	1214.3	1213.0	1408.0	1483.7

FIGURE 8: Comparison in terms of matrix multiplication execution times of the two experiments.

Following matrix combination, the 32 threads in the warp can be kept busy, which indicates that the optimization method can exhibit good performance.

5.4. Evaluation of Small Matrix Inversion. Two experiments are conducted on 4×4 matrix inversion in (5). cuBLAS has provided a large matrix inversion function with low efficiency for batch small matrices inversion. The small batch matrix inversion method is implemented on GPU kernel, in which one submatrix is processed by one thread. The small batch matrix inversion function is called in Exp1 once, and 100 inversion threads are scheduled on the GPU. The cuBLAS matrix inversion function is called 100 times in Exp2. The results show that the small matrix inversion method yields better performance (Exp1 925 ms versus Exp2 1055 ms) in Figure 9.

6. Conclusion and Future Work

In this paper, GPU-based static state security analysis is proposed for power systems. The GPU-based multifrontal method is implemented to solve a large sparse matrix, and sensitivity analysis is chosen for static state security analysis on GPU. To make full use of GPU device, several optimization methods of matrix operations are presented, such as data combination in multiple small-scale matrix multiplication operations and the partition matrix method for matrix inversion.

Experimental results indicate that the proposed algorithm on GPU can significantly improve system performance.

FIGURE 9: Comparison in terms of small matrix inversion times of the two experiments.

Our results show a speedup of 1.7–1.9 with power system simulation cases from a scale of 3,000 to 6,000.

In future work, it may be desirable to further improve performance that the system and methods could be ported to more scalable distributed memory environment, such as multi-GPUs [22]. We can also use the compilers to speed up system migration and dynamic task scheduling in CPU-GPU heterogeneous parallel systems. Our way of dealing with small-scale matrices can be used in scientific calculations in other fields, and the processing method of special-dimensional matrix can be extended to all-dimensional matrices.

Competing Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (no. 61133008), the National 973 Key Basic Research Plan of China (no. 2013CB2282036), Major Subject of State Grid Corporation of China (no. SGCC-MPLG001(001-031)-2012), the National 863 Research and Development Program of China (no. 2011AA05A118), and the National Science and Technology Pillar Program (no. 2012BAH14F02).

References

- [1] J. D. Glover, M. S. Sarma, and T. Overbye, *Power System Analysis and Design*, Cengage Learning, 2016.

- [2] P. Ding, Y. Li, D. Xu, F. Tian, J. Yan, and Z. Yu, "Improved algorithm of fast static state security analysis of power systems," *Proceedings of the Chinese Society of Electrical Engineering*, vol. 30, no. 31, pp. 77–82, 2010.
- [3] G. Zhou, X. Zhang, Y. Lang et al., "A novel GPU-accelerated strategy for contingency screening of static security analysis," *International Journal of Electrical Power & Energy Systems*, vol. 83, pp. 33–39, 2016.
- [4] J. D. Owens, D. Luebke, N. Govindaraju et al., "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.
- [5] Y. Chen, H. Jin, H. Jiang, D. Xu, R. Zheng, and H. Liu, "GPU-based static state security analysis in power systems," in *Proceedings of the 9th Asia-Pacific Services Computing Conference (APSCC '15)*, pp. 258–267, Bangkok, Thailand, December 2015.
- [6] X. Li and Z. Guo, "The transmission interface real power flow control based on N-1 static safety restriction," *Electric Power*, vol. 38, no. 3, pp. 26–28, 2005.
- [7] K. Purchala, L. Meeus, D. Van Dommelen, and R. Belmans, "Usefulness of DC power flow for active power flow analysis," in *Proceedings of the IEEE Power Engineering Society General Meeting*, pp. 454–459, San Francisco, Calif, USA, June 2005.
- [8] X. Wang, W. Fang, and Z. Du, *Modern Power System Analysis*, Science Press, 2016.
- [9] P. Bientinesi, J. A. Gunnels, M. E. Myers, E. S. Quintanaort, and R. A. van de Geijn, "The science of deriving dense linear algebra algorithms," *ACM Transactions on Mathematical Software*, vol. 31, no. 1, pp. 1–26, 2005.
- [10] NVIDIA, "cuBLAS," <https://developer.nvidia.com/cublas>.
- [11] R. Zheng, W. Wang, H. Jin, S. Wu, Y. Chen, and H. Jiang, "GPU-based multifrontal optimizing method in sparse Cholesky factorization," in *Proceedings of the 26th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '15)*, pp. 90–97, IEEE, Ontario, Canada, July 2015.
- [12] X. Chen, L. Ren, Y. Wang, and H. Yang, "GPU-accelerated sparse LU factorization for circuit simulation with performance modeling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 3, pp. 786–795, 2015.
- [13] D. J. Sooknanan and A. Joshi, "GPU computing using CUDA in the deployment of smart grids," in *Proceedings of the SAI Computing Conference (SAI '16)*, pp. 1260–1266, London, United Kingdom, July 2016.
- [14] L. Xuan, L. Tianqi, and L. Xingyuan, "A novel evolving model for power grids," *Science China Technological Sciences*, vol. 53, no. 10, pp. 2862–2866, 2010.
- [15] S. Wang, Z. Zheng, and C. Wang, "Power system transient stability simulation under uncertainty based on Taylor model arithmetic," *Frontiers of Electrical and Electronic Engineering in China*, vol. 4, no. 2, pp. 220–226, 2009.
- [16] J. L. Greathouse and M. Daga, "Efficient sparse matrix-vector multiplication on GPUs Using the CSR storage format," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '14)*, pp. 769–780, November 2014.
- [17] A. Gómez and L. G. Franquelo, "An efficient ordering algorithm to improve sparse vector methods," *IEEE Transactions on Power Systems*, vol. 3, no. 4, pp. 1538–1544, 1988.
- [18] R. Betancourt, "An efficient heuristic ordering algorithm for partial matrix refactorization," *IEEE Transactions on Power Systems*, vol. 3, no. 3, pp. 1181–1187, 1988.
- [19] J. W. Liu, "The multifrontal method for sparse matrix solution: theory and practice," *SIAM Review*, vol. 34, no. 1, pp. 82–109, 1992.
- [20] S. G. Li, C. J. Hu, J. C. Zhang, and Y. Q. Zhang, "Automatic tuning of sparse matrix-vector multiplication on multicore clusters," *Science China Information Sciences*, vol. 58, no. 9, pp. 1–14, 2015.
- [21] 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 Transactions on Power Systems*, vol. 26, no. 1, pp. 12–19, 2011.
- [22] D. Chen, W. Chen, and W. Zheng, "CUDA-Zero: a framework for porting shared memory GPU applications to multi-GPUs," *Science China Information Sciences*, vol. 55, no. 3, pp. 663–676, 2012.