SIMD-Based Large-Scale Transient Stability Simulation on the Graphics Processing Unit

Vahid Jalili-Marandi, *Student Member, IEEE*, and Venkata Dinavahi, *Senior Member, IEEE*

***Abstract*—This paper presents a single-instruction-multiple-data (SIMD) based implementation of the transient stability simulation on the Graphics Processing Unit (GPU). Two programming models to implement the standard method of the transient stability simulation are proposed and implemented on a single GPU. In the first model the CPU is responsible for part of the simulation, while the onerous computations were offloaded to the GPU, creating a hybrid GPU-CPU simulator. In the second model, the GPU performs all the computations, while the CPU simply monitors the flow of the simulation. The accuracy of the proposed methods are validated using the PSS/E software for several large test systems. A substantial increase in speed was observed for the GPU-based simulations.**

***Index Terms*—Graphics processors, large-scale systems, parallel programming, power system simulation, power system transient stability.**

I. Introduction

A POWER system is constantly mutating as the grid conditions change dynamically. System operators utilize the energy management system (EMS) to keep track of these changes and take necessary control actions on the grid. Transient stability evaluation is a crucial part of this decision-making process formally referred to as the dynamic security assessment (DSA) program executed on the control center computers [1]. Currently, this analysis takes several hours to complete for a realistic-size power system. The overriding reason is that transient stability simulation of a large system involves computationally onerous time-domain solution of thousands of nonlinear differential algebraic equations (DAEs), which must be solved for each condition of a large set of possible outage conditions on the grid. Since the 1960s several approaches have been developed to perform this simulation faster for large-scale systems on parallel computing hardware [2]–[4]. A commonality of these methods is *task-parallelism* which executes the component subsystems on parallel and distributed hardware composed of clusters of CPUs. Nevertheless, there are two limitations in these methods that contribute to the speed bottleneck: 1) data-sequential computation on the CPUs of the cluster, and 2) limited data bandwidth, and latency of the inter-processor communication channels of the cluster.

Recently, the graphics processing unit (GPU) has revealed the potential to revolutionize state-of-the-art research in *data-parallel* computing. In the early 2000s the GPU was a fixed-function accelerator originally developed to meet the needs for fast graphics in the video game and animation industries [5]. The demand to render more realistic and stylized images in these applications increased with time; a lifelike rendering of an image requires billions of pixels per second with each pixel taking hundreds of operations. The existing obstacle in the fixed-function GPU was the lack of generality to express complicated graphical operations such as shading and lighting that are imperative for producing high quality visualizations. The answer to this problem was to replace the fixed-function operations with user-specified functions. Developers, therefore, focused on improving both the application programming interface (API) as well as the GPU hardware. The result of this evolution is a powerful programmable processor with enormous arithmetic capability which could be exploited not only for graphics applications but also for general purpose computing (GPGPU). Compared to a CPU, a GPU can deliver a much higher performance at a lower cost and lower power consumption. Taking advantage of the GPUs' massively parallel architecture, the GPGPU applications quickly mushroomed to include intensive computations such as those in molecular biology, image and video processing, n-body simulations, large-scale database management, and financial services [6].

In this paper we introduce GPU-based transient stability simulation for large-scale power systems. The motivation for this work is twofold: the mathematical complexity along with the large data-crunching need in this simulation, and the substantial opportunity to exploit parallelism. Both these characteristics are uniquely suited to the GPU. However, since the GPU's architecture is markedly different from that of a conventional CPU, it requires a completely different algorithmic approach for implementation. The GPU thrives on applications that have a large computational requirement, and where data-parallelism can be exploited. A data-parallel application consists of large streams of data elements in the form of matrices and vectors that have identical computation codes (kernels) applied to them. The data-parallel characteristic of the GPU gives it a single-instruction-multiple-data (SIMD) architecture. Therefore any computation that is desired to be implemented on the GPU must be in the SIMD format, otherwise the GPU cannot deliver its computational benefits. Here we propose two SIMD-based programming models to exploit the GPU's resources for transient stability simulation. In the first model the GPU is used as a co-processor for the CPU to offload certain computational tasks, whereas in

Manuscript received June 01, 2009; revised December 31, 2009. First published March 01, 2010; current version published July 21, 2010. This work was supported by the Natural Science and Engineering Research Council of Canada (NSERC). Paper no. TPWRS-00403-2009.

The authors are with the Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB T6G 2V4, Canada (e-mail: v_jalili@ece.ualberta.ca; dinavahi@ece.ualberta.ca).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TPWRS.2010.2042084

the second model the GPU works as a stand-alone processor and the CPU only controls the flow of the simulation. Both these models were implemented using an inexpensive off-the-shelf GPU (Geforce GTX 280 from NVIDIA®) for the transient stability simulation. The accuracy and efficiency of the simulation were evaluated for several large test power systems with sizes up to 1248 buses and 320 generators.

The paper is organized as follows: Section II gives a brief summary of parallel processing based computation of transient stability. In Section III an overview of the GPU's hardware and programming tools is presented. The SIMD-based formulation for transient stability simulation is explained in Section IV. In Section V a detailed analysis of the accuracy and efficiency of the GPU-based simulation and discussions are presented. Finally, Section VI gives the conclusion of the study.

II. Parallel Transient Stability Simulation

The need for parallel processing in transient stability simulation of large-scale systems was recognized decades ago due to its overwhelming computational demand. Although there is no agreed classification of these efforts, a primary categorization of these approaches is hardware-based and software-based schemes.

Exploring the literature reveals that from the hardware point of view several types of multiple-instruction multiple-data (MIMD) and SIMD parallel architectures [7] have been employed to accelerate transient stability simulation. Supercomputers [8], multiprocessor networks [9], [10], array-processors [11], [12], and PC-cluster based real-time simulators [13] all have been examined and reported for this application. Although these hardware-based approaches helped to speed-up the simulations, they were stymied by significant drawbacks. The cost of the supercomputers, communication issues and difficulties in the control of multiprocessor-based systems, difficulties in the programming and required algorithms for fitting in array-processors, and limitation on maximum system size of the simulated networks in real-time simulators are some of the noteworthy bottlenecks which limited their widespread application.

Along with advancements in the processor hardware, a substantial amount of research has also been done in search of solution algorithms to efficiently exploit the parallel hardware architecture. The diakoptics method introduced by Kron [14] tears the problem into several subtasks that can be run concurrently on parallel computers. In parallel-in-space [15] and waveform relaxation [16] methods the system is decomposed into smaller subsystems and the computation is allocated to parallel computers. There are also parallel-in-time methods [17], [18] which concurrently solve multiple time steps on parallel processors. While *parallel processing* usually refers to simulation techniques in which closely coupled processors are simultaneously working on the transient stability computation, the concept of *distributed processing* employs a number of loosely coupled and geographically distributed computers to simulate large-scale power systems [19].

The new capability of the modern GPU, as a massively parallel processor, for general purpose high performance computation is the beginning of a new era in computing science. Although at first glance the SIMD architecture of the GPUs and

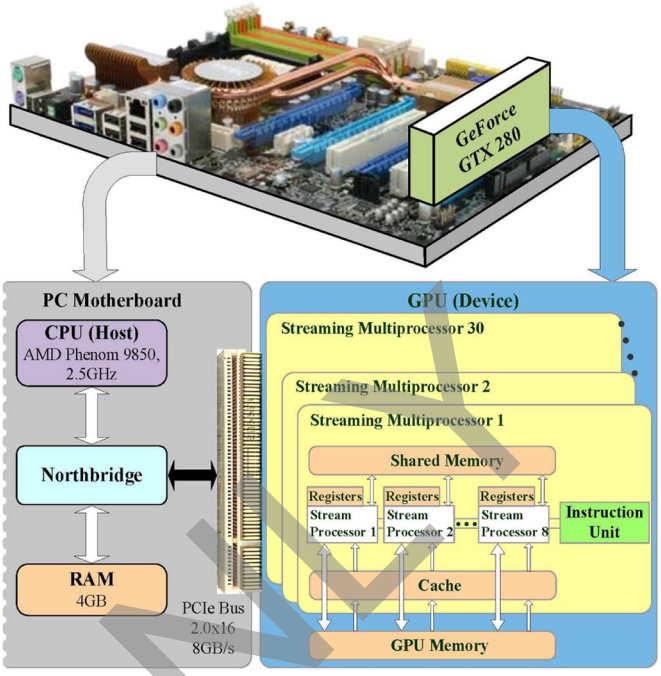

Fig. 1. Hardware architecture of GPU mounted on the PC motherboard.

that of array-processors might look similar, there are significant differences between these two technologies which will be explained in the following section. Moreover, issues related to processing elements' communication overhead, programming complexity, and cost effectiveness have been solved for the GPU [20]. The advantages of modern GPUs, the demand for fast simulation, and the structure of the transient stability computation makes the GPU very suitable for this application.

III. Graphics Processor: An Overview

A. GPU Hardware Architecture

Fig. 1 illustrates the architecture of the GPU [21] plugged into the motherboard of a 2.5-GHz quad-core AMD® Phenom CPU supported by 4 GB of RAM. The GPU and CPU communicate via the PCIe 2.0 × 16 bus that supports up to 8 GB/s transfer rate. The GPU runs its own specified instructions independently but it is controlled by the CPU. The computing element in the GPU is called a *thread*. When a GPU instruction is invoked, blocks of threads (with the maximum size of 512 threads per block) are defined to assign one thread to each data element. All threads in one block run the same instruction on one streaming multiprocessor (SM). The multiprocessor creates, manages, and executes concurrent threads in hardware with zero scheduling overhead. Fast barrier synchronization together with lightweight thread creation and zero-overhead thread scheduling efficiently support fine-grained parallelism [20].

Each SM includes eight stream processor (SP) cores, an instruction unit, and on-chip memory that comes in three types: registers, shared memory, and cache. Threads in each block have access to the shared memory in the SM, as well as to a global memory in the GPU. Unlike a CPU, the architecture of the GPU is developed in such a way that more transistors are devoted to

TABLE I
GEFORCE GTX 280 GPU SPECIFICATIONS

Number of SM's	30
Number of SP's	240
Device memory	1024MB
Memory bandwidth	141.7GB/s
Clock rate	1296MHz
Warp size	32 threads
Active warps/SM	32

data processing rather than data caching and flow control. When a SM is assigned to execute one or more thread blocks, the instruction unit splits and creates groups of parallel threads called *warps*. The threads in one warp are managed and processed concurrently on the eight stream processors. The target GPU specifications in this study are given in Table I.

B. GPU Programming

Depending on the application, the GPU programming paradigms can be divided into two main categories: graphics related programming, and GPGPU programming. In both these categories the GPU follows the SIMD model.

1) Graphical Functionality: From the graphics point of view the GPU has two types of programmable processors: *vertex* and *fragment* processors [22]. In computer graphics 3-D objects are typically represented with triangular meshes. The vertex processors apply a vertex program (also called a vertex shader) to transform each vertex based on its position relative to the camera, and then each set of three vertices is used to compute a triangle from which streams of fragments are generated. A fragment contains all information, such as color and depth, needed to generate a shaded pixel in the final image. The fragment processors apply a fragment program (also called a pixel shader) to each fragment in the stream to compute the final color of each pixel.

2) General Purpose Functionality: Early GPGPU programming directly used the existing graphics API's to express computations in terms of an image. In other words, the computational elements (i.e., vectors and matrices) were mapped onto the graphical elements (i.e., vertices or pixels) which are the components of shading programs. Shading languages used for this purpose include Cg, HLSL, and OpenGL. Although this technique has been successfully used in many research areas, including power system load flow computations [23], it was largely unapproachable by the common programmer. Firstly, because there is a mismatch between traditional programming and the graphics APIs, and secondly, because it is difficult to debug or modify these programs.

3) High Level Programming Languages: Dedicated GPU programming tools that bypassed the graphics functions of the GPU were created for general purpose computing, starting with Brook and Sh, and ultimately leading to commercial tools such as AMD's HAL and NVIDIA's CUDA.

Compute unified device architecture (CUDA) provides a C-like syntax to execute and manage computations on the GPU as a data-parallel computing device. A CUDA program consists of multiple phases that are executed on either the CPU (*host*) or the GPU (*device*). The phases of the program that exhibit little or no data-parallelism are run in the host-code after compiling with the host's standard C compiler, whereas the phases that exhibit fine-grained parallelism are implemented in the device-code in the form of *kernels*, the synonym for GPU functions. Host-code uses a CUDA-specific function-call syntax to invoke the kernel code. Calling a kernel distributes the tasks among the available multiprocessors to be simultaneously run on a large number of parallel threads. The programmer organizes these threads into a grid of thread blocks. Thread creation, scheduling, and resource management are performed in hardware. Each thread of a CUDA program is mapped to a physical thread resident in the GPU, and each running thread block is physically resident on a SM.

Moreover, a library of the basic linear algebra subprograms (BLAS) is provided that allows the integration with C++ code. By using this library, called CUBLAS, portions of a sequential C++ program can be executed in SIMD-form on the GPU, as shown in Fig. 2, while other parts of the code are executed sequentially on the CPU [24]. Wherever a kernel is invoked a grid consisting of several blocks with equal numbers of threads/block is created. Each block within a grid, and each thread within a block are identified by individual indices that make them accessible via the built-in variables in CUDA. Threads determine the task they must do and the data they will access by inspecting their own thread and block IDs. Therefore in an application with highly intensive computations, the onerous computation tasks can be offloaded to the GPU, and performed faster in parallel, whereas mundane tasks such as the flow control of the program, required initial calculations, or the updating and saving of the variables can be done by the CPU. This co-processing configures a hybrid GPU-CPU simulator.

C. Data-Parallel Computing

Here the fundamental difference in the computing model of a GPU and a CPU is discussed, which leads us to a new methodology to implement the transient stability simulation on the data-parallel architecture of GPU. Let us look at a simple example. Suppose we want to evaluate $z = x + y$, where x, y, and z are $n \times 1$ real vectors. On the CPU, a single *for* loop is typically used over all array elements as follows:

$$\text{for}(i = 0; i < n; i++)$$
$$z[i] \leftarrow x[i] + y[i].$$

In this model two levels of computation are active: outside the loop, the loop counter i is increasing and compared with the length of vectors n, whereas inside the loop, the arrays are accessed at a fixed position determined by the loop counter and the actual computation is performed (addition on each data element). The calculations performed on each data element in the vectors are independent of each other, i.e., for a given output position, distinct input memory locations are accessed and there are no data dependencies between elements in the result vector.

If we had a vector processor capable of performing operations on whole vectors of length n or even n CPUs, we would not need the *for* loop at all. This is the core idea of SIMD programming. The computation on the GPU is performed by separating

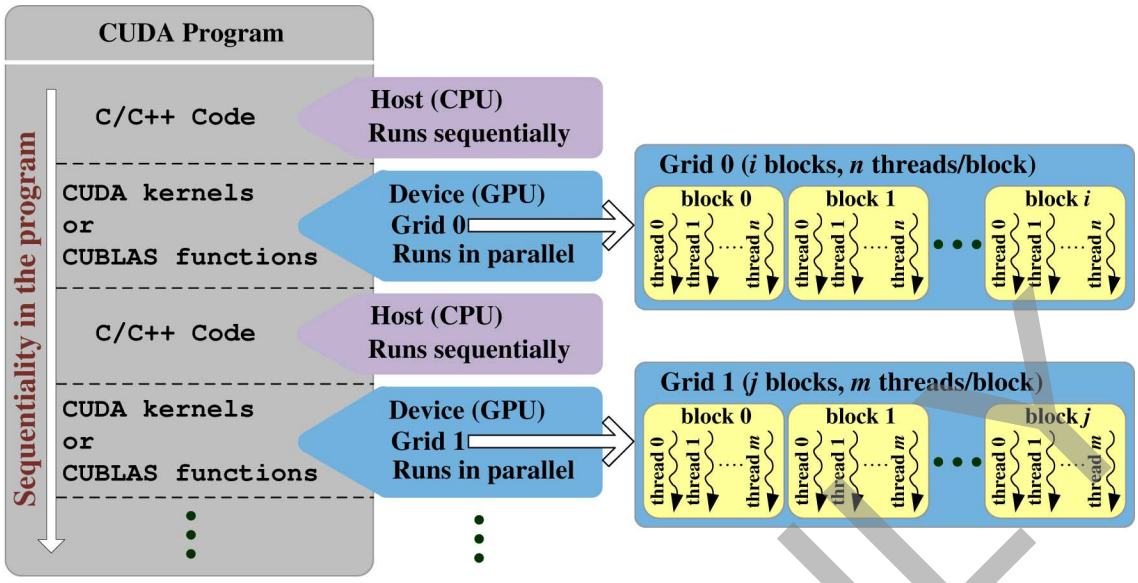

Fig. 2. Cooperation of the host and device to execute a CUDA program, and the hierarchy of computing structure in a GPU.

the outer loop from the inner calculations. The inner loop calculations are extracted into a computational kernel as follows:

$$\text{if}(\text{index} < n)$$
$$z[\text{index}] \leftarrow x[\text{index}] + y[\text{index}]$$

where *index* is the ID of the threads assigned to elements of the vectors. Note that the kernel is no longer a vector expression but a scalar template of the underlying math that forms a single output value from a set of input values. For a single output element, there are no data dependencies with other output elements, and all dependencies to input elements can be described relatively. Whenever a kernel is called, the driver logic hardware schedules each data item into the different multiprocessors (SM)—this is not programmable. Although internally the computation is split up among the available SMs, one cannot control the order in which they are working. One can therefore assume that all work is done in parallel without any data interdependence.

IV. SIMD-Based Transient Stability Simulation on the GPU

A. Transient Stability Simulation

The general form of DAEs which describe the dynamics of a multi-machine power system is given as

$$\dot{\mathbf{x}} = \mathbf{f}(\mathbf{x}, \mathbf{V}, t) \quad (1)$$
$$\mathbf{0} = \mathbf{g}(\mathbf{x}, \mathbf{V}, t) \quad (2)$$
$$\mathbf{x}(t_0) = \mathbf{x_0} \quad (3)$$

where $\mathbf{x}$ is the vector of state variables, $\mathbf{x_0}$ is the initial values of state variables, and $\mathbf{V}$ is the vector of bus voltages. (1) describes the dynamic behavior of the system, while (2) describes the network constraints on (1). In this work the detailed model of synchronous generator including AVR and PSS is used. Each generating unit is modeled using a ninth-order Park's model with an individual dq reference frame fixed on the generator's field winding [25]. In this model the magnetic saturation of the synchronous machines has been ignored, however, the AVR and PSS include windup limits. The network, including transmission lines and loads, is modeled using algebraic equations in a common DQ reference frame. Thus, in a power network with m synchronous generators and n buses, $\mathbf{x}$ is a $9m \times 1$ vector and $\mathbf{V}$ is a $2n \times 1$ vector. The complete system representation is given in Appendix A.

The standard approach to solve these nonlinear and coupled DAEs involves three steps [26]:

Step 1) The continuous-time differential equations are first discretized and converted to discrete-time algebraic equations. Using the implicit Trapezoidal integration method, discretizing (1) results in a new set of nonlinear algebraic equations:

$$0 = \frac{h}{2}[\mathbf{f}(\mathbf{x}, \mathbf{V}, t) + \mathbf{f}(\mathbf{x}, \mathbf{V}, t-h)] - (\mathbf{x}(t) - \mathbf{x}(t-h)) \quad (4)$$

where h is the integration time-step.

Step 2) The existing nonlinear algebraic equations are linearized by the Newton–Raphson method (for the jth iteration) as

$$J(\mathbf{z}_{j-1}) \cdot \Delta\mathbf{z} = -\mathbf{F}(\mathbf{z}_{j-1}) \quad (5)$$

where J is the Jacobian matrix, $\mathbf{z} = [\mathbf{x}, \mathbf{V}]$, $\Delta\mathbf{z} = \mathbf{z}_j - \mathbf{z}_{j-1}$, and $\mathbf{F}$ is the vector of nonlinear function evaluations.

Step 3) The resulting linear algebraic equations are solved to obtain the system state. (5) is solved using the LU factorization followed by the forward-backward substitution method.

Appendix A shows how to solve the network equations.

B. SIMD Formulation for Transient Stability Solution

The generator and the network model shown in Appendix A are suitable for sequential computation on the CPU. To be able to perform these computations on the GPU, the equations must

be expressed in the SIMD format, i.e., instead of using single-element values vectors or matrices of them must be used. This is straightforward for the generator equations and is accomplished by replacing all variables and parameters with the vectors whose elements relate to each generator. For instance, the rotor angle equation for the ith generator, as given in (27), is discretized by (4) as follows:

$$0 = \omega_R^i \frac{h}{2}[\Delta\omega^i(t) + \Delta\omega^i(t-h)] - (\delta^i(t) - \delta^i(t-h)). \quad (6)$$

Allocating vectors whose length are the number of existing generators, this equation can be expressed for SIMD computing as follows:

$$0 = \omega_R^v \frac{h}{2}[\Delta\omega^v(t) + \Delta\omega^v(t-h)] - (\delta^v(t) - \delta^v(t-h)) \quad (7)$$

where the superscript v indicates the vector-format variable, and

$$\delta^v = [\delta^1, \delta^2, \ldots, \delta^i \ldots, \delta^n]^t \quad (8)$$

and

$$\Delta\omega^v = [\Delta\omega^1, \Delta\omega^2, \ldots, \Delta\omega^i \ldots, \Delta\omega^n]^t. \quad (9)$$

Similarly other differential equations described in Appendix A, i.e., (27)–(29), can be implemented in the SIMD format.

For the network side, expressing the computations in SIMD format is more complicated and requires a revision of the equations. For the n generators in the system, the objective in the network side equations is to compute two vectors whose elements are the I_D and I_Q of each generator given by (43) and (44). In the sequential computing model, as shown by (45)–(47), I_D and I_Q are constructed based on the parameters S_1 to S_4, and A_1 to A_8. Therefore, these parameters must first be computed in the SIMD format to get the I_D and I_Q vectors. We start with the allocation of six vectors on the GPU as follows:

$$U_k^v = \left[u_k^1, u_k^2, \ldots, u_k^i, \ldots, u_k^n\right]^t; k = 1 \cdots 6 \quad (10)$$

where the superscript v indicates the vector-format variable. u_k^i for the ith generator is computed based on (47). Having these vectors on the GPU, a kernel was prepared to lay these vectors on the main diagonal of a square matrix. Thus, six matrices are built on the GPU whose diagonal elements are the U_k^v and whose off-diagonal elements are zero, as

$$U_k^m = \mathrm{diag}(U_k^v); k = 1 \cdots 6 \quad (11)$$

where superscript m indicates the matrix-format variable. From the reduced admittance matrix shown in (37) two sets of vectors and matrices are then extracted. The vectors are the real and imaginary parts of the diagonal elements of the $Y_R = G_R + jB_R$:

$$G^v = [G_{11}, G_{22}, \ldots, G_{ii}, \ldots, G_{nn}]^t \quad (12)$$

$$B^v = [B_{11}, B_{22}, \ldots, B_{ii}, \ldots, B_{nn}]^t. \quad (13)$$

The two matrices contain the off-diagonal elements of G_R and B_R:

$$G_{\mathrm{off-diag}}^m = G_R - \mathrm{diag}(G^v) \quad (14)$$

$$B_{\mathrm{off-diag}}^m = B_R - \mathrm{diag}(B^v). \quad (15)$$

Note that vectors G^v and B^v, and matrices $G_{\mathrm{off-diag}}^m$ and $B_{\mathrm{off-diag}}^m$ can be built and saved offline for any number of contingencies and transferred to the GPU during the initialization step. Once all of the above vectors and matrices are allocated on the GPU, the SIMD computation for S_1 to S_4 can be expressed as follows:

$$S_1^v = \left(U_1^m \cdot G_{\mathrm{off-diag}}^m - U_3^m \cdot B_{\mathrm{off-diag}}^m\right) \cdot I_D^v \quad (16)$$

$$S_2^v = (U_2^m \cdot G_R - U_4^m \cdot B_R) \cdot I_Q^v \quad (17)$$

$$S_3^v = (U_3^m \cdot G_R - U_1^m \cdot B_R) \cdot I_D^v \quad (18)$$

$$S_4^v = \left(U_4^m \cdot G_{\mathrm{off-diag}}^m + U_2^m \cdot B_{\mathrm{off-diag}}^m\right) \cdot I_Q^v. \quad (19)$$

The SIMD computations of A_5 to A_8 is given as

$$A_5^v = I^v + B^v * U_3^v - G^v * U_1^v \quad (20)$$

$$A_6^v = I^v - G^v * U_4^v - B^v * U_2^v \quad (21)$$

$$A_7^v = G_R \cdot U_5^v - B_R \cdot U_6^v \quad (22)$$

$$A_8^v = G_R \cdot U_6^v + B_R \cdot U_5^v \quad (23)$$

where $I^v = [1, 1, \ldots, 1]_{n\times 1}^t$, and the star operation '$*$' refers to a kernel that multiplies two equal-length vectors element by element. Based on (16)–(23) the SIMD format of equations I_D and I_Q, and then E_D and E_Q is achieved.

C. Hybrid GPU-CPU Simulation

The standard method of transient stability explained in the previous section was implemented in a hybrid GPU-CPU configuration. Benchmarking revealed that a majority of execution time in the simulation was spent for Step 2) and Step 3), i.e., the nonlinear iterative solution using Newton–Raphson, and the linear algebraic equation solution. Therefore these two steps of the simulation were off-loaded to the GPU to be processed in parallel, while the remaining tasks such as discretizing, updating, and computation of intermediate variables were executed sequentially on the CPU. The entire simulation code was developed in C++ integrated with CUDA.

In this programming model discretization of the differential equations, and building of the Jacobian matrix are done on the CPU, whereas the network algebraic equations and the Jacobian matrix solutions are performed on the GPU. At each time-step, the Jacobian matrix is transferred to the GPU. It should be noted that CUDA stores matrices on the GPU in a column-major format. Therefore after the Jacobian matrix is constructed on CPU, it needs to be transposed before or after being transferred to the GPU. The other option is to construct the Jacobian matrix in the column-major format from the beginning on the CPU obviating the need for an extra transpose operation and thus saving computation time. Here the latter option has been adopted in this programming model.

With the Jacobian matrix in the proper format on the GPU, the SIMD-format of the LU factorization method was used on the GPU to handle large matrix decomposition. This factorization scheme was implemented using a blocked algorithm, employing bulk matrices, specifically suited for data-parallelism [27]. After transforming the Jacobian matrix into its upper and

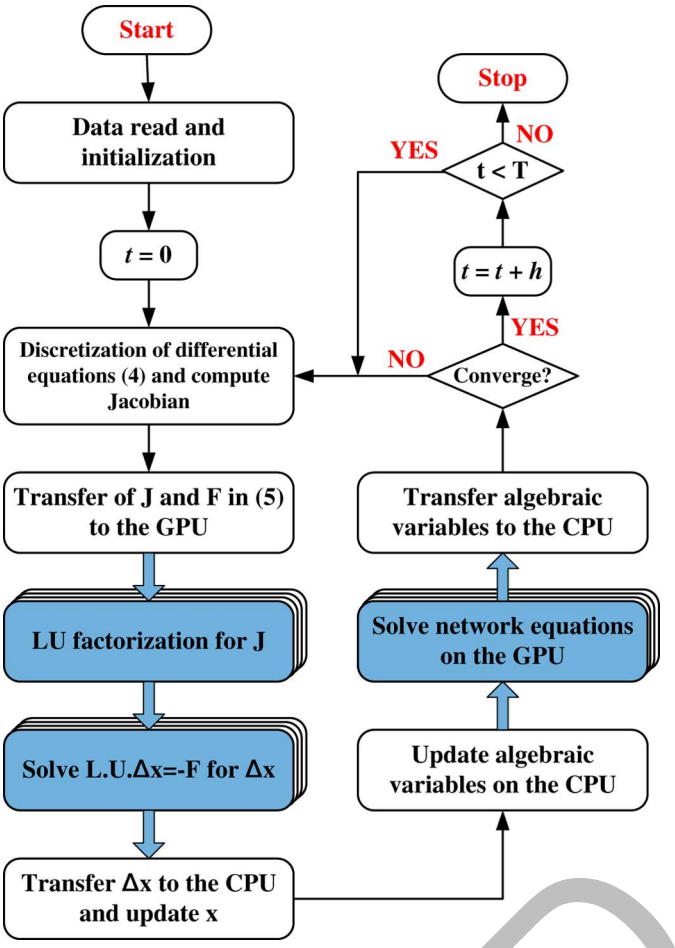

Fig. 3. Flowchart for the hybrid GPU-CPU transient stability simulation. The colored boxes refer to the GPU operations, and white boxes refer to the CPU operations.

lower triangular matrices, the BLAS2 function $cublasStrsv()$ of the CUBLAS library is used to solve the equation $L{\cdot}U{\cdot}\Delta\mathbf{x} = -\mathbf{F}(\mathbf{x}_{j-1})$ for $\Delta\mathbf{x}$. The results are transferred back to the CPU to update the state variables for the next iteration's calculations. Then the computations are continued on the GPU for the network side solution as described by (10)–(23). Once the iteration process converges, the time-step is advanced. These steps are illustrated in the flowchart given in Fig. 3.

D. GPU-Only Simulation

In the second programming model the transient stability simulation was carried out as a GPU-only computation. In this model the CPU initializes the GPU with the system data, and then all the three steps of the simulation are done on the GPU, while the CPU monitors and controls the flow of the simulation. In this programming model the Jacobian matrix is completely constructed on the GPU. Constructing the Jacobian matrix for a multi-machine power system is well suited for exploiting data-parallel programming. For example, in the row-major-saved Jacobian matrix, the first column is the derivative of all nonlinear functions with respect to one of the variables. Although these functions have different formulations, they reoccur for all generators in the system. Therefore the derivatives of all identical functions (that have the same formulation but different data-input) can be evaluated in a data-parallel model. After the sub-matrices of the Jacobian are calculated they are combined together to form the full Jacobian matrix on the GPU, and the simulation continues the same as hybrid GPU-CPU model. At the end of simulation, the value of the required variables is transferred to the CPU memory to be saved or plotted. The flowchart in Fig. 4 illustrates this process.

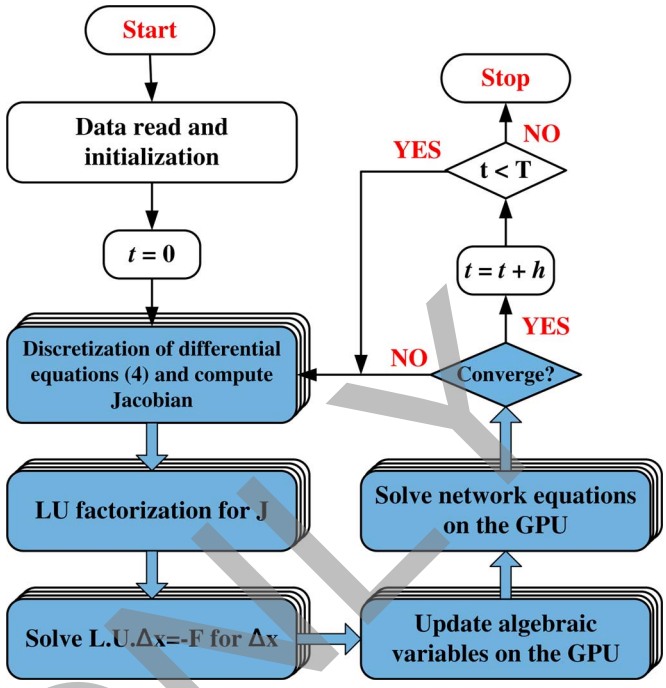

Fig. 4. Flowchart for the GPU-only transient stability simulation. The colored boxes refer to the GPU operations, and white boxes refer to the CPU operations.

V. Experimental Results

In this section we demonstrate results to verify the accuracy and efficiency of the SIMD-based programming models described in the previous section for the transient stability simulation of large-scale power systems on the GPU. As mentioned in Section II, the target GPU in this work is connected to the motherboard of a quad-core CPU. Although the CPU has four cores, to precisely control the execution of the CPU and to have a clear comparison, thread programming was used to force the CPU to run the C++ code on only one of the four cores. Otherwise, the operating system decides each core's execution at each instant which leads to a vague evaluation.

A. Simulation Accuracy Evaluation

The accuracy of the programming models was validated using PTI's PSS/E software program. The case study used in this section is the IEEE 39-bus New England test system whose one-line diagram is shown in Fig. 5. All generator models are detailed and equipped with AVR and PSS described in Appendix A. The complete system can be described by 90 nonlinear differential equations and 20 algebraic equations. Several fault locations have been tested and the results were compared with those of PSS/E. In all cases results from the proposed programming models match the PSS/E results very well. A sampling of these

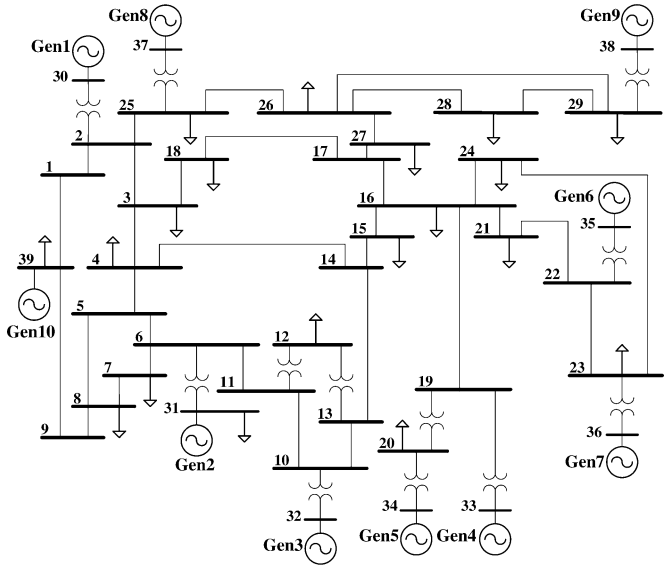

Fig. 5. One-line diagram for the IEEE 39-bus power system (*Scale 1* system).

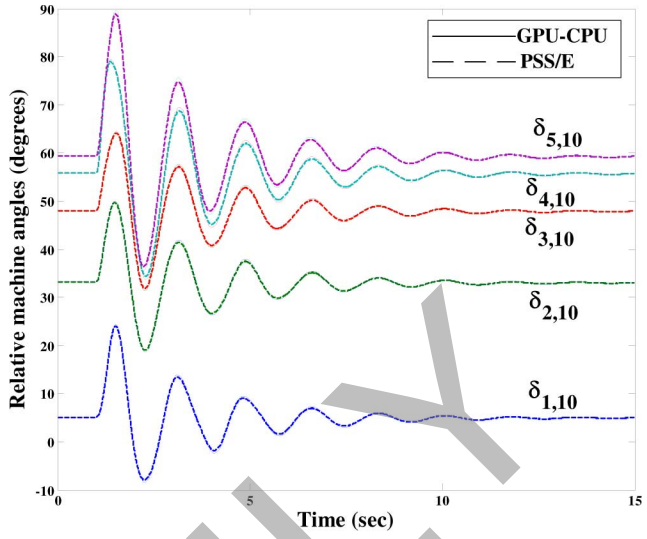

Fig. 6. Comparison of relative machine angles collected from hybrid simulator and PSS/E simulation for IEEE 39-bus test system: $\delta_{i,10} = \delta_i - \delta_{10}$; $i = 1 \ldots 5$, for a three-phase fault at Bus 21.

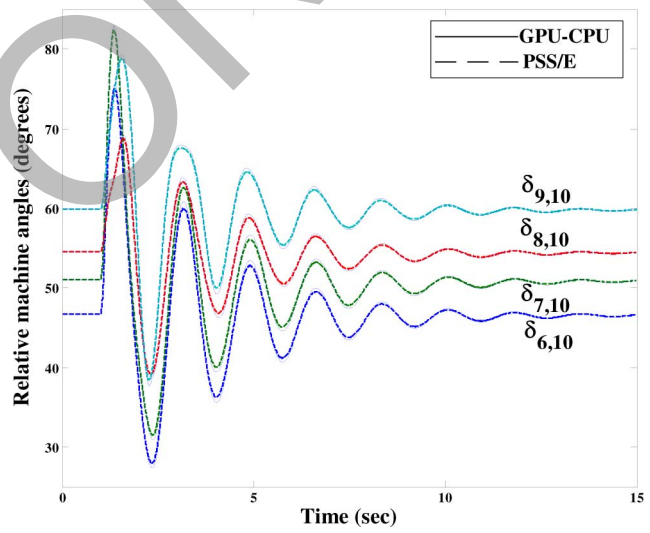

Fig. 7. Comparison of relative machine angles collected from hybrid simulator and PSS/E simulation for IEEE 39-bus test system: $\delta_{i,10} = \delta_i - \delta_{10}$; $i = 6 \ldots 9$, for a three-phase fault at Bus 21.

results obtained from the hybrid GPU-CPU simulation is presented here. A three-phase fault happens at Bus 21, at $\mathrm{t} = 1$ s and it is cleared after 100 ms. $Gen10$ is the reference generator and the relative machine angles are shown in Figs. 6 and 7. For comparison PSS/E results are superimposed in these two figures. As can be seen the transient stability code is completely stable during the steady-state of the system, i.e., $\mathrm{t} < 1$ s. During the transient state and also after the fault is cleared, the program results closely follow the results from PSS/E. The maximum discrepancy between generator angles from GPU-CPU co-processing and the PSS/E simulation was found to be 1.46%, based on (24):

$$\varepsilon_\delta = \frac{\max|\delta_{\mathrm{PSS/E}} - \delta_{\mathrm{GPU-CPU}}|}{\delta_{\mathrm{PSS/E}}} \quad (24)$$

where $\delta_{\mathrm{PSS/E}}$ and $\delta_{\mathrm{GPU-CPU}}$ are the relative machine angles from PSS/E and GPU-CPU co-processing simulation, respectively.

B. Computational Efficiency Evaluation

To investigate the efficiency of the proposed SIMD-based programming models for the transient stability simulation, we show comparative results in this section. Several test systems of increasing sizes have been used for this evaluation whose specifications are listed in Table II. The *Scale 1* system is the IEEE's New England test system, illustrated in Fig. 5 and verified in the previous section. The *Scale 1* system was duplicated several times to create systems of larger scales. Thus, we obtained test systems of 78, 156, 312, 624, and 1248 buses. In these systems a flat start was used, i.e., voltage and angle of all buses set to 1.0 $\angle 0°$ p.u., and they were modeled in the PSS/E software to find the load flow results. These results were then fed into the prepared simulation codes.

Three separate simulation codes were prepared: the first code is purely in C++ to be run sequentially on the CPU (CPU-only), the second is C++ integrated with CUDA to be completely run on the GPU (GPU-only), and the third is also the integration of C++ and CUDA; however, it uses GPU as the coprocessor (GPU-CPU). The difference between the second and third programming models was explained in Section III. The execution time of these three codes was compared for the test systems. In Table II the columns indicated by CPU-only, GPU-only, and GPU-CPU list the computation time of each programming model to simulate a duration of 1580 ms with a time-step of 10 ms for all systems. The CPU execution time of PTI's PSS/E software program is also included in Table II for reference. Fig. 8 plots the computation times with respect to the system size.

The application of GPU (in both GPU-only and GPU-CPU models) is truly advantageous for parallel computing on a large set of input data. For small size of data, the communication overhead and memory latency in the GPU are not insignificant compared to the computation time. As such, we did not expect better performance for *Scale 1* and *Scale 2* systems. When the size of

TABLE II
SYSTEM SIZE VERSUS COMPUTATION TIME FOR VARIOUS CONFIGURATIONS FOR A SIMULATION DURATION OF 1580 MS

System scale	Generators	Buses	State variables	Algebraic variables	CPU-only	GPU-only	GPU-CPU	PSS/E Program
1	10	39	90	20	0.9s	5.5s	2.8s	0.35s
2	20	78	180	40	6.4s	7.7s	5.2s	0.40s
4	40	156	360	80	49.8s	12.3s	10.5s	0.43s
8	80	312	720	160	7.2min	21.5s	21.1s	0.46s
16	160	624	1440	320	1hr	41.0s	44.8s	0.55s
32	320	1248	2880	640	10hr	1min15.2s	1min44.4s	0.83s

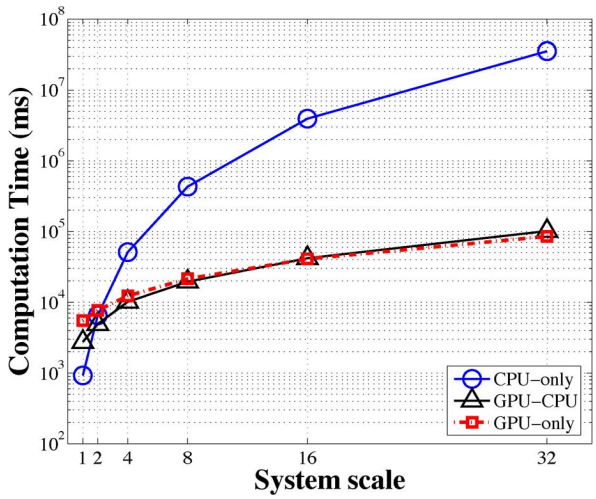

Fig. 8. Computation time variation with respect to system size.

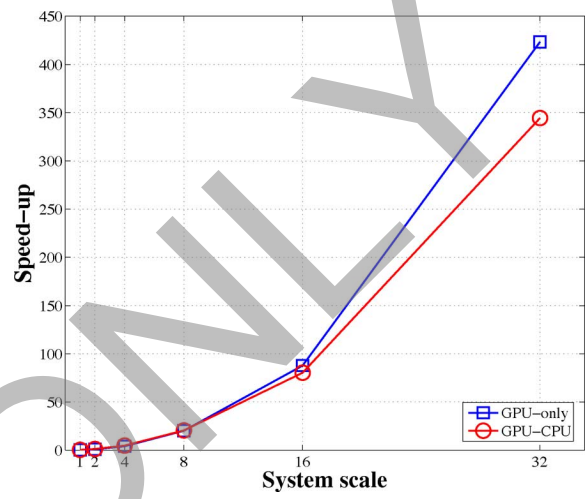

Fig. 9. Speed-up of GPU-based processing.

TABLE III
SPEED-UP COMPARISON

System scale	$\beta_{GPU-only}$	$\beta_{GPU-CPU}$
1	×0.16	×0.3
2	×0.83	×1.2
4	×4.05	×4.7
8	×20.1	×20.5
16	×87.8	×80.3
32	×423.5	×344.8

system increases, however, the latency is dwarfed by the computation time, and involving the GPU into the simulations results in a significant acceleration. For example, for *Scale 32*, the GPU-CPU takes 1 min 44.4 s for simulation, whereas the CPU-only needs 10 h. Table III lists the speed-up factors, defined by (25) and (26), for the two GPU-based simulations:

$$\beta_{\text{GPU-only}} = \frac{\text{CPU\_only processing time}}{\text{GPU\_only processing time}} \quad (25)$$

$$\beta_{\text{GPU-CPU}} = \frac{\text{CPU\_only processing time}}{\text{GPU\_CPU processing time.}} \quad (26)$$

The speed-up factors for the two simulations are plotted in Fig. 9. As can be seen for the *Scale 32* system GPU-CPU co-processing is more than 340 times faster than CPU-only processing.

C. Discussion

The tabulated results and graphs reveal that for small systems the hybrid GPU-CPU programming model is faster than the GPU-only model, whereas for large-scale systems GPU-only model is faster. This result is consistent with the performance of a single GPU in other reported applications. However there are applications where a GPU may need to communicate with other entities or use services which may not be accessible to a GPU, for example in a multiple GPU application. In such cases and with the currently available GPU technologies, the hybrid GPU-CPU programming model is not only useful but also unavoidable. Similar to all programmable processors, efficient GPU programming needs a good understanding of its hardware architecture. Managing the number of required active threads for each kernel call, and the number of device-host interactions are essential to make a timesaving program.

Another useful observation found from the achieved results is the scalability of the proposed hybrid simulator. A system whose performance improves after adding a specific hardware component, proportionally to the capacity added, is said to be a scalable system. In a single GPU expanding the size of data-under-process asks for the co-operation of more SPs which translates to adding more computing resources. In our experiments the size of test systems and the hybrid simulators' elapsed computation time change approximately at the same rate. In the CPU-only simulation cases, however, the computation time increases at a rate that is approximately the cube of the system size increment rate.

The last column in Table II indicates PSS/E's computation time for the test systems. It is important to put the lower execution times of PSS/E into proper perspective so as not to misjudge the performance of the GPU. PSS/E is a mature software program developed over several decades which incorporates specialized techniques to optimize computational

efficiency for large-scale system simulation. The intent of this work is to illustrate the advantage of a GPU-based simulation over a CPU-based one utilizing the same computational methods; the efficiency comparison with PSS/E is provided only for the sake of completeness, however, PSS/E utilizes very different numerical methods and algorithms. In terms of the numerical methods, the key factors that contribute to PSS/E's speed are as follows:

1) PSS/E uses the modified Euler integration algorithm which is an explicit integration technique [31, Ch. 26]. This is a simple and fast method, however, it can suffer from numerical instability. For a given system configuration, depending on the time-step value the simulation may either converge or diverge. If it did not converge, the time-step must be decreased and program must be rerun [32]. In the proposed GPU-based simulation, the implicit trapezoidal integration method was used. This method avoids numerical instability, however, it requires an iterative solution of the resulting nonlinear algebraic equations at each time-step.
2) The outputs of an explicit integrator are not instantaneous functions of their inputs, and a straightforward non-iterative procedure would be enough to find the derivatives of states, as it happens in PSS/E. However the proposed GPU-based simulator uses the Newton–Raphson method which consumes a large portion of the simulation time, albeit it gives accurate convergence. At each time-step the Jacobian matrix is calculated, LU factorized, and then solved using forward-backward substitutions. It is also possible to use dishonest types of the Newton-Raphson method to avoid refactorization of the Jacobian matrix in each iteration for improved computational efficiency.
3) PSS/E takes advantage of the sparsity of the system matrices. This is an important factor in its fast and successful management of computer storage [31. Ch. 5]. The purpose of this paper was to show how the same implementation of a transient stability algorithm on a GPU is advantageous with respect to a CPU implementation. Therefore sparsity was not used in any of the proposed programming models.

VI. Conclusion

Transient stability simulation of large-scale power systems is computationally very demanding. This paper investigated the potential of using a GPU to accelerate this simulation by exploiting its SIMD architecture. Two SIMD-based programming models to implement the standard method of the transient stability simulation were proposed and implemented on a single GPU. The simulation codes are quite flexible and extensible; they are written entirely in C++ integrated with GPU-specific functions. The accuracy of the proposed methods were validated by the PSS/E software. The efficiency was evaluated for several large test cases. Based on the results obtained, it can be concluded that:

- Using a GPU for transient stability simulations is highly advantageous when the system size is large. As such, for simulating realistic-size power systems, the application of GPU looks promising.
- For small-scale systems the hybrid GPU-CPU simulation was faster than the GPU-only simulation, while for large-scale systems the GPU-only model was faster.

It is anticipated that the GPU can play an important role in realizing the ultimate goal of online or real-time dynamic security assessment in the energy control center. The transient stability simulation algorithm in this paper employed the implicit Trapezoidal integration method along with the iterative Newton–Raphson procedure. However, as classified in [26], there exist many other possibilities for solving the nonlinear DAEs applicable to the transient stability problem, such as:

- explicit or implicit integration methods;
- iterative and non-iterative solvers;
- simultaneous or partitioned solution approaches;
- application of sparse methods to solve linear algebraic equations [33].

In addition, it is also possible to implement improved parallel methods for large-scale systems such as the parallel-in-space or relaxation-based methods [4] for higher speed-ups. It is predicted that if a method accelerates the CPU-based simulation, it would also accelerate the GPU-based model, if that approach was efficiently implemented on the GPU. Research is currently underway to use multiple GPUs to handle larger system sizes, to provide a comprehensive library of power system components, and to simulate multiple contingencies.

Appendix A
System Representation for the Transient Stability Analysis

The detailed model of a synchronous generator used in this paper is given here.

1) Equations of motion (swing equations or rotor mechanical equations):

$$\begin{aligned}\dot{\delta}(t) &= \omega_R \cdot \Delta\omega(t)\\ \dot{\Delta\omega}(t) &= \frac{1}{2H}[T_e(t) + T_m - D \cdot \Delta\omega(t)].\end{aligned} \tag{27}$$

2) Rotor electrical circuit equations: This model includes two windings on the d axis (one excitation field and one damper) and two damper windings on the q axis:

$$\begin{aligned}\dot{\psi}_{fd}(t) &= \omega_R \cdot [e_{fd}(t) - R_{fd} i_{fd}(t)]\\ \dot{\psi}_{1d}(t) &= -\omega_R \cdot R_{1d} i_{1d}(t)\\ \dot{\psi}_{1q}(t) &= -\omega_R \cdot R_{1q} i_{1q}(t)\\ \dot{\psi}_{2q}(t) &= -\omega_R \cdot R_{2q} i_{2q}(t).\end{aligned} \tag{28}$$

3) Excitation system: Fig. 10 shows a bus-fed thyristor excitation system, classified as Type ST1A in the IEEE Standard [28]. This system includes an AVR and PSS:

$$\begin{aligned}\dot{v}_1(t) &= \frac{1}{T_R}[v_t(t) - v_1(t)]\\ \dot{v}_2(t) &= K_{\text{stab}} \cdot \dot{\Delta\omega}(t) - \frac{1}{T_w} v_2(t)\\ \dot{v}_3(t) &= \frac{1}{T_2}[T_1 \dot{v}_2(t) + v_2(t) - v_3(t)].\end{aligned} \tag{29}$$

4) Stator voltage equations:

$$e_d(t) = -R_a i_d(t) + L''_q i_q(t) - E''_d(t)$$
$$e_q(t) = -R_a i_d(t) - L''_d i_d(t) - E''_q(t) \quad (30)$$

where

$$E''_d \equiv L_{aq}\left[\frac{\psi_{q1}}{L_{q1}} + \frac{\psi_{q2}}{L_{q2}}\right]$$
$$E''_q \equiv L_{ad}\left[\frac{\psi_{fd}}{L_{fd}} + \frac{\psi_{d1}}{L_{d1}}\right]. \quad (31)$$

5) Electrical torque:

$$T_e = -(\psi_{ad} i_q - \psi_{aq} i_d) \quad (32)$$

where

$$\psi_{ad} = L''_{ad}\left[-i_d + \frac{\psi_{fd}}{L_{fd}} + \frac{\psi_{d1}}{L_{d1}}\right]$$
$$\psi_{aq} = L''_{aq}\left[-i_q + \frac{\psi_{q1}}{L_{q1}} + \frac{\psi_{q2}}{L_{q2}}\right] \quad (33)$$

where $\omega_R, H, D, R_{fd}, R_{1d}, R_{1q}, R_{2q}, R_a, L_{fd}, L_{d1}, L_{q1}, L_{q2}, L''_d, L''_q, L_{ad}, L_{aq}, L''_{ad}, L''_{aq}, T_R, T_w, T_1, T_2$, and K_{stab} are constant system parameters whose definition can be found in [29].

According to this formulation the vector of state variables in (1) and (2) for the synchronous generator is given as

$$x = [\delta \Delta\omega\, \psi_{fd}\, \psi_{1d}\, \psi_{1q}\, \psi_{2q}\, v_1\, v_2\, v_3]^t. \quad (34)$$

6) Network equations [25]: Stator equations are solved together with the network equations. All nodes except the generator nodes are eliminated and the admittance matrix for the reduced network is obtained. The procedure for network reduction is shown below. The nodal equation for the network can be written as

$$I = YV \quad (35)$$

where $I_{(n+r)\times 1} = [I_{n\times 1} \vdots 0_{r\times 1}]^t$. n denotes the number of generator nodes and r denotes the number of remaining nodes. The matrices Y and V are partitioned as

$$\begin{bmatrix} I_n \\ 0_r \end{bmatrix} = \begin{bmatrix} Y_{nn}\, Y_{nr} \\ Y_{rn}\, Y_{rr} \end{bmatrix} \begin{bmatrix} V_n \\ V_r \end{bmatrix}. \quad (36)$$

Expanding (36) to find I_n based on V_n, we obtain

$$I_n = (Y_{nn} - Y_{nr} Y_{rr}^{-1} Y_{rn}) V_n. \quad (37)$$

Thus

$$I_n = Y_R \cdot E_n \quad (38)$$

where the matrix $Y_R = Y_{nn} - Y_{nr} Y_{rr}^{-1} Y_{rn}$ is the desired reduced matrix. It has the dimensions $n \times n$ where n is the number of generators. This matrix must be computed for steady-state, during the transient state, and after the clearing of the transient phenomena.

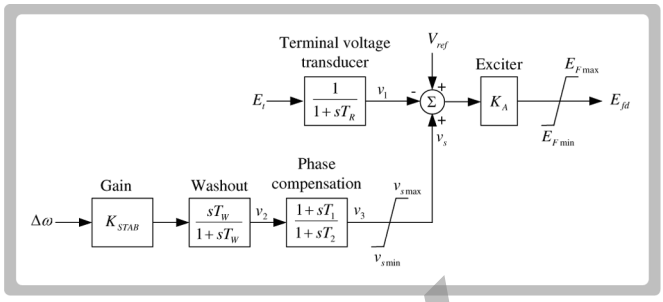

Fig. 10. Excitation system with AVR and PSS.

As the components in the network common DQ frame are of interest the above complex matrix equation can be written as two separate real matrix equations:

$$I_{Dn} = G_R \cdot E_{Dn} - B_R \cdot E_{Qn} \quad (39)$$
$$I_{Qn} = G_R \cdot E_{Qn} + B_R \cdot E_{Dn}. \quad (40)$$

To relate the components of voltages and currents expressed in the d, q reference of each individual machine to the common reference frame (DQ), the following reference frame transformation is used:

$$i_{dq} = I_{DQ} \cdot \exp(-j\delta) \quad (41)$$
$$e_{dq} = E_{DQ} \cdot \exp(-j\delta) \quad (42)$$

where δ is the rotor angle of the synchronous machine.

The kth iteration current components in the common reference frame can be expressed as [30]

$$I_D(k) = \frac{1}{A_5}(S_1 + S_2 + A_7) \quad (43)$$
$$I_Q(k) = \frac{1}{A_6}(S_3 + S_4 + A_8) \quad (44)$$

where the S and A parameters are defined as follows:

$$S_1 = \sum_{\substack{j=1 \\ j\neq k}}^{n} I_D(j) \cdot A_1,\ S_2 = \sum_{j=1}^{n} I_Q(j) \cdot A_2$$
$$S_3 = \sum_{j=1}^{n} I_D(j) \cdot A_3,\ S_4 = \sum_{\substack{j=1 \\ j\neq k}}^{n} I_Q(j) \cdot A_4 \quad (45)$$

and

$$A_1 = G_R(k,j) \cdot u_1(j) - B_R(k,j) \cdot u_3(j)$$
$$A_2 = G_R(k,j) \cdot u_2(j) - B_R(k,j) \cdot u_4(j)$$
$$A_3 = G_R(k,j) \cdot u_3(j) - B_R(k,j) \cdot u_1(j)$$
$$A_4 = G_R(k,j) \cdot u_4(j) + B_R(k,j) \cdot u_2(j)$$
$$A_5 = 1 + B_R(k,j) \cdot u_3(j) - G_R(k,j) \cdot u_1(j)$$
$$A_6 = 1 - G_R(k,j) \cdot u_4(j) - B_R(k,j) \cdot u_2(j)$$
$$A_7 = \sum_{j=1}^{n} G_R(k,j) \cdot u_5(j) - B_R(k,j) \cdot u_6(j)$$
$$A_8 = \sum_{j=1}^{n} G_R(k,j) \cdot u_6(j) + B_R(k,j) \cdot u_5(j) \quad (46)$$

where

$$u_1 = \frac{\omega}{\omega_0}(L''_{ad} - L''_{aq})\cos(\delta)\cdot\sin(\delta) - R_a$$
$$u_2 = \frac{\omega}{\omega_0}(L''_{ad}\sin^2(\delta) + L''_{aq}\cos^2(\delta) + L_l)$$
$$u_3 = -\frac{\omega}{\omega_0}(L''_{ad}\cos^2(\delta) + L''_{aq}\sin^2(\delta) + L_l)$$
$$u_4 = -\frac{\omega}{\omega_0}(L''_{ad} - L''_{aq})\cos(\delta)\cdot\sin(\delta) - R_a$$
$$u_5 = -\cos(\delta)\cdot E''_d - \sin(\delta)\cdot E''_q$$
$$u_6 = \cos(\delta)\cdot E''_q - \sin(\delta)\cdot E''_d. \quad (47)$$

Having the above parameters and thus I_D and I_Q, the components of the bus voltages can be expressed as

$$E_D(k) = I_D(k)\cdot u_1(k) + I_Q(k)\cdot u_2(k) + u_5(k) \quad (48)$$
$$E_Q(k) = I_D(k)\cdot u_3(k) + I_Q(k)\cdot u_4(k) + u_6(k). \quad (49)$$

REFERENCES

[1] R. Schainker, P. Miller, W. Dubbelday, P. Hirsch, and G. Zhang, "Real-time dynamic security assessment," *IEEE Power and Energy Mag.*, vol. 4, no. 2, pp. 51–58, Mar./Apr. 2006.

[2] J. S. Chai and A. Bose, "Bottlenecks in parallel algorithms for power system stability analysis," *IEEE Trans. Power Syst.*, vol. 8, no. 3, pp. 9–15, Aug. 1993.

[3] IEEE Task Force on Computer and Analytical Methods, "Parallel processing in power systems computation," *IEEE Trans. Power Syst.*, vol. 7, no. 2, pp. 629–638, May 1992.

[4] V. Jalili-Marandi and V. Dinavahi, "Instantaneous relaxation based real-time transient stability simulation," *IEEE Trans. Power Syst.*, vol. 24, no. 3, pp. 1327–1336, Aug. 2009.

[5] D. Blythe, "Rise of the graphics processor," *Proc. IEEE*, vol. 96, no. 5, pp. 761–778, May 2008.

[6] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proc. IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[7] M. J. Flynn, "Very high speed computing systems," *Proc. IEEE*, vol. 54, no. 12, pp. 1901–1909, Dec. 1966.

[8] H. H. Happ, C. Pottle, and K. A. Wirgau, "An assessment of computer technology for large scale power system simulation," in *Proc. Power Industry Computer Applications Conf.*, May 1979, pp. 316–324.

[9] F. M. Brasch, J. E. Van Ness, and S. C. Kang, "Simulation of a multiprocessor network for power system problems," *IEEE Trans. Power App. Syst.*, vol. PAS-101, no. 2, pp. 295–301, Feb. 1982.

[10] S. Y. Lee, H. D. Chiang, K. G. Lee, and B. Y. Ku, "Parallel power system transient stability analysis on hypercube multiprocessors," *IEEE Trans. Power Syst.*, vol. 6, no. 3, pp. 1337–1343, Aug. 1991.

[11] H. Taoka, S. Abe, and S. Takeda, "Fast transient stability solution using an array processor," *IEEE Trans. Power App. Syst.*, vol. PAS-102, no. 12, pp. 3835–3841, Dec. 1983.

[12] M. Takatoo, S. Abe, T. Bando, K. Hirasawa, M. Goto, T. Kato, and T. Kanke, "Floating vector processor for power system simulation," *IEEE Trans. Power App. Syst.*, vol. PAS-104, no. 12, pp. 3361–3366, Dec. 1985.

[13] P. Forsyth, R. Kuffel, R. Wierckx, J. Choo, Y. Yoon, and T. Kim, "Comparison of transient stability analysis and large-scale real time digital simulation," in *Proc. IEEE Power Tech.*, Sep. 2001, vol. 4, pp. 1–7.

[14] G. Kron, *Diakoptics The Piecewise Solution of Large-Scale Systems*. London, U.K.: MacDonald, 1963.

[15] M. La Scala, M. Brucoli, F. Torelli, and M. Trovato, "A Gauss-Jacobi-Block-Newton method for parallel transient stability analysis," *IEEE Trans. Power Syst.*, vol. 5, no. 4, pp. 1168–1177, Nov. 1990.

[16] M. L. Crow and M. Ilic, "The parallel implementation of the waveform relaxation method for transient stability simulations," *IEEE Trans. Power Syst.*, vol. 5, no. 3, pp. 922–932, Aug. 1990.

[17] F. L. Alvarado, "Parallel solution of transient problems by trapezoidal integration," *IEEE Trans. Power App. Syst.*, vol. PAS-98, no. 3, pp. 1080–1090, May/Jun. 1979.

[18] M. La Scala, R. Sbrizzai, and F. Torelli, "A pipelined-in-time parallel algorithm for transient stability analysis," *IEEE Trans. Power Syst.*, vol. 6, no. 2, pp. 715–722, May 1991.

[19] M. Shahidehpour and Y. Wang, *Communication and Control in Electric Power Systems: Applications of Parallel and Distributed Processing*. New York: Wiley, 2003.

[20] NVIDIA, NVIDIA CUDA: Programming Guide, Jun. 2008.

[21] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, Mar.-Apr. 2008.

[22] M. Pharr, *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Reading, MA: Addison-Wesley Professional, 2005.

[23] A. Gopal, D. Niebur, and S. Venkatasubramanian, "DC power flow based contingency analysis using graphics processing units," in *Proc. IEEE Power Tech.*, Jul. 2007, pp. 731–736.

[24] NVIDIA, CUDA CUBLAS Library, Mar. 2008.

[25] P. M. Anderson and A. A. Fouad, *Power System Control and Stability*. Ames, IA: Iowa State Univ. Press, 1977.

[26] B. Stott, "Power system dynamic response calculations," *Proc. IEEE*, vol. 67, no. 2, pp. 219–241, Jul. 1979.

[27] M. Garland, S. Le Grand, J. Nicolls, J. Anderson, J. Hardwick, S. Morton, E. Phillips, Y. Zhang, and V. Volkov, "Parallel computing experiences with CUDA," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, Jul.-Aug. 2008.

[28] *IEEE Recommended Practice for Excitation System Models for Power System Stability Studies*, IEEE Std 421.5-2005, Apr. 2006, IEEE Power Eng. Soc., pp. 1–85.

[29] P. Kundur, *Power System Stability and Control*. New York: McGraw-Hill, 1994.

[30] W. Janischewskyj and P. Kundur, "Simulation of the non-linear dynamic response of interconnected synchronous machines, Part I—Machine modelling and machine-network interconnection equations," *IEEE Trans. Power App. Syst.*, vol. PAS-91, no. 5, pp. 2064–2069, Sep. 1972.

[31] PSS/E™ 31, Program Application Guide, Dec. 2007.

[32] D. N. Ewart and F. D. deMello, "FACE: A digital dynamic analysis program," in *Proc. Power Industry Computer Applications Conf.*, Sep. 1967, pp. 1–8.

[33] M. Wang, H. Klie, M. Parashar, and H. Sudan, "Solving sparse linear systems on NVIDIA Tesla GPUs," in *Computational Science—ICCS 2009*. Berlin, Germany: Springer, May 2009, pp. 864–873.

Vahid Jalili-Marandi (S'06) received the B.Sc. and M.Sc. degrees in power systems engineering from the University of Tehran, Tehran, Iran, in 2003 and 2005, respectively. Currently, he is pursuing the Ph.D. degree at the University of Alberta, Edmonton, AB, Canada.

His research interests include transient stability studies, and digital real-time simulations in power systems.

Venkata Dinavahi (S'94–M'00–SM'08) received the B.Eng. degree in electrical engineering from Visveswaraya Regional College of Engineering, Nagpur, India, in 1993, the M.Tech. degree from the Indian Institute of Technology, 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 joined the University of Alberta, Edmonton, AB, Canada, in July 2001, where he is presently a Professor. His research interests include real-time simulation of power systems and power electronic systems, large-scale system simulation, and parallel and distributed computing.