A Successful Parallel Implementation of NSGA-II on GPU for the Energy Dispatch Problem on Hydroelectric Power Plants

[1]Lucas B. de Oliveira, [1]Carolina G. Marcelino, [1]Anolan Milanés, [1]Paulo E. M. Almeida and [2]Leonel M. Carvalho
[1]Computer Engineering Departament, CEFET-MG, Belo Horizonte, Brasil
[2]Centre for Power and Energy Systems - CPES, [2]INESC TEC - Portugal
Email: [1]lucas.braga.deo@gmail.com, [2]lcarvalho@inesctec.pt

***Abstract*—Nowadays, hydraulic sources are responsible for most of the Brazil's energy production. Hydroelectric power plants (HPP) operators in Brazil usually distribute equally the total power required among the generator units available in the plant. However, studies show that this configuration does not guarantee that each generator unit operate close to its optimal operation point. The energy dispatch optimization problem consists in determining which generation units need to be on or off and what is their respective power-set, so that both the overall HPP costs is minimized and the power required by the plant is met. This paper presents a parallel implementation of NSGA-II on GPU, to solve the energy dispatch problem of a HPP complaying with the real time restrictions posed by the operation of a real HPP from the reception of the power demand to the energy dispatch. Our implementation obtains better solutions than the sequential implementation currently available.**

I. Introduction

Economy and population growth in Brazil contributes to an increasing demand to its electric energy production. According to the 2014 annual report of the Brazilian Energy Planning Company ("Empresa de Planejamento Energético"- EPE, in Portuguese), most of the country's energy is produced by renewable sources, hydraulic sources being responsible by 70.6% of it, as shown on Table I. Hydroelectric power plants (HPP) operate to supply an electrical demands requested by National Electric System Operator (Operador Nacional do Sistema Elétrico - ONS), a government agency which coordinates and controls the electrical energy production and its transmission.

TABLE I
Brazil's Domestic Electricity Supply by Source on 2014 [1]

Source	Percentage
Hydro	70.6%
Natural Gas	11.3%
Biomass	7.6%
Oil Products	4.4%
Coal and Coal Products	2.6%
Nuclear	2.4%
Wind	1.1%

An HPP is composed by several turbines connected to electrical generators (generator units). Most hydroelectric plants in Brazil operate by equally distributing equally distribute the total power required by ONS among the generator units available in the plant. However, studies of Marcelino et al. [2] [3] show that this equal dispatch does not represent optimal efficiency, since it does not make each generator unit operating close to its optimal operation point. In both works the objective of the energy dispatch optimization problem consisted on determining which generation units need to be on or off and what is their respective power set-point (in MW), so that the overall HPP costs is minimized while able to meet the power required by the plant.

The energy dispatch problem is a significant matter, considering that even small improvements in efficiency can reduce the water demand by the HPP. In consequence, the cost is reduced, allowing a higher power generation. This topic has been covered by different approaches, including linear programming, lagrangian relaxation and neural networks [4]–[8].

A mono-objective mathematical model was proposed by Marcelino et al. [2] to solve the HPP energy dispatch problem. This model was solved using an EA called Differential Evolution (DE) [10]. [9]. The DE/best/1/bin strategy was the most efficient solution to this model.

Marcelino et al. [3] improved the mathematical model made in the previous work [2] by considering the same problem as a MO approach. A second objective was added to the model, in order to measure the distance between two operational modes. The first mode is the "Normal Mode of Operation" (NMO) which equally distributes the power request among the generator units. The second mode is the "Optimized Control Mode" (OCM) which is the optimized energy dispatch found by the first objective. The latter objective was proposed because the HPP technical staff was not used to employ OCM, therefore NMO was preferred. This objective shows that there are optimal operational points near NMO, increasing the staff's confidence on OCM.

In order to solve this new mathematical model, two well established multi-objective evolutionary algorithms were used, the NSGA-II [10] and SPEA2 [11]. Both algorithms were able to solve the energy dispatch problem with results near the operating points of NMO and with high productive efficiency. In Brazilian HPP, the technical staff must define how to dispatch the energy on plants' generator units in less than 10

seconds, right after receiving the power demand from ONS. The implementations of NSGA-II and SPEA2 algorithms in Marcelino et al. work [3] were made using Matlab scripts and focused mostly on quality of the solutions than execution time. Thus, those algorithms take around 5 minutes to execute, making it impossible to use them on a real HPP.

Like many other evolutionary algorithms, NSGA-II functions have an inherently parallelism on its behavior, considering that most of them iterates for each individual of a population and each iteration usually does not interfere with another. This makes the use of graphics processing units (GPU) an interesting solution, since its hardware is specialized in the execution of large quantities of threads simultaneously. Many studies proved that GPUs can provide great speedups to NSGA-II and other Multiobjective Evolutionary Algorithms (MOEA), as in test problems such as ZDT and DTLZ [12]–[14], as well as real world applications e.g. traffic light signaling optimization [15] and optimization of fuel treatment for mitigating wildfire hazard [16].

In this paper, we propose a parallel implementation of NSGA-II using a GPU to solve Marcelino et al. [3] mathematical model with an execution time short enough to be used on a HPP, i.e. less than 10 seconds, while not deteriorating the quality of the solutions. We first give an overview of GPU and Nvidia's platform Compute Unified Device Architecture (CUDA) capabilities on Section 2. In Section 3 we present the multi-objective model created by Marcelino et al. [3] for the HPP energy dispatch problem and how this model is solved with NSGA-II. We discuss our parallel implementation of NSGA-II in Section 4. The experiments and results are reported in Section 5. Finally, Section 6 presents the conclusion and possible future extensions.

II. GPU and CUDA Capabilities

Graphics processing units were originally conceived to supply the demand of multimedia, games and 3D rendering fields. The computation performed in these industries is heavily parallel, therefore GPU is focused on high throughput. Comparing the hardware of a GPU and a CPU, the latter allocates most of its space to control and cache units and the remaining space for its arithmetic logic units (ALUs). GPU, on the other hand, dedicate much more space to ALUs and a few to cache and control units. Figure 1 illustrates this comparison.

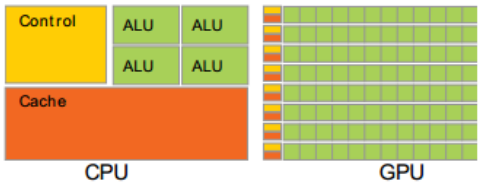

Fig. 1. Comparison of CPU and GPU hardware [17]

The increasing flexibility in GPUs architecture and software made the parallel power of GPUs available also to general purpose computations. CUDA [17] is the Nvidia's platform for general-purpose computing on their graphics processing units (GPGPU).

In CUDA, the instructions that run on GPU are structured in functions called kernels. Kernel's threads are organized by blocks, which are 3D structures that define the number of threads on each dimension. A kernel can be launched from the CPU or from another kernel executing on the GPU. Nvidia calls the latter "dynamic parallelism" [17]. It can be used to reduce the communication between CPU and GPU, which has a cost due to the lower bandwidth on this communication comparing to GPU and CPUs memory bandwidth.

Nvidia adopted a Single Instruction Multiple Threads (SIMT) execution model on their GPUs. Threads of a block are executed in sets of 32 threads called warps. All threads on a same warp must execute the same instruction at same time. When some threads in a warp need to execute a different instruction (eg. following an *if* clause), there is a warp divergence, and each group of threads in the warp will execute sequentially instead of simultaneously. In order to minimize warp divergence, branch instructions and loops with different number of iterations should be avoided.

CUDA defines a memory hierarchy that includes the global and the shared memory. The global memory can be accessed by all executing threads on the GPU and can be used to transfer data between the CPU and other GPUs. The shared memory is faster than the global but has smaller capacity. All threads on the same block can access it and the stored data in this memory is lost when the block execution ends. Moving data that is frequently used by the same block to shared memory can improve GPU performance. The memory access pattern is also highly important to kernel performance. A sequence of threads should load from memory in an aligned and sequential way called coalesced access.

III. Multi-objective Model for the Energy Dispatch Problem of an HPP and Proposed Solution

The multi-objective optimization model proposed by Marcelino et al. [3] has two objective. The first objective is to maximize the hydroelectric productivity of the plant using the objective function 1. The second objective is to minimize the distance between NMO and OCM using the objective function 2. The optimization variables are the water flow rate for each of the six generator units, represented by vector x:

$$x = [q_{1t}, q_{2t}, ..., q_{jt}],$$

and the bi-objective problem is described as:

$$\text{Maximize } F_1(x) = \frac{\sum_{j=1}^{J(r)} ph_{jt}}{\sum_{j=1}^{J(r)} q_{jt}}, \tag{1}$$

$$\text{Minimize } F_2(x) = \sqrt{\sum_{j=1}^{J(r)} (q_{jt} - q_{cc})^2}, \tag{2}$$

subject to:

$$\sum_{j=1}^{J(r)} ph_{jt} \cong Dm, q_{jt}min \leq q_{jt} \leq q_{jt}max, \quad (3)$$

$$ph_{jk}^{min} \sum_{k=1}^{\emptyset_j} Z_{jk} \leq ph_{jt} \leq ph_{jk}^{max} \sum_{k=1}^{\emptyset_j} Z_{jk}, \quad (4)$$

$$Z_{jk} \in \{0,1\}, \sum_{k=1}^{\emptyset_j} Z_{jk} \leq 1. \quad (5)$$

The model parameters are described in Table II and Table III presents the efficiency coefficients.

TABLE II
DESCRIPTION OF PARAMETERS USED IN MODEL [3].

Parameter	Description
$\boldsymbol{ph_{jt}}$	Power generated by unit j at time t
$\boldsymbol{ph_{jt}min}$	Minimum Power
$\boldsymbol{ph_{jt}max}$	Maximum Power
$\boldsymbol{Dm}$	Requested Demand (MW)
$\boldsymbol{q_{jt}}$	Water Discharge of unit j at time t
$\boldsymbol{q_{cc}}$	Water Discharge at NMO
$\boldsymbol{q_{jt}min}$	Minium Water Discharge
$\boldsymbol{q_{jt}max}$	Maximum Water Discharge
$\boldsymbol{Z_{jk}}$	Operative Zone of Generator Unit j at Time k

TABLE III
EFFICIENCY COEFFICIENTS [3].

Coefficients	Value	Coefficients	Value
p_{0j}	1,4630e-01	p_{3j}	-3,5254e-03
p_{1j}	1,8076e-02	p_{4j}	-1,1234e-03
p_{2j}	5,0502e-03	p_{5j}	1,4507e-05

The first objective function determines how much power the plant is able to produce with a given volume of water. Maximizing this function means a higher power production with less water. The numerator of F1 is the production function: as this number increases, the objective function value also increases. When the denominator of F1 is decreased, the productivity ratio is also reduced [3].

The second objective function, F2, measures the distance between the water discharge used in NMO and the one used in OCM. This shows that there are operation points in OCM, which are closer to NMO but still ensure the maximization of energy production. This contributes to a new culture development by the HPP operational staff, increasing their confidence on OCM [3].

The first constraint indicates that the power to be delivered should be equal to the power requested to the HPP. The second constraint states that the calculated flow rate must comply with the minimum and maximum flow capacites of each generation unit. The third constraint requires the corresponding generated power to comply with minimum and maximum power capacity of each generation unit. The fourth constraint ensures that each generation unit maintains its operating status, i.e. it stays on or off during the whole production period [3].

The proposed solution for a HPP consists in a software that receives a power demand as an input parameter, then thirty (30) independent NSGA-II runs solve the mathematical model. A new population is created combining the individuals on the first front of each NSGA-II run and a dominance routine is applied in order to generate a final Pareto front. A selectable-point graph is displayed for this final front, allowing the staff to choose a point, which indicates the water discharge values for each generator unit.

IV. PARALLEL IMPLEMENTATION OF NSGA-II ON GPU

A. Parallel Model

In the following sections we shall explain how we parallelized one NSGA-II execution, in Subsection **D** we describe how thirty (30) NSGA-II executions occur simultaneously. In order to create our parallel implementation of NSGA-II, we initially identified following steps in the algorithm:

1) Generate the initial parent population.
2) Evaluate the cost of the parent population individuals (fitness evaluation).
3) Separate the parent population into fronts with non-dominated sorting operator.
4) Evaluate the crowding distance of the parent population individuals.
5) Create the offspring population using individuals from the parent population, employing selection, crossover and mutation operators.
6) Evaluate the cost of the offspring population individuals.
7) Separate all individuals, i.e. parent and offspring, into fronts.
8) Evaluate the crowding distance of all individuals.
9) Select the best individuals based on their front and crowding distance values and move them to the parent population location. These individuals will then turn into the parent population of the next generation.
10) If the stopping criteria is satisfied, finish the algorithm, otherwise return to step 3.

Since some of those steps can be executed by the same function, we identified the following six tasks to be implemented:

1) Generate initial parent population.
2) Evaluate the costs of the individuals.
3) Separate the individuals into fronts with non-dominated sorting operator.
4) Evaluate crowding distance of the individuals.
5) Create offspring population.
6) Select the best individuals for the next generation.

One kernel was used for tasks 1, 2 and 5. Task 3 was subdivided into two different subtasks. The first one takes one kernel, which calculates: A) the domination count n_p, the number of other solutions that dominates the solution p, and; B) the dominated set S_p, which is a set of solutions dominated by p. After calculating these variables, the kernel defines the first front. The second subtask defines other fronts. We used one kernel that launches three other kernels with dynamic parallelism, whose functioning will be detailed later in Subsection **C**.

The subdivision explained above was made because the n_p and S_p calculations and the first front definition need data of all individuals in order to occur. The definition of other fronts consists in an iterative method, which requires data of the last defined front individuals. Therefore, the first subtask will always use a static number of threads, which can be either the number of individuals in parent population or the number of all individuals. The second subtask uses a dynamic number of threads and we used the dynamic parallelism to handle the thread number.

Tasks 5 and 6 also use dynamic parallelism. In Task 5, it is first necessary to sort the individuals based on their cost evaluated in Task 2, then calculate the crowding distance. The sorting phase is done on a separated kernel, which implements an insertion sort. This kernel is launched via dynamic parallelism by the main kernel of Task 5.

Task 6 is also subdivided into two subtasks, being the first one similar to Task 5. We first sort the individuals based on their crowding distance using the insertion sort kernel, then each thread storages their individual's new position on global memory. The second subtask uses one kernel to move the best individuals' costs and decision variables to the parent population positions.

B. Data Structure

In this work, we focused on shorter execution times, thus we used shared memory whenever it was beneficial. However, this created a limitation with this algorithm. As explained in Section 2, shared memory can only be accessed by threads within same block. In our implementation, most kernel threads represent one individual of the population, consequently the maximum number of individuals equals to the maximum number of threads population data is stored using six arrays in a block, which in current Nvidia's GPU is 1024.

- Position, for the decision variables;
- Cost, for fitness evaluation;
- Rank, in order to determine which front the individual belongs to;
- Crowding Distance, for density estimation;
- Dominated set, to identify which individuals an individual dominates;
- Domination count, to quantify how many individuals are dominated by another one.

These arrays are allocated in the GPU's global memory with a static size and all individuals share the same arrays. The data type chosen for these arrays and their size are presented by Table IV

TABLE IV
ARRAYS FOR INDIVIDUALS DATA

Array Name	Data Type	Array Size
Position	Double	6 * Total Polulation Size
Cost	Double	2 * Total Polulation Size
Rank	Unsigned Integer	Total Polulation Size
Crowding Distance	Double	Total Polulation Size
Dominated Set	Boolean	Total Polulation Size^2
Domination Count	Unsigned Integer	Total Polulation Size

The fronts created by the non-dominated sorting operator are stored using two integer arrays. One of them identify the front members and the other one receives the size of the fronts. In both arrays, elements are added in a sequential way, by using two integers in global memory that are manipulated with CUDA atomic operations, in order to avoid data race conditions. Figure 2 displays how these two arrays identifies the individuals on each front.

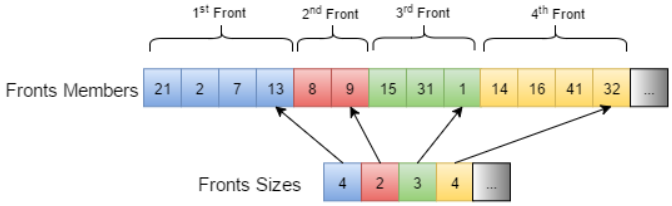

Fig. 2. Data Structure for Pareto Fronts

A similar structure is also used in crowding distance calculation and by selection of individuals for the next generation. Since both need to sort the fronts individuals by either the cost of an individual on each objective or by the crowding distance, we allocated two pairs of key-value arrays in global memory. Before the kernels of crowding distance and next generation selection, launches the insertion sort kernel, they make a copy of fronts members and their fitness value or crowding distance to this pair of arrays, so the fronts members can be sorted without affecting the original arrays.

C. Kernels Implementation

This section details some kernels implementation. The flowchart in Figure 3 presents the kernel names and, in parenthesis, on which population (parent, offspring or total) they are operating. The arrows and rectangles in green indicates that the kernel was launched via dynamic parallelism.

In order to explain how these kernels work, it is necessary to define the following variables:

- pop_pos, array for decision variables.
- pop_cost, array for fitness evaluation.
- pop_rank, array for individuals ranks.
- pop_cdist, array for individuals crowding distance.
- pop_dset, array for domination sets.
- pop_dcount, array for domination count.
- $f\_members$, array to identify members on each front.
- f_size, array of fronts sizes.
- f_index, integer to control where a new individual must be inserted on $f\_members$ and f_size.
- $c1\_key$, array of individuals' identification to be used on sorting.
- $c1\_value$, array of individuals' values to be used on sorting.
- $c2\_key$, array of individuals' identification to be used on sorting.
- $c2\_value$, array of individuals' values to be used on sorting.
- $n\_parents$, integer for the number of parent individuals.

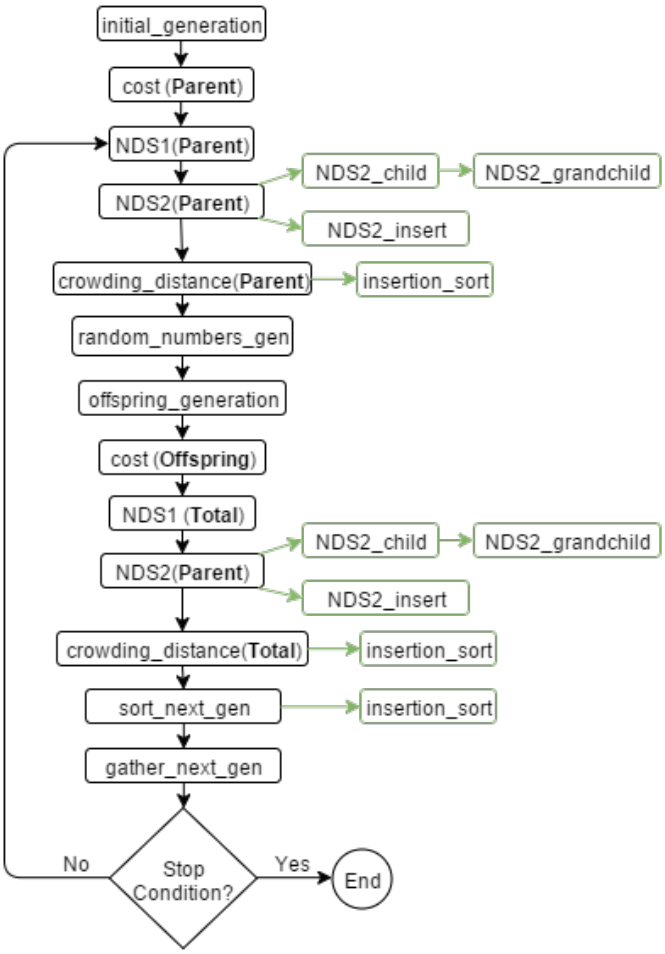

Fig. 3. Flowchart of Kernels

- $n\_offspring$, integer for the number of offspring individuals.
- n_total, integer for the sum of two previous values.

The $initial\_generation$ kernel creates the first parent population and each of this kernel's threads generates one random number within the range of 70-140 (range of water discharge on the generator units), which represents a decision variable. This value is stored in pop_pos array and the number of threads for this kernel is $6 * n\_total$.

The $cost$ kernel does the fitness evaluation of the individuals for each of the two objectives. The calculations done is this kernel are identical to the ones made in Matlab scripts by Marcelino work [3], except for some adaptations to code it in C programming language. After the fitness evaluation, the results are stored in pop_cost array. This kernel can operate on parent or offspring population, thus each kernel represents one individual and the number of threads is equal to $n\_parents$ or $n\_offspring$.

The $NDS1$ kernel calculates the domination sets and dominated counts for each individual, and then it defines the first front. The first step is making a copy of the population costs from pop_cost array to the shared memory, in order to reduce access time. In this kernel, each thread represents an individual. After initializing the shared memory, each thread calculates its individual domination count and dominated set. When all threads finish calculating these variables, each thread verifies if their domination count is zero. If it equals to zero, the thread sets its individual rank to 1, adds its individual identification number in the f_member array and increases by 1 a counter in shared memory, which will be used to quantify the number of individuals in the first front. The addition to $f\_members$ and counter is done using atomic operations. Finally, the individual rank, dominated set and domination count are stored on the pop_rank, pop_dset and pop_dcount arrays, respectively. The counter that quantifies the individuals on first front is stored in $f\_sizearray$. The number of threads equals to n_parent or n_total.

The $NDS2$ kernel is responsible to control the creation of other fronts. It controls the loop that determines if the process is completed or not. It also defines the parameters and the numbers of threads of $NDS2\_child$ and launch the $NDS2\_child$ and $NDS2\_insert$ kernels via dynamic parallelism. This kernel requires only one thread.

$NDS2\_child$ receives from the $NDS2$ kernel the number of the last front created and, with this number, it calculates the index of the first member of that front on $f\_members$ array, by adding up the sizes of the previous fronts in f_size array. With this index, each thread then identifies one member of the last front created and launches the $NDS2\_grandchild$ kernel, passing this member identification as a parameter. This kernel has a dynamic number of threads, which is equal to the number of members of the last front created. Each thread corresponds to one of these members.

$NDS2\_grandchild$ performs the subtraction of the domination count. Each individual has a set of positions in pop_dset array and each position identifies if another individual is dominated or not. For example: if position 3 has value as true, then the owner of that dominated set dominates the third individual. In this kernel each thread corresponds to one position on the dominated set of one individual. If their position value is true then the thread performs the subtraction on the domination count of the individual represented by this position. In order to avoid data race condition, the subtraction is done using atomic operations. The number of threads in this kernel equals to n_total. However, note that this kernel is launched using dynamic parallelism by each thread of $NDS2\_child$, thus there are multiple kernels running simultaneously.

After $NDS2\_child$ and $NDS2\_grandchild$ finish their execution, the $NDS2$ kernel launches $NDS2\_insert$. This kernel verifies which individuals have their domination count on zero and adds them to f_member array. It sets their rank in pop_rank array and increases the counter of this new front on f_size array. This verification is done on all individuals of the population, therefore the number of threads is equal to n_parent or n_total and each thread represents one individual.

After $NDS2$ finishes the non-dominating sorting process,

the crowding distance is calculated for each individual by a namesake kernel ($crowding\_distance$). In this kernel each thread represents one individual. First, the threads initialize the $c1\_key$, $c1\_value$, $c2\_key$ and $c2\_value$ arrays, by copying the f_member array to $c1\_key$ and $c2\_key$; and copying each cost from pop_cost array to $c1\_value$ and $c2\_value$; $c1\_value$ receives cost for the first objective and $c2\_value$ receives the cost for the second objective. After that, each thread verifies if their corresponding individual is the first individual of their front. If it is, then the thread launches two $insertion\_sort$ kernels, one receives $c1\_key$ and $c1\_value$ arrays and the other receives $c2\_key$ and $c2\_value$. All threads wait for the conclusion of all $insertion\_sort$ kernels. After the threads, we copy the sorted arrays to the shared memory. Finally, each thread calculates their individual crowding distance using the sorted arrays. Then they store the value in pop_cdist array. The number of threads in this kernel is either equal to n_parent or n_total.

The $insertion\_sort$ kernel is a serial implementation of a namesake sorting algorithm. We choose this algorithm and the serial version because of the relatively small size of the arrays that need to be sorted. Only one thread is necessary when this kernel is launched, yet multiple kernels may be launched at same time by $crowding\_distance$ and $sort\_next\_gen$ kernels.

The $random\_number\_gen$ kernel generates the random numbers used by the selection, crossover and mutation operators in $offspring\_generation$ kernel. We used the cuRAND library [17] with MTGP32 generator. With this library, we can generate random numbers directly on GPU, which is faster than generating those in CPU then copy them to GPU. All numbers are stored on global memory and on each NSGA-II iteration new numbers are generated.

The $offspring\_generation$ kernel creates the offspring population by combining the selection, crossover and mutation operators in a single kernel. Each thread of this kernel creates two offspring individuals. First, the threads copy the pop_pos, pop_rank and pop_cdist arrays to shared memory. Then two pairs of parent individuals are chosen using the random numbers generated on previous kernel. For each pair, we select the lower ranked parent. If ranks are equal, the one with higher crowding distance is selected. The chosen parents are used on the crossover, in order to generate two offspring individuals, then each one of these may be mutated. Finally, offspring individuals are stored in pop_pos array. Since each thread creates two offspring individuals, the number of threads is half of $n\_offspring$.

The $sort\_next\_gen$ is similar to $crowding\_distance$ kernel, but the threads initializes the $c1\_key$ and $c1\_value$ arrays, by copying $f\_members$ to $c1\_key$ and pop_cdist to $c1\_values$. Then for each front only one $insertion\_sort$ will be launched. The sorted $c1\_key$ array is used on next kernel. The number of threads in this kernel is n_total.

Finally, $gather\_next\_gen$ moves the best individuals to the parent population, using the parallel patern called gather, so that those individuals are used on next iteration of NSGA-II. Given a collection of locations (addresses or arrays indices) and a source array, the gather pattern collects all the data from the source array at given locations and places them into an output collection [18]. This kernel uses the $c1\_key$ array as the collection of locations, the $c1\_key$ array is already sorted by the fronts and each front is sorted by the crowding distance of its members. Therefore the individuals on the first fronts and with highest crowding distance are moved. The source and output arrays are pop_pos, pop_cost and pop_rank; since the source and output are the same, we first copy all data to the shared memory and then write it back on the given locations, so that we avoid data race conditions. Each thread on this kernel represents one individual and the number of threads is equal to n_parent.

D. Multiple Runs of NSGA-II

In section B, we explained why we used shared memory and how that limits our population size to 1024 and kernels blocks to 1. In order to parallelize multiple runs of NSGA-II, we made that each kernel block is an NSGA-II run.

The only change in the data structure is the array sizes, which were multiplied by the number of NSGA-II runs, i. e. 30. Each run has its own range of addresses, so that one run does not access the addresses of another, for instance: with a population size of 100 individuals, the NSGA-II run number 1 uses the 0-599 indices of pop_pos array, the run number 2 uses the 600-1199 and so on.

V. Experiments and Results

Our experiments compare the two solutions to the HPP energy dispatch problem: the one made by Marcelino et. al. [3] with Matlab scripts and our implementation with CUDA. The approach considers execution time as well as the quality of the solutions. The parameters for NSGA-II are the same used in the experiments of Marcelino et. al. [3], which are: power demand of 320MW, parent population size of 50, offspring population size of 40, mutation probability of 2%, 30 NSGA-II runs and 50 iterations.

We used an Nvidia Geforce GTX 980 GPU, which has 2048 cores operating at 1,126GHz; an Intel Core i7 4790 CPU, which has 4 cores with hyperthreading operating at 3,6GHz. The CUDA toolkit version is the 7.0, Matlab version is the 2014a and the operational system is CentOS 7.

Figure 4 presents the final Pareto front, representing nondominated frontier of the 30 executions, created with the individuals on the Pareto front of each NSGA-II run,for Matlab and CUDA implementations. In this graph, the horizontal axis represents the energy efficiencyand vertical axis represents the distance between NMO and OCM. The Pareto front curve from the CUDA implementation is very similar to the curve from Matlab and it has a slight advantage at some points. This behavior suggests that our CUDA implementation maintains the quality of solution found by Matlab.In order to validate the quality of CUDA solutions, we performed a statistical analysis using S-metric of each NSGA-II run of both implementations and ANOVA with Tukey test.

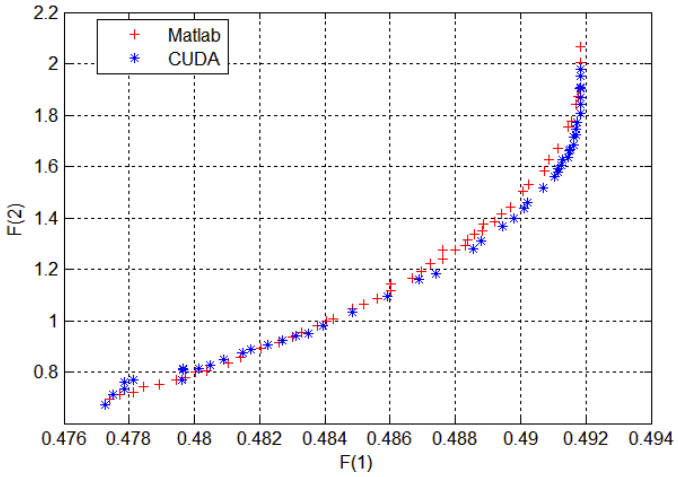

Fig. 4. Final Pareto Front Generated by CUDA and Matlab implementations

The S-metric is a quality measure to compare Pareto fronts generated by multiobjective optimizers [19]. This metric calculates a hypervolume of a region delimited by a reference point β, thus calculating a region that β dominates; a higher hypervolume means higher quality solutions.

ANOVA is a statistical technique that evaluates hypothesis about the population means. This analysis assumes that chance only produces small deviations, the major differences being generated by real causes. The null and alternative hypothesis to verify the variance analysis are: Null hypothesis, H0, in which the population means are equal. As for the alternative hypothesis, H1, the population means are different, i.e., at least one of the means is different from the others [20].

TABLE V
ANOVA TEST RESULTS

Source	SS	df	MS	F	Prob>F
Columns	3,2982	1	3,29816	20,03	**3,61462e-05**
Error	9,549	58	0,16464		
Total	12,8472	59			

The ANOVA results indicates that there is a difference between the two implementations, because the result of P-Value is found lower than the significance level adopted (equal to 0.05) in this test as shown in Table V, however this test does not determine this difference. Given its ability to analyze multiple data sets, this study used ANOVA with Tukey test or Honestly Significant Difference (HSD) [21] in order to find some information that differentiates the implementations.

The HSD test performed indicates with 95% confidence that CUDA implementation presents better mean results compared to Matlab's. Therefore, our CUDA produces solutions with higher quality than Matlab's.

During the development, we noticed that the random number generator affects the quality of the solutions obtained by NSGA-II. There are three different pseudorandom generators in cuRAND library [17]: XORWOW, MRG32k3a and the MTGP32, which is an adaption of Mersenne Twister generator [22] for GPU. XORWOW and MRG32ka generators are not able to produce solutions as good as Matlab's. Therefore, we believe that the only reason why our CUDA implementation is capable of generating better solutions is the MTGP32 generator.

In order to compare the execution time of each implementation, we ran both 20 times and calculated their averages, as displayed on Table VI. Note that each test on both implementations includes 30 NSGA-II runs. Therefore, we ran NSGA-II 600 times for each implementation and the average time is the time to complete 30 NSGA-II runs. The CUDA implementation provides a speedup of 284.72 to the Matlab one, however this comparison is not completely fair, since Matlab scripts are interpreted while CUDA code is compiled. Our implementation average time also satisfies the 10 seconds restriction from the HPP, thus a real HPP can apply our solution.

TABLE VI
AVERAGE EXECUTION TIME OF CUDA AND MATLAB IMPLEMENTATIONS

	Average (s)	**Deviation (s)**
Matlab	355.627	10.69
CUDA	1.249	0.0152

We also analyzed how our implementation performs with higher population sizes regarding execution time, GPU usage and memory usage. For each population size, we performed 20 runs. Table VII presents the results of this experiment.

TABLE VII
CUDA IMPLEMENTATION SCABILITY

	Execution Time			
Population Size	**Average (s)**	**Deviation (s)**	**Memory Usage (MB)**	**GPU Usage**
90	1.249	0.0152	660	99%
128	1.68	0.0625	661	99%
256	12.781	1.1562	664	99%
384	31.863	2.0628	668	99%
512	52.396	1.5539	673	99%

We can observe that CUDA implementation is still faster than Matlab's, even with a population of 512 individuals, although only the population size of 128 is viable to use on a HPP. The memory usage reaches high values during the execution and there is a small variance for each test. This behavior is due to the way we used the dynamic parallelism: when a parent kernel launches a child kernel and then makes a synchronization call, the GPU needs to allocate some memory in order to handle the switch of these kernels, thus most of GPU memory is used to handle the dynamic parallelism.

Using Nvidia Visual Profile [23] we analyzed the percentage of each kernel on the execution and the results are shown in Table VIII. Note that the percentages of the kernels consider the time from when the kernel starts until it ends, therefore part of the percentage of the kernels that uses dynamic parallelism is caused by its child kernels.

In Table VIII, it is noticeable that the kernels responsible for the non-dominated sorting operator are the ones that occupy most of the execution time. This is an expected result

since this operator is the one that defines NSGA-II complexity of $O(MN^2)$ (where M is the number of objectives and N the population size).

TABLE VIII
KERNEL PROFILING

Kernel Name	
NDS2	65.7%
·NDS2_child	54%
··NDS2_grandchild	13.9%
·NDS2_insert	2.7%
crowding_distance	19.8%
·insertion_sort	15.2%
sort_next_gen	7.2%
·insertion_sort	5.5%
random_numbers_gen	4%
cost	2.6%
NDS1	0.6%
offspring_generation	0.1%
gather_next_gen	0%
initial_generation	0%

VI. CONCLUSION

This paper presented a parallel implementation of NSGA-II on the GPU, to tackle the energy dispatch problem of a HPP using the mathematical model developed by Marcelino et. al. [3]. The goal of this parallel implementation is to reduce the execution time to fullfil the operating restrictions of a real HPP, that is, a maximum of 10 seconds to dispatch the energy after receiving a power demand. Our implementation is capable of executing 30 simultaneous runs of NSGA-II in less than 1.29 seconds on average, with better solutions than the sequential implementation of Marcelino et. al. work [3].

Compared to other parallel NSGA-II on GPU [12]–[14], our implementation makes intensive use of the shared memory, which speeds up the execution but, on the other hand, it limits the population size to a maximum of 1024 individuals on current GPU architectures. There are, however, well known methods in the parallel programming area to manage storage restrictions (e.g. tiling) that may be explored in future works in order to improve the scalability of the implementation. The way we used CUDA dynamic parallelism allows for multiple runs of NSGA-II simultaneously, but has a great impact on GPU's memory capacity, affecting also the scalability. Since the maturity of dynamic parallelism in CUDA has improved in recent versions, it is expected that new implementations can achieve better scalability. Regarding to the HPP energy dispatch problem, we suggest analyzing if more NSGA-II runs and bigger population sizes can improve the quality of the solutions.

VII. ACKNOWLEDGEMENT

The authors acknowledge the CEFET-MG for the infrastructure provided and CAPES, CNPQ and FAPEMIG for the financial support. This work is financed by the ERDF (European Regional Development Fund) through the Operational Programme for Competitiveness and Internationalization (COMPETE) 2020 Programme within project ≪POCI-01-0145-FEDER-006961≫, and by National Funds through the FCT - Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) as part of project ≪UID/EEA/50014/2013≫

REFERENCES

[1] EPE, "National energy balance," Energy Planning Company (in portuguese), Tech. Rep., 2014.

[2] C. Marcelino, E. Wanner, and P. Almeida, "A novel mathematical modeling approach to the eletric dispatch problem: Case study using differential evolution algorithms," *IEEE Congress on Evolutionary Computation (CEC)*, pp. 400–407, 2013.

[3] C. G. Marcelino, L. M. Carvalho, P. E. M. Almeida, E. F. Wanner, and V. Miranda, "Application of Evolutionary Multiobjective Algorithms for solving the problem of Energy Dispatch in Hydroelectric Power Plants," *In: 8th International Conference on Evolutionary Multi-Criterion Optimization, 2015*, pp. 1–15, 2015.

[4] A. Arce, T. Ohishi, and S. Soares, "Optimal dispatch of generating units of the itaipú hydroelectric plant," *Power Systems, IEEE Transactions on*, vol. 17, no. 1, pp. 154–158, 2002.

[5] E. Finardi and E. Da Silva, "Unit commitment of single hydroelectric plant," *Electric Power Systems*, vol. 75, no. 2, pp. 116–123, 2005.

[6] F. Y. Takigawa, E. L. da Silva, E. C. Finardi, and R. N. Rodrigues, "Solving the hydrothermal scheduling problem considering network constraints," *Electric Power Systems Research*, vol. 88, pp. 89–97, 2012.

[7] S. Liu and X. Li, "Hydroelectric unit commitment by enhanced pso," in *E-Product E-Service and E-Entertainment (ICEEE), 2010 International Conference on*. IEEE, 2010, pp. 1–4.

[8] P. d. L. Abrão, E. F. Wanner, and P. E. M. d. Almeida, "A novel movable partitions approach with neural networks and evolutionary algorithms for solving the hydroelectric unit commitment problem," in *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM, 2013, pp. 1205–1212.

[9] K. Price, R. M. Storn, and J. A. Lampinen, *Differential Evolution: A Practical Approach to Global Optimization (Natural Computing Series)*. Springer, 2005.

[10] K. Deb, "A fast and elitist multiobjective genetic algorithm: NSGA II." *Evolutionary Computation, IEEE Transactions on,*, vol. 6, 2002.

[11] E. Zitzler, M. Laumanns, L. Thiele, E. Zitzler, E. Zitzler, L. Thiele, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm," 2001.

[12] F. R. Padurariu and C. Marinescu, "NSGA-II: Implementation and Performance Metrics Extraction for CPU and GPU," in *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, no. Section 5, 2014, pp. 494–499.

[13] S. Gupta and G. Tan, "A scalable parallel implementation of evolutionary algorithms for multi-objective optimization on gpus," in *IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2015, pp. 1567–1574.

[14] M. L. Wong, "Parallel multi-objective evolutionary algorithms on graphics processing units," in *Proceedings of the 11th Annual Conference Companion on Genetic and Evolutionary Computation Conference: Late Breaking Papers*. ACM, 2009, pp. 2515–2522.

[15] Z. Shen, K. Wang, and F.-Y. Wang, "Gpu based non-dominated sorting genetic algorithm-ii for multi-objective traffic light signaling optimization with agent based modeling," in *IEEE Conference on Intelligent Transportation Systems-(ITSC)*. IEEE, 2013, pp. 1840–1845.

[16] B. Arca, T. Ghisu, and G. A. Trunfio, "Gpu-accelerated multi-objective optimization of fuel treatments for mitigating wildfire hazard," *Journal of Computational Science*, vol. 11, pp. 258–268, 2015.

[17] Nvidia, "Cuda C Programming Guide," 2014. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[18] M. McCool, J. Reinders, and A. Robison, *Structured Parallel Programming: Patterns for Efficient Computation*, 1st ed. Morgan Kaufmann, 7 2012.

[19] E. Zitzler, *Evolutionary algorithms for multiobjective optimization: Methods and applications*. Citeseer, 1999, vol. 63.

[20] E. G. Carrano, E. F. Wanner, and R. H. Takahashi, "A multicriteria statistical based comparison methodology for evaluating evolutionary algorithms," *Evolutionary Computation, IEEE Transactions on*, vol. 15, no. 6, pp. 848–870, 2011.

[21] H. Abdi and L. J. Williams, "Tukeys honestly significant difference (hsd) test," *Encyclopedia of Research Design. Thousand Oaks, CA: Sage*, pp. 1–5, 2010.

[22] M. Saito and M. Matsumoto, "A Variant of Mersenne Twister Suitable for Graphic Processors," *ACM Transactions on Mathematical Software (TOMS)6*, vol. 39, no. 2, p. 10, 2010.

[23] Nvidia, "Nvidia visual profiler," 2015. [Online]. Available: http://developer.nvidia.com/nvidia-visual-profiler