Distribution System Optimization on Graphics Processing Unit

Vincent Roberge, Mohammed Tarbouchi, and Francis A. Okou

Abstract—**Power distribution networks operate in a radial topology, but also include extra tie switches to allow for their reconfiguration in case of scheduled maintenance or unexpected failure. With the implementation of the smart grid and the development of fast high power switching devices, it is now possible to automatize this reconfiguration to also adjust to demand fluctuation and always operate the network in the optimal topology, minimizing power transmission losses. This automation requires the development of highly efficient and powerful optimization algorithms that can compute the optimal configuration with minimum delay. This paper presents a parallel genetic algorithm on graphics processing unit for distribution feeder reconfiguration. By exploiting the massively parallel architecture of graphics processors, the execution time of the solver is reduced by a factor of 66.2×, resulting in a very fast solver. Moreover, the metaheuristic uses a unique solution encoding based on the minimum spanning tree to maintain the radial structure of the candidate topologies. This novel encoding drastically improves the effectiveness of the genetic algorithm and allows for the optimal reconfiguration of networks up to 4400 buses; five times larger than any of the references surveyed.**

Index Terms—**Distribution feeder reconfiguration, genetic algorithm, graphics processing unit, minimum spanning tree, parallel programming.**

I. Introduction

THE POWER distribution network is the last stage in the delivery of electricity. It connects the distribution substations to the customers. Typically arranged in a radial topology, distribution networks include additional tie switches to allow for the reconfiguration of the network in case of planned maintenance, unexpected failure or demand fluctuation. An example of a 16-bus distribution network in shown in Figure 1. With the upgrade of the power infrastructure and the implementation of the smart grid, it is possible to automatize the network reconfiguration in order to always operate in a highly optimal topology minimizing outage time, power transmission losses and ultimately, operating costs. As explained in [1], this motivates the development of algorithms capable of calculating the network reconfiguration in real-time in order to quickly react to changes, especially following a hardware fault in order to minimize down time. Yet, calculating this optimal topology is not trivial. In fact, the distribution feeder reconfiguration (DFR) problem is a large-scale, non-convex, non-linear, combinatorial optimization problem. It can be formulated as finding the radial topology that:

Manuscript received June 6, 2015; revised August 27, 2015; accepted November 17, 2015. Date of publication December 7, 2015; date of current version June 19, 2017. Paper no. TSG-00626-2015.

The authors are with the Department of Electrical and Computer Engineering, Royal Military College of Canada, Kingston, ON K7K 7B4, Canada (e-mail: vincent.roberge@rmc.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/TSG.2015.2502066

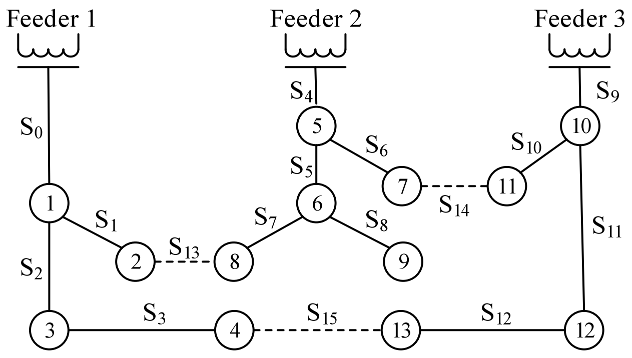

Fig. 1. 16-bus system (dash lines represent open switches).

minimizes

$$f(\bar{x}) = \sum_{i=1}^{N_{branches}} P_{loss\ i} \tag{1}$$

subject to

$$|V_i|_{min} \le |V_i| \le |V_i|_{max} \quad \text{for } i = 1 to N_{buses} \tag{2}$$

$$|S_i| \le |S_{i\ max}| \quad \text{for } i = 1, \ldots, N_{branches} \tag{3}$$

where $\bar{x}$ is the solution vector (topology of the network), P_{lossi} are the real power losses on branch i, $|V_i|$ is the voltage magnitude at bus i and $|S_i|$ is the apparent power on branch i. $|V_i|_{min}$, $|V_i|_{max}$ and $|S_{imax}|$ are the limits on the bus voltage and the branch rating. The biggest challenge in solving the DFR problem is to maintain the radial topology of the network when generating potential solutions.

In this paper, we propose a parallel genetic algorithm on graphics processing units (GPU) to compute the optimal DFR that minimizes real power losses. The proposed algorithm fully exploits the massively parallel architecture of graphic processors and provides a maximum speedup of 66.2x, resulting in an extremely fast solver. Moreover, the algorithm uses a unique encoding and relies on the calculation of the minimum spanning tree (MST) to decode the solution. This novel encoding ensures the radial topology of the network without the need for complex operators. The algorithm is tested on networks from 16 to 4400 buses and is proven to be very efficient and powerful. For all test cases, it finds solutions of equal or better quality than previous works from the literature while maintaining a much shorter execution time.

This paper makes two important contributions. The first one is the parallelization of a DFR solver on GPU which is truly advantageous in terms of execution time and has never been done before. The second is the unique solution encoding proposed which is very different from any of the encodings used by other references. Yet this encoding truly improves the scalability of metaheuristics to very large distribution networks, five times larger than any of the references surveyed.

The remainder of this paper is organized as follows: Section II reviews key publications on DFR; Section III provides an overview of the GPU architecture; Section IV describes the optimization strategy including the solution encoding used by the proposed solver; Section V explains the parallelization on GPU; and finally, experimental results are presented in Section VI.

II. Previous Works

DFR solvers are often classified into three categories: heuristics methods, conventional programming and metaheuristics [2]. Heuristics are problem specific and developed from experience. Two examples are the branch exchange [3] and the loop segmentation [4] methods. The branch exchange method starts with a known solution and opens and closes branches in pairs to generate new candidate topologies. Starting with all branches closed, the loop segmentation method identifies the fundamental loops and opens one branch per loop to finally obtain the radial network. Heuristics methods are intuitive and simple to implement. However, their search strategy is greedy and they cannot escape local optima.

Recent works relying on conventional programming techniques to solve the DFR problem include mixed-integer linear programming (MILP) [5], mixed-integer conic programming (MICP) [5] and mixed-integer quadratic programming (MIQP) [6], [7]. In these approaches, the mixed-integer non-linear programming (MINLP) problem is simplified and formulated so it becomes solvable by classic optimization methods. Unlike heuristics, conventional programming methods do not require an initial solution and guarantee convergence to the global optimum. However, these two advantages comes at the cost of a very long execution time and the solution found still depends on the simplified formulation of the problem. Conventional programming was used in [6] and [7] to successfully reconfigure a 880-bus network, but the execution times were respectively 3192 and 1134 seconds. Direct mathematical approaches to the DFR problem is also possible. In [8], a Benders' decomposition is used to divide the reconfiguration problem into two sub-problems, each solved using a commercial non-linear programming (NLP) solver. Although the results reported were encouraging, the NLP solver cannot guarantee the optimum solution within reasonable time [7].

Finally, metaheuristics are non-deterministic optimization algorithms used to solve problems intangible by classic methods. They require very little information about the problem and work by improving candidate solutions over many iterations until a quasi-optimal solution is found. Many metaheuristics have been used for DFR including the ant colony optimization (ACO) [9], the genetic algorithm (GA) [10], the artificial immune system (AIS) [11], the particle swarm optimization (PSO) [12], the bees algorithm (BA) [13] or the imperialist competitive algorithm (ICA) [14]. Advantages of metaheuristics include their versatility, their ability to escape local optima and their capacity to consider non-differentiable objective functions. On the down side, using a metaheuristic for DFR is challenging because the candidate solutions cannot be truly random, but must satisfy the radial constraint on the network. In some works such as [15], the metaheuristics is executed as is and the fitness function is defined to severely penalize solutions that are not radial. Although this technique works relatively well for small networks it does not scale to larger ones as the probability of randomly generating radial topologies becomes extremely small. A second and more common approach consists of developing complex operators to maintain the radial property of the network such as in [10] or [16]. These operators are however limited to the neighborhood of the solution and somewhat reduce the exploration of the search space. Moreover, because of the stochastic aspect of metaheuristic, the number of candidate solutions used and the number of iterations required can be very large resulting long and sometimes impractical execution times. For this reason, metaheuristics have rarely been used for the reconfiguration of networks with more than a few hundred buses. In fact, the largest network found in the literature that was optimized by metaheuristics had 476 buses [11].

To address this limitation, this paper suggests a parallel implementation on GPU to significantly accelerate the execution of a metaheuristic-based DFR solver. The proposed algorithm is also rendered extremely efficient by the use of a unique solution encoding that avoids complex operators. The resulting algorithm is able to reconfigure networks up to 4400 buses, much larger than for any of the references surveyed.

GPUs have never been used in the context of DFR. However, there exists many parallel implementations of metaheuristics for GPUs including a parallel PSO [17], a parallel GA [18], and a parallel ACO [19]. A strategy for the development of parallel hybrid metaheuristics on GPU was published in [20]. A software framework called *gpuMF* was introduced in [28] as a tool to facilitate the deployment of parallel metaheuristics on GPU. For some optimization problems, the performance gained by the parallel implementations in [20] and [28] surpassed 100 folds which motivates us to develop a parallel metaheuristic-based DFR solver on GPU. The proposed solver makes use of *gpuMF* to implement the optimization algorithm and rely on a unique solutions encoding. Although *gpuMF* implements both the PSO and the GA, the proposed solver uses the GA as it was shown in [21] that the GA converges in fewer iterations than the PSO for the DFR problem. This improved convergence is key to the development of a fast DFR solver.

III. GPU Architecture

The architecture of an NVIDIA GPU is shown in Figure 2. These processors are designed with a very large number of cores called *streaming processors* (SP). The SP are grouped

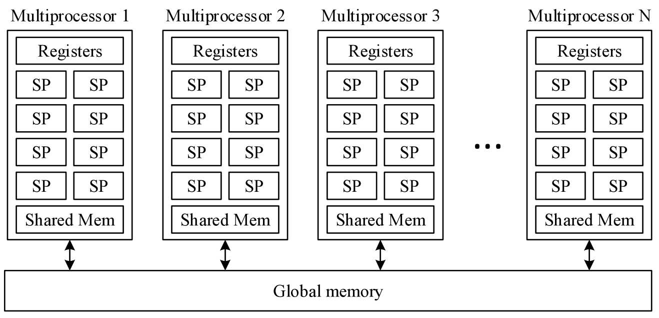

Fig. 2. Typical architecture of graphics processing units.

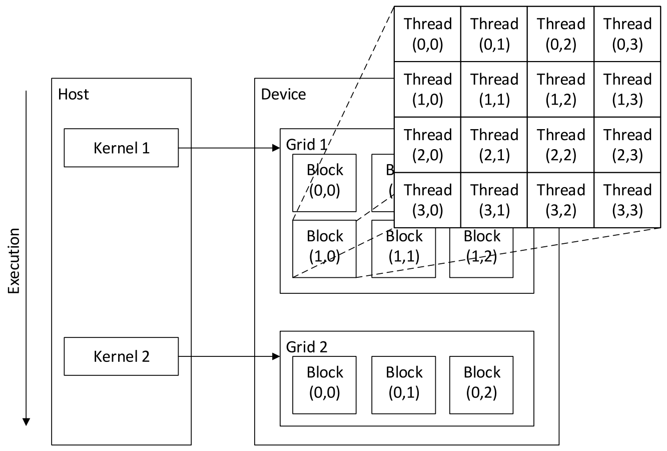

Fig. 3. Execution model of a CUDA program.

into *multiprocessor* (MP) units and share a bank of registers and an on-chip *shared memory*. The GPU has a large off-chip random access memory accessible by all MPs called the *global memory*. The GPU is connected to the CPU through the PCIe bus. The experimental tests presented in Section VI were conducted on a NVIDIA GTX Titan card. This card has a GK110 chip, 2880 SPs, 15 MPs, 64 K 32-bit registers per MP, 64 KB of shared memory per MP and 6 GB of DDR5 random access memory. The SPs operates at 876 MHz and the global memory has an effective frequency of 6008 MHz delivering a bandwidth of 288 GB/s.

NVIDIA GPUs are programmed in CUDA^{TM} language by writing parallel functions called *kernels*. The execution model is illustrated in Figure 3. The flow of the program is controlled by the CPU (or host) and heavy calculations are offloaded to the GPU (or device) by calling *kernels*. To execute a *kernel,* the GPU launches a grid of threads divided into thread blocks. Each thread block is mapped to a MP and the multiple SPs allow for the concurrent execution of the threads.

IV. Proposed Optimization Strategy

A. Genetic Algorithm

The GA is an optimization algorithm inspired from the natural evolution of species. Starting with a population of candidate solutions, the GA relies on selection, crossover and mutation operations to improve the solutions mimicking how living organisms evolve and adapt to their environment. The pseudo code of the GA is listed in algorithm 1.

Algorithm 1 Pseudo code of the Genetic Algorithm

```
1: generate population of candidate solutions
2: for $i = 0$ to num_iterations do
3:     compute fitness of candidate solutions
4:     select parent solutions
5:     generate children solutions by crossover
6:     mutate children solutions
7:     replace parent solutions
8: return best solution
```

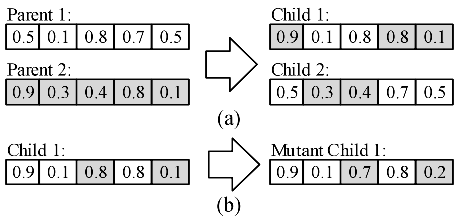

Fig. 4. Uniform crossover (a) and the uniform random mutation (b).

In the proposed implementation, the candidate solutions are encoded as a vector of real numbers between 0 and 1 and initialized randomly using a uniform distribution. The selection is done by tournament from a pool of 3 candidates [22]. Each pairs of parents generates two children by uniform crossover [22]. Children are subject to a uniform random mutation [22] based on a selection probability of 0.1 and a probability of mutation of 0.2. These two operators are illustrated in Figure 4.

B. Solution Representation

Most of previous works on metaheuristics for DFR encode the candidate solutions using (a) a vector of binary numbers representing the state (open/close) of every branch or (b) a vector of integer numbers identifying the index of the open branches [10]. In both cases, preserving the radial topology of the network is a major challenge as explained in Section II.

In this work, the candidate solutions are represented differently using vectors of real numbers. Each element varies between 0 and 1 and the length of the vectors is equal to the number of branches in the network. An example of a randomly generated solution to the 16-bus network is shown in Figure 5. To decode a solution, the feeder buses are grouped into a single root node and the distribution network is represented as an undirected graph. Each element of the solution vector is assigned to a branch and acts as a weight. The Boruvka's MST algorithm [23] is computed on the weighted graph to get the radial topology associated with the candidate solution as shown in Figure 6. Finally, the state (open(0)/close(1)) of each branch is recorded in Figure 7.

The solution encoding proposed here has never been used in any other metaheuristic-based DFR solver. It is novel, simple, well suited for a parallel implementation on GPU, and it ensures the radial topology of the candidate solutions without the need for complex operators. As we will see in Section VI, this encoding will significantly improve the performance of the DFR solver and represents a key contribution of this paper.

Branch ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Solution	0.5	0.1	0.5	0.7	0.5	0.0	0.8	1.0	0.0	0.7	0.4	0.3	0.6	0.9	0.0	0.8

Fig. 5. Encoded solution vector.

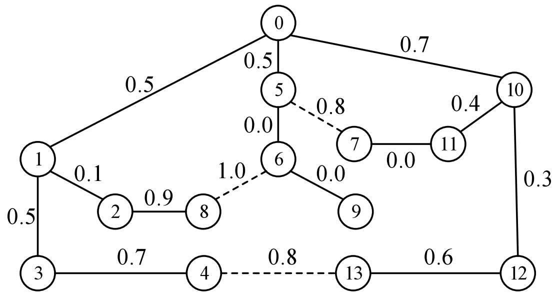

Fig. 6. Configuration of the 16-bus network after execution of the minimum spanning tree algorithm.

Branch ID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Solution	1	1	1	1	1	1	0	0	1	1	1	1	1	1	1	0

Fig. 7. Decoded solution vector.

C. Backward-Forward Power Flow Analysis

Once the candidate solution are decoded, a power flow (PF) analysis using the Backward-Forward (B-F) algorithm [24] is run to obtain the bus voltage. The B-F algorithm is the method of choice for this application as it is specially designed to analyze radial networks where the R/X ratio is generally high causing other PF techniques such as Newton-Raphson (N-R) or Gauss-Seidel (G-S) to diverge [25]. Moreover, because the B-F method is limited to radial networks, the calculations involved are simpler than for the N-R method and the number of iterations required before convergence is smaller than for the G-S method. Finally, previous works have shown that a parallel implementation on GPU of the B-F method can provide higher speedups and lower execution times than parallel implementations of the N-R or the G-S methods [26], [27]. The B-F algorithm works by performing a series of backward and forward sweeps on the radial network and therefore require the system to be organized in layers as in Figure 8. This is achieved using a graph traversal operation to compute the depth of every node.

The details of each B-F iteration t are given below:

1) **Nodal current injection:** Calculate the current $I^t_{node\ i}$ injected at every node i of using:

$$I^t_{node\ i} = \left(\frac{S_i}{V_i^{t-1}}\right)^* - Y_i V_i^{t-1} \tag{4}$$

where S_i is the power load; V_i^{t-1} is the voltage as computed at the previous iteration, Y_i is the total bus shunt admittance, i is the node index and '*' denotes the complex conjugate operator.

2) **Backward sweep:** from the deepest layer to layer 1, calculate the current $I^t_{br\ i,j}$ in the branch connecting node i to its parent j using:

$$I^t_{br\ i,j} = -I^t_{node\ i} + \sum_{node\ k\ \in\ children\ of\ i} I^t_{br\ i,k} \tag{5}$$

where $I^t_{br\ i,k}$ is the current in the branch connecting node i to its child k.

3) **Forward sweep:** from layer 1 to the deepest layer, calculate the voltage V_i^t at node i using:

$$V_i^t = V_j^{t-1} + Z_{br\ i,j} I^t_{br\ i,j} \tag{6}$$

where $Z_{br\ i,j}$ is the impedance of the branch connecting node i to its parent j.

4) **Nodal power injection:** compute the power S_i^t injected at every node i using:

$$S_i^t = V_i^t \left(\sum_{node\ k\ \in\ children\ of\ i} Y^*_{br\ i,k} V_k^{t*} \right) \tag{7}$$

where $Y_{br\ i,k}$ is the susceptance of the branch connecting node i to its child k and and '*' denotes again the complex conjugate operator.

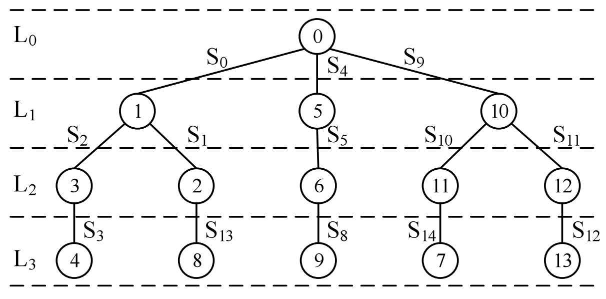

Fig. 8. Layers of the radial network.

This iterative process stops when the largest error between the calculated power injected and the scheduled power injected is smaller than a specified tolerance.

D. Fitness Function

Once the PF analysis is completed, the fitness function $\mathcal{F}(\bar{x})$ can be computed. Firstly, the objective function $f(\bar{x})$ must be evaluated using equation (1) and normalized between 0 and 1:

$$f_{NORM}(\bar{x}) = \frac{1}{1 + f(\bar{x})} \tag{8}$$

Secondly, a violation factor is computed to account for the inequality constraints. Each bus is checked and any excess on the voltage is quantified using:

$$E(|V_i|) = \begin{cases} \dfrac{|V_i| - |V_i|_{max}}{|V_i|_{max} - |V_i|_{min}}, & |V_i| > |V_i|_{max} \\ \dfrac{|V_i|_{min} - |V_i|}{|V_i|_{max} - |V_i|_{min}}, & |V_i| < |V_i|_{min} \\ 0, & |V_i|_{min} \le |V_i| \le |V_i|_{max} \end{cases} \tag{9}$$

where $E(|V_i|)$ is the relative excess on the voltage at bus i. In equation (9), the difference is divided by the range in order to get a relative value. This allows for a meaningful comparison when the voltage limits vary from one bus to another. The same approach is used to compute the relative excess $E(|S_i|)$ on the branch rating for all branches i. The values are then summed to get the total relative excesses $E_{total}(\bar{x})$ for the whole network:

$$E_{total}(\bar{x}) = \sum_{i=1}^{N_{bus}} E(|V_i|) + \sum_{i=1}^{N_{branch}} E(|S_i|) \tag{10}$$

As for the objective function, this value is normalized between 0 and 1 to form the normalized violation factor $\mathcal{V}_{NORM}(\bar{x})$:

$$\mathcal{V}_{NORM}(\bar{x}) = \frac{1}{1 + E_{total}(\bar{x})} \tag{11}$$

Thirdly, the normalized objective function and the normalized violation factor are grouped to get the fitness $\mathcal{F}(\bar{x})$ of the candidate solution using:

$$\mathcal{F}(\bar{x}) = \begin{cases} 1 + f_{NORM}(\bar{x}), & \mathcal{V}_{NORM}(\bar{x}) = 1 \\ \mathcal{V}_{NORM}(\bar{x}), & \mathcal{V}_{NORM}(\bar{x}) < 1 \end{cases} \tag{12}$$

so that infeasible solutions all have a fitness between 0 and 1 and feasible solutions, between 1 and 2.

V. Parallel Implementation

Designed with thousands of cores, GPUs form a new family of computing devices referred to as massively parallel processors. Whereas task level parallelism is adequate for multicore CPUs, a much higher level of parallelism must be exploited in order to take advantage of GPUs. This level of parallelism can only be achieved through data level parallelism. Parallel programs for GPU execute in a sequential logic order, but every operation is run in parallel on a large amount of data using different parallel primitives.

In this section, we discuss the parallelization on GPU of the proposed solver for DFR. The solver is designed to offer a clear separation between the optimization algorithm and the problem considered. This modularity allows the use of *gpuMF* [28], a framework for metaheuristics on GPU, to implement the GA module. Compared to other frameworks [29] that simply offload some of the calculations to the GPU, *gpuMF* fully parallelizes every step of the GA providing a speedup up to 276x compared to a sequential execution on CPU. Since the implementation details of *gpuMF* are available in [28], they are not repeated here and the rest of this section is dedicated to the second module of the solver, the parallel evaluation of the fitness function.

As discussed in the previous section, the evaluation of the fitness function involves a) the calculation of the MST to get the radial topology, b) a graph traversal operation to compute the depth of every node, c) a B-F power flow analysis to get the bus voltage, and finally d) the evaluation of $\mathcal{F}(\bar{x})$, the actual fitness function. All four operations are implemented in batch mode where the candidate solutions are evaluated concurrently using one thread block per solution (or network). In this section, we review key parallel primitives and show how they are used to implement the four operations.

A. Parallel primitives

1) Parallel Map Function: Sequential loops where each iteration runs independently of each other are easily parallelized using a parallel map primitive. This operation consists

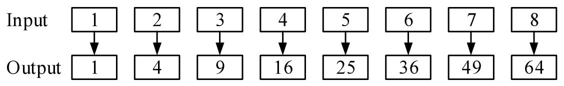

Fig. 9. Parallel map to compute the square of the elements of a vector.

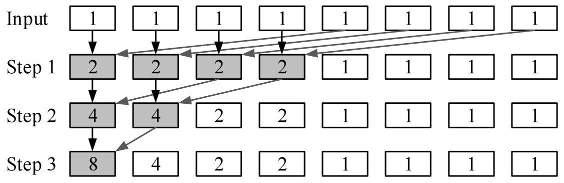

Fig. 10. Parallel reduction to compute the sum of the elements of a vector.

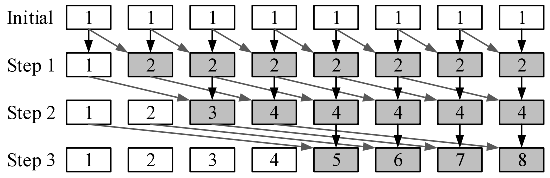

Fig. 11. Parallel scan to compute the prefix sum of the elements of a vector.

of executing the multiple iterations concurrently using one thread per iteration. An example is shown in Figure 9 to compute the square of every element in a vector. The parallel map leads to highest possible speedup as no inter-thread synchronization is required.

2) Parallel Reduction: The parallel reduction processes a series of inputs and produces a single output. An example is shown in Figure 10 to compute the sum of the elements of a vector. In a sequential program, this operation is implemented using a loop and executes in N steps compared to log_2N for the parallel reduction.

3) Parallel Scan: Finally, the parallel scan computes a partial reduction for every element of the input vector, i.e., element i from the output vector is the reduction of elements 0 to i from the input vector. A parallel scan based on the Kooge-Stone prefix graph [30] for a sum operation is shown in Figure 11. It takes log_2N steps to complete compared to N steps for a sequential loop.

B. Parallel Minimum Spanning Tree

The first step in the calculation of the fitness function is to the calculation of the MST. Three common algorithms for MST are Prim's, Kruskal's and Boruvka's. The Prim's algorithm uses a dense adjacency matrix and two nested loops. The algorithm has been parallelized on GPU, but the parallelization was limited to the inner loop resulting in a modest speedup of two folds [31]. The Kruskal's algorithm works by selecting the branch with the minimum weight from the set of all branches. The branch selected is added to the tree only if it does not create a cycle. The Kruskal's algorithm has also been ported on GPU, but offered a limited performance gain of 3x [32]. Between the three methods, Boruvka's algorithm offers the best potential for parallelization. Its operation is illustrated in Figure 12 to find the MST of the 16-bus network shown earlier in Figure 6. At the first iteration, each node is visited to identify the branch with the minimum weight.

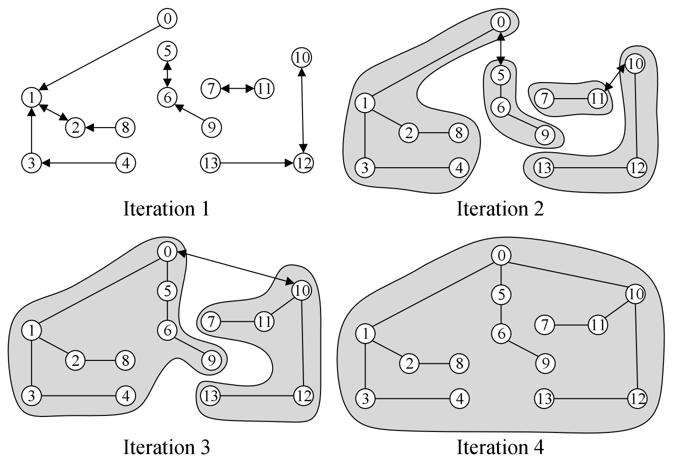

Fig. 12. Operation of the Boruvka's MST algorithm for the 16-bus network shown earlier in Figure 6.

The sub-trees are grouped to form super-nodes represented by the gray regions. At the second iteration, each super-node is visited and the branch with the minimum weight is identified. This process continues until there remains a single super-node which represents the MST of the graph. An implementation of Boruvka's algorithm on GPU was proposed in [33] and provided good speedup. However, the parallel approach used irregular memory access patterns and was criticized for requiring heavy hand-tuned code to perform well [23]. Recently, da Silva Sousa *et al.* [23] published a parallelization better adapted to the GPU architecture that only uses regular memory access patterns. The resulting algorithm is highly efficient and allowed speedups up to 16.2x.

The MST algorithm used in the proposed DFR solver is based on this last implementation, but modified to work in batch mode, solving many MST concurrently. By working in batch mode and limiting the integer number to two bytes, we are able to store all temporary data in the GPU shared memory and run the entire algorithm in a single CUDA *kernel* resulting in a very fast execution. As we will see in the experimental results section, our parallel MST implementation achieved a maximum speedup of 50x compared to sequential execution on CPU.

Our implementation of the Boruvka's MST algorithm uses a compressed sparse row (CSR) adjacency matrix to represent the meshed graph. The methods works only for undirected graphs and every branch must be entered twice, once for each direction. The CSR representation of the graph introduced earlier in Figure 6 is shown here in Figure 13. As the graph contains 16 branches, the CSR matrix holds 32 entries. At iteration 1, a parallel map function is used to process all nodes concurrently and identify their minimum branch. Mirrored branches are removed and the remaining ones added to the MST. A second parallel map function is used to identify the super-node associated with each node. This operation requires multiple passes where every node's successor is replaced by the successor's successor until each node points to the root of its super-node (or sub-tree). New IDs are assigned to each super-node using a parallel scan operation. Finally, the CSR adjacency matrix for the next iteration is built using a map

Algorithm 2 Pseudo code to compute node depth

```
1:  for all i = 0 to num_nodes do in parallel
2:      depth[i] = -1
3:  depth[root_node] = 0
4:  search_completed = false
5:  while (search_completed == false)
6:      search_completed = true
7:      for all i = 0 to num_edges do in parallel
8:          if (depth[f_node[i]] == -1) && (depth[t_node[i]] != -1)
9:              depth[f_node[i]] = depth[t_node[i]] + 1;
10:             search_completed = false;
11:         else if (depth[f_node[i]] != -1) && (depth[t_node[i]] == -1)
12:             depth[t_node[i]] = depth[f_node[i]] + 1;
13:             search_completed = false;
14: return depth[ ]
```

and a scan function. The entire process is repeated until there is a single super-node left. The details of the operations are given in [23].

C. Parallel Graph Traversal

The graph traversal operation is used to compute the depth of every node in the radial network obtained at the previous step. Two common methods are the depth-first search (DFS) or the breath-first search (BFS). Both methods are efficient and have a linear complexity given that the graph is represented using a sparse adjacency matrix. Unfortunately, the DFS is intrinsically sequential and cannot be parallelized. On the other hand, the BFS can exhibit a fair level of parallelism if the out degree of the nodes is large. Yet, the memory access pattern is irregular and does not map nicely to the GPU architecture. Parallel BFS for GPUs were proposed in [34]–[36] and resulted in limited speedups of respectively 2.3x, 2.5x and 3x.

To achieve a better speedup, one can resort to a simple quadratic parallelization. As explained in [37], this parallel method consists of visiting the root node first and marking all others as not-visited. At each iteration, every branch ($n_i \rightarrow n_j$) is inspected and a node n_j is processed only if n_i is marked as visited and n_j as not-visited. The search terminates when there is no node left to visit. The pseudo code for a quadratic parallelization of the graph traversal to compute the depth of the nodes of a graph is given at algorithm 2. In this code, the array *depth* is used to record the depth of every node, but also to identify their status as nodes that have not been visited yet are assigned a depth of -1. The inspection of the branches is done in parallel and the shared variable *search_completed* is set to false as soon as a node is visited in the current iteration. This shared variable is used to control the iterative process defined by the while loop.

Quadratic parallelization strategies have been used several times to port graph traversal algorithms on GPU [38], [39]. They are easily mapped to the GPU architecture and require a very simple synchronization mechanism as shown in the example code above. On the down side, they do not scale well to larger networks as they are not work efficient. Yet, in the context of the DFR solver, the graph traversal is performed concurrently on a large number of small to medium size graphs and, as we will see in Section VI, the speedup achieved compared to a sequential DFS is excellent.

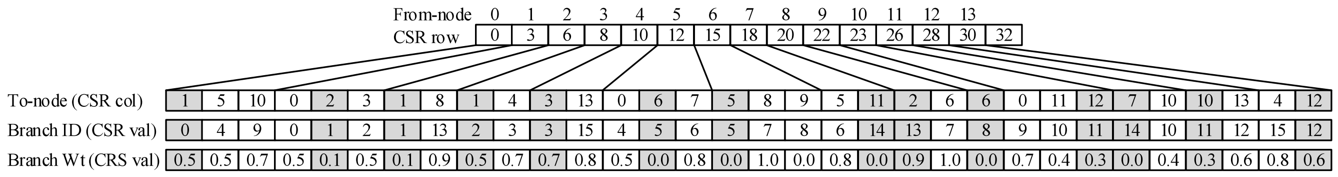
From-node 0 1 2 3 4 5 6 7 8 9 10 11 12 13
CSR row 0 3 6 8 10 12 15 18 20 22 23 26 28 30 32
To-node (CSR col) 1 5 10 0 2 3 1 8 1 4 3 13 0 6 7 5 8 9 5 11 2 6 6 0 11 12 7 10 10 13 4 12
Branch ID (CSR val) 0 4 9 0 1 2 1 13 2 3 3 15 4 5 6 5 7 8 6 14 13 7 8 9 10 11 14 10 11 12 15 12
Branch Wt (CRS val) 0.5 0.5 0.7 0.5 0.1 0.5 0.1 0.9 0.5 0.7 0.7 0.8 0.5 0.0 0.8 0.0 1.0 0.0 0.8 0.0 0.9 1.0 0.0 0.7 0.4 0.3 0.0 0.4 0.3 0.6 0.8 0.6

Fig. 13. CSR sparse adjacency matrix representing 16-bus network shown earlier in Figure 6.

TABLE I
PSEUDO-CODE OF OUR IMPLEMENTATION OF THE BACKWARD-FORWARD POWER FLOW SOLVER WITH AVERAGE RUNTIME (MS) AND SPEEDUP (880-BUS TEST CASE, 1000 INSTANCES OF THE NETWORK, 20 TRIALS)

Pseudo code	Average time (ms)		Speedup
	CPU	GPU	
1: *find depth of every bus*	247.05	0.91	271x
2: *build a depth-bus connection matrix*	68.91	0.68	102x
3: *build a bus-children connection matrix*	58.22	1.05	55x
4: **while** *(iterations < max_iterations)*	-	-	-
5: **&&** (max error > tolerance)	-	-	-
6: *compute bus current injection*	41.41	0.14	300x
7: *perform backward sweep*	30.17	1.81	17x
8: *perform forward sweep*	31.18	1.48	21x
9: *compute complex power S mismatch*	86.36	0.18	469x
10: *find the max error*	0.89	0.04	25x
11: *iteration = iteration + 1*	-	-	-
12: *update network data with calculated V*	30.92	0.08	391x
13: *compute power S required at feeder bus*	0.30	0.03	10x
14: **return**			
TOTAL	**595.408**	**6.404**	**93.0x**

Quadratic parallelization strategies have been used several times to port graph traversal algorithms on GPU [38], [39]. They are easily mapped to the GPU architecture and require a very simple synchronization mechanism as shown in the example code above. On the down side, they do not scale well to larger networks as they are not work efficient. Yet, in the context of the DFR solver, the graph traversal is performed concurrently on a large number of small to medium size graphs and, as we will see in Section VI, the speedup achieved compared to a sequential DFS is excellent.

D. Parallel Backward-Forward Power Flow Analysis

Once the depth of every node has been calculated, the B-F power flow is executed. The pseudo code of the B-F method is listed in Table I. Execution times for the sequential and parallel implementations are included in the table in order to give a clear view of the efficiency of the parallelization and identify the bottlenecks. The runtimes are averaged over 20 trials and were measured on a Dell Precision T7600 workstation equipped with two 8-core Intel Xeon E5-2650 CPUs and a NVIDIA GTX Titan GPU. Here, the software was used to solve 1000 instances of a 880-bus network taken from [7]. At the exception of line 11, all functions runs in batch mode using one thread block per network.

In Table I, the data is kept in the GPU global memory and each line is implemented by one or more CUDA *kernels*. When the network size allows it, the data is loaded into shared memory at the beginning of each *kernel* to avoid any non-coalescent memory access to global memory. The operation at line 1 is a graph traversal and was described at the previous section. At lines 2 and 3, two connection matrices in CSR format are built. Those sparse matrices are key to the fast execution of the algorithm and allow for a direct access to the nodes within a layer and to the children of a node. Those matrices are built directly on the GPU using parallel primitives such as a sort, a reduction by key and a scan. The detailed procedure is available in [40], but was modified to allow for batch operation. The computation of the nodal current injection at line 6 is done in a single step using a parallel map function with one thread per node. The backward sweep at line 7 starts at the lowest level and sequentially moves up to level 1. All nodes within a level are processed concurrently using one thread per node. This strategy is possible because of the two connection matrices built earlier. The forward sweep at line 8 starts at level 1 and iterates down to the lowest level. The parallelization technique used here is the same as for the backward sweep. Once the voltage estimates for all nodes have been updated, the nodal complex power injected is calculated at line 9, again using a parallel map function with one thread per bus. The maximum error between the calculated and scheduled nodal power is found at line 10 using a parallel reduction primitive. When the largest error falls below the specified tolerance or when the maximum number of iterations has been exceeded, the B-F algorithm terminates. The complex voltage are saved in memory at line 12 and the power required at the feeder bus is calculated using equations (7) S^t at line 13.

From the runtimes in Table I, it can be noted that the proposed parallelization on GPU provide a significant speedup for most of the functions. This is caused by the very high level of parallelism exploited. As an example, when computing the complex power at lines 9 for the 1000 instances of the 880-bus network, the program launches a massive parallel map function with 880 000 threads and fully utilizes the parallel architecture of the GPU. In the case of the backward and the forward sweep, the level of parallelism exploited is not as high since only the nodes from the same layer are processed concurrently. The level of parallelism is further reduced when computing the power at the feeder bus at line 13 as only one thread per network is launched. Despite those bottlenecks, the overall speedup provide by the GPU is excellent and will be key to the fast execution of the parallel GA-based DFR solver.

E. Parallel Evaluation of the Fitness Function

Once the B-F power flow analysis is completed and the node voltages are known, the transmission real power losses can be calculated for every branch using a parallel map function with one thread per branch and summed using a parallel reduction.

TABLE II
DETAILS OF THE TEST CASES

Test case	Test case details				GA config.	
	Feeders	Buses	Branches	Load (MVA)	Num. soln	Num. ite.
16-bus	3	13	16	28.7 + *i* 17.3	128	10
33-bus	1	32	37	3.7 + *i* 2.3	512	20
70-bus	2	68	79	4.5 + *i* 3.1	1024	100
83-bus	11	83	96	28.4 + *i* 20.7	1024	100
136-bus	1	135	156	18.3 + *i* 7.9	1024	200
415-bus	55	415	480	141.8 + *i* 103.5	1024	600
880-bus	7	873	900	124.9 + *i* 74.4	1024	600
1760-bus	14	1746	1820	249.7 + *i* 147.7	1024	900
4400-bus	35	4365	4550	624.4 + *i* 371.8	1024	150

These two operations are done in batch mode, processing every candidate topology concurrently. The excesses on the inequality constraints at equation (9) are calculated in the same manner. Finally, the fitness function at equation (12) is evaluated using a map primitive with one thread per network before the GA optimizer continues its execution.

VI. EXPERIMENTAL RESULTS

In this section, the parallel DFR solver for GPU is tested on the nine distribution networks listed in Table II. The complete data description is available for download in an online repository at [41]. Seven of them come from the literature and two were created especially for this work. The 1760-bus network was assembled using two instances of the 880-bus network and 20 extra tie switches to interconnect the two networks. The 4400-bus network was built in the same manner, but using five instances and 50 extra tie switches. All tests are run on a Dell Precision T7600 workstation equipped with two 8-core Intel Xeon E5-2650 CPUs and a NVIDIA Titan GPU. The CPU version is implemented in C++ in Visual Studio 2013 and compiled with full optimization for best performance. The CPU code relies on the Boost C++ library [42] to implement some of the basic functions such as sorting arrays when building the sparse adjacency matrices. The GPU version is also implemented in Visual Studio 2013 with full optimization. The parallel primitives are coded using the block-wide collective functions from the NVIDIA CUB library [43] to allow for batch operation and ensure maximum performance.

A. Evaluation of the parallel strategies

In the first test, the speedups gained from the parallelization of the Boruvka MST, the graph traversal and the B-F power flow analysis are measured and listed in Table III in order to provide insights on the efficiency of the parallel strategy used in the development of these three software modules. The timings and speedups are also plotted in Figures 14, 15 and 16. All tests were performed in batch mode processing 1000 instances for each network. Solving such a large number of instances may seem extravagant, but it is typical for the GA solver proposed. The runtimes reported in Figures 14, 15 and 16 do not include data transfers between the CPU and the GPU as the data required for the fitness evaluation of the candidate solutions already resides on the GPU. When the solver is first started, the network description is copied to the GPU and all

TABLE III
AVERAGE RUNTIME AND SPEEDUP FOR THE PARALLEL BORUVKA MST, THE PARALLEL QUADRATIC GRAPH TRAVERSAL AND THE PARALLEL BACKWARD-FORWARD POWER FLOW ANALYSIS ALGORITHMS (20 TRIALS)

Test case	Boruvka MST			Graph Search			B-F analysis		
	Runtime (ms)		SU	Runtime (ms)		SU	Runtime (ms)		SU
	CPU	GPU		CPU	GPU		CPU	GPU	
14	2.8	0.10	29x	1.7	0.03	65x	14.7	1.7	8x
33	6.1	0.15	40x	3.9	0.06	68x	41.8	3.6	12x
70	12.5	0.25	50x	8.5	0.05	182x	74.3	2.7	27x
83	14.6	0.30	48x	9.8	0.05	200x	107.2	3.4	32x
136	24.2	0.63	38x	21.4	0.10	222x	153.4	3.9	39x
415	64.0	2.82	23x	74.4	0.12	625x	593.2	8.2	73x
880	114.5	3.03	38x	245.8	0.73	335x	1025.5	14.7	70x
1760	229.0	8.27	28x	532.8	1.69	315x	2148.0	31.8	67x
4400	624.0	55.14	11x	1405.7	4.12	342x	5587.5	78.1	72x

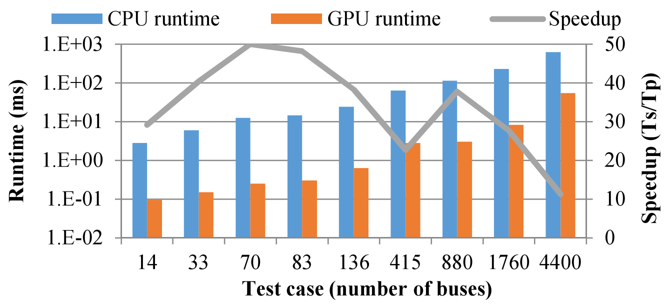

Fig. 14. Average Runtime and Speedup for the Boruvka Minimum Spanning Tree Algorithm (20 trials).

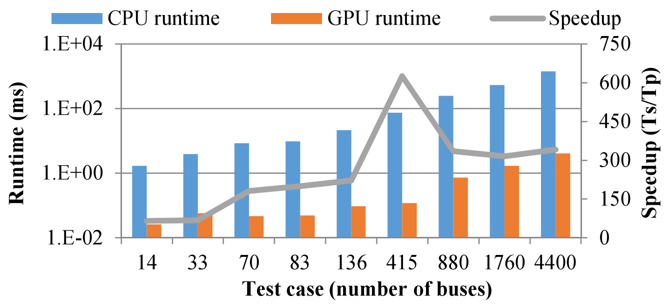

Fig. 15. Average Runtime and Speedup for the Graph Search Algorithm (20 trials).

other data is created and kept on the GPU. At the very end, once the iterative process of the GA has completed, only the network topology with the associated bus voltages are transferred back to the CPU. Throughout the optimization process, no data transfer is required. This data management strategy avoids the bottleneck of the PCIe bus and ensures maximum performance.

For the parallel MST, we note in Table III a maximum speedup of 50x which is three times better than in [23]. This is due to the batch operation of the proposed implementation which allows for an efficient use of the GPU shared memory and the execution of the entire algorithm in a single CUDA *kernel*. However, for the case of the 4400-bus network, the data is too large to fit in shared memory and must be stored in the GPU global memory resulting in a lower, but still very gainful, speedup of 11x.

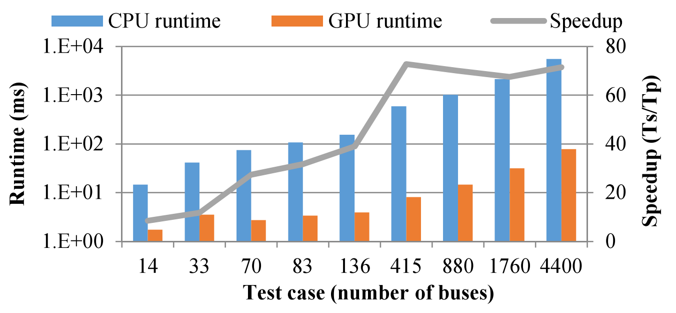

Fig. 16. Average Runtime and Speedup for the Backward-Forward Algorithm (20 trials).

The quadratic parallelization strategy used for the graph traversal module revealed to be extremely advantageous. As we can see in Table III, the speedup gained reached 625x compared to a sequential DFS. In fact, because all branches are processed concurrently, the number of steps required to traverse the graph is equal to the maximum depth, much less than with a DFS.

The advantage of the parallelization on GPU is also significant for the concurrent power flow analysis of multiple networks. As shown in Figure 16, the acceleration reaches 73 folds for the 415-bus network. Finally, although the parallelization for each of the three operations is critical to the overall performance of the proposed DFR solver, the parallel B-F module is the most effecting one to reduce the overall calculation time as calculating the power flow represents the most time-consuming part of the fitness evaluation of the candidate solutions.

B. Optimal reconfigurations

In the second test, the GA-based solver is run to compute the optimal reconfiguration that minimizes the real power losses. The numbers of candidate solutions and iterations used by the GA was set experimentally for each network based on their complexity. These configurations are listed in Table II. The other GA parameters such as the mutation probability are given in Section IV-A. In Table IV, we list the summary of the solutions found by the proposed method and those of other references. The details of the solutions vectors found, are available online at [41]. From the results obtained, we see that the proposed algorithm finds solutions of equal or better quality for all test cases. We also note that the execution of the proposed solver is significantly faster, up to 266 times, than the references cited which all used sequential implementations on CPU. Even though each reference used different computer hardware, the performance gap is large enough to clearly show the advantage of the parallel implementation on GPU. Finally, due to the unique solution encoding, the proposed solver is able to efficiently reconfigure networks five times larger than any of the references surveyed.

C. Speedup of the proposed GA-based solver on GPU

Finally, in the third test, we run 20 trials with different random number generator seeds for the parallel GA-based solver on GPU for each network and list in Table V the average runtimes and the average real power losses of the final solutions. To make the comparison more interesting, we also include in Table V the runtime of a parallel implementation on multi-core CPU. This parallelization is achieved by sharing the fitness evaluation of the candidate solutions between multiple cores using OpenMP®. For this experimentation, 32 threads were launched in order to make full use of the 16-core multithreaded computer system. All runtimes include the data transfers between the CPU and the GPU at the beginning and at the end of the optimization process. From the table, we note that the maximum speedup on multicore CPU is 13.4x while 66.2x can be achieved on the GPU demonstrating

TABLE IV
COMPARISON OF THE SOLUTIONS OBTAINED BY THE PROPOSED ALGORITHM AND OTHER REFERENCES WHEN MINIMIZING REAL POWER LOSSES (P_{loss})

Test case	Source	P_{loss} (kW)	Q_{loss} (kVAR)	V_{min} (p.u.)	V_{avg} (p.u.)	Runtime (s)
16-bus	Initial	511.4	590.4	0.969	0.985	-
	ACO [9]	466.1	544.9	0.972	0.987	1.81
	GA [10]	466.1	544.9	0.972	0.987	2.1
	GA-GPU	466.1	544.9	0.972	0.987	0.02
33-bus	Initial	211.0	143.0	0.904	0.945	-
	AIS [11]	139.6	102.3	0.938	0.965	16.9
	PSO [12]	139.6	102.3	0.938	0.965	5.69
	GA-GPU	139.6	102.3	0.938	0.965	0.09
70-bus	Initial	227.5	204.9	0.905	0.950	-
	GA [10]	203.2	186.6	0.931	0.953	4.64
	GA [45]	203.9	191.1	0.927	0.953	160
	GA-GPU	201.4	185.1	0.931	0.954	0.53
83-bus	Initial	532.0	1374.3	0.929	0.970	-
	PSO [12]	471.1	1252.1	0.952	0.972	36.1
	AIS [11]	469.9	1248.0	0.953	0.972	160
	GA-GPU	469.9	1248.0	0.953	0.972	0.50
136-bus	Initial	320.3	702.7	0.931	0.975	-
	GA [46]	280.7	611.0	0.961	0.977	32.6
	MICP [5]	280.1	611.1	0.959	0.977	1800
	GA-GPU	280.1	611.1	0.959	0.977	1.1
415-bus	Initial	2660.0	6871.6	0.929	0.969	-
	MICP [5]	2359.9	-	-	-	1800
	MILP [5]	2350.7	-	-	-	1800
	GA-GPU	2349.4	6240.0	0.953	0.972	9.3
880-bus	Initial	1496.4	1396.5	0.956	0.987	-
	MIQP [6]	461.4	-	0.982	0.990	3192
	MIQP [7]	461.0	566.7	0.992	0.995	1134
	GA-GPU	457.0	563.3	0.992	0.995	12.0
1760-bus	Initial	2992.9	2793.0	0.956	0.987	-
	GA-GPU	821.7	1014.9	0.992	0.996	39.3
4400-bus	Initial	7482.2	6982.5	0.956	0.987	-
	GA-GPU	1905.3	2399.5	0.992	0.996	226.2

TABLE V
AVERAGE RUNTIME OF THE PROPOSED GPU-BASED SOLVER AND AVERAGE POWER LOSSES OF THE FINAL SOLUTIONS (20 TRIALS)

Test case	Avg runtime (s)			Speedup		P_{loss} (kW)	
	CPU	OMP	GPU	OMP	GPU	Avg	Std
16	0.05	0.01	0.02	5.2x	2.2x	466.1	0.00
33	0.69	0.06	0.09	10.6x	7.3x	140.3	0.93
70	14.1	1.2	0.53	12.1x	26.5x	201.4	0.13
83	17.3	1.4	0.50	12.4x	35.5x	469.9	0.00
136	51.9	4.3	1.1	12.2x	45.9x	280.2	0.05
430	532.1	39.8	9.3	13.4x	58.8x	2,349.5	0.11
880	793.8	63.6	12.0	12.5x	66.2x	457.2	0.61
1760	2,488.7	195.7	39.3	12.7x	63.4x	823.2	0.80
4400	10,806.1	931.2	226.2	11.6x	47.8x	1911.8	3.31

its significant advantage for DFR. Also, the average power losses show that the solutions listed in Table IV were not hand-selected, but representative of the effectiveness of the proposed solver.

VII. Conclusion

This paper presented a parallel genetic algorithm on GPU for distribution feeder reconfiguration (DFR). This is a significant contribution as it is the very first time this problem is solved using graphic processors. Designed with thousands of cores, the GPU provided an impressive speedup of 66.2x compared to a sequential implementation on CPU. This acceleration is a true benefit as it allows for a fast reconfiguration of the distribution network in order to always operate in a highly optimal topology following a disturbance, a failure or simply a demand fluctuation. In addition to the parallelization, this paper makes a second contribution by proposing a unique solution encoding based on the minimum spanning tree in order to maintain a radial topology when generating candidate solutions. The proposed encoding revealed itself to be very efficient and allowed the metaheuristics to solve networks five times larger than any of the reference papers surveyed. The proposed solver was tested on networks ranging from 16 to 4400 buses and found solutions with smaller power losses, while keeping a much smaller execution time than other references.

References

[1] P. L. Cavalcante *et al.*, "Centralized self-healing scheme for electrical distribution systems," *IEEE Trans. Smart Grid*, [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=7175042&sortType%3Dasc_p_Sequence%26filter%3DAND%28p_Publication_Number%3A5165411%29%26pageNumber%3D6

[2] L. Tang, F. Yang, and J. Ma, "A survey on distribution system feeder reconfiguration: Objectives and solutions," in *Proc. IEEE Innov. Smart Grid Technol. (ISGT)*, Kuala Lumpur, Malaysia, 2014, pp. 62–67.

[3] Q. Peng and S. H. Low, "Optimal branch exchange for feeder reconfiguration in distribution networks," in *Proc. IEEE 52nd Annu. Conf. Decis. Control (CDC)*, Florence, Italy, 2013, pp. 2960–2965.

[4] F. Ding and K. A. Loparo, "A simple heuristic method for smart distribution system reconfiguration," in *Proc. IEEE Energytech*, Cleveland, OH, USA, May 2012, pp. 1–6.

[5] R. A. Jabr, R. Singh, and B. C. Pal, "Minimum loss network reconfiguration using mixed-integer convex programming," *IEEE Trans. Power Syst.*, vol. 27, no. 2, pp. 1106–1115, May 2012.

[6] J. A. Taylor and F. S. Hover, "Convex models of distribution system reconfiguration," *IEEE Trans. Power Syst.*, vol. 27, no. 3, pp. 1407–1413, Aug. 2012.

[7] H. Ahmadi and J. R. Marti, "Distribution system optimization based on a linear power-flow formulation," *IEEE Trans. Power Del.*, vol. 30, no. 1, pp. 25–33, Feb. 2015.

[8] H. M. Khodr, J. Martinez-Crespo, M. A. Matos, and J. Pereira, "Distribution systems reconfiguration based on OPF using benders decomposition," *IEEE Trans. Power Del.*, vol. 24, no. 4, pp. 2166–2176, Oct. 2009.

[9] C.-F. Chang, "Reconfiguration and capacitor placement for loss reduction of distribution systems by ant colony search algorithm," *IEEE Trans. Power Syst.*, vol. 23, no. 4, pp. 1747–1755, Nov. 2008.

[10] B. Enacheanu *et al.*, "Radial network reconfiguration using genetic algorithm based on the matroid theory," *IEEE Trans. Power Syst.*, vol. 23, no. 1, pp. 186–195, Feb. 2008.

[11] L. W. de Oliveira *et al.*, "Artificial immune systems applied to the reconfiguration of electrical power distribution networks for energy loss minimization," *Int. J. Elect. Power Energy Syst.*, vol. 56, pp. 64–74, Mar. 2014.

[12] W.-C. Wu and M.-S. Tsai, "Application of enhanced integer coded particle swarm optimization for distribution system feeder reconfiguration," *IEEE Trans. Power Syst.*, vol. 26, no. 3, pp. 1591–1599, Aug. 2011.

[13] H. B. Tolabi, M. H. Ali, S. B. M. Ayob, and M. Rizwan, "Novel hybrid fuzzy-Bees algorithm for optimal feeder multi-objective reconfiguration by considering multiple-distributed generation," *Energy*, vol. 71, pp. 507–515, Jul. 2014.

[14] S. H. Mirhoseini, S. M. Hosseini, M. Ghanbari, and M. Ahmadi, "A new improved adaptive imperialist competitive algorithm to solve the reconfiguration problem of distribution systems for loss reduction and voltage profile improvement," *Int. J. Elect. Power Energy Syst.*, vol. 55, pp. 128–143, Feb. 2014.

[15] B. Amanulla, S. Chakrabarti, and S. N. Singh, "Reconfiguration of power distribution systems considering reliability and power loss," *IEEE Trans. Power Del.*, vol. 27, no. 2, pp. 918–926, Apr. 2012.

[16] A. Mazza, G. Chicco, and A. Russo, "Optimal multi-objective distribution system reconfiguration with multi criteria decision making-based solution ranking and enhanced genetic operators," *Int. J. Elect. Power Energy Syst.*, vol. 54, pp. 255–267, Jan. 2014.

[17] V. Roberge and M. Tarbouchi, "Efficient parallel particle swarm optimizers on GPU for real-time harmonic minimization in multilevel inverters," in *Proc. 38th Annu. Conf. IEEE Ind. Electron. Soc. (IECON)*, Montreal, QC, Canada, 2012, pp. 2275–2282.

[18] V. Roberge, M. Tarbouchi, and F. Okou, "Strategies to accelerate harmonic minimization in multilevel inverters using a parallel genetic algorithm on graphical processing unit," *IEEE Trans. Power Electron.*, vol. 29, no. 10, pp. 5087–5090, Oct. 2014.

[19] L. Dawson and I. A. Stewart, "Accelerating ant colony optimization-based edge detection on the GPU using CUDA," in *Proc. IEEE Congr. Evol. Comput. (CEC)*, Beijing, China, Jul. 2014, pp. 1736–1743.

[20] V. Roberge, M. Tarbouchi, and F. Okou, "Collaborative parallel hybrid metaheuristics on graphics processing unit," *Int. J. Comput. Intell. Appl.*, vol. 14, no. 1, Mar. 2015, Art. ID 1550002.

[21] A. Swarnkar, N. Gupta, and K. R. Niazi, "Distribution network reconfiguration using population-based AI techniques: A comparative analysis," in *Proc. IEEE Power Energy Soc. Gen. Meeting*, San Diego, CA, USA, Jul. 2012, pp. 1–6.

[22] E.-G. Talbi, *Metaheuristics: From Design to Implementation* (Wiley Series on Parallel and Distributed Computing). Hoboken, NJ, USA: Wiley, 2009.

[23] C. da Silva Sousa, A. Mariano, and A. Proença, "A generic and highly efficient parallel variant of Boruvka's algorithm," presented at the 23rd Euromicro Int. Conf. Parallel Distrib. Netw. Based Process., Turku, Finland, 2015.

[24] D. Shirmohammadi, H. W. Hong, A. Semlyen, and G. X. Luo, "A compensation-based power flow method for weakly meshed distribution and transmission networks," *IEEE Trans. Power Syst.*, vol. 3, no. 2, pp. 753–762, May 1988.

[25] M. F. AlHajri and M. E. El-Hawary, "Exploiting the radial distribution structure in developing a fast and flexible radial power flow for unbalanced three-phase networks," *IEEE Trans. Power Del.*, vol. 25, no. 1, pp. 378–389, Jan. 2010.

[26] D. Ablakovic, I. Dzafic, and S. Kecici, "Parallelization of radial three-phase distribution power flow using GPU," in *Proc. 3rd IEEE PES Int. Conf. Exhibit. Innov. Smart Grid Technol. (ISGT Europe)*, Berlin, Germany, Oct. 2012, pp. 1–7.

[27] C. Guo *et al.*, "Performance comparisons of parallel power flow solvers on GPU system," in *Proc. IEEE 18th Int. Conf. Embedded. Real-Time Comput. Syst. Appl. (RTCSA)*, Seoul, Korea, Aug. 2012, pp. 232–239.

[28] V. Roberge, M. Tarbouchi, and F. Okou, "gpuMF: A framework for parallel hybrid metaheuristics on GPU with application to the minimisation of harmonics in multilevel inverters," *Int. J. Process Syst. Eng.*, vol. 3, nos. 1–3, pp. 20–41, 2015.

[29] N. Melab, T. V. Luong, K. Boufaras, and E.-G. Talbi, "ParadisEO-MO-GPU: A framework for parallel GPU-based local search metaheuristics," in *Proc. 15th Annu. Conf. Genet. Evol. Comput.*, Amsterdam, The Netherlands, 2013, pp. 1189–1196.

[30] S.-W. Ha and T.-D. Han, "A scalable work-efficient and depth-optimal parallel scan for the GPGPU environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 24, no. 12, pp. 2324–2333, Dec. 2013.

[31] W. Wang, S. Guo, F. Yang, and J. Chen, "GPU-based fast minimum spanning tree using data parallel primitives," in *Proc. 2nd Int. Conf. Inf. Eng. Comput. Sci. (ICIECS)*, Wuhan, China, Dec. 2010, pp. 1–4.

[32] S. Rostrup, S. Srivastava, and K. Singhal, "Fast and memory-efficient minimum spanning tree on the GPU," *Int. J. Comput. Sci. Eng.*, vol. 8, no. 1, pp. 21–33, Feb. 2013.

[33] V. Vineet, P. Harish, S. Patidar, and P. J. Narayanan, “Fast minimum spanning tree for large graphs on the GPU,” in *Proc. Conf. High Perform. Graph.*, New Orleans, LA, USA, 2009, pp. 167–171.

[34] M. Daga, M. Nutter, and M. Meswani, “Efficient breadth-first search on a heterogeneous processor,” in *Proc. IEEE Int. Conf. Big Data*, Washington, DC, USA, Oct. 2014, pp. 373–382.

[35] S. Dashora and N. Khare, “Implementation of graph algorithms over GPU: A comparative analysis,” in *Proc. IEEE Stud. Conf. Elect. Electron. Comput. Sci. (SCEECS)*, Bhopal, India, Mar. 2012, pp. 1–8.

[36] D. S. Banerjee, S. Sharma, and K. Kothapalli, “Work efficient parallel algorithms for large graph exploration,” in *Proc. 20th Annu. Int. Conf. High Perform. Comput. (HIPC)*, Bengaluru, India, Dec. 2013, pp. 433–442.

[37] D. Merrill, M. Garland, and A. Grimshaw, “Scalable GPU graph traversal,” in *Proc. 17th ACM SIGPLAN Symp. Prin. Pract. Parallel Program.*, New Orleans, LA, USA, 2012, pp. 117–128.

[38] P. Harish and P. J. Narayanan, “Accelerating large graph algorithms on the GPU using CUDA,” in *Proc. 14th Int. Conf. High Perform. Comput.*, Goa, India, 2007, pp. 197–208.

[39] Y. Deng, B. D. Wang, and S. Mu, “Taming irregular EDA applications on GPUs,” in *IEEE ACM Int. Conf. Comput.-Aided Design Tech. Dig. Papers (ICCAD)*, San Jose, CA, USA, Nov. 2009, pp. 539–546.

[40] CUSP. [Online]. Available: https://developer.nvidia.com/cusp, accessed May 29, 2015.

[41] V. Roberge. *Distribution Feeder Reconfiguration Test Cases*. [Online]. Available: http://roberge.segfaults.net/joomla/index.php/dfr, accessed May 7, 2015.

[42] *Boost C++ Libraries*. [Online]. Available: http://www.boost.org/, accessed Aug. 26, 2015.

[43] D. Merrill. *NVIDIA CUB*. [Online]. Available: http://nvlabs.github.io/cub/index.html, accessed Jan. 13, 2015.

[44] B. Tomoiagă, M. Chindriş, A. Sumper, R. Villafafila-Robles, and A. Sudria-Andreu, “Distribution system reconfiguration using genetic algorithm based on connected graphs,” *Elect. Power Syst. Res.*, vol. 104, pp. 216–225, Nov. 2013.

[45] A. M. Eldurssi and R. M. O’Connell, “A fast nondominated sorting guided genetic algorithm for multi-objective power distribution system reconfiguration problem,” *IEEE Trans. Power Syst.*, vol. 30, no. 2, pp. 593–601, Mar. 2015.

Authors’ photographs and biographies not available at the time of publication.