The published version of this accepted manuscript can be found at:
https://doi.org/10.1016/j.enconman.2020.112535

A parallel solution with GPU technology to predict Energy Consumption in Spatially Distributed Buildings using Evolutionary Optimization and Artificial Neural Networks

J.R.S. Iruela[a], L.G.B. Ruiz[a], M.C. Pegalajar[a], M.I. Capel[b]

[a]*Department of Computer Science and Artificial Intelligence, University of Granada, Spain*
[b]*Department of Software Engineering, University of Granada, Spain*

Abstract

Today all governments talk about climate change and its consequences. One of the ways to tackle this problem is by studying the energy consumption of the buildings around us. The study of energy consumption may give us relevant information to make better decisions, and thus reduce costs and pollution. However, ANN training models, in order to achieve those goals, has a high computational cost in terms of time. To solve that problem, this paper presents a GPU-based parallel implementation of NGSA-II to train ANNs whose evaluation has also been implemented in a parallel GPU scheme. Our methodology is designed to predict the energy consumption of a series of public buildings, and thus, to model consumption, save energy and improve the energy efficiency of these buildings without compromising their performance obtaining the prediction in a very short period of time. We compared the sequential implementation of the evolutionary algorithm NSGA-II with our new version developed in parallel and the parallel implementation gets better results in much faster execution time.

Keywords: Energy consumption forecasting, Artificial Neural Networks, GPU, Evolutionary Algorithm

1. Introduction

In the last 50 years, the planet has degraded more than in 100 centuries [1]. According to the United Nations, 60% of all-natural resources are exploited

in a non-sustainable manner. It becomes, therefore, necessary to carry out efficient use of the energy resources for the time to come. Moreover, predicting energy consumption (EC) in a set of buildings associated with an institution is a difficult task to perform Feng et al. [2] that large companies and administrations around the world are aiming today Liu et al. [3], Park et al. [4], and to solve it is of paramount importance for the achievement of efficient EC by 2020. One of the most important areas in which to tackle this problem is in the local EC (EC) forecasting, which will allow us to anticipate future events and, in this way, propitiate to make wise decisions regarding energy savings. In this realm, to analyze the EC collected by sensors at an individual level, over delimited zones or buildings, can help to reduce energy costs and environmental impact created by energy production. To make, however, predictions of EC within a time range, e.g., a forecast temporal horizon (time of prediction of the model) larger than 1 hour is not usually useful, and it is actually an extremely difficult task to carry out because of the massive amount of data to be processed within an established time-span.

In this work we propose to use ANN to predict EC in buildings that, to the best of our knowledge, has not been tried before in conjunction with a multi-objective function optimization over a population, which is carried out with the well-known NSGA-II algorithm to optimize both models. In order to obtain the necessary high performance of the application for making useful EC predictions, we propose an implementation based on GPU to leverage all the performance capabilities of that implementation of the models can bring to us. The models were tested by using different buildings of the University of Granada, and the obtained results, by making a daily prediction in an approximate time of 60 seconds, are shown.

Several studies provided solutions to the problem of predicting EC in buildings by using Evolutionary Algorithm (EA) and ANN [5, 6]. However, the main drawback of those methods lies in their non-dependable time response. So far, some approximations have been proposed to solve this problem [5]. As a consequence, there is still a lot of work to do in this research line whereas some interesting GPU implementations of EAs have been proposed in [7, 8, 9, 10, 11] in which different data structures, configurations, data size and complexity were studied to solve the problem. The main issue in these solutions lies in the fact that all of them only parallelize the repetitive process of the evolutionary algorithm needed to find the fitness of the population, without going any further in terms of the parallel structure of the sub-tasks of that algorithm. To fill up this gap, we propose here to advance in the

design of the EA by taking full advantage of parallelism not only in the parallelization of the EA intrinsic scheme but also at the level of the evaluation of each individual of the mentioned population. Thus, by the evaluation of the ANN associated with each individual, we designed a parallel solution to deal with the fitness function calculation of the population of the EA.

Although, as mentioned above, the problem of parallelization of EAs has been addressed multiple times with success, however the proposed solutions up today present the hurdle of ignoring time restrictions. The function that usually takes the longest time to be calculated is the fitness function, which is the weak point regarding the performance of all the previously mentioned parallelization proposals of these algorithms [10]. In order to achieve the above requirements and solve the problem of EC prediction, a new design of a parallel multi-objective evolutionary algorithm of the NSGA-II is presented here. By an original assignment of blocks to threads of the GPU that consists of starting a processing structure of 2 levels of GPU-threads, we obtained that the evaluation with genetic operators of each individual of the population was independently performed. Thus, the iterative loop performed in each ANN to evaluate all data is done in parallel without safety violation (race conditions, non-mutual exclusion, ...) that might hinder one another evaluation.

The main point to solve the mentioned problems is how to cope with the high time cost of calculations. To do so, parallel computing was adopted as a fundamental tool for speeding up the creation of new prediction models [12]. However, there are two different methodologies within the area of parallelism to follow. The first one, based on the parallelization of the algorithmic parts of the calculation only at the CPU level, which is expensive in terms of computation resources and limited by the small acceleration factor of the calculations. The second approach is aimed at the deployment of software at the GPU level, and thus the entire algorithm and data are programmed as CUDA kernels on the GPU. This approach is usually more economical than the former one and has better performance since it can have up to 6000 cores available (e.g., NVIDIA Tesla graphics) that are efficient for the execution of matrix operations. For this reason, we selected this parallelism in GPU, which proved to work successfully on modelling EC in our University.

Our proposal has proven to be an excellent solution to deal the high time cost needed to obtain good forecasting models, achieving substantial improvements in terms of time and an enormous difference between sequential and parallel evaluation of the population. The rest of the work has been divided into the following sections: Section 2 presents a brief description

of the GPU architecture. Section 3 describes the proposed algorithms to solve the power consumption prediction problem. Section 4 explains how the proposed multi-objective evolutionary algorithm has been parallelized and implemented on GPUs with the goal of reducing the run-time of the entire algorithm. Section 5 presents the results of the experiments, which were obtained from the UGR data-set and the implementation of the developed algorithm. Section 6 concludes with a summary of the main conclusions obtained and future research work.

2. Related work

Studying energy consumption can help to reduce energy costs and the environmental impact created by its production. For this reason, to improve energy efficiency in buildings is a concern in many situations nowadays. When studying energy consumption we are mainly interested in obtaining behavioural patterns [13] to detect anomalies [14, 15] and to make predictions of energy demand [16], as well as to acquire consumption profiles of different buildings[17].

In the related literature there are many techniques to predict energy consumption in different scenarios, such as ARIMA[18], Grey Models[19], Regression Tree[20], Support Vector Machine[21], Artificial Neural Networks (ANN) [22] or even combination of different techniques to optimize solutions, such as, neural fuzzy systems[23], regression trees using clustering methods [24], ANN and Evolutionary Algorithm [25], among others Wu et al. [26], Fan et al. [27]. Particularly, ANN has proven to be a powerful technique to predict energy consumption. Several kinds of ANN can be found in literature to solve these problems. In [28] an Artificial Neural Network Inverse is utilized to optimize multiple variables in an absorption heat transformer in order to provide a method for optimizing energy usage. An study of local perceptron networks can be found in [29] where authors discuss several approaches to carry out forecasting on time series. They demonstrate that ANNs many often outperform traditional prediction techniques. For a deeper analysis and a comprehensive study of artificial neural networks, a fundamental reference is [30].

Nevertheless, all those techniques lack of a method to optimize and improve their results, and therefore, they are highly improvable very often. Hence, other techniques must be used to deal with this drawback, and these are Evolutionary Algorithms (EA)-based ones Krzywanski et al. [31]. The

EAs bring the possibility of optimizing models in many senses, e.g., simplifying its structure, dealing local minimum, optimizing relations among parameters. Several studies have provided solutions to the problem of predicting energy consumption in a series of buildings using EA, with which sequential solutions are proposed to obtain accurate results. However, the cost due to the execution time of EA is often seen as very high to get results on time [5], especially in those systems which presents time criticality. On the other hand, EA has proven to be very suitable to obtain its parallelization and has successfully solved numerous problems. In [9] a flight planner was developed with EA, the results are compared in the study with the sequential version. With the parallel algorithm, the execution time was reduced by a factor of 290x compared to sequential execution. A similar study with different techniques is carried out in [7] where the problem of energy consumption forecasting of a hydroelectric power plant was solved in real-time successfully by using a parallel GPU implementation of a evolutionary algorithm. In that study, results were shown competitive compared to the serial version. And several approaches were proposed in many other studies [8, 10], where different proposals of parallel-based evolutionary algorithms were investigated, and so distinct different data structures, configurations of the algorithm, data size and complexity of the problem, have been compared over recent years [11].

Since in our approach we do not only look for accuracy but also intend to speed up the algorithm execution, we focus the research on parallelizing a multiobjective evolutionary algorithm with neural networks in order to obtain the results in the shortest period. In order to achieve these objectives of precision and acceleration, the execution of the evolutionary algorithm has been parallelized along with the evaluation of individuals through neural networks.

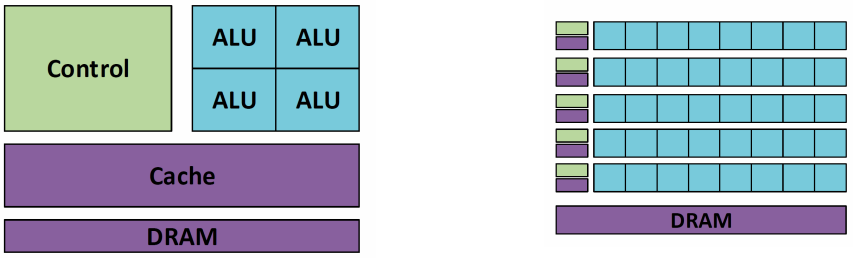
3. Preliminary concepts

GPU was originally designed to be used in video games or image rendering [32] as it presents a high degree of parallelism that is required in these kinds of problems. But as of today it has been assumed in many fields of research, and it has yielded the technical term known as GPGPU (General-Purpose computation on GPU) [32, 33]. Note that this section is intended to introduce some essential concepts concerning the GPU, so the reader can skip this section if he is an expert in this field, or otherwise he should not since the

association between our solution and the structure of the GPU could not be easy to follow.

Therefore, the GPU's particular structure is the first thing we must point out. Thus, figure 1 compares the architectures of the CPU and GPU. Whereas CPU reserves more space for control units and their associated storage, and thus the remaining space is intended for the logical arithmetic units (ALU)(see Figure 1a), on the contrary the GPU allocates more space to ALU and less space for control and storage units in order to reach a higher degree of explicit parallelism and throughput(see Figure 1b).

CPU is biased to process serial instructions that use a lot of memory, while a GPU is a better choice for processing parallel instructions that use much less memory. Another difference is that CPU has few powerful processing cores, while GPU has thousands of these. The GPU architecture works better with highly parallelizable applications while the CPU gets more computational performance by parallelizing applications with longer sequential code [34]. This point is important as we leverage these features in the representation of our solution (the ANN) as it shows a high level of potential parallelism [7], which will be detailed in the next sections.



(a) (b)

Figure 1: CPU (a) and GPU (b) architecture.

3.1. GPU notations

The architecture and software that support the use of GPUs allow us to use them for general-purpose calculations. There are different frameworks with which it is possible to develop software and generate code for the GPU. The two most important frameworks are OpenCL [35] and CUDA [36], both of which are platforms that allow the GPU to be used for high-level programming. OpenCL is a cross-platform programming language with no hardware

constraints that is intended to be used on heterogeneous platforms, whereas CUDA is a parallel computing platform that includes a compiler and tools for programming algorithms in NVIDIA GPUs. The fact that CUDA was developed exclusively to work with NVIDIA graphics cards gives it a clear advantage over OpenCL by obtaining better performance results [37]. On the other hand, OpenCL is supported by more software applications than CUDA actually is.

In the CUDA environment, CPU is referred as the *host*, and GPU as the *device*. The set of instructions running on GPUs are structured into functions called kernels. These kernels will be the most critical point in our work as the success of our solution regarding performance improvement depends on the optimal implementation of them. They will be the operations that our algorithm will carry out, i.e., the creation of the solutions, its evaluation, etcetera.

A kernel can be launched from either the CPU or GPU. The advantage of calling a kernel from the GPU is that it reduces the bottleneck caused by the necessary communication between GPU and CPU memories, and thus we take advantage of this feature and therefore try to keep data into the GPU as long as possible to speed up the computations. The kernels are executed by the different threads available on the GPU. As can be seen in the figure 2 that shows the internal structure of the GPU, threads are organized in blocks, which have a 3D structure that defines the number of threads in each dimension. At the same time, blocks are organized in a 2D grid. When a kernel is launched, it runs as a grid of parallel threads. The threads in a grid are organized in 2 hierarchy levels but only to reference them in the program. At the top level, each grid is made up of one or more blocks of threads. All blocks have the same number of threads. Each block is unequivocally identified by two coordinates assigned by CUDA, for more detail we refer to [36].

NVIDIA uses "Single Instruction Multiple Threads" (SIMT) as the runtime model. The threads of each block are executed in sets of 32 threads called *warps*. All threads in the same warp must execute the same instruction at the same time instant. When some threads belonging to the same warp need to execute a different instruction, a divergence occurs, and the group of threads of the same warp is executed sequentially, thus no simultaneously. In order to minimize convergence, it is advisable to avoid instructions that produce thread bifurcation. However, in our solution, this would be required as some functions are needed for it. For instance, the mutation

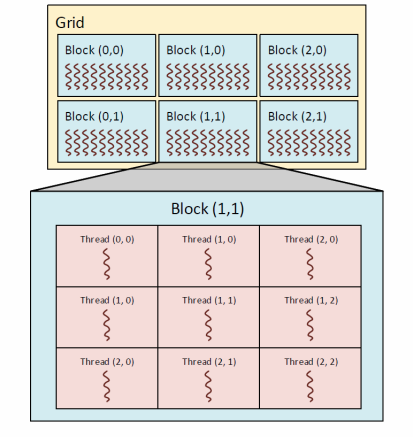


Figure 2: Structure block and threads.

operator, because some individuals will change their values and others will not, depending on a probability. All of this will be discussed in detail later on.

CUDA defines its own memory hierarchy, which includes, among the different memories shared and non-shared by threads, there exist global memory and shared memory. Global memory can be accessed and modified from the host and the device, therefore all the threads can access to the global memory. Shared memory is faster than global memory but its capacity is smaller. All the threads of the same block can access the shared memory. All data stored in this memory are lost when the block ends its execution. Data that are frequently used by the same block must be moved to the block's shared memory, and so the performance offered by the GPU can be improved.

4. Methodology

In this work, we combine two machine learning techniques to address the problem of EC forecasting applied to prevent energy waste in buildings. The first technique is artificial neural networks (ANN), which are used to model and predict energy usage. The second one is the multi-objective evolutionary algorithm NSGA-II deployed in our study to provide a procedure to obtain optimal forecasting models, i.e., the neural network with the least number of hidden neurons and which makes the least error in the set of examples. Note that in our solution, we will not use any classic machine learning algorithm,

but we will use the same EA to modify the weights of the ANNs so we carry out the two task we pursue at once. In this way, we minimize the error and the complexity of the model simultaneously. Since the cost of deploying these techniques for EC forecasting is very high in terms of time, the models supporting them were designed for implementation on a GPU framework. With this fact in mind, in the following subsections, we will go over these concepts.

4.1. Artificial neural networks

ANNs are algorithm based on brain functioning which are widely known for their good results, and used in tasks such as process control, optimization, pattern recognition, prediction, etc. [38, 39, 40].

One of the most recognized models, used in multivariate regression problems, are the neural networks called feed-forward. Its topology can be observed in 3 and, as this figure shows, the ANN connect their input neurons to the hidden neurons, and then these to the output neurons. This type of ANN uses the values of the input neurons and the weights assigned to each link, which connect them to the hidden neurons, to assign values to the states of the network. Once the states of the hidden neurons and the weights associated with the output layer are calculated, the value for each of the output neurons will be finally obtained.

On the other hand, Back-propagation (BP) is one of the most popular algorithms used for training feed-forward neural networks. BP is a method based on gradient descent and may be subject to convergence at a premature local optimal solution. As a consequence, the solutions found by this technique depend on the fair initial randomness and on the lack of that fairness during the calculus of solutions, which is not easy to prevent [41]. Therefore, ANNs can make errors caused by multiple local optimal solutions and the application of the BP method may yield results from a local solution that are not the global optimal [42]. In addition to this, the use of ANN, like many other techniques, implies determining several criteria before its execution, such as to obtain the number of neurons and the selection of the training procedure. These decisions can be made by an expert through trial and error, as usual, on account of high computing time of carrying out the entire procedure, it would be better to reduce the number of trials. We must keep in mind that making these decisions is generally a difficult task, since the change of a single parameter, among the possible configurations of an ANN,

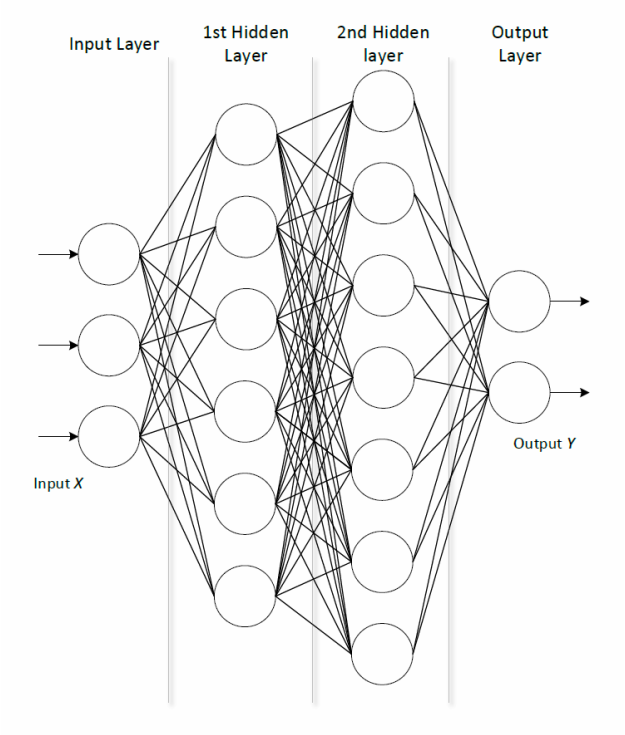


Figure 3: Typology of a feed-forward ANN.

may lead to a substantial negative effect on the performance of the learning algorithm, which makes it difficult to find the optimal global and how it affects the performance of the neural network. As a solution to those drawbacks, we propose to deploy an evolutionary algorithm capable of carrying out these tasks at the same time. In other words, we utilize the evolutionary algorithm not only to adjust some of ANN's parameters, such as the number of neurons, but also to combine the information of the different solutions to learn about the data and thus to provide an implicit training mechanism, thus fitting in this way the different weights of the models. Indeed, this approach has shown its potential to solve many problems like this one up to now [43][44].

4.2. Evolutionary Algorithms and Multi-objective Evolutionary Algorithms

Evolutionary algorithms (EA) are a global search method based on a population of individuals (associated with sub-optimal solutions), inspired by the natural mechanisms of the genetic evolution of biological species. On

the whole, a EA employs three operators: selection, mutation and crossover. Those procedures are used to generate new solutions (called individuals) that will lead to exploration of new search points. EA may become very effective in a wide variety of problems, however, their execution time may become a limiting factor for deploying it when programming the solution of some large problems. This fact is due to the great number of operations which should be done to achieve the final solution. One of the most expensive, in terms of computation time, of such procedures is the evaluation of the population, also called calculus of the *fitness* function. This is because all individuals must be evaluated many times to estimate the accuracy of their associated solutions. Fortunately, evaluations can be carried out independently for each individual in the population, and it makes the EA a candidate method to be parallelized with high performance [45].

In the specific literature, EA are usually designed to solve a problem with a single objective, e.g., error, saving, space, etc. However, it is not uncommon in real problem solving to seek solutions that require finding values of parameters according to multiple criteria [46]. However, it is normal that the optimal way to combine the different objectives is not known or it is difficult to do so. For these problems with multiple objectives, there is not always a single solution that can be considered the best one, but a set of solutions that represent the best compromises between the different criteria. This set is called the optimum Pareto set and its representation in the target space is called the Pareto front [47, 48].

On the other hand, EAs are recognized for their great effectiveness in solving difficult optimization problems [6] but addressing them requires large amounts of computing resources as every solution has to be evaluated many times. In this vein, it becomes necessary to take advantage of parallelization, which has proven to be the fastest and most effective approach [49] [50]. For these reasons, our design, developed by using this technology, will be explained in the sequel.

4.3. Multi-objective evolutionary algorithms to optimize ANN

In this work, the multi-objective evolutionary algorithm NSGA-II is used to combine the knowledge of every solution so as to optimize the topology of the networks and to train the weights of the ANNs. By doing so, one can obtain the least error with the optimum number of neurons [51]. This algorithm sets the ground to develop a software component of the method based on fair-randomness of solutions choice that allows us to better explore the search

space, making it easier not to fall into local optimal solutions but to improve solution searching by performing several procedures properly optimized.

In the following section, we will detail and adapt all the components of the NSGA-II algorithm in order to obtain an optimal ANN with the final aim of predicting EC in buildings.

4.3.1. Coding of individuals

In the calculation of individuals two parts are differentiated: a first one that will represent the number of hidden neurons in the sought networks and a second part that will represent the set of associated weights to the links bound to these neurons; note that an appropriate value of the weights will provide better forecasting in the model. These two parts can be seen represented graphically in figure 4. This figure shows a diagram with the coding used in this study, where N is the number of hidden neurons that make up the network and W_{ij} and V_{ij} are the different weights of the network. The gene associated with the number of hidden neurons will be an integer value and the genes associated with the set of network's weights will be coded as real values.

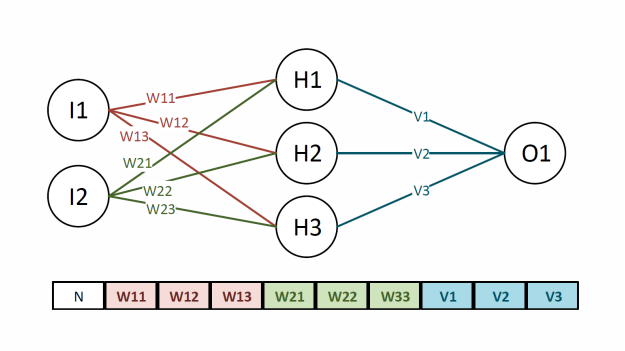


Figure 4: Data structure.

4.3.2. Objectives to optimize

In this study, the main objective is to obtain the optimum neural network. To do so, the following objectives have been taken into account:

The prediction error: In our problem, for an ANN, we can say the lower the error is, the better solution is found. Therefore, to minimize the error obtained in the training set will be one of the objectives of our method.

In equation 1 we see the definition of MSE, this refers to the first objective, where T is the number of instances available for training, *Y(t)* is the expected output and *O(t)* is the output calculated by the network at time *t*.

$$Min\{f_1(s)\} = min\left\{\frac{1}{T}\sum_{t=1}^{T}(Y(t) - O(t))^2\right\} \quad (1)$$

Number of hidden neurons: This objective allows us to compare networks of different sizes with the intention of finding networks with fewer hidden neurons. In function 2 you can see the definition of this objective, where h(s) is the number of hidden neurons for the network s.

$$Min\{f_2(s)\} = min\{h(s)\} \quad (2)$$

Eventually, the NSGA-II algorithm will focus on finding individuals that minimize these objectives, i.e., solutions that have the lowest prediction error and obtained with the simplest model architecture. It is common to find a better solution if the model is more complex, therefore discovering a simpler model while keeping the error level of a more complex one is not a trivial task [43][44][52]. Next sections will detail the whole scheme of the proposed algorithm.

4.3.3. Initialization of the population

Each of the individuals who are part of the population will represent a possible, non-optimal, solution to the problem. Each solution represents a distinct ANN. The generation of the individuals is made randomly, in order to explore the most of the search space. To generate the population, in our problem, each individual will be assigned a random number of hidden neurons between a previously declared minimum and a maximum. Therefore, in this study, 4 and 32 respectively were established as these values. Once the *length* of the individual has been defined, its structure will be completed by randomly assigning weights between a previously defined range [-30, 30].

4.3.4. Fitness function

The fitness function is used to know how good is a solution. In our case and, since it is a classic evolutionary algorithm, it is normal to choose as fitness function the error returned by the neural network evaluated with the training set. In that case, as it is a multi-objective algorithm, fitness will be

determined by 2 objectives, the error obtained by ANN and the number of hidden neurons.

4.3.5. Selection

Selection is the first genetic operator to be used in a EA. This operator, based on the fitness of each individual, is used to improve the search for parent chromosomes. The main objective of the selection is to promote and make survive parents with better fitness. There are several strategies to carry out the selection, some of the most used are tournament selection, random selection and roulette selection.

In this work, the selection by tournament has been used, in which two parents are chosen at random among all the population and these are confronted by means of a comparison of their fitness.

4.3.6. Crossover

The crossing is a genetic operator that is in charge of generating new children from the parents. This operator simulates the mating and genetic reproduction of nature. For getting so, this operator selects two parents, and then two things can happen:

1. Both parents have the same number of hidden neurons; in this case, two children are generated of the same length of the parents, and the genes of the children will be obtained by crossing the ones of both parents.
2. The parents have a different number of hidden neurons; in this case, two children will also be generated. The first child will have the same number of hidden neurons as the first parent and the second child will have the same number of hidden neurons as the second parent. The genes of the children will be generated by crossing the genes of both parents. When the number of genes is different, then these will be completed with the longest parent's genes.

The heuristic Wright's crossover [53] has been used in this work, and its application has yielded good results, indeed [43, 54]. Figure 5 shows graphically the heuristic Wright's crossover where two parents are crossed based on a crossing probability if the children do not cross because the probability of crossing is not satisfied, the parents replace the children in the next generation. If they can finally be crossed, the children will be obtained in the following way:

Let a_i and b_i be the parents to be crossed and r is a random real number (float) belonging to [0,1] then a child is obtained by the following equation:

$$h_1 = r \times (b_i - a_i) + a_i \tag{3}$$

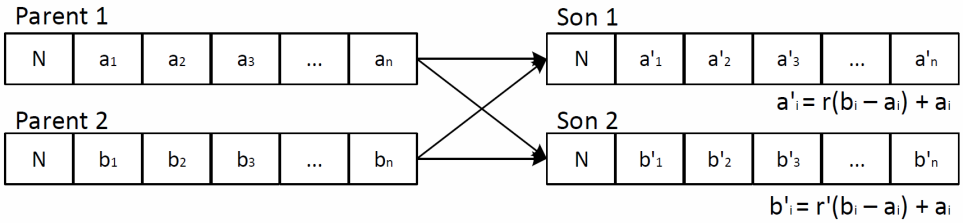


Figure 5: Heuristic Wright crossover.

4.3.7. Mutation

The mutation operator is applied in EA to preserve the diversity of the population. This operator creates new individuals by slightly modifying or drastically the genes of existing individuals. Thus, in our work, two types of mutation can be distinguished: structural and genetic, respectively. Figure 6 shows a graphical representation of both mutations. While the structural mutation modifies the topology of the network (neuron numbers) and causes a drastic change in the individual, the genetic mutation modifies one or several genes corresponding to the weights of the network. When the size of the network is changed, the chromosome must be restructured by expanding or decreasing its size; this benefits the exploration of the search space.

In the following sub-section, the implementation of the proposed NSGA-II algorithm is detailed.

4.4. NSGA-II description

The NSGA-II (Elitist Non-Dominated Sorting Genetic Algorithm) was proposed by Deb et al., in 2002. Since this problem requires to work with several targets to optimize, it is necessary to introduce the concept of dominance. Therefore, prior to explaining the algorithm, a series of definitions must be given to properly understand how NSGA-II works:

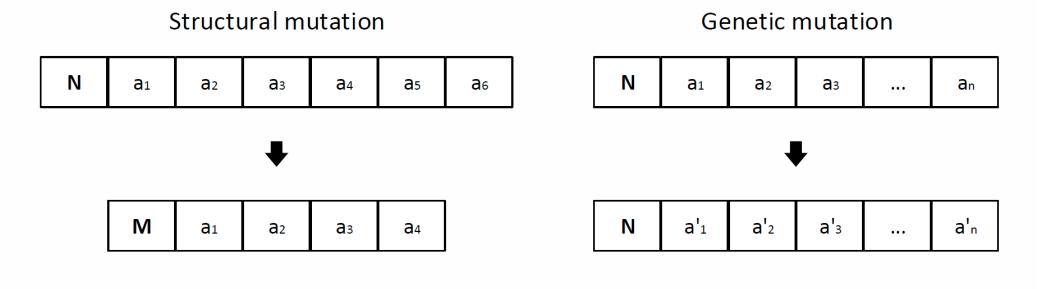


Figure 6: Structural and genetic mutation.

Definition 4.1. Solution A dominates solution B if the following two conditions are met:

1. If $A <= B$ for all M objectives and
2. If $A < B$ for at least one of the M objectives.

In the event that one of the above conditions is not met, solution A would not dominate solution B. For example, if a ANN A has better error and less neurons than another ANN B, then A dominates B; otherwise, it doesn't. Considering a population of N solutions each with M values of target functions, the following procedure is used to find the non-dominated set of solutions [55]:

1. Do $i = 1$.
2. For all $j \neq i$, compare solutions x^i and x^j to determine dominance, using the 2 conditions cited above.
3. If for any j, x^i is dominated by x^j, mark x^i as dominated. Increase i by one and go to step 2.
4. If all solutions $(i = j)$ in the set have been considered, go to step 5; otherwise increase i by one and go to step 2.
5. All solutions that are not marked as dominated are non-dominated solutions.

By applying the above definition, we can divide the population of individuals into different groups according to their dominance number. The first frontier will be formed by individuals who are not dominated by any other individual in the population and then, the dominance number is 0. The second and following frontiers will be calculated by performing an iterative loop

where individuals who are not dominated by other individuals of the same group are singled out and the rest are inserted into a new front. This process will be repeated until all individuals in the population are organized into fronts. Note that the first front is a set that only contains the individuals non-dominated by others, and thus the set of P solutions associated to these individuals is known as the Pareto front [51].

In order to avoid the selection of two *too similar* parents, NSGA-II uses the individual's attribute known as *distance crowding*, which allows to preserve the diversity of the population. This procedure guarantees the diversity of the population within the same Pareto front by blending all the objectives. In this way, when the population converges towards the optimal Pareto front, the algorithm ensures that the solutions are distant enough to acknowledge dissimilarity from each other [55].

Having defined the main concepts necessary for NSGA-II [34], we will now explain how it works, firstly by showing the pseudocode and then a brief explanation of it:

```
1  generationsCount = 0;
2  currentPopulation = generateRandomIndividuals(N);
3  evaluateFitness(currentPopulation);
4  nonDominatedSort(currentPopulation);
5  while generationsCount <X do
6  |   while newPopulation.count <X do
7  |   |   parent1, parent2 = select(currentPopulation);
8  |   |   child1, child2 = crossover(parent1, parent2);
9  |   |   mutate(child1, child2);
10 |   |   newPopulation.Add(child1, child2);
   |   end
11 |   nextGeneration = currentPopulation + newPopulation;
12 |   nonDominatedSort(nextGeneration);
13 |   currentGeneration = selectBestN(nextGeneration);
14 |   generationsCount = generationsCount + 1;
   end
```

Algorithm 1: NSGA-II.

Initially, the NSGA-II algorithm creates a random population of P_0 parents, on line 2. The population is sorted (within different fronts) on line 4 according to their number of non-dominance, once they have been evaluated (see lines 2-3). Using the selection, crossover and mutation operators, the

descendant population Q_0 will be generated, on lines 7-9. The population of P_t parents of size N is used to create the descendant population Q_t of size N, line 10. The two populations are then joined to form R_t of size $2N$, line 11. Once the populations have joined, an non-dominated operator is used to classify the R_t population on different Pareto fronts and the new population is generated according to that procedure, as shown on lines 12-13. First, the individuals belonging to the best non-dominated front F_1 will be added, then individuals from the second front F_2, third front F_3, ..., so on a so forth. The adding of individuals to different fronts continues until obtaining N fronts, where N is the number of solutions.

For a more detailed description of the implementation of the algorithm see [56].

5. Parallel Multiobjective Evolutionary Algorithms for ANN optimization

So far, the general scheme of the implemented EA has been introduced as well as many necessary concepts for understanding our solutions. In this section, the bulk of our contribution is presented. This section details the strategy that has been carried out to parallelize the multi-objective evolutionary algorithm NSGA-II and also to calculate in parallel the computation error that all the population of individuals also makes. These two approaches that we followed provided a substantial increase in speed with regard to their sequential versions. Logically, parallel versions follow a different development strategy that the serial versions, since the design of an algorithm for being deployed in a CPU is not valid to be run on a GPU directly, because memory and thread management in a GPU differ from the CPU ones, yet the algorithm intention shares the same philosophy.

In figure 7 we can see the parallelization of the algorithm's general scheme. We can observe that two sections are differentiated; the sequential section, where the load of the dataset is carried out and the definition of parameters of the algorithm. That section is also where the selection and ordering of the best individuals that have been obtained after executing the whole process (main loop of algorithm 1) is carried out to finally check the stop criterion. The parallel section is made up of the evaluation of the individuals and all the functions belonging to the evolutionary algorithms NSGA-II. In the following sections we will design the entire algorithm for its optimization on the GPU.

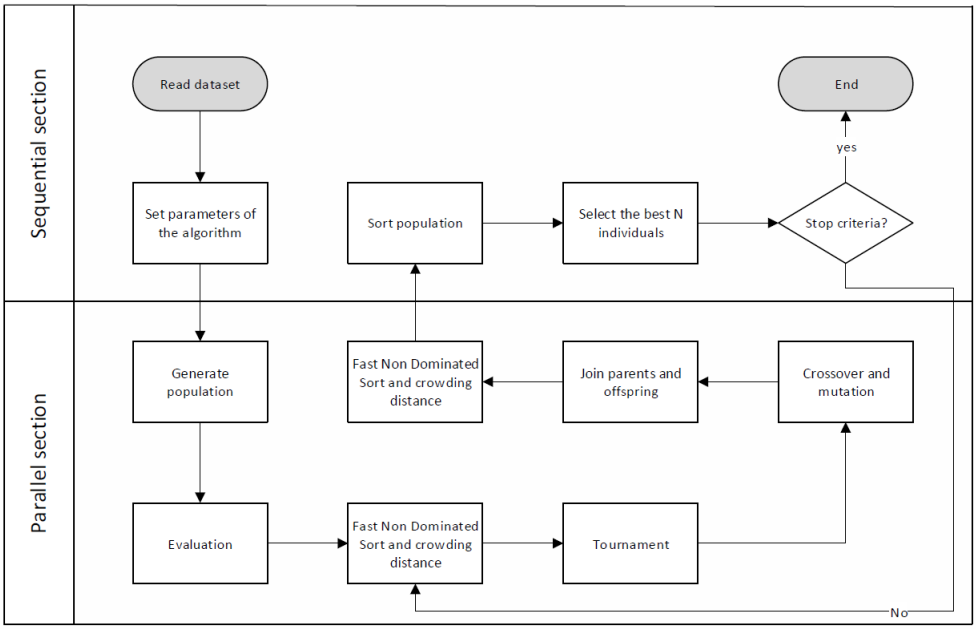


Figure 7: Flowchart of kernels.

5.1. GPU-based Parallel Strategy for evaluations of ANNs in the population

As we have mentioned in previous sections, the ANN used in our study and the one to be evaluated are the feed–forward neural networks. The evaluations of each of the networks are independent of each other, so we can consider launching them on the GPU at the same time, in parallel. In order to evaluate all individuals at once, we developed a kernel in which each individual is assigned to a specific block, as shown in figure 8. This block is called a thread-block in CUDA, which is a programming abstraction that represents, with CUDA Toolkit 10, a group of up to 1024 threads. Threads in the same block run on the same stream processor and can communicate with each other via shared memory or atomic operations.

Therefore, as in the kernel designed, we associated one individual to each block of the grid for parallel computation with GPU. The number of threads per block will be equal to the maximum number of neurons present in the initial parameters of the algorithm $<< Pop\_size, N\_max >>$. To map the grid's blocks into the individuals of NSGA-II, we used a global float matrix

in which each row corresponds to an individual. Each individual will be composed of the network weights, the number of hidden neurons and the error obtained after executing the evaluation.

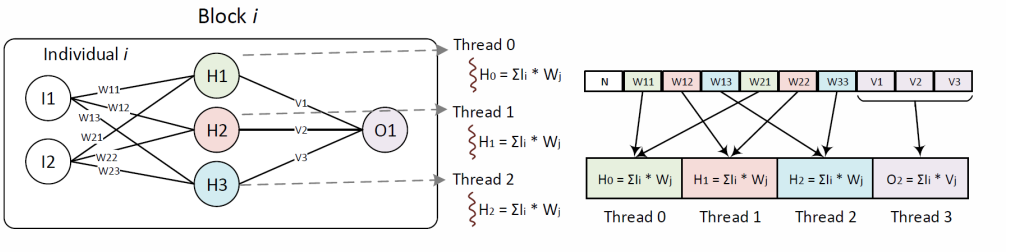


Figure 8: Parallel Strategy for Evaluating the ANN.

5.2. Parallel Strategy for GPU Evaluation of NSGA-II Operators

The strategy followed to parallelize the functions corresponding to the NSGA-II algorithm are detailed below. The GPU parallelization design is simpler than the one carried out in the ANN evaluation of the population's individuals of the previous subsection. Now, only a single thread-block of the GPU will be used to execute all individual actions. All kernels corresponding to this part are executed individually and serially. They have been executed with a block size equal to 1 whose number of threads is equal to the population size $<< 1, pop\_size >>$. The reason of this is because of the low cost in terms of space of those operations and the advantage of using a faster memory, i.e., in the same block all elements may access to a small space of memory with substantially lower latency than uncached global memory. As we can see in the figure 9, the data will be read from shared memory and global memory. In this way, each one of the operations associated to the functions of the algorithm is carried out in parallel by each population individual with the consequent advantage of being able to share memory of quick access, and thus obtaining a greater gain in the execution of each one of the functions of the evolutionary algorithm. These data will be processed by the different threads of the same block. Depending on the kernel to execute, the threads of the unique block will perform one task or another in order to get the following results, for example, for f_{GI} will apply the first function that will result in R_{GI}, which translates into an array with randomly initialized individuals:

- **Generation of individuals (GI):** A matrix will be obtained with a new, randomly generated, population.
- **Non_dominated_sorting (NDS):** It will return the population sorted by fronts.
- **Crowding distance (CD):** It will return, in a similar way to NDS, the crowding distance of each individual with respect to its neighbors that belong to its same front.
- **Selection operator (SO):** A set of the best parents will be obtained.
- **Crossover and mutation operators (CMO):** It will return a new population of children from the parent selection carried out in SO.

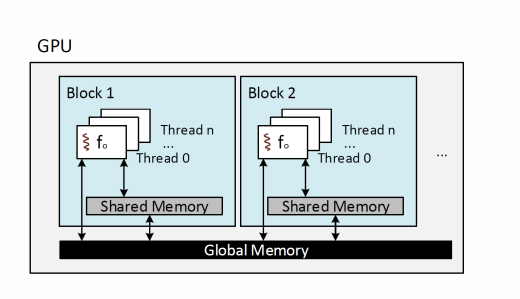


Figure 9: Memory management and design for each kernel f_i leveraging shared memory.

GI: A kernel has been created in which each thread randomly generates an individual. The xoroshiro128+ [57] algorithm has been used to create random numbers in the GPU. These new values will be stored in a float matrix where each row will correspond to an individual, and the columns will be the weights of the network that will later be evaluated in order to obtain an error. As many threads as individuals in the population will be used.

NDS: In this kernel, each thread is in charge of evaluating the dominance of each individual with respect to the others. To this end, the measures of goodness of the individuals have been loaded into shared memory in order to accelerate access to these data. Once the dominance of each of the individuals has been calculated, a single thread will be in charge of creation of the different fronts. The data structure used to classify individuals in fronts

has been shown in figure 12, where the elements of a first array of integer type contains the indices of individuals sorted by fronts and the components of a second array of integer type holds the size of each front. To execute this kernel, as many threads as individuals in the population will be assigned to the thread block.

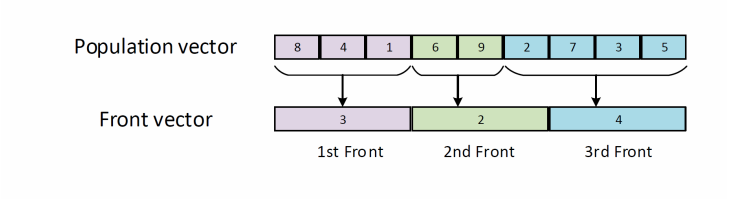


Figure 10: Data structure for fronts.

CD: Each of the threads is in charge of assigning the crowding distance between individuals of a specific front. To do this, each thread is responsible for obtaining the solutions with maximum and minimum fitness values, respectively, in each front, along with the crowding distance. As many threads as fronts will be used, so the kernel will be launched with a number of threads equal to the size of the population, which is meant to address the extreme case of each individual belongs to a single front. Two vectors have been created in shared memory of the integer and float types, respectively, which contain the measures of goodness. Finally, the distance between individuals of the same front is calculated and stored in a data structure similar to the one described in figure 12.

SO: In this kernel each of the threads is responsible for conducting a tournament between two individuals from which a winner is obtained. In order to optimize the speed of access to these data, shared memory area has been used to store the information of the fronts and the crowding distance. Two float and integer vectors have been created for this purpose.

CMO: Unlike the other kernels, this kernel's execution initiates with a number of threads per block equal to $pop\_size/2$. This is because each thread is responsible for making a cross and a mutation through which two new individuals, also called children, will be generated. This kernel works with two float matrices that store parents and new children, both stored in global memory due to their large size. In addition, several variables stored in the registers of each thread are created to know the size of each individual, fitness and probabilities of crossing and mutation.

Data (Bytes)		
Global memory	Constant memory	Shared memory (per block)
8114×10ˆ3	65535	4915
Other GeForce characteristics		
Registers per block	Warp size	Maximum Threads per MP
65536	32	2048

Table 1: Features of the NVIDIA Geforce GTX 1080.

6. Experiments and results

This section shows the experiments and results carried out to solve the problem of EC prediction in a set of distributed buildings of the University of Granada. Developed models analyze historical data and are trained by following our parallel-optimized design of the NSGA-II evolutionary algorithm. Four different facilities have been used in this study whose dataset has about four years hourly data. Although the UGR has more buildings available, the rest of them show a very similar energy expenditure to one of those selected. In summary, each data set is made up of 35,000 instances each one thus a building provides hourly information about its consumption, i.e., about 35,000 samples per building. Most of the pre-processing consisted in compiling consumption by days; in this way, we will work with the past 24 hours as time window for predicting.

All measurements were performed using a graphics card NVIDIA Geforce GTX 1080, which has a total of 2560 cores, a maximum of 1024 threads per block and the memory hierarchy that is depicted in Table 1. In this work, Python was used due to its versatility and its broad compatibility with other packages related to Machine Learning methods and algorithms. Numba is utilized to work with CUDA, a Python compiler supported by CUDA that can compile the code for its execution in GPU.

NSGA-II need to set different parameters before algorithm's execution, all of which were previously tested, and by modifying them one can observe the algorithm yields more exploratory results or more convergence of results depending on its execution behavior. In order not to hamper the final aim of this contribution, some of these parameters were skipped as we want to focus on the significant improvement in terms of execution time of our imple-

mentation of the algorithm. Once this has been done, we decided to choose as fundamental parameters for the NSGA-II algorithm the ones shown in Table 2 only.

Probability	
Crossing	0.9
Tournament's winner	0.9
Structural mutation	0.5
Genetic mutation	0.1

Table 2: Chosen parameters for NSGA-II implementation

Although several population sizes were tested for the NSGA-II algorithm implementation, its size was set to 160 individuals owing to the results that will be explained in the following paragraphs. The ANN instances deployed in our study were set as to have a range of neurons between 4 and 32. The *energy consumption* (EC) data have been normalized between 0 and 1 to have the same range of values for each input to the ANN, thus ensuring that the model will not give more weight to those attributes which values belong to a wider range. The data will be reconstructed as shown in Figure 11.

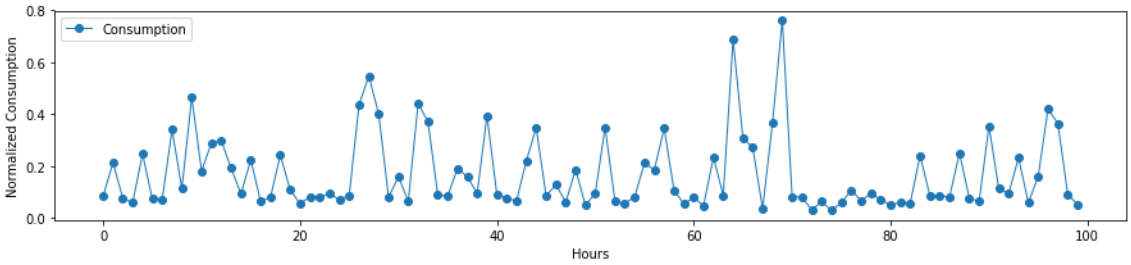


Figure 11: Normalized consumption over a 100-hour period.

The NSGA-II developed in this work allows us to obtain a set of optimal solutions, which are those that form the Pareto front. On figure XX is shown a visual example of the Pareto front obtained for a particular building of our study. In this graph, the X axis shows the error committed (MSE) and the Y axis the number of hidden neurons, each one of the graph-points represents a non-dominated optimal solution and the entire set of points represent the Pareto front

On this front of Pareto we will be see the solutions that have obtained the least error in the prediction of energy consumption with a minimum network size

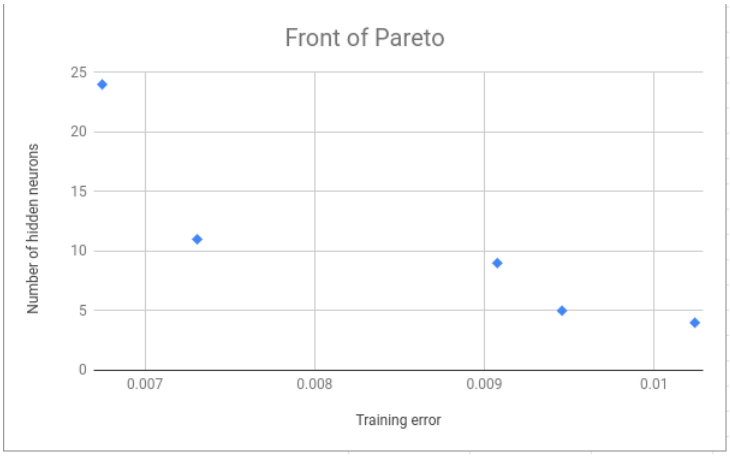


Figure 12: The Pareto optimal front provided by the NSGA-II

We run ten executions for each algorithm and building, so that we could perform a statistical analysis of the results. Each dataset was randomly divided into training data (70%) and test data (30%), to prevent over-training.

After the parameters are established, the first experiment is intended to compare both implementations of the NSGA-II, the sequential and our parallel proposal. Notice that both implementations are exactly the same. The sequential design is performed by launching one block and just one thread in the GPU. Table 3 illustrates the speed up achieved by our design. In this table, all kernels decrease their execution time in the parallel version of the evolutionary algorithm with respect to the sequential one. Some functions need more computational cost than others, for example, the generation of individuals (GI) and the cross-mutation operator (CMO) take similar time because their computations are focused on simple operations over the population. In the first case, GI is responsible for the creation of random numbers to generate all individuals. In the second case, CMO not only creates some random values but also modifies some genes of individuals to perform the mutation and crossover computations. For this reason, the implementation of CMO operator shows more speedup than that of the GI. Another example of an effortless function is the selection operator (SO) as it just needs to decide which individuals are selected for crossing. Differently to execution

times shown by those functions, the non-dominated sort operator (NDS) is the most time-consuming function. In this case, more barriers that synchronise and cause contention of threads arise in the NDS function because of the high dependency among fronts to build each one of them, and this is why the lowest speed-up is obtained here. Along with the NDS, the crowding distance (CD) is the basis of the NSGA-II algorithm. The CD, however, may be optimized in a better way than NDS because CD calculates the *distance* matrix among solutions, and this task may be done separately. This procedure takes considerable time as all individuals must be computed each other, for this reason, the third-best speed-up is attained here. Finally, the most consuming function is often associated with the fitness function (FF), due to its iterative behaviour. This function used to be launched over and over during the algorithm. As a consequence, the emphasis has been placed on looking for a good parallel implementation of FF in our proposal. Thus, in this table one may observe a great difference in execution times of all functions in its sequential version. The same happens if we observe the parallel version results. Nevertheless, the optimized design of the FF function empowers the algorithm and gets a surprising speed-up up to 787.90. What's more, it enables to decrease the execution time of the algorithm from several hours to some minutes.

Kernel Name	Sequential	Parallel	Speedup
GI	17.85	0.259	68.92
NDS	479.67	62.6	7.66
CD	1026.50	17.5	58.66
SO	3.70	0.122	30.33
CMO	17.56	1.02	17.22
FF	104350	132.44	787.90

Table 3: Time (ms) of each kernel, generation of individuals (GI), non-dominated sort and crowding distance functions (NDS and CD, respectively), selection operator (SO), crossover and mutation operators (CMO) and the fitness function (FF).

Some experiments were launched using different population size. Those results are gathered in table 4. The execution time in nearly all cases is the same, although it is observed that it increases a little as the population grows. There is almost no difference because there is still some memory in the GPU which may be used. For this reason, from 200 individuals on it takes longer.

In addition to this, the mean squared error achieved is slightly better as the population increases. However, the implementation run typically reaches a point between 160 and 200 individuals beyond which the algorithm does not provide any further improvement. This is on account of the amount of the information that individuals may provide, and therefore the amount of useful knowledge each individual can share.

Population Size	Time (s)	MSE	Memory Usage (MB)	GPU Usage
80	22.989	0.0122	143	92%
160	19.792	0.0098	145	95%
200	20.129	0.0094	147	98%
250	24.345	0.0099	151	100%
300	25.932	0.0105	160	100%

Table 4: Scalability of the designed algorithm using different population size.

Similarly, table 5 shows the average execution time of each individual with a different number of individuals if all of them got the same number of threads. The first column shows the number of individuals, the second column the minimum time, the third column the maximum time and the last column the time average. That table provides information about how each individual moderately increases their time cost as the number of individuals increases. This is due to the increase in data that are loaded into the device as the number of individuals increases. The amount of data is relatively small, as a consequence, all data are located in shared memory except the information related to each individual, and this fact explains this little variability.

Individuals	Min	Max	Avg
80	0.8249	1.1721	0.9286
160	0.8443	1.1491	0.9492
200	0.8713	1.0945	0.9620
250	0.8717	1.0833	0.9684
300	0.8802	1.0643	0.9820

Table 5: Population size - Time (sec) execution of an individual.

The next experiment is intended to compare the accuracy of both solutions. In table 6 we can see the experimentation for sequential and parallel models. Columns 2,3,4 and 5 show the experimentation for the buildings B1, B2, B3 and B4, respectively. Column 1 describes the metric used. For sequential and parallel experiments we show the average fitness, which contains the average MSE of 10 executions of each algorithm; the best fitness and worst fitness with the minimum and maximum MSE obtained with 10 runs and the standard deviation. The number of evaluations has been fixed at 5000 and a total of 150 individuals. As we can see in the table 6, the results obtained by both implementations are quite alike. However, the parallel version shows lower error values on average.

For a better analysis of the results in table 6, we have included boxplots of the MSE distribution. Figure 13 contains the boxplots of the MSE for the different buildings in sequential and parallel models. The median value in boxplots is highlighted with an orange line. The whiskers plot illustrates in all buildings present a mean error closer to the Q1 in sequential, and their parallel versions get closer to the Q3. Just in the fourth instance, these results slightly change due to the higher complexity of the behaviour presented by that building. This fact give us an idea that all results are quite similar among them, and their differences are caused by the randomness in the algorithm. In nearly all cases, the algorithm reaches a set of solutions comparable in every experiment. However, in the situation that the building presents a consumption harder to predict, then their solutions vary in a higher range as it is not so easy to find the optimal model, yet note that this range is quite small in terms of error as illustrated in the following figures.

The implementation carried out in parallel maintains the quality, understood as the lower possible MSE of solutions obtained with the algorithm, with respect to the solutions found in sequential implementation as shown in table 6. With the intention of validating the quality of the results obtained, a statistical test has been used. First, the normality of the errors was verified with the Shapiro test. Since all error distributions follow a normal distribution, a parametric test was carried out using the T-test with 99% confidence to compare the results in the sequential and parallel version for each of the buildings. Each cell contains the p-value resulting from the t-test. A value < 0.01 means that there are significant differences between the samples compared in the test, i.e., the sequential and parallel versions of the algorithm in our case, and a value > 0.01 means that there are no differences between the two versions. We may conclude from this table that both versions show

	Measures	B1	B2	B3	B4
Sequential	Average Fitness	0.0168	0.0118	0.0182	0.0174
	Best Fitness	0.0140	0.0079	0.0130	0.0165
	Worst Fitness	0.0197	0.0145	0.0237	0.0245
	Standard Deviation	$\pm$0.0018	$\pm$0.0022	$\pm$0.0031	$\pm$0.0035
Parallel	Average	0.0153	0.0095	0.0152	0.0162
	Best Fitness	0.0117	0.0061	0.0107	0.0130
	Worst Fitness	0.0176	0.0122	0.0191	0.0218
	SD	$\pm$0.0017	$\pm$0.0018	$\pm$0.0025	$\pm$0.0027
t-Test		0.0821	0.0227	0.0305	0.0329

Table 6: Average MSE of sequential and parallel implementations.

similar behaviour in terms of accuracy. There are no significant differences except in the fourth case, caused by the variability in the solutions obtained experiments that correspond to the sequential implementation. Note that Figure 13 shows a greater box in that building. This is because the behaviour of that building is harder to model. Consequently, finding a model with good accuracy is more complicated.

Table 7 shows the percentage of GPU utilization of each of the kernels. Starting from the bottom upwards, the GI is the function that makes less use of the device, followed by the SO and the CMO. This is due to their computations are notably simpler than the rest. After them, CD is placed in the third place, this kernel must be launched every time all individuals are evaluated. It takes longer than the previous functions because it must calculate distances among all individuals. The function that needs the second more time to get their results is the NDS as this kernel has to perform a sorting algorithm according to the dominance of the solutions and this kind of algorithms often takes a great amount of time. Finally, as expected, the core of the algorithm is mainly focused on the FF. This kernel takes the longest time on account of the data that need to process and the number of iterations that need to be done to evaluate all individuals with all data. Nearly half of the time required to execute the algorithm is spent in FF. For this reason, our efforts were focused on improving and optimizing this part of the evolutionary algorithm by associating one individual per block. Thus, the number of threads was optimized according to the maximum num-

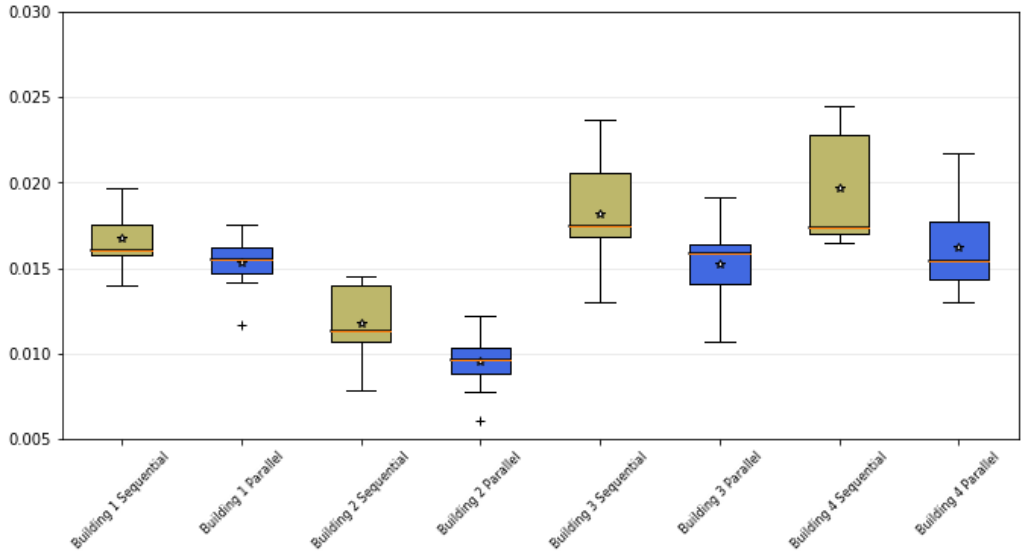


Figure 13: Boxplots of the error rate (MSE) for buildings.

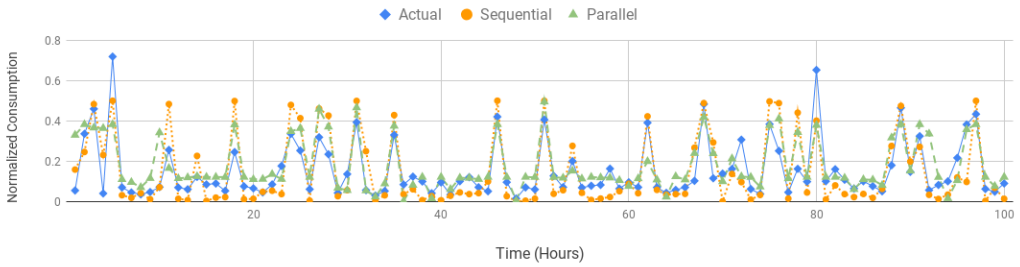
ber of neurons and therefore, all computations can be executed in parallel, achieving the excellent speed-up obtained. This fact demonstrates that our implementation might be considered a significant contribution in the current state-of-art since not only improves the parallel design of a hybrid solution of the NSGA-II with ANN but is also showing better figures regarding speed up in particularly complex functions of the algorithm. Additionally, it lessens the MSE with respect to other NSGA-II implementations on GPU.

Kernel Name	Usage Rate
FF	48.11%
NDS	39.55%
CD	11.46%
CMO	0.23%
SO	0.07%
GI	0.01%

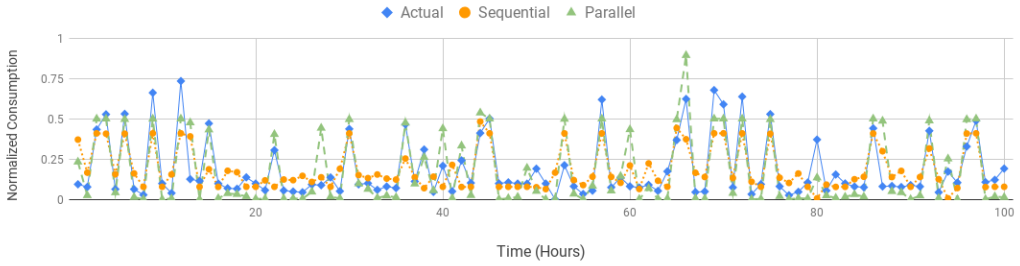
Table 7: Kernel profiling.

Finally, figure 14 shows a graphical representation of the forecast of EC for a total of 100 hours of a set of test data. In these plots, the horizontal

axis represents the hour that is predicted and the vertical axis represents the normalized consumption for that hour. Figure 14a shows the best result obtained and figure 14b shows the worst result of the parallel and sequential versions. As we can see in the figures the predicted results fit well to the real values in both cases. The results of both versions have been put together to visually verify how the fit does not differ considerably from each other.



(a)



(b)

Figure 14: Example of the (a) best and worst (b) predictions of energy consumption by both parallel and sequential models.

6.1. Performance comparison between NSGA-II and Backpropagation.

After showing the good results obtained in the performance of our parallel NSGA-II algorithm in comparison to its sequential version, an experiment has been carried out to compare the algorithm developed in this paper with the well-known Backpropagation (BP) one. BP is one of the most popular local search algorithms used for neural network training.

Given the long execution time that the BP algorithm implementation takes, a total of 9 runs of the algorithm were launched for each of the buildings included in this study. Each of the executions has been performed to a total of 50 iterations. The number of neurons in the Nsga-II experiment has been set between a range of 4 and 32, however the range chosen for BP algorithm execution was coarser (by increasing in two from 2 to 32), since the average execution time of the BP took too long. The average time taken to run the parallel NSGA-II and then obtain a set of optimal solutions is 25.932 s., while the BP had an average execution time of 10 hours to obtain the results with respect to all the different number of neurons aforementioned. In table 8 we can see the performance comparison for both algorithms. This table shows, for each building, the best model obtained after conducting the experiment. The first column shows the identifier of each building. In column 2 and 5 we can see the number of hidden neurons, which the best model obtains for the different buildings with the NSGA-II and the BP respectively. In columns 3 and 6 it can be seen that the error given by the NSGA-II and BP for the B2 building is almost equal, and it is slightly better in the rest of the buildings as for the BP results. Finally, columns 4 and 7 show the execution time of the NSGA-II and BP, it can be seen that our algorithm is much faster than the BP. Although the error of the models is important, in this work we have focused more in improving the execution time sacrificing some tenths of error since, since in some problems is most relevant to obtain faster and a little less precise models.

7. Conclusion

This paper presents a parallel implementation of NSGA-II developed in GPU to train an ANN whose evaluation has also been implemented in parallel in GPU to solve the problem of predicting EC. The objective of our proposal is to reduce the execution time of the different functions that compose the algorithm. Our implementation is able to obtain an optimal neural network with the least number of neurons necessary to learn the set of examples and also

Building	NSGA-II neurons	NSGA-II MSE	NSGA-II time (seconds)	BP neurons	BP MSE	BP time (seconds)
B1	4	0.0117	26.998	14	0.0044	7821.44
B2	4	0.0061	25.965	4	0.0049	595.40
B3	4	0.0107	25.441	18	0.0054	12200
B4	4	0.0130	25.899	18	0.0045	13089.30

Table 8: NSGA-II and BP performance comparison.

provides the weights trained making a minimum error for a specific building, and thus taking less than 60 seconds with a total of 20000 evaluations and an average MSE of 0.0093, achieving in some functions of the algorithm a speedup of 788 compared with the sequential version.

In the experimentation section, different results of the behaviour of the algorithms developed in this work have been shown. Our proposal obtains good results in time and accuracy with respect to the sequential version on account of the good utilization of the resources of the GPU as they are the deployment of memory of fast access. It is worth mentioning that 100% of GPU use has been achieved, for which it has been necessary an elaborate CUDA development work and extensive experimentation. On the other hand, the use of the GPU limits the parameters of the algorithm since it cuts down the size of the population of individuals that can be used due to constraints by GPU resources availability. Finally, the version developed in parallel has shown to obtain better results in execution time without loss of precision. This assertion has been verified with statistical tests, which have shown that there are no significant differences between the accuracy obtained in the serial version and the parallel version, even though there is a great difference in the performance of an advantageous way by the parallel version.

In future work we would like to explore and solve our current device limits by deploying our implementation on a GPU with more computational resources, as the Nvidia Tesla with the intention of knowing if a higher population size and a higher number of executions will produce better results without increasing too much the execution time of the entire algorithm. We are also confident that CUDA's future capabilities will be improved by helping us to scale our problem. Besides, as a continuation of this work, several lines of research remain open and in which it is possible to continue working. These lines have arisen during the development of this work and we hope to

be able to work in the near future. One of these future lines is to develop more complex networks for the evaluation of the models, intending to achieve an improvement in the precision of the results. On the other side, we would like to analyze the aggregation of other data sources to reinforce the results obtained. These new data sources could be measurements of temperature, humidity, luminosity and occupancy.

Acknowledgement

This work has been supported by the project TIN201564776-C3-1-R.

References

[1] S. Belt Ibérica, En los ultimos 50 años, el planeta se ha degradado más que en 100 siglos, http://www.belt.es/noticias/2005/abril/26/planeta.htm, 2005.

[2] Y. Feng, D. Gong, Q. Zhang, S. Jiang, L. Zhao, N. Cui, Evaluation of temperature-based machine learning and empirical models for predicting daily global solar radiation, Energy Conversion and Management 198 (2019) 111780.

[3] Y. Liu, Y. Zhou, D. Wang, Y. Wang, Y. Li, Y. Zhu, Classification of solar radiation zones and general models for estimating the daily global solar radiation on horizontal surfaces in china, Energy Conversion and Management 154 (2017) 168 – 179.

[4] J.-K. Park, A. Das, J.-H. Park, A new approach to estimate the spatial distribution of solar radiation using topographic factor and sunshine duration in south korea, Energy Conversion and Management 101 (2015) 30 – 39.

[5] L. G. B. Ruiz, M. I. Capel, M. C. Pegalajar, Parallel memetic algorithm for training recurrent neural networks for the energy efficiency problem, Applied Soft Computing 76 (2019) 356–368.

[6] L. G. B. Ruiz, R. Rueda, M. P. Cuéllar, M. C. Pegalajar, Energy consumption forecasting based on elman neural networks with evolutive optimization, Expert Systems with Applications 92 (2018) 380–389.

[7] L. B. d. Oliveira, C. G. Marcelino, A. Milanés, P. E. M. Almeida, L. M. Carvalho, A successful parallel implementation of NSGA-II on GPU for the energy dispatch problem on hydroelectric power plants, in: 2016 IEEE Congress on Evolutionary Computation (CEC), pp. 4305–4312.

[8] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. P. Turian, D. Warde-Farley, Y. Bengio, Theano : A CPU and GPU Math Compiler in Python.

[9] V. Roberge, M. Tarbouchi, G. Labonté, Fast Genetic Algorithm Path Planner for Fixed-Wing Military UAV Using GPU, IEEE Transactions on Aerospace and Electronic Systems 54 (2018) 2105–2117.

[10] R. S. Sinha, S. Singh, S. Singh, V. K. Banga, Accelerating Genetic Algorithm Using General Purpose GPU and CUDA.

[11] D. D'Agostino, G. Pasquale, I. Merelli, A Fine-Grained CUDA Implementation of the Multi-objective Evolutionary Approach NSGA-II: Potential Impact for Computational and Systems Biology Applications, in: C. DI Serio, P. Liò, A. Nonis, R. Tagliaferri (Eds.), Computational Intelligence Methods for Bioinformatics and Biostatistics, Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 273–284.

[12] I. M. Coelho, V. N. Coelho, E. J. d. S. Luz, L. S. Ochi, F. G. Guimarães, E. Rios, A GPU deep learning metaheuristic based model for time series forecasting, Applied Energy 201 (2017) 412–418.

[13] O. Guerra Santin, Behavioural Patterns and User Profiles related to energy consumption for heating, Energy and Buildings 43 (2011) 2662–2672.

[14] D. B. Araya, K. Grolinger, H. F. ElYamany, M. A. M. Capretz, G. Bitsuamlak, An ensemble learning framework for anomaly detection in building energy consumption, Energy and Buildings 144 (2017) 191–206.

[15] Y. Zhang, W. Chen, J. Black, Anomaly detection in premise energy consumption data, in: 2011 IEEE Power and Energy Society General Meeting, pp. 1–8.

[16] B. B. Ekici, U. T. Aksoy, Prediction of building energy consumption by using artificial neural networks, Advances in Engineering Software 40 (2009) 356–362.

[17] S. Heiple, D. J. Sailor, Using building energy simulation and geospatial modeling techniques to determine high resolution building sector energy consumption profiles, Energy and Buildings 40 (2008) 1426–1436.

[18] S. Barak, S. S. Sadegh, Forecasting energy consumption using ensemble ARIMA–ANFIS hybrid algorithm, International Journal of Electrical Power & Energy Systems 82 (2016) 92–104.

[19] Y.-S. Lee, L.-I. Tong, Forecasting energy consumption using a grey model improved by incorporating genetic programming, Energy Conversion and Management 52 (2011) 147–152.

[20] J. Fan, X. Wang, L. Wu, F. Zhang, H. Bai, X. Lu, Y. Xiang, New combined models for estimating daily global solar radiation based on sunshine duration in humid regions: A case study in south china, Energy Conversion and Management 156 (2018) 618 – 625.

[21] B. Dong, C. Cao, S. E. Lee, Applying support vector machines to predict building energy consumption in tropical region, Energy and Buildings 37 (2005) 545–553.

[22] L. Ekonomou, Greek long-term energy consumption prediction using artificial neural networks, Energy 35 (2010) 512–517.

[23] K. Li, H. Su, J. Chu, Forecasting building energy consumption using neural networks and hybrid neuro-fuzzy system: A comparative study, Energy and Buildings 43 (2011) 2893–2899.

[24] B. Choubin, G. Zehtabian, A. Azareh, E. Rafiei-Sardooi, F. Sajedi-Hosseini, Kişi, Precipitation forecasting using classification and regression trees (CART) model: a comparative study of different approaches, Environmental Earth Sciences 77 (2018) 314.

[25] A. Azadeh, S. F. Ghaderi, S. Tarverdian, M. Saberi, Integration of artificial neural networks and genetic algorithm to predict electrical energy consumption, Applied Mathematics and Computation 186 (2007) 1731–1741.

[26] L. Wu, G. Huang, J. Fan, F. Zhang, X. Wang, W. Zeng, Potential of kernel-based nonlinear extension of arps decline model and gradient boosting with categorical features support for predicting daily global solar radiation in humid regions, Energy Conversion and Management 183 (2019) 280 – 295.

[27] J. Fan, X. Wang, L. Wu, H. Zhou, F. Zhang, X. Yu, X. Lu, Y. Xiang, Comparison of support vector machine and extreme gradient boosting for predicting daily global solar radiation using temperature and precipitation in humid subtropical climates: A case study in china, Energy Conversion and Management 164 (2018) 102 – 111.

[28] R. A. Conde-Gutiérrez, U. Cruz-Jacobo, A. Huicochea, S. R. Casolco, J. A. Hernández, Optimal multivariable conditions in the operation of an absorption heat transformer with energy recycling solved by the genetic algorithm in artificial neural network inverse, Applied Soft Computing 72 (2018) 218–234.

[29] Y. Dong, J. Wang, Z. Guo, Research and application of local perceptron neural network in highway rectifier for time series forecasting, Applied Soft Computing 64 (2018) 656–673.

[30] G. S. Georgiou, P. Christodoulides, S. A. Kalogirou, Implementing artificial neural networks in energy building applications — A review, in: 2018 IEEE International Energy Conference (ENERGYCON), pp. 1–6.

[31] J. Krzywanski, K. Grabowska, F. Herman, P. Pyrka, M. Sosnowski, T. Prauzner, W. Nowak, Optimization of a three-bed adsorption chiller by genetic algorithms and neural networks, Energy Conversion and Management 153 (2017) 313 – 322.

[32] M. Heitzler, J. C. Lam, J. Hackl, B. T. Adey, L. Hurni, GPU-Accelerated Rendering Methods to Visually Analyze Large-Scale Disaster Simulation Data, Journal of Geovisualization and Spatial Analysis 1 (2017) 3.

[33] S. S. Stone, J. P. Haldar, S. C. Tsao, B. Sutton, Z.-P. Liang, et al., Accelerating advanced mri reconstructions on gpus, Journal of parallel and distributed computing 68 (2008) 1307–1318.

[34] A. Syberfeldt, T. Ekblom, A comparative evaluation of the GPU vs. the CPU for parallelization of evolutionary algorithms through multiple

independent runs, International Journal of Computer Science & Information Technology (IJCSIT) 9 (2017) 1–14.

[35] OpenCL - The open standard for parallel programming of heterogeneous systems, https://www.khronos.org/opencl/, 2013.

[36] J. Sanders, E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 1st edition, 2010.

[37] OpenCL vs. CUDA: Which Has Better Application Support, https://create.pro/blog/opencl-vs-cuda/, 2017.

[38] C. H. Dagli, Artificial Neural Networks for Intelligent Manufacturing, Springer Science & Business Media, 2012. Google-Books-ID: K4ftCAAAQBAJ.

[39] S. Samarasinghe, Neural Networks for Applied Sciences and Engineering : From Fundamentals to Complex Pattern Recognition, Auerbach Publications, 2016.

[40] D. Roffman, G. Hart, M. Girardi, C. J. Ko, J. Deng, Predicting non-melanoma skin cancer via a multi-parameterized artificial neural network, Scientific Reports 8 (2018) 1701.

[41] J. Škutová, Weights initialization methods for mlp neural networks, Vysoká škola báňská-Technická Univerzita Ostrava (2008).

[42] L. G. B. Ruiz, M. Cuéllar, M. Delgado, M. C. Pegalajar, An application of non-linear autoregressive neural networks to predict energy consumption in public buildings, Energies 9 (2016) 684.

[43] M. Delgado, M. C. Pegalajar, A multiobjective genetic algorithm for obtaining the optimal size of a recurrent neural network for grammatical inference, Pattern Recognition 38 (2005) 1444–1456.

[44] A. Blanco, M. Delgado, M. C. Pegalajar, A real-coded genetic algorithm for training recurrent neural networks, Neural Networks 14 (2001) 93–105.

[45] P. Pospichal, J. Jaros, J. Schwarz, Parallel Genetic Algorithm on the CUDA Architecture, in: C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. I. Esparcia-Alcazar, C.-K. Goh, J. J. Merelo, F. Neri, M. Preuß, J. Togelius, G. N. Yannakakis (Eds.), Applications of Evolutionary Computation, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2010, pp. 442–451.

[46] E. Zitzler, K. Deb, L. Thiele, Comparison of Multiobjective Evolutionary Algorithms: Empirical Results, Evol. Comput. 8 (2000) 173–195.

[47] C. Von Lücken, A. Hermosilla, B. Barán, Algoritmos evolutivos para optimización multiobjetivo: un estudio comparativo en un ambiente paralelo asíncrono, X Congreso Argentino de Ciencias de la Computación (2004).

[48] C. A. Coello Coello Coello, A Short Tutorial on Evolutionary Multiobjective Optimization, in: E. Zitzler, L. Thiele, K. Deb, C. A. Coello Coello, D. Corne (Eds.), Evolutionary Multi-Criterion Optimization, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2001, pp. 21–40.

[49] G. Wu, J. L. Greathouse, A. Lyashevsky, N. Jayasena, D. Chiou, GPGPU performance and power estimation using machine learning, in: 2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), pp. 564–576.

[50] M. C. Pegalajar, M. Capel, L. G. B. Ruiz, M. Delgado, A parallel approach to intelligent data analysis for efficient energy management in distributed facilities (pioneer), in: European Space Projects: Developments, Implementations and Impacts in a Changing World - Volume 1: EPS Porto 2017,, INSTICC, SciTePress, 2017, pp. 28–49.

[51] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A Fast and Elitist Multiobjective Genetic Algorithm: NSGA-II, Trans. Evol. Comp 6 (2002) 182–197.

[52] M. Delgado, M. P. Cuellar, M. C. Pegalajar, Multiobjective hybrid optimization and training of recurrent neural networks, IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 38 (2008) 381–403.

[53] A. H. Wright, Genetic Algorithms for Real Parameter Optimization, in: Foundations of Genetic Algorithms, Morgan Kaufmann, 1991, pp. 205–218.

[54] M. Delgado, M. P. Cuellar, M. C. Pegalajar, Multiobjective Hybrid Optimization and Training of Recurrent Neural Networks, IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics) 38 (2008) 381–403.

[55] C. A. C. Flórez, R. A. Bolaños, A. M. Cabrera, Algoritmo Multiobjetivo Nsga-Ii Aplicado Al Problema De La Mochila., Scientia Et Technica XIV (2008) 206–211.

[56] K. Deb, D. Kalyanmoy, Multi-Objective Optimization Using Evolutionary Algorithms, John Wiley & Sons, Inc., New York, NY, USA, 2001.

[57] Xoshiro, xoroshiro generators and the PRNG shootout, http://xoshiro.di.unimi.it/, 2018.