Carlos VILACHÁ PÉREZ, Julio César MOREIRA MEIRA, Edelmiro MÍGUEZ GARCÍA, Antonio FERNÁNDEZ OTERO

University of Vigo, Department of Electrical Engineering

Massive Jacobi Power Flow Based On SIMD-Processor

***Abstract**. This paper presents an implementation of the Jacobi power flow algorithm to be run on a single instruction multiple data (SIMD) unit processor. The purpose is to be able to solve a large number of power flows in parallel as quickly as possible. This well-known algorithm was modified taking into account the characteristics of the SIMD architecture. The results show a significant speed-up of the algorithm compared to the time required to solve the algorithm in a conventional CPU, even when a more efficient sequential algorithm, such as the Newton-Raphson, is used. The accuracy of the performance has been validated with the results of the IEEE-118 standard network. This paper also shows a case where the proposed algorithm is used to calculate a statistical load power flow using the Monte Carlo's method.*

Streszczenie.** W artykule przedstawiono implementację algorytmu rozpływu mocy Jacobiego przeznaczonych do uruchamiania na procesorze typu SIMD (jedna instrukcja wiele danych). Celem jest rozwiązanie dużej liczby rozpływów mocy równolegle w jak najkrótszym czasie. Ten dobrze znany algorytm został zmodyfikowany z uwzględnieniem cech architektury SIMD. Wyniki wskazują na znaczne przyspieszenie tego algorytmu w porównaniu do czasu potrzebnego do rozwiązania problemu za pomocą konwencjonalnych CPU, nawet jeśli zastosujemy najbardziej efektywny algorytm sekwencyjny, taki jak metoda Newtona-Raphsona. Dokładność uzyskanych wyników została potwierdzone dzięki porównaniu z wynikami uzyskanymi w sieci standardowej IEEE-118. W artykule przedstawiono również przypadek, gdy proponowany algorytm jest używany do obliczeń statystycznych rozpływów mocy przy użyciu metody Monte Carlo. **(Obliczanie równoległe rozpływu mocy za pomocą procesora SIMD)

Keywords: CUDA, GPU, Jacobi, Monte Carlo method, parallelization.
Słowa kluczowe: CUDA, GPU, algorytm Jacobiego, metoda Monte Carlo, paralelizacja.

Introduction

Today, a new architecture based on graphic processor units (GPU) exists that provides high performance parallel computing. At first, the GPUs were designed to improve computer graphics, but, little by little, in only a few years, their performance improved, primarily due to the needs of the video game industry. As a result, today, very realistic graphics are generated due to the powerful GPU units available. In only a few years' time, these units have begun to be used for general purpose calculations, yielding important results in Electromagnetism [1], and Biology [2]. The number of fields GPUs can be employed in continues to grow.

In the electrical power engineering field there are some problems that can be solved by GPUs, and some papers have already been published regarding this. Indeed, recently, a single-instruction-multiple- thread (SIMD) algorithm was presented to solve a transient stability problem [3] but with no improvement over already existing commercial software. A real-time power flow SIMD algorithm using symmetrical components is explained in [4]. Another approach for accelerating a power flow was published in [5], where Gauss-Jacobi and Newton-Raphson algorithms are used, with a speed-up of 10 times in a large-scale system. A Newton method and a biconjugate gradient algorithm are described in [6]. This author parallelizes one power flow obtaining an improvement of 2.1 times.

The previously cited papers show the difficulties in parallelizing the equations of the load flow and obtaining a significant improvement using the GPUs' architecture. Additionally, in [6] Amdahl's law is used to establish an acceleration limit in the load flow parallelization.

Power load flow is one of the most common and important analyses in the electrical power system [7]. It consists of obtaining bus voltages, line powers and generated power in electric power plants. Decisions about design, planning and operation are made using these data; these are the main cornerstones for the proper functioning of the network.

Nowadays, a single load flow can be solved relatively quickly, but in power system analysis there are more complicated problems that include solving a large number of power flows. Among these problems are steady state security assessment [8], reliability evaluation, stochastic power flow by Montecarlo simulations or planning optimization.

This paper presents an implementation of a Jacobi algorithm for solving power flows on a massive scale, which would allow solving the previously mentioned problems more quickly. It has five sections. Firstly we explain the general architecture of the graphics processor units (GPUs). Secondly, we adapt the traditional Jacobi's method to SIMD structure. Then, we show a comparative between a well-programmed-CPU algorithm versus the proposed method. Lastly, we solve a Montecarlo problem to demonstrate the benefits of the algorithm.

GPU Architecture

A graphics processing unit or GPU is a device designed to quickly manage memory in such a way so as to accelerate the composition of images in a buffer for output to a visualization target such as a monitor.

Current GPUs have an extraordinary performance at manipulating computer graphics because of their parallel architecture. These also are more efficient than central units processors (CPU) for algorithms where are processed large blocks of data in parallel.

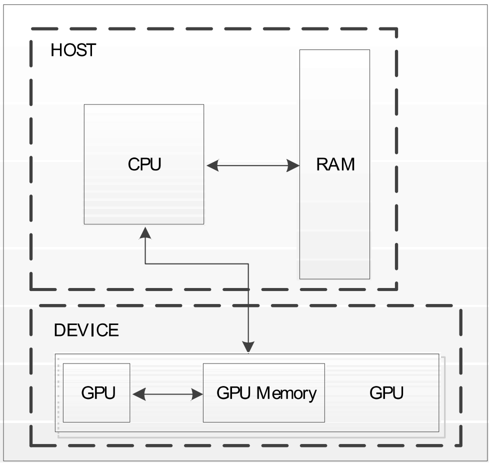

Fig. 1. Hardware architecture. Connection between the Host (CPU and RAM) and the Device (GPU).

Currently, GPUs are connected to the computer motherboard by means of an interface called PCI Express (PCIe). In the GPU-programming setting, the GPU is called the "device" and the motherboard, that includes the RAM and the CPU, is called the "host" as it is shown in the Fig. 1.

The GPUs have two kinds of programmable processors or arithmetic logic units (ALU): vertex and fragment [3]. Vertex processors work with meshes, doing mathematical operations and defining the position of the elements in the screen. The fragment processors, on the other hand, define the colors, light and textures.

Performing general-purpose computations with a GPU is difficult, hard work using a graphics-only language such as OpenGL or DirectX, because graphics instructions are being used to solve mathematical problems. As a result of the need for easier programming in graphics processors, high level languages HAL and CUDA were developed by the two main GPU builders, AMD and NVIDIA respectively. In this paper we will use CUDA.

CUDA stands for Computer Unified Device Architecture [9-11]. This technology allows the user to interact with the graphical processor in an easier fashion, even if the user is not an experienced programmer. This architecture uses all the ALUs to perform the general-purpose computations. Moreover, CUDA complies with the IEEE requirements for single precision floating-point arithmetic.

In CUDA language, an algorithm is a kernel that is run by the graphical unit. Several kernels can be executed at the same time. Each kernel is executed by a collection of threads which are gathered in blocks Fig. 2. All these blocks make up a grid.

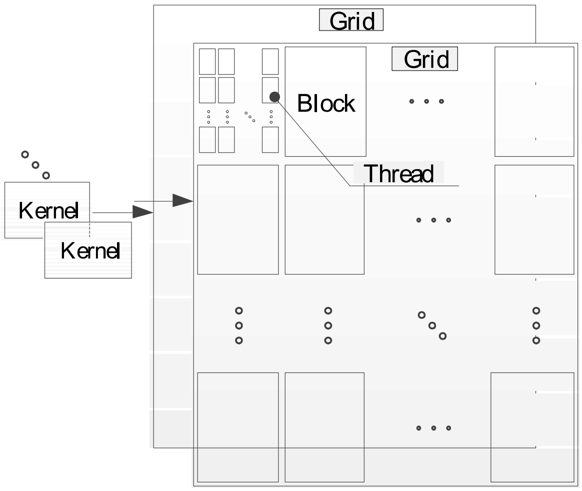

Fig. 2. CUDA architecture. Hierarchy of blocks and threads representation.

All the threads execute the same code. In other words, a single instruction is run by multiple threads; this is an SIMD. There is a mechanism to identify each thread inside the block and each block inside the grid, using indexes. These indexes are commonly used to read and write in the memory.

From the programmer's perspective, the kernel is called from the host. When the device has finished the calculations, the execution thread returns to the host. The user must be careful to properly copy the data from the host to the device and vice versa, and also set up the size of the grid and the blocks in accordance to the objective of the problem.

Moreover, CUDA provides a local memory and three specific memories that can improve the execution time if they are used properly, because each one is designed for a specific access pattern. The local memory is global and can be read and written from the host and accessed by all threads. The shared memory allows threads of a block to share data between them. The constant memory is an overall data storage that can be accessed by all threads. It provides a good performance if threads access sequentially at a same location of this memory. Finally, the texture memory is also common for all threads. This memory speeds up the algorithm if threads access to a memory location according with their position in the grid.

A GPU does not always perform well on some tasks that CPUs are built to perform well. Therefore, it is expected that most applications use both CPUs and GPUs, executing the sequential parts on the CPU and numeric intensive parts on the GPUs. This is why CUDA supports both CPU and GPU execution of an application.

SIMD-Based Jacobi Power Flow

In this section, a short explanation of Jacobi's method, its characteristics, and the implementation of CUDA in the GPU, in order to calculate a large number of power flows, is presented.

Jacobi's algorithm

Jacobi's method is a well-known iterative solver suitable for calculating a large linear equation system that has no zeros along its main diagonal. Each diagonal element is solved for. Then an approximated value is plugged in. This process continues until it converges if it does.

There are improvements of this method that have a faster convergence, such as the one by Gauss-Seidel. Nevertheless, Jacobi's iterative solver has an advantage over the others in that it is very suitable for parallelizing, as is shown in [12-15].

A linear set of equations can be represented as:

$$A \cdot x = b \tag{1}$$

where A is the coefficient matrix, x is the variable vector and b is the vector of the independent terms.

Each element of the vector x is calculated in the following manner:

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{i \neq j} a_{ij} x_j^k\right), i, j = 1, 2, ..., n \tag{2}$$

where x_j^k is an element of the vector x in the iteration k, a_{ij} is an element of the matrix A, and b_i is an element of the vector b.

As can be seen in Eq. 2 it is not necessary to know the value of the remaining variables in the iteration k+1 to be able to calculate x_i^{k+1} nor the previous value of itself. For this reason it is highly parallelizable.

Massive implementation

As per what was explained in the previous subsection, the calculations in the Jacobi's iterative method do not need to be kept in any sort of synchronization during iteration. Nevertheless, it is necessary to:

- To synchronize between each iteration to avoid that in the same load power flow
- To access the vector x^k by each x_j^{k+1}.

These restrictions are very important to be able to decide how the kernel must be programmed to obtain the highest performance.

From the power flow perspective, the algorithm had to obtain the voltage of each electrical node. As is well known, there are three kinds of nodes. There are nodes with a defined consumption of active and reactive power and with an unknown voltage (PQ node). Nevertheless, in the generation nodes for instance, the modulus of the voltage and the active power (PV node) is known. And lastly, a slack bus is set so the generated power and the consumption of power are equal. These nodes require a slight change in the algorithm.

The input data are: a vector with the node type T, the specified voltages for nodes PV (U^{esp}), the active (P) and reactive (Q) power vector for both nodes PQ and PV, and the admittance matrix. The latter is made up of the diagonal elements stored in vector (Y^D) and a compact matrix (Y^R) to store the remainder elements. We decided to store in two different data structures to minimize the amount of memory necessary because the vast majority of elements of the admittance matrix are zero (approximately the 94%), so we also reduced the transfer time. Another matrix (I^R) is used to store the column indexes for the elements. The results (the voltages) of the iteration k+1 are stored in vector (U_{k+1}). Finally, another vector is defined in order to store the voltage values of the iteration k.

Algorithm 1 CUDA Kernel – Jacobi Power Flow

```
1:  $\rightarrow enter;$
2:  shared bool error, error flag
3:  $error \leftarrow true;$
4:  $shared\_Memory \leftarrow local\_Memory;$
5:  $k \leftarrow 0;$
6:  while $error\&\&it\_max>k$ do
7:      $synchronize$, synchronize all the treads of the block
8:      $error \leftarrow false;$
9:      if $T_j == PQ\_Node$ then
10:         $U_j^{k+1} \leftarrow f(U^k, P, Q, Y^D, Y^R, I^R);$
11:     else if $T_j == PV\_Node$ then
12:         $U_j^{k+1} \leftarrow f(U^k, U_j^{esp}, P, Q, Y);$
13:         $Q_j \leftarrow f(U^k, Q, Y);$
14:     end if
15:     if $\left|U_j^{k+1} - U_j^k\right| \geq ERR$
16:         $error \leftarrow true;$
17:     end if
18:     $k \leftarrow k+1;$
19:     $synchronize$, synchronize all the treads of the block
20: end while
21: $\rightarrow exit;$
```

To obtain the best performance, each block of threads will solve a load flow. A copy of the admittance matrix and vectors, the specified voltage for the PV nodes, the active power for nodes PV and PQ and the reactive power for PQ nodes is transferred to the shared memory. Results are also stored in the shared memory and transferred to the local memory at exit and finally to the host.

In Alg. 1 the structure of the code programmed in the graphic unit is presented. In line 2, a shared boolean variable that indicates when the solution is reached, is defined. When the solution is reached the block becomes inactive. In line 3 the algorithm makes a copy of the input data in the shared memory. The remaining code is a common implementation of a Jacobi's iterative method, except for the synchronized flags. The first synchronization, in line 6, ensures that all the threads reach this point and no thread checks the break condition of the while loop due to some threads can be in line 16 while others are checking the condition in line 5 at the same iteration k. The last synchronization does not permit a thread to begin iteration before the current one has finished avoiding the divergence; all threads must to calculate the same iteration k.

The algorithm has two criteria to exit. The loop can finalize when the load power flow has converged or when a number of iterations are reached. We use the previous exit criterion to ensure that the algorithm ends up finalizing in all situations even when the load power flow has no solution.

Algorithm 1 CUDA Kernel – Jacobi Power Flow

```
$\rightarrow enter;$
$shared\_Memory \leftarrow local\_Memory;$
$synchronize$, synchronize all the treads of the block
$I_{ij} \leftarrow f(Y_{ij}, U_i, U_j);$
$\rightarrow exit;$
```

The Alg. 2 presents a kernel to calculate currents that flow through all lines. Each thread is in charge of the calculation of only one current. The necessary input data are the admittance between bus-bars Y and the vector of bus-bar voltages U.

This algorithm is very simple but cuts down the calculation time significantly mainly in large networks because it fits perfectly to GPU processors philosophy. Speed-ups in small networks are not very impressive, nevertheless, the higher the network and the number of power flows are, the more acceleration it gets.

In this paper the previously mentioned algorithm is run on an NVidia Tesla C2050. This is a device designed specifically to process general purposed calculations [17]. It allows a maximum setting of shared memory for each block and each block can contain 1024 threads. In the proposed method the number of nodes is limited. In fact, the shared memory runs out before the maximum number of threads per block is used. Nonetheless, this paper addresses the idea of solving a high number of power flows on a graphics unit.

Experimental Results

In this section, we present two cases in order to evaluate the performance of our proposed algorithm.

The following cases are run on an NVidia Tesla C2050 with 3 GB of memory connected to a PCI Express graphics bus in 16x mode and an Intel i5 750 at 2.67 GHz with 8 GB of RAM.

We consider that this hardware is a representative configuration of the current state of the art in the computing calculation based on GPU.

Algorithm Benchmarking

This subsection show an analysis consists of a series of executions where a number of load flows are solved at the same time. The network used is the IEEE 118, which is calculated several times in each case. In order to calculate the total CPU execution time, it has been taken into consideration that solving N power flows need N times more time. The convergence criterion is that the error defined in Alg. 1 is lesser than 10^{-8}. All the calculations are done using single precision floating-point format.

In order to make a realistic comparison, the results presented in [5] are used. The authors make a comparison between their proposal, a single power flow solved on graphic unit, and a CPU implementation on the Intel Math Kernel Library (MKL).

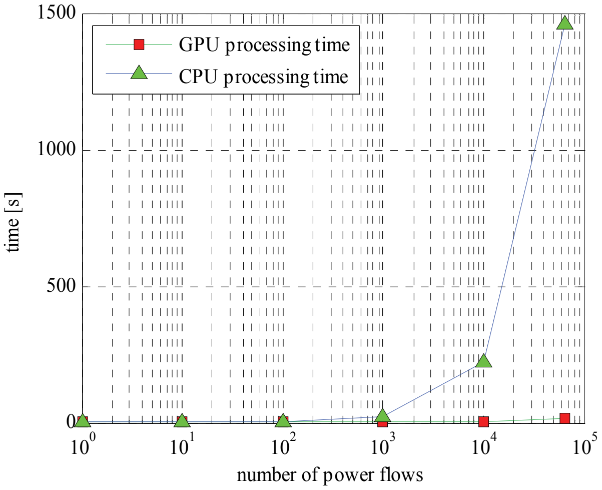

Fig. 3. Total simulation time for different number of power flow. CUDA architecture.

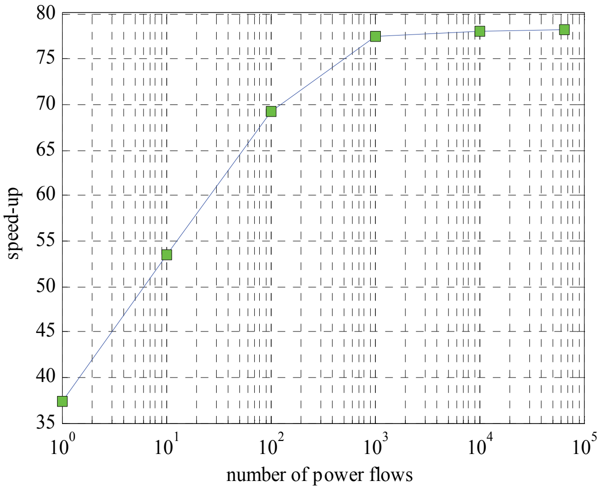

Fig. 4. Speed-up obtained compared with the CPU time for different number of power flow.

As can be seen in Fig. 3, the CPU needs much more time. It takes almost 25 minutes to solve 65000 power flows. By contrast, the GPU takes only 18.6 seconds. The improvement is very high, as is shown in Fig. 4, and the time reduction is very close to 80 times.

With a stressed network, near the collapsed point, the proposed method requires more iterations, over to 4 times further. Even so the speed-up keeps on being big, because the Newton Raphson needs more iterations too (near 2 times).

When a large number of blocks are thrown in CUDA, the performance is no faster than in the beginning, Fig. 4. But in the case of 118 nodes, only 118 threads are used, which is not the number that provides the best performance [9].

Another important point is that the CPU reference times were obtained using a much better algorithm. For instance, a Newton-Raphson method probably finds a solution in 5 iterations, but with the Jacobi's method 728 iterations are needed.

The previous algorithm, Alg. 2, is very simple but cuts down the calculation time in almost 8 times when 65000 load power flows are calculated over the IEEE-118 network.

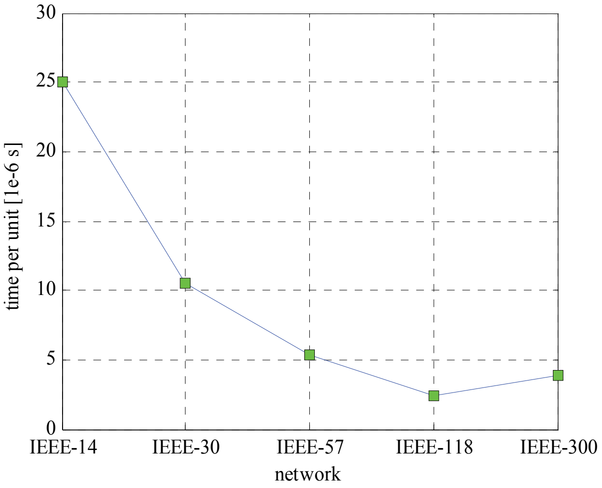

Fig. 5. Algorithm scalability.

The number of threads per block has a high impact on the execution time [9] as the Fig. 5 shows. This figure represents some IEEE standard networks versus the time that takes per each bus-bar to simulate a single power flow. In this case the best performance per node is obtained around 118 threads per block.

CUDA accepts up to 1024 threads per block. However is not recommendable to use all of them. Measure the execution time for different block setups is a good practice because the performance depends on kernel, used memories and hardware.

The current kernel needs the same number of threads as bus-bars the simulated network has. This paper does not take into account the previous issue, however future papers should consider it in order to obtain the maximum acceleration.

Montecarlo Simulation

Montecarlo method is a computational algorithm that relies on repeated random sampling to compute their results. This method is most suited to calculation by a computer and tends to be used when it is infeasible to compute an exact result with a deterministic algorithm.

Montecarlo method is often used in simulating statistical load power flow because of each time there are more non deterministic generation connected to network [18, 19].

In this subsection, we run a statistical load power flow using the Montecarlo method over the IEEE-118. This is an example where GPUs show their benefits over traditional implementations on CPUs.

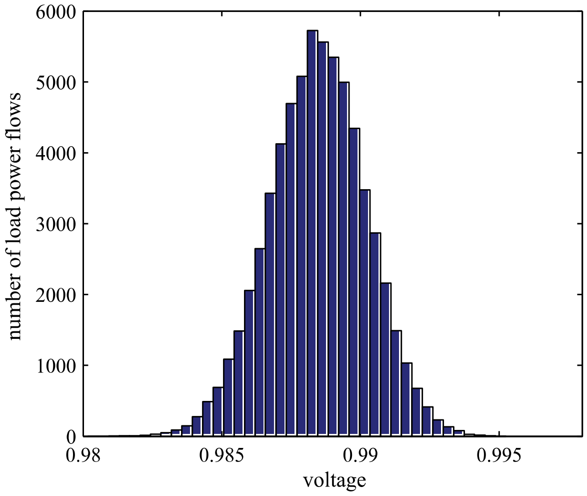

Fig. 6. Voltage histogram of a node.

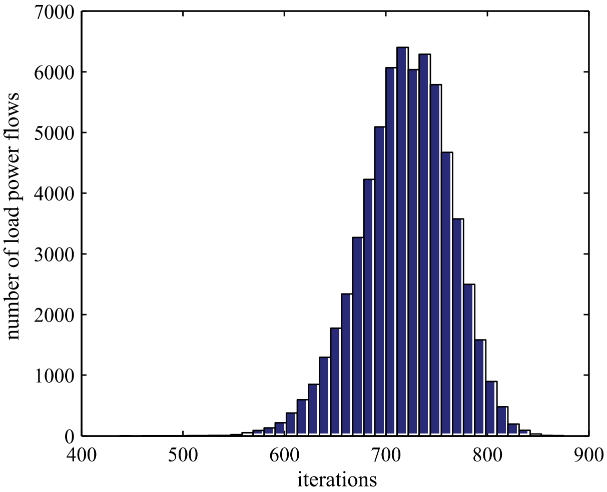

Fig. 7. Convergence histogram of a node.

We generate random values using a Gaussian function by means the Intel MKL libraries. We set 0 as mean value and 0.02 as standard deviation and we add the returned values to IEEE-118 voltage vector.

The Fig. 6 shows the histogram of the simulated voltages in a specific node.

The iterations of the algorithm can be seen in the Fig. 7. We set as a non-convergence criterion 10000 iterations. However all simulations converged into a value.

Conclusion

Previously published references focus on a parallel implementation in order to solve a single power flow. The speed-ups obtained are in the range of 1 to 2 times, some of them even report a performance of almost 10 times in large-scale systems. These speed-up figures are not amazing and are limited by the size of the sequential part of the algorithms used. Moreover, the transfer data time between the host (the computer) and the device (GPU) is significant when compared to the overall execution time of a single power flow.

The results obtained in this paper suggest it would be better to focus on power system analysis problems where a large number of power flow executions need to be done, including problems such as reliability assessment, probabilistic load flows or steady state security analysis.

Despite using a very basic iterative method with a very poor convergence rate, a speed-up of 80 times is achieved when compared to an efficient multi-core CPU algorithm. Even when this implementation has a limit on the number of nodes, the obtained results are promising.

In conclusion, the authors think that specific algorithms which take advantage of the simultaneous solving of multiple power flows should be developed in the future.

Acknowledgment

This work was supported by the Galician autonomous government under the contract INCITE08PXIB303262PR.

REFERENCES

[1] N. Godel, N. Nunn, T. Warburton, and M. Clemens, “Scalability of Higher-Order Discontinuous Galerkin FEM Computations for Solving Electromagnetic Wave Propagation Problems on GPU Clusters,” IEEE Trans. Magn., vol. 46, no. 8, pp. 3469–3472, Aug. 2010.

[2] G. Chalkidis, M. Nagasaki, and S. Miyano, “High Performance Hybrid Functional Petri Net Simulations of Biological Pathway Models on CUDA.” IEEE/ACM transactions on computational biology and bioinformatics / IEEE, ACM, no. 99, pp. 1–1, Nov. 2010.

[3] V. Jalili-Marandi and V. Dinavahi, “SIMD-Based Large-Scale Transient Stability Simulation on the Graphics Processing Unit,” IEEE Trans. Power Syst., vol. 25, no. 3, pp. 1589–1599, Aug. 2010.

[4] Dzafic and H.-T. Neisius, “Real-time power flow algorithm for shared memory multiprocessors for European distribution network types,” in 2010 Conference Proceedings IPEC. IEEE, Oct. 2010, pp. 152–158.

[5] J. Singh and I. Aruni, “Accelerating Power Flow Studies On Graphics Processing Unit,” in 2010 Annual IEEE India Conference (INDICON), 2010.

[6] N. Garcia, “Parallel power flow solutions using a biconjugate gradient algorithm and a Newton method: A GPU-based approach,” in Power and Energy Society General Meeting, no. 5, 2010, pp. 1–4.

[7] J. Grainger and W. D. S. Jr., Power Systems Analysis (Power & Energy). McGraw-Hill Publishing Co., 1994.

[8] F. Capitanescu and L. Wehenkel, “A New Iterative Approach to the Corrective Security-Constrained Optimal Power Flow Problem,” IEEE Trans. Power Syst., vol. 23, no. 4, pp. 1533–1541, Nov. 2008.

[9] J. Sanders and E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming. Addison-Wesley Professional, 2010.

[10] D. B. Kirk and W.-m. W. Hwu, Programming Massively Parallel Processors: A Hands-on Approach. Morgan Kaufmann, 2010.

[11] Nvidia, “Nvidia Cuda C. Programming Guide,” 2010. [Online]. Available: http://developer.nvidia.com/page/home.html

[12] J. Lin and Y. Chen, Design and Implementation of Jacobi Algorithms on GPU. IEEE, Oct. 2010.

[13] Z. Zhang, Q. Miao, and Y. Wang, CUDA-Based Jacobi’s Iterative Method. IEEE, 2009.

[14] R. Amorim, G. Haase, M. Liebmann, and R. Weber dos Santos, Comparing CUDA and OpenGL implementations for a Jacobi iteration. IEEE, Jun. 2009.

[15] W. Tao, Y. Yuan, H. Lin, Z. Dan, and Z. Yuanyuan, implementation of Jacobi iterative method on graphics processor unit. IEEE, Nov. 2009.

[16] Guerrero,J. and Naar,L. and Romero,G., Power system losses estimation using Montecarlo algorithm. IEEE, Nov. 2009.

[17] Nvidia, “TESLA C2050 / C2070 GPU Computing Processor Supercomputing.” [Online]. Available: http://www.nvidia.com/object/personal-supercomputing.html

[18] Z.N.C Viray and W.C. Nerves, “Integrated energy and reserve electricity market analysis using probabilistic optimal power flow,” in TENCON 2010 - 2010 IEEE Region 10 Conference. IEEE, Nov. 2010, pp. 598-603.

[19] C. Zapata, L. Garces, and O. Gomez, “Reliability Assessment of Energy Limited Systems Using Sequential Montecarlo Simulation,” in IEEE/PES Transmission and Distribution Conference and Exposition: Latin America. IEEE, Aug. 2006, pp.1-6.

***Authors**: M.Sc. C. Vilachá, M.Sc. J. C. Moreira, Ph.D. E. Míguez and Ph.D. Antonio F. Otero are with the Department of Electrical Engineering of University of Vigo, Galicia, Spain, E-mails: cvilacha@uvigo.es; jcmeira@uvigo.es; edelmiro@uvigo.es; afotero@uvigo.es.*