Expert Systems With Applications 230 (2023) 120661

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

CUDA-bigPSF: An optimized version of bigPSF accelerated with graphics processing Unit

D. Criado-Ramón [a,1,*], L.B.G. Ruiz [b,2], M.C. Pegalajar [a,3]

[a] Department of Computer Science and Artificial Intelligence, University of Granada C/Periodista Daniel Saucedo Aranda s.n, 18014, Granada, Granada, Spain
[b] Department of Software Engineering, University of Granada C/Periodista Daniel Saucedo Aranda s.n, 18014, Granada, Granada, Spain

ARTICLE INFO

Keywords:
Time series forecasting
Hybrid models
CUDA
Energy
Big data

ABSTRACT

Accurate and fast short-term load forecasting is crucial in efficiently managing energy production and distribution. As such, many different algorithms have been proposed to address this topic, including hybrid models that combine clustering with other forecasting techniques. One of these algorithms is bigPSF, an algorithm that combines K-means clustering and a similarity search optimized for its use in distributed environments. The work presented in this paper aims to improve the time required to execute the algorithm with two main contributions. First, some of the issues of the original proposal that limited the number of cores simultaneously used are studied and highlighted. Second, a version of the algorithm optimized for Graphics Processing Unit (GPU) is proposed, solving the previously mentioned issues while taking into account the GPU architecture and memory structure. Experimentation was done with seven years of real-world electric demand data from Uruguay. Results show that the proposed algorithm executed consistently faster than the original version, achieving speedups up to 500 times faster during the training phase.

1. Introduction

Since electricity was discovered, humanity has created a steadily growing number of devices that make use of electricity. Most of the time, people use electricity simultaneously for multiple applications such as lighting, refrigeration, cooling, or heating, among others. The energy required for this is usually provided via an interconnected electricity network known as "power grid".

However, many complex factors have to be taken into account in the management of the power grid, such as the use of renewable energy sources, which rely on weather conditions or electricity transmission losses. Thus, it is common to use Artificial Intelligence (AI) systems to assist in the management of the power grid, particularly in the prediction of energy demand and renewable energy production (Bose, 2017).

Over the last two decades, technical advancements have led to the higher use of smart meters (Zheng et al., 2013), devices that measure the electricity imported and exported from the grid by the consumer in real time. These devices also provide the energy provider with energy consumption data periodically, which can be used to optimize energy production and distribution in entire regions. With the adoption of these devices and the increasing energy consumption transparency of public entities and governments, researchers have a wide range of data available to study energy consumption. However, in many cases, usage of this type of data poses considerable challenges, as the sheer amount of data may significantly increase the computational power required to train these AI systems.

The relevance of energy in our current society has led to its study under many different scenarios. The algorithms used for this task (Kong et al., 2019) cover a wide range from easy-to-understand and interpret models, such as ARIMA, to highly accurate black-box models: neural networks, deep learning, and ensembles of different models, among others. Pattern Sequence-based Forecasting (PSF) (Martinez Alvarez

Abbreviations: ANN, Artificial Neural Network; CNN, Convolutional Neural Network; CUDA, Compute Unified Device Architecture; GPU, Graphics Processing Unit; LSTM, Long-Short Term Memory; MAE, Mean Absolute Error; MAPE, Mean Absolute Percentage Error; PSF, Pattern Sequence-Based Forecasting; RDD, Resilient Distributed Dataset; RMSE, Root Mean Square Error; SOM, Self-Organizing Map.

* Corresponding author.
E-mail addresses: dcriado@ugr.es (D. Criado-Ramón), bacaruiz@ugr.es (L.B.G. Ruiz), mcarmen@decsai.ugr.es (M.C. Pegalajar).
[1] 0000-0003-3030-792X.
[2] 0000-0001-6716-5115.
[3] 0000-0001-9408-6770.

https://doi.org/10.1016/j.eswa.2023.120661
Received 31 January 2023; Received in revised form 6 May 2023; Accepted 30 May 2023
Available online 2 June 2023
0957-4174/© 2023 The Author(s). Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

et al., 2011) is an interesting middle-ground approach that has previously provided remarkable results in the energy field. This algorithm creates hybrid models that combine clustering and additional methods to extract patterns and make computations based on those patterns. PSF and many of its improved versions present some interesting properties in big data scenarios, e.g., the clustering-based pattern extraction reduces the computational cost for the second step of the algorithm, the pattern sequence-based forecast. However, they still require intense computational power as each prediction requires an independent clustering and pattern sequence-based forecast, severely hindering the time needed to train and predict with these models.

Parallel and distributed approaches are frequently used to reduce the time needed to train algorithms with high computational demands. An improved specialized version for Apache Spark clusters called "bigPSF" was presented in 2020 (Pérez-Chacón et al., 2020). However, there is no work to this date that studies PSF algorithms under parallel architectures. In this paper, a new version of the bigPSF algorithm accelerated with Graphic Processing Units (GPUs) is proposed, hereafter referred to as "CUDA-bigPSF". Two main contributions are provided to this research field in this work:

- The first GPU implementation of a pattern sequence-based algorithm is developed, reducing significantly the time required to train and use this model.
- Some of the issues of the original BigPSF proposal are highlighted and how they could be solved to obtain better performance when using a distributed environment.

This manuscript is structured as follows: Section 2 reviews relevant related papers on pattern sequence-based forecasting and GPU algorithms with a focus on big data energy problems. Section 3 describes the CUDA/GPU architecture and explains the CUDA-bigPSF algorithm. Section 4 studies the GPU implementation's accuracy, speedup, and scalability. Lastly, section 5 draws the most relevant accomplishments of this work and proposes future lines of research.

2. Related works.

This section is structured in two independent parts and reviews the most relevant related works in the field. In the first part, works related to the PSF algorithm are reported and discussed. In the second part, we review the use of the GPU in AI and, more specifically, in the energy field.

The PSF algorithm was published in 2011 (Martinez Alvarez et al., 2011). This algorithm starts by applying K-means clustering to transform the time series before the prediction date into a sequence of cluster identifiers (labels). Afterwards, the algorithm splits the labeled sequence using a sliding window of size W. In order to make the prediction, the algorithm looks for similar patterns to the last created with the sliding window, i.e., the pattern of the W days before the prediction date. The final prediction is the average of all the occurrences found using the original time series.

PSF has shown excellent results when working with energy data, and, as such, it is its primary use. Nevertheless, it has also been used to forecast energy prices (Jin et al., 2015), wind speed (Bokde et al., 2017), solar power (Fujimoto & Hayashi, 2012), or even to impute missing data (Bokde et al., 2018). Several authors have proposed variants and improvements to overcome some of the original algorithm's limitations. In (Jin et al., 2015) the authors used the Self-Organizing Map (SOM) and neural networks to create a specialized version that preserves the input space's topological properties. Similarly, in (Martínez-Álvarez et al., 2019) the authors proposed a specialized version for functional data (funPSF) through the use of a functional clustering algorithm, funHDDC (Bouveyron & Jacques, 2011). They also created a version with specialized models for each day of the week (7-funPSF) that provided significantly better results than the previous one. In (Shen et al., 2013) the authors evaluated using PSF with five different clustering methods (K-means, SOM, K-medoids, Hierarchical clustering, and Fuzzy C-means) individually and in an ensemble. (Jin et al., 2014) introduced a weighted mean that gives more relevance to the most frequent patterns each day of the week. Lastly, the algorithm our work is based in (Pérez-Chacón et al., 2020) proposes adapting the original PSF algorithm for clusters with Apache Spark. Beyond the distributed approach, this algorithm also included a weighted mean that gives more relevance to the matches closer to the prediction date and a grid search of hyperparameters to find the best solution at the expense of more computational power.

The rise of big data and many other data science methodologies that are computationally expensive, such as AutoML, have led to a higher interest in parallel and distributed algorithms capable of providing similar results in less time. Researchers and companies have published open-source access to GPU-accelerated implementations of traditional machine learning algorithms. Facebook's FAISS library (Johnson et al., 2021) optimizes similarity search and clustering of dense vectors, providing fast K-means clustering and nearest neighbour search algorithms. ThunderSVM (Wen et al., 2018) provides a GPU adaptation of Support Vector Machines with the standard kernels used for classification and regression. Most gradient boosting machines provide GPU-accelerated implementations, such as XGBoost (Chen & Guestrin, 2016) or LightGBM (Ke et al., 2017). NVIDIA recently launched cuML (Raschka et al., 2020), a CUDA-specific open-source library to accelerate all the algorithms included in the popular Python package scikit-learn.

Table 1
Summary of related works using the GPU on the energy field.

Citation	Framework	Application	Contributions
(Kintsakis et al., 2015)	No	Demand and price forecast	They propose a parallelized version of Particle Swarm Optimization to train Local Linear Wavelet Neural networks.
(Coelho et al., 2017)	No	Appliance load forecast	They propose a hybrid model combining fuzzy rules and metaheuristics accelerated with GPU.
(Tian et al., 2019)	PyTorch	Smart meters	They developed a transfer learning methodology to train large sets of smart meters based on similarity.
(Kim & Cho, 2019)	Keras	Residential buildings consumption	They propose a combination of Convolutional Neural Networks (CNN) and Long-Short Term Memory (LSTM).
(Iruela et al., 2020)	No	Public buildings consumption	They develop a parallel version of the NSGA-II metaheuristic to train feed-forward neural networks.
(Iruela et al., 2021)	Tensorflow	Public buildings consumption	They present a methodology to simultaneously train specialized neural network models for each hour of the day.
(Said & Alanazi, 2022)	Keras	Solar energy production	They combined the use of autoenconders for feature extraction with LSTM neural networks.
(Haque & Rahman, 2022)	Tensorflow	Commercial buildings consumption.	They combined the use of regularized LSTM and Recurrent Neural Networks (RNN) and developed a heuristic to find the optimal neural network configuration.
(Chen et al., 2023)	Tensorflow	Smart meters	They developed a federated framework for smart meters that makes of generative adversarial networks (GAN) to create privacy-preserving synthetic data.

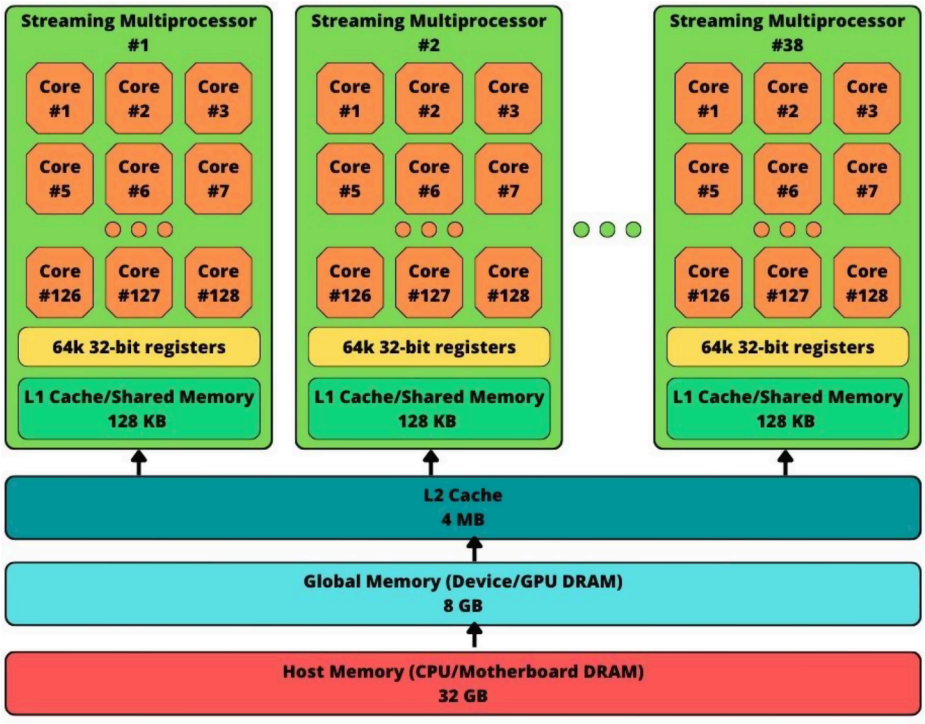

Fig. 1. A schematic of the memory layout and multiprocessors of the GPU device used in this research.

Neural network frameworks, such as Tensorflow (Abadi et al., 2016) or PyTorch (Paszke et al., 2019), provide GPU-accelerated implementations optimized for deep neural networks and represent the broadest use of GPU in AI research nowadays.

The energy field is no different, and most relevant recently published works use GPU-accelerated neural network frameworks or use parallelized metaheuristics to train neural networks. Table 1 presents a summary of the most relevant works on the energy field using the GPU.

Although PSF algorithms have previously shown excellent results in energy forecasting, to the best of our knowledge, the use of GPU for PSF algorithms has yet to be studied. As such, the study and proposal of our GPU-accelerated algorithm, CUDA-bigPSF, is justified.

3. Materials and Methods.

3.1. The CUDA architecture.

GPUs were initially conceived to accelerate graphical computation. However, the massively parallel architecture of the GPU was also of interest in many other fields that could use it to accelerate their applications and simulations, leading to an evolution of the GPU programming model towards the paradigm known nowadays as General-Purpose GPU (GPGPU). As part of this evolution, new user-friendly languages were created to avoid the complexity of writing general-purpose code through graphical APIs or assembly. An example of this is Compute Unified Device Architecture (CUDA), a proprietary extension of C++, made to facilitate GPGPU programming with NVIDIA graphics cards.

In CUDA, the set of instructions to be executed by each GPU thread are written in special functions called kernels. The programmer specifies the kernel's total number of threads by dividing the total number of threads in a grid of blocks. Each block always contains a fixed number of threads, and all threads within the same block can be synchronized and access a special programmer-managed cache for fast collaboration. The grid indicates the total number of blocks required to execute the kernel. The number of threads in a given block can be provided in one, two or three dimensions to overcome some limitations and to provide easier abstractions in some algorithms involving complex structures such as matrices. The same applies to the dimension of a grid.

A CUDA-capable GPU has one or more streaming multiprocessors, each containing a set of cores, registers, cache memory and a scheduler. When a kernel is launched, the blocks are distributed through the different multiprocessors. All threads within the same block are executed concurrently, and multiple blocks can be executed concurrently by the same multiprocessor. At its core, the CUDA architecture uses a Single Instruction Multiple Threads (SIMT) approach where 32 contiguous threads (a "warp") will execute the same instruction independently of the number of threads used in a block. As such, branching code can negatively impact the performance of the GPU algorithm, as both options must be evaluated before proceeding with the next instruction, even if only one thread in the warp takes the alternative branch.

Memory accesses are one of the primary bottlenecks of GPU-accelerated algorithms. As such, understanding the GPU memory hierarchy (Fig. 1), its advantages and caveats is critical in GPU algorithm development. The GPU presents a slower and bigger global memory used to communicate with the CPU (host) that all threads of the GPU can access. Furthermore, it presents up to two levels of cache memory (L1 cache for each multiprocessor and L2 cache for all multiprocessors). When writing a kernel, the developer can decide whether to store the variable in the global scope (global memory), local scope, and the specialized section of the L1 cache to cooperate with threads within the same block called "shared memory". Variables in the local scope follow similar rules to those in the global memory, but the compiler can also store them in the registers under certain circumstances. Nevertheless, accesses to global and local memory can also be fast if we use a predictable access pattern, as they will be cached once a store or load happens. Only cache misses will hinder the performance. Lastly, we must note that there are some other specialized memory abstractions, such as the constant memory (read-only) or the texture memory, that we have decided to omit for simplicity as they are irrelevant to this paper.

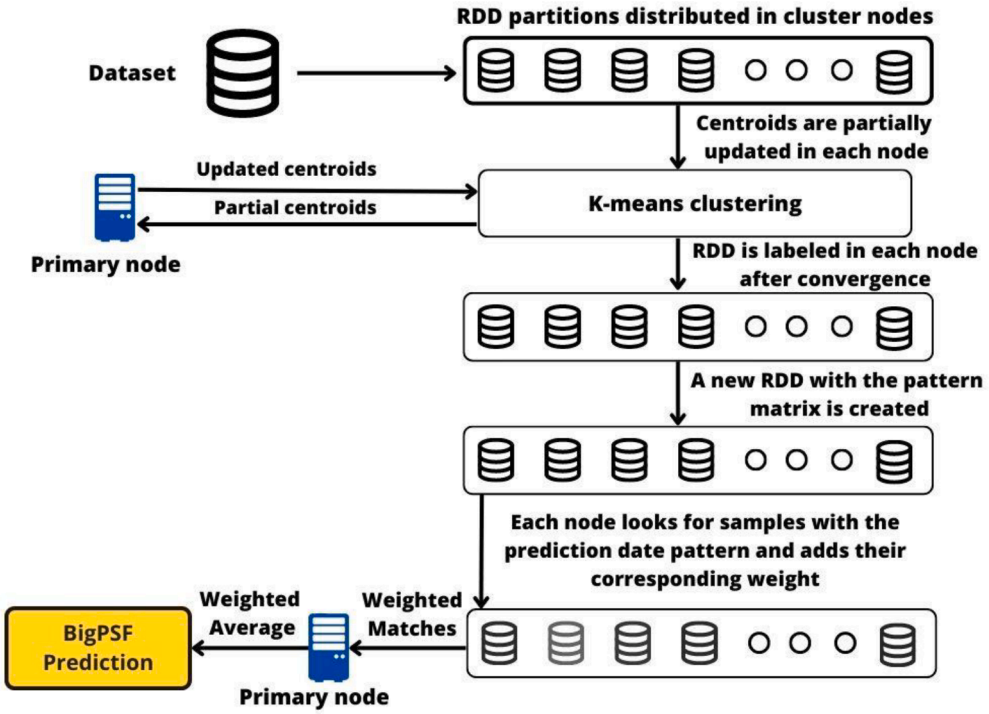

Fig. 2. A general scheme of the steps done by the bigPSF algorithm for each prediction.

3.2. The bigPSF algorithm.

BigPSF provides an improved PSF algorithm for distributed environments. The training process of the algorithm finds the optimal hyperparameters (number of clusters and window size) through a grid search evaluated in a validation partition. The training and test processes are done sequentially over all the days on their corresponding partition, using the additional computational power to accelerate each prediction.

The BigPSF accelerated prediction algorithm (Fig. 2) starts by creating a distributed structure denominated RDD (Resilient Distributed Dataset) from the original dataset samples before the prediction date.

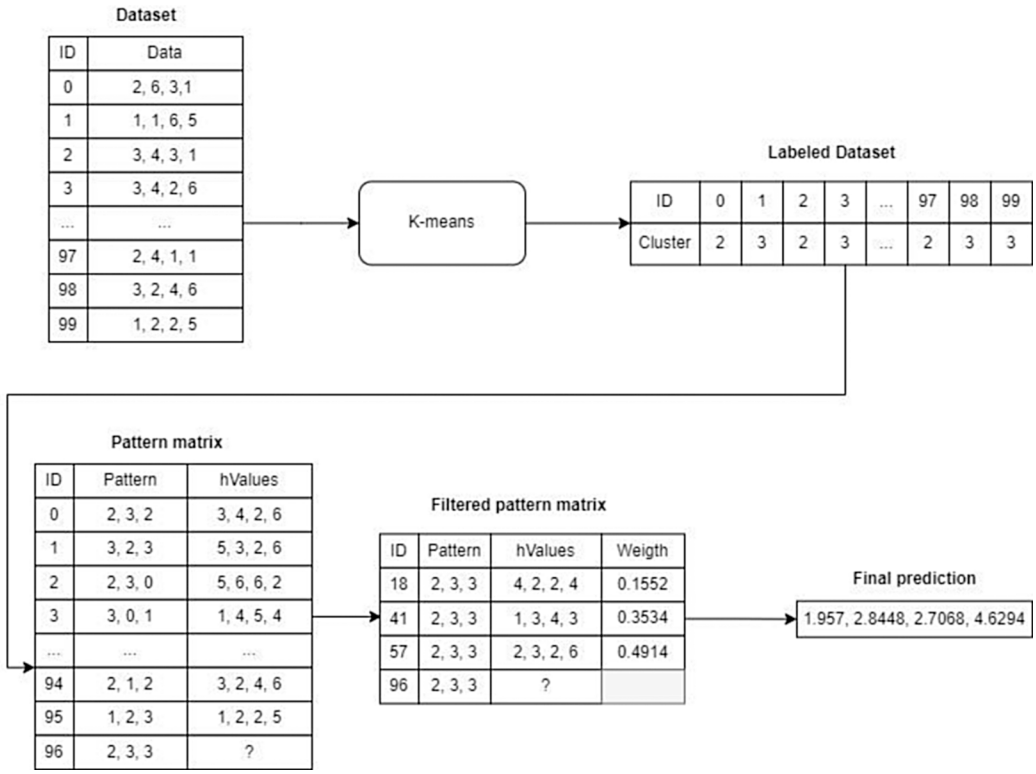

Fig. 3. An example of how the BigPSF algorithm calculates a prediction in a simulated dataset for K = 5 and W = 3.

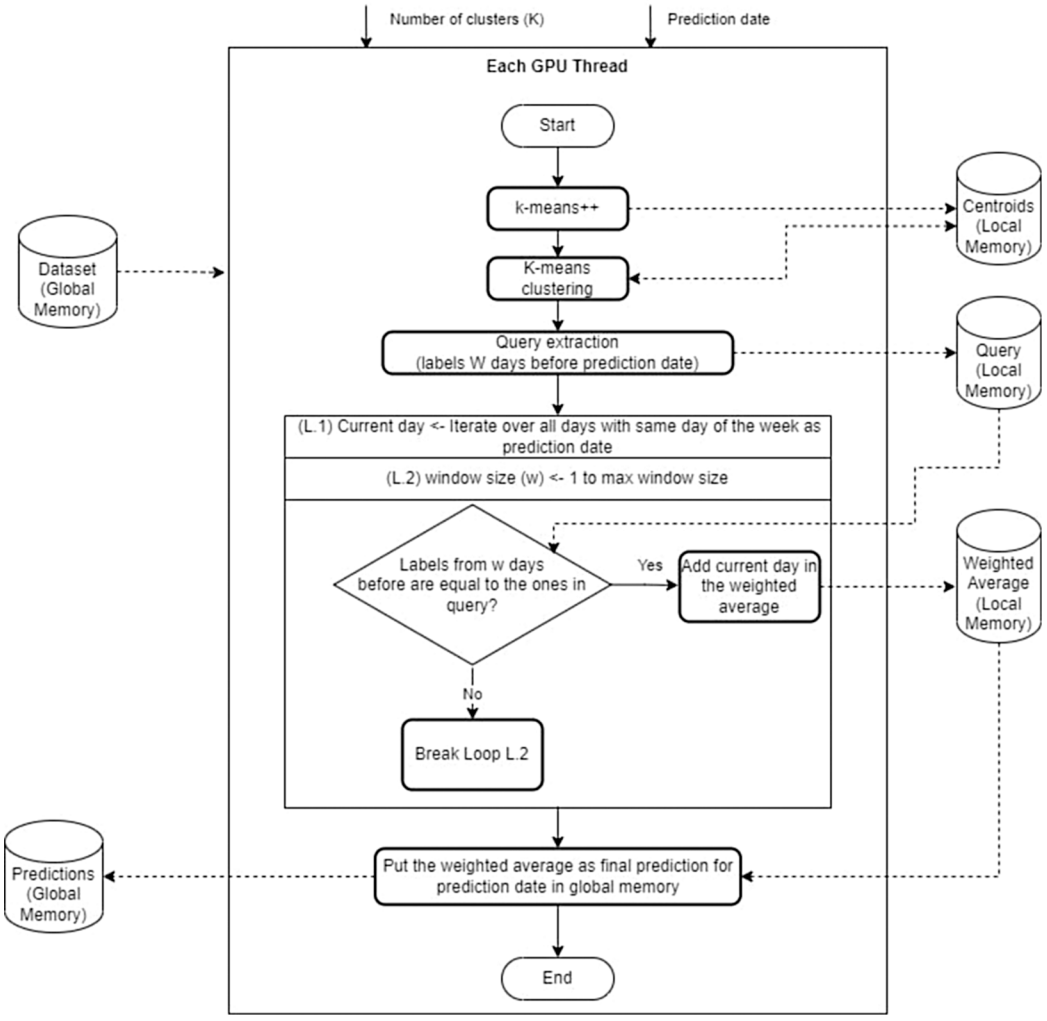

Fig. 4. A flowchart of the work executed by each thread in CUDA-bigPSF.

This RDD is shuffled into random partitions distributed on the nodes available in the cluster. The algorithm continues by applying K-means over the RDD. Centroids are initialized using the k-means++ algorithm (Arthur & Vassilvitskii, 2007).

Afterwards, each node finds the closest cluster for the partitions of the RDD available in the node and computes a partial centroids update. After each iteration, partial centroids are communicated to the primary node to obtain the final centroids of the iteration. K-means clustering ends after reaching a maximum number of iterations or convergence. The clustering process finishes with the creation of a new RDD, in which each sample is transformed to its closest cluster identifier. Each compute node does this last step independently, as synchronization is unnecessary. Then, the algorithm creates its more complex structure, the "pattern matrix", in a new RDD. Each row of this RDD contains a row identifier *id*, a sequence of *W* labels from the days between *id* and $id + W - 1$, and a data copy (hValue) of the day $id + W$ of the original dataset. This structure is generated by grouping all the possible sequences of labels of length *W* from the previous RDD. Finally, the algorithm filters all rows in the pattern matrix that share the same pattern and day of the week of the prediction date. The prediction is the weighted average of the data copies sharing the same sequence of labels and day of the week. This weight for each match is calculated as:

$$w_i = \frac{id_i}{\sum_{j \in matches} id_j} \quad (1)$$

where id_i is the row identifier of the match and *matches* contains all pattern occurrences in the pattern matrix.

A small example of how the bigPSF calculates a prediction is provided in Fig. 3, where the algorithm is computing the prediction for the day with ID 100 with a window size of W = 3 and a number of clusters K = 5. The algorithm starts by applying K-means with all the data prior to the day to be predicted and labeling them with their corresponding best cluster (upper row of the figure). Then, making use of the labeled dataset and the window size, the pattern matrix is constructed. The last row of the pattern matrix will indicate the pattern of the day to be predicted. All previous rows in the pattern matrix containing the same pattern are filtered and the final prediction is made with the weighted average of the *hValues* of the rows selected (using the weights provided in eq. 1).

3.3. CUDA-bigPSF.

The bigPSF algorithm shows some level of parallelism in two primary ways. In the first one (data parallelism), the computation for each sample in the dataset in parallel is done in paralle, as it proposed in the original bigPSF algorithm. In the second one, each prediction is made sequentially in each thread. There are several reasons why the second approach better when using the GPU. First, to obtain a significant speedup, it is imperative to keep all GPU threads busy. However, if a data parallelism strategy is used, there would be several instances in which some threads would have to wait until all the others finish for synchronization purposes. For example, after each K-means iteration,

the algorithm would need synchronization to ensure centroids are updated before the next iteration begins. Second, if the number of days in the dataset is smaller than the number of cores available in the GPU, using the first approach would keep more CUDA cores busy as long as three or more different numbers of clusters are being evaluated simultaneously. Last, the limited memory available in the GPU makes using the second approach better for scalability as memory accesses are local to the thread except for reading the dataset and writing the final result. Therefore, after each thread finishes its work, the local memory resources can be released to be used by another thread, significantly improving the scalability of the proposed approach.

As such, CUDA-bigPSF (Fig. 4) distributes the work in independent threads, each computing the prediction for a given date. They have access to the entire dataset and the final output structure in global memory. The thread identifier will be used to determine up to which date of the input dataset they should be able to access and where they must write their predictions in the output structure. The kernel (algorithm 1) will launch using a bi-dimensional grid of quantity of number of clusters to be evaluated by the minimum number of blocks to cover the validation (or test) partition.

Algorithm 1 CUDA-bigPSF (Each thread)

```
1: cluster_centers = KMeans(K, input, max_iterations, ∈)
2: query[0:max w-1] = closest_cluster(cluster_centers, input[rows(input)w:rows
   (input)])
3: weekday = n mod 7
4: for all i in weekday, weekday + 7, …, n-7 do
5:   for all w in 1,2,…,max_w do
6:     label = closest_cluster(cluster_centers, sample[i-w])
7:     if label = query[max_w-w] then
8:       weight = i - w + 1
9:       prediction_weights[w] += weight
10:      my_predictions[w-1] += weight * input[i]
11    else
12:      break
13:    end if
14:  end for
15: end for
16: for all w in 1, 2, …, max_w do
17:  if prediction_weights[w] != 0 then
18:    my_predictions[w] = my_predictions[w] / prediction_weights[w]
19:  else
20:    if w = 1 then
21:      Repeat for loop at line 4 with i from 0 to n-1
22:      my_predictions[w] = my_predictions[w] / prediction_weights[w]
23:    else
24:      my_predictions[w] = my_predictions[w-1]
25    end if
26:  end if
27: end for
28: Put my_predictions in its corresponding place in global memory
```

The kernel (algorithm each thread executes) starts with a standard implementation of Lloyd's K-means algorithm, initializing the centroids with the K-means++ algorithm. The clustering process finishes after reaching a maximum number of iterations or convergence. The objective function of the K-means algorithm is to minimize the Within Set Sum of Squared Errors (WSSSE) of each cluster, which is defined as follows (eq. 2):

$$WSSSE = \sum_{j=1}^{K} \sum_{x_i \in C_j} d\left(x_i, c_j\right)^2 \tag{2}$$

where $d\left(x_i, c_j\right)^2$ is the Euclidean distance between each sample x_i of the cluster Cj and the centroid of that cluster c_j. The algorithm iterates over the entire dataset once in each iteration, calculating the closest cluster to each sample, adding the sample to a new array to compute the centroids for the next iteration, and incrementing by one another structure used to count the number of samples in each cluster. The centroids for the next iteration are obtained by dividing these last two data structures

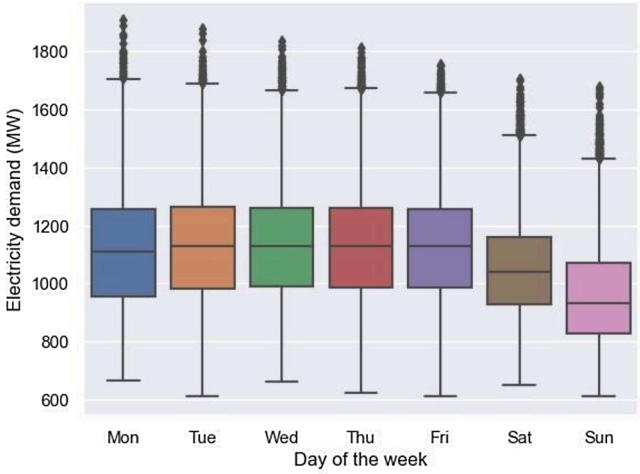

Fig. 5. Box plot of the energy consumption each day of the week.

(computing the mean).

Next, the pattern sequence-based algorithm starts. First, the query is calculated, i.e., the labels (cluster identifiers) for the w samples before the prediction date. Then, the algorithm strides weekly over the days in the dataset that share the same day of the week of the prediction date. To evaluate a dataset sample i, the label of the sample w days before it is computed. A match is found for a window size of one if it shares the same label as the position w of the query in reverse order. The same conditions apply for any window size w except the previous window size $w-1$ also needs to have a match. The computation for each w is done in ascending order to avoid any unnecessary calculations.

Every match found indicates that we must use the sample in the weighted average for the current prediction date and window size. To use only a stride over the entire dataset, two data structures are required to compute the weighted average, similar to the procedure previously used for the k-means centroids. Since the weights of BigPSF are a division that has the sum of all numerators in the denominator, whenever a match is found the sample is partially weighted by multiplying by the numerator and stored in a data structure and an additional data structure is used to eventually compute the sum of all numerators (denominator).

Lastly, the thread computes the division of the previous two data structures to obtain the prediction for a given data for all possible values of w that we are using. As the BigPSF algorithm specifies, the prediction obtained by a window of size $w-1$ is used if there are no matches for a window size of w. Occasionally the algorithm may fail for a window size of one. In those scenarios, all samples before the prediction date are used, regardless of the day of the week. The kernel finishes by putting the local structure containing the predictions for all possible values of w in their corresponding place in the global memory so the CPU can access the results.

As a last note, different clustering algorithms could be used instead of K-means. Although a similar approach to the one proposed for K-means could be used for any clustering algorithm, the optimal GPU implementation of the algorithm will change significantly depending on the data structures and computations required by each algorithm. Nevertheless, using K-means provides several advantages that will lead to substantially faster execution times than most clustering methods. This is due to the fact that only one hyper-parameter has to be tuned for K-means (the number of clusters) and only to store a really small data structure per execution of K-means is required in memory (the cluster centroids) that will usually always fit in the cache memory even when there are many predictions and, as such, clustering processes, being computed in parallel.

Table 2
MAPE (%) for the grid search during the training phase for CUDA-bigPSF and bigPSF (enclosed in parentheses). Best values for each method in bold.

	K = 2	K = 3	K = 4	K = 5	K = 6	K = 7	K = 8	K = 9	K = 10	K = 11	K = 12	K = 13	K = 14	K = 15
W = 1	7.54	6.75	6.16	5.65	5.27	4.99	4.87	4.79	4.70	4.64	4.60	4.55	4.51	**4.49**
	(7.12)	(6.43)	(6.02)	(5.47)	(5.33)	(5.18)	(4.96)	(4.95)	(4.89)	(4.73)	(4.65)	(4.59)	(4.67)	(4.67)
W = 2	7.27	6.50	5.88	5.39	5.10	4.88	4.78	4.72	4.65	4.64	4.61	4.59	4.58	4.56
	(6.70)	(6.30)	(5.83)	(5.39)	(5.22)	(4.99)	(4.83)	(4.85)	(4.89)	(4.73)	(4.65)	**(4.52)**	(4.61)	(4.61)
W = 3	7.12	6.42	5.71	5.26	5.08	4.89	4.84	4.79	4.74	4.73	4.71	4.70	4.70	4.69
	(6.59)	(6.34)	(5.76)	(5.38)	(5.20)	(5.05)	(4.95)	(4.95)	(4.93)	(4.84)	(4.77)	(4.64)	(4.68)	(4.77)
W = 4	7.04	6.46	5.70	5.21	5.14	4.97	4.94	4.90	4.86	4.86	4.84	4.85	4.84	4.85
	(6.55)	(6.34)	(5.77)	(5.40)	(5.31)	(5.19)	(4.97)	(5.04)	(5.08)	(4.94)	(4.89)	(4.80)	(4.88)	(4.88)
W = 5	6.90	6.50	5.70	5.19	5.16	5.02	5.02	4.99	4.97	4.98	4.96	4.98	4.99	4.98
	(6.50)	(6.51)	(5.83)	(5.51)	(5.41)	(5.26)	(5.05)	(5.17)	(5.24)	(5.12)	(5.02)	(4.97)	(4.99)	(4.95)
W = 6	6.79	6.56	5.74	5.20	5.21	5.08	5.08	5.07	5.05	5.06	5.05	5.07	5.07	5.07
	(6.46)	(6.64)	(5.90)	(5.59)	(5.80)	(5.37)	(5.14)	(5.25)	(5.32)	(5.18)	(5.10)	(5.07)	(5.11)	(5.00)
W = 7	6.80	6.66	5.82	5.28	5.27	5.14	5.14	5.13	5.10	5.12	5.10	5.12	5.12	5.11
	(6.52)	(6.74)	(5.99)	(5.66)	(5.57)	(5.38)	(5.17)	(5.29)	(5.37)	(5.25)	(5.17)	(5.15)	(5.13)	(5.02)
W = 8	6.81	6.75	5.87	5.35	5.33	5.18	5.19	5.18	5.15	5.16	5.13	5.14	5.14	5.14
	(6.53)	(6.86)	(6.09)	(5.71)	(5.67)	(5.42)	(5.23)	(5.39)	(5.14)	(5.28)	(5.20)	(5.21)	(5.15)	(5.05)
W = 9	6.84	6.84	5.91	5.43	5.41	5.25	5.26	5.23	5.19	5.19	5.16	5.17	5.16	5.15
	(6.60)	(6.96)	(6.18)	(5.77)	(5.73)	(5.48)	(5.24)	(5.46)	(5.44)	(5.31)	(5.25)	(5.23)	(5.15)	(5.05)
W = 10	6.91	6.89	5.96	5.49	5.48	5.31	5.31	5.29	5.24	5.23	5.20	5.19	5.18	5.16
	(6.70)	(7.04)	(6.23)	(5.84)	(5.73)	(5.50)	(5.26)	(5.48)	(5.48)	(5.34)	(5.26)	(5.24)	(5.19)	(5.05)

4. Discussion

4.1. Experimental Setup

We have used the same dataset used in the bigPSF paper to compare our results. This dataset contains electricity consumption data from Uruguay between 2007 and 2014 recorded hourly. The average demand observed is 1092.21 MW, with a minimum of 609.87 MW and a maximum of 1907.55 MW. Fig. 5 displays the energy consumption distribution by day of the week. We can observe from this figure that energy demand on weekends is lower than on weekdays, as it is expected (Raza & Khosravi, 2015). We did not need additional preprocessing since the dataset did not present any missing observations or extreme outliers. The dataset was split in 70 % training and 30 % test, with the last 30 % of the training partition used as validation for the hyper-parameter optimization, as it is specified in the bigPSF paper.

All experiments were done with a personal computer with an AMD 5 Ryzen 2600X CPU running at 3.6 GHz, an NVIDIA GeForce RTX 3060 Ti 8 GB graphics card, and 32 GB of DDR4 RAM. The code was written using Python 3.11 and CUDA 11.8. CUDA experiments were repeated 30 times with seeds from 1996 to 2025. For the CUDA-BigPSF kernel, we used 32 threads per block, as it provided the fastest results.

4.2. Implementation accuracy.

In this section, we will compare the accuracy of our implementation with the results provided in the original paper. Even though we have implemented the same algorithm with different approaches, we cannot obtain the same results as the original authors due to the randomness in the initialization of k-means and the fact that the original authors did not seed their experiments. As such, we can only evaluate if we have obtained reasonably similar results during training and test.

During the training phase, the Mean Absolute Percentage Error (MAPE) was used, as it is done in bigPSF. This metric (eq. 2) has the advantage of being scale-independent and easy to interpret as it represents the average distance between forecasted and expected value in percentage. For all equations, n represents the total number of samples, y_i the forecasted sample at index i and $\hat{y}_i$ the expected values.

$$MAPE(\%) = \frac{100}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \tag{2}$$

Table 2 displays the difference in MAPE during training between the average of 30 repetitions of CUDA-bigPSF and BigPSF (enclosed in parentheses). As we can observe, both algorithms provide relatively similar results considering the randomness of k-means initialization. The most significant difference in MAPE between the approaches is 0.59 % with $k = 6$ and $w = 6$. The best averaged MAPE found by CUDA-bigPSF was 4.51 % with $k = 14$ and $w = 1$, while the best MAPE for BigPSF was 4.52 % with $k = 13$ and $w = 2$. In 1 of the experiment's repetitions with $k = 15$, CUDA-bigPSF could not provide at least one prediction, even removing the day of the week constraint. Thus, we have excluded that seed (1998) from the average displayed in the table for $k = 15$. In 27 out of the 30 experiment repetitions, a window size of one provided the best results, questioning whether it is advantageous to study the use of a broader window size or whether we should limit the window size from the start to reduce the algorithm's computational complexity. In 18 out of the 30 experiment repetitions, a window size of $k = 15$ provided the best results, followed by 5 repetitions with $k = 13$ and 4 repetitions with $k = 14$.

Table 3
Summary of results obtained by the algorithms in quality metrics for the test partition.

Algorithm	MAPE	MAE	RMSE
bigPSF	4.70	57.15	61.23
CUDA-bigPSF (Average)	4.75	56.84	83.54
CUDA-bigPSF (Worst)	4.88	58.43	87.15
CUDA-bigPSF (Best)	4.62	55.29	80.40

We applied a similar methodology to compare the results in test using the 30 seeds with their optimal hyperparameters. For test, two additional metrics are used: the Mean Absolute Error (MAE) and the Root Mean Squared Error. The MAE (eq. 3) provides the average difference between the forecasted value and the expected value in the original scale of the data while the RMSE (eq. 4) gives a higher penalization to large errors between forecasted values and expected values.

$$MAE = \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{n} \tag{3}$$

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n}} \tag{4}$$

Table 3 summarizes the results of our 30 repetitions for CUDA-bigPSF and the results reported for bigPSF. Our implementations obtain similar results on average for MAPE and MAE, and the best experiment done with CUDA even improves the results reported in bigPSF substantially. However, there is an unexpected difference in the

Table 4
Execution time per version of the algorithm in hh:mm:ss.

Dataset	CPU-Seq		CUDA-bigPSF			bigPSF(Spark)	
	Training	Test	Training	Test	N° Cores	Training	Test
N (7 years)	00:16:44.17	00:02:37.05	00:00:01.87	00:00:00.47	2	00:18:54	00:01:30
2 N (14 years)	01:03:56.76	00:09:18.47	00:00:09.71	00:00:00.89	4	00:22:03	00:01:45
4 N (28 years)	04:00:29.14	00:37:04.03	00:00:38.49	00:00:09.48	4	00:29:24	00:02:20
8 N (56 years)	14:58:00.58	02:25:40.14	00:02:36.62	00:00:42.00	4	00:42:50	00:03:24
16 N (112 years)	56:08:18.72	09:23:37.46	00:10:33.48	00:02:46.42	4	1:07:25	00:05:21

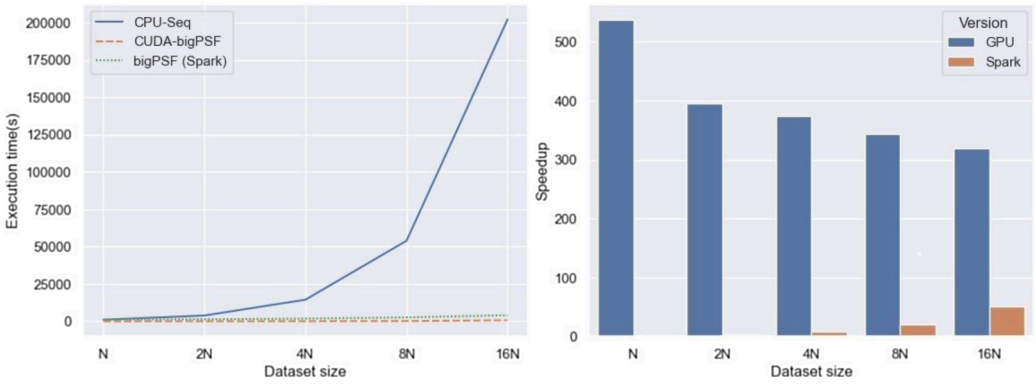

Fig. 6. On the left, line plot of the time spent in training by each method. On the right, speedup obtained by the Spark and CUDA versions over a sequential implementation.

RMSE metric that we cannot explain. A comparison of the results provided by the bigPSF / CUDA-bigPSF algorithm with other forecasting algorithms such as neural networks, ARIMA and gradient boosting trees can be found in (Pérez-Chacón et al., 2020).

4.3. *Implementation speedup and scalability.*

At last, we compare the executing times of the Spark version, the CUDA version, and a sequential CPU version we will use as a baseline. Table 4 reports the performance of each architecture with the original dataset and synthetic datasets made by repeating the original dataset, as is done in the BigPSF paper.

First, it is important to note that even though the number of cores used for bigPSF seems small, authors reported that using a higher number of cores does not improve the results but rather makes them go even slower. This situation happens because many algorithm steps of bigPSF using their data distribution approach require synchronization and node cooperation, unlike our GPU approach. As such, even though it takes almost 19 min to train the algorithm with Spark, our GPU version can train it (find the optimal number of clusters and window size) in under two seconds using the full potential of all its cores. Interestingly, our sequential implementation was slightly faster than the Spark version, training 2 min faster, although it is easily explained as our CPU has a much higher clock speed and the Spark version only uses two cores. The evolution of training time for all approaches and the speedup obtained by bigPSF and CUDAbigPSF are displayed in Fig. 6, where the speedup is calculated by dividing the sequential version time by the accelerated version time. However, the Spark approach struggles to obtain a significant speedup until using 28 years of data. Meanwhile, our GPU approach can produce results over 500 times faster than both methods for seven years of data and still manages to make results at least 300 times faster when using the highest amount of data evaluated in this paper (112 years).

From the previously discussed results, it is clear that using a CUDA device will produce faster results than the Spark approach in most situations. In fact, the Spark approach only uses a significant number of cores once training with an unreasonably large dataset. It is also important to note that due to the weighting system used in bigPSF, older samples influence the prediction at a much lower rate. As such, at some point, adding more data, at best, will be no more than a rounding error in the final forecast. The only situation in which the CUDA version proposed in this paper should perform significantly worse than reported is with GPU devices that cannot store all the data structures in the device memory. During the implementation and explanation of our algorithm, we have considered this and used local memory whenever possible so that once a thread finishes its work, another thread can use that memory. As a last resource, the user can reduce the number of clusters evaluated simultaneously to reduce the amount of local memory used per thread. Nevertheless, this algorithm should provide good results in most cases, even using low-end NVIDIA graphics cards.

5. Conclusion

The main objective of the work presented in this paper was to create a high-performance GPU implementation of an algorithm for load forecasting made for distributed algorithms, bigPSF. The proposed algorithm was evaluated with the same dataset of energy consumption from Uruguay used in bigPSF, allowing a direct comparison between both methods. The design of the GPU version took into account some of the limitations of the bigPSF algorithm through two main contributions.

First, CUDA-bigPSF uses a completely different approach to distribute the work between the cores, removing almost all the need for synchronization and communication between nodes. Second, CUDA-bigPSF takes into account several factors to avoid any unnecessary computations and removes one of the costly data structures used in bigPSF, the pattern matrix.

Results show that CUDA-bigPSF provides a correct implementation of bigPSF capable of achieving speedups during the training phase up to 500 times faster than the original bigPSF. As such, the work presented in this paper makes bigPSF more accessible to researchers and practitioners, as the availability of GPU devices is more widespread and cheaper than access to a distributed cluster. Furthermore, many of the solutions proposed in this paper for the GPU can also be used to improve the distributed version of the algorithm.

There are several directions for future work on the algorithm presented in this paper. One possibility is to evaluate and optimize the use of different clustering methods or ensembles of them, evaluating the training time and accuracy of them in different datasets. Additionally, it may be useful to develop versions of the algorithm for multivariate time series. Another possible direction for future work is to combine the use of this algorithm in an ensemble with other forecasting algorithms to potentially improve forecast accuracy.

CRediT authorship contribution statement

David Criado Ramón: Conceptualization, Methodology, Software, Validation, Writing – original draft, Writing – review & editing. **Luis Gonzaga Baca Ruiz:** Conceptualization, Methodology, Writing – original draft, Writing – review & editing, Supervision. **María del Carmen Pegalajar Jiménez:** Conceptualization, Methodology, Writing – review & editing, Supervision, Project administration, Funding acquisition.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

Funding for open access charge: Universidad de Granada / CBUA. This work has been developed with the support of the Department of Computer Science and Artificial Intelligence of the University of Granada, TIC111. We acknowledge financial support from Grant PID2020-112495RB-C21 funded by MCIN/ AEI /10.13039/501100011033 and the I + D + i FEDER 2020 project B-TIC-42-UGR20. We thank Drs. Pérez-Chacón and Martínez-Álvarez (Data Science and Big Data Lab, Pablo de Olavide University) for all the help provided to reproduce their algorithm.

References

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., … Zheng, X. (2016). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv preprint. arXiv:1603.04467.

Arthur, D., & Vassilvitskii, S. (2007). k-means++: The advantages of careful seeding. *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1027–1035.

Bokde, N., Beck, M. W., Martínez Álvarez, F., & Kulat, K. (2018). A novel imputation methodology for time series based on pattern sequence forecasting. *Pattern Recognition Letters, 116*, 88–96. https://doi.org/10.1016/j.patrec.2018.09.020

Bokde, N., Troncoso, A., Asencio-Cortés, G., Kulat, K., & Martínez-Álvarez, F. (2017). Pattern sequence similarity based techniques for wind speed forecasting. *Proceedings of the International Work-Conference on Time Series, Granada, Spain, 2*, 786–794. https://itise.ugr.es/pdf/ITISE2017_vol2.pdf.

Bose, B. K. (2017). Power Electronics, Smart Grid, and Renewable Energy Systems. *Proceedings of the IEEE, 105*(11), 2011–2018. https://doi.org/10.1109/JPROC.2017.2745621

Bouveyron, C., & Jacques, J. (2011). Model-based clustering of time series in group-specific functional subspaces. *Advances in Data Analysis and Classification, 5*(4), 281–300. https://doi.org/10.1007/s11634-011-0095-6

Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (pp. 785–794). https://doi.org/10.1145/2939672.2939785

Chen, Z., Li, J., Cheng, L., & Liu, X. (2023). Federated-WDCGAN: A federated smart meter data sharing framework for privacy preservation. *Applied Energy, 334*, Article 120711. https://doi.org/10.1016/j.apenergy.2023.120711

Coelho, I. M., Coelho, V. N., Luz, E. J. da S., Ochi, L. S., Guimarães, F. G., & Rios, E. (2017). A GPU deep learning metaheuristic based model for time series forecasting. *Applied Energy*, *201*, 412–418. doi: 10.1016/j.apenergy.2017.01.003.

Fujimoto, Y., & Hayashi, Y. (2012). Pattern sequence-based energy demand forecast using photovoltaic energy records. *International Conference on Renewable Energy Research and Applications (ICRERA), 2012*, 1–6. https://doi.org/10.1109/ICRERA.2012.6477299

Haque, A., & Rahman, S. (2022). Short-term electrical load forecasting through heuristic configuration of regularized deep neural network. *Applied Soft Computing, 122*, Article 108877. https://doi.org/10.1016/j.asoc.2022.108877

Iruela, J. R. S., Ruiz, L. G. B., Capel, M. I., & Pegalajar, M. C. (2021). A TensorFlow Approach to Data Analysis for Time Series Forecasting in the Energy-Efficiency Realm. *Energies, 14*(13), Article 13. https://doi.org/10.3390/en14134038

Iruela, J. R. S., Ruiz, L. G. B., Pegalajar, M. C., & Capel, M. I. (2020). A parallel solution with GPU technology to predict energy consumption in spatially distributed buildings using evolutionary optimization and artificial neural networks. *Energy Conversion and Management, 207*, Article 112535. https://doi.org/10.1016/j.enconman.2020.112535

Jin, C. H., Pok, G., Lee, Y., Park, H.-W., Kim, K. D., Yun, U., & Ryu, K. H. (2015). A SOM clustering pattern sequence-based next symbol prediction method for day-ahead direct electricity load and price forecasting. *Energy Conversion and Management, 90*, 84–92. https://doi.org/10.1016/j.enconman.2014.11.010

Jin, C. H., Pok, G., Park, H.-W., & Ryu, K. H. (2014). Improved pattern sequence-based forecasting method for electricity load. *IEEJ Transactions on Electrical and Electronic Engineering, 9*(6), 670–674. https://doi.org/10.1002/tee.22024

Johnson, J., Douze, M., & Jégou, H. (2021). Billion-Scale Similarity Search with GPUs. *IEEE Transactions on Big Data, 7*(3), 535–547. https://doi.org/10.1109/TBDATA.2019.2921572

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., & Liu, T.-Y. (2017). LightGBM: A highly efficient gradient boosting decision tree. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (pp. 3149–3157).

Kim, T.-Y., & Cho, S.-B. (2019). Predicting residential energy consumption using CNN-LSTM neural networks. *Energy, 182*, 72–81. https://doi.org/10.1016/j.energy.2019.05.230

Kintsakis, A. M., Chrysopoulos, A., & Mitkas, P. A. (2015). Agent-based short-term load and price forecasting using a parallel implementation of an adaptive PSO-trained local linear wavelet neural network. In *2015 12th International Conference on the European Energy Market (EEM)* (pp. 1–5). https://doi.org/10.1109/EEM.2015.7216611

Kong, W., Dong, Z. Y., Jia, Y., Hill, D. J., Xu, Y., & Zhang, Y. (2019). Short-Term Residential Load Forecasting Based on LSTM Recurrent Neural Network. *IEEE Transactions on Smart Grid, 10*(1), 841–851. https://doi.org/10.1109/TSG.2017.2753802

Martinez Alvarez, F., Troncoso, A., Riquelme, J. C., & Aguilar Ruiz, J. S. (2011). Energy Time Series Forecasting Based on Pattern Sequence Similarity. *IEEE Transactions on Knowledge and Data Engineering, 23*(8), 1230–1243. https://doi.org/10.1109/TKDE.2010.227

Martínez-Álvarez, F., Schmutz, A., Asencio-Cortés, G., & Jacques, J. (2019). A Novel Hybrid Algorithm to Forecast Functional Time Series Based on Pattern Sequence Similarity with Application to Electricity Demand. *Energies, 12*(1), Article 1. https://doi.org/10.3390/en12010094

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., … Chintala, S. (2019). In *PyTorch: An imperative style, high-performance deep learning library* (pp. 8026–8037). Curran Associates Inc.

Pérez-Chacón, R., Asencio-Cortés, G., Martínez-Álvarez, F., & Troncoso, A. (2020). Big data time series forecasting based on pattern sequence similarity and its application to the electricity demand. *Information Sciences, 540*, 160–174. https://doi.org/10.1016/j.ins.2020.06.014

Raschka, S., Patterson, J., & Nolet, C. (2020). Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence. *Information, 11*(4), Article 4. https://doi.org/10.3390/info11040193

Raza, M. Q., & Khosravi, A. (2015). A review on artificial intelligence based load demand forecasting techniques for smart grid and buildings. *Renewable and Sustainable Energy Reviews, 50*, 1352–1372. https://doi.org/10.1016/j.rser.2015.04.065

Said, Y., & Alanazi, A. (2022). AI-based solar energy forecasting for smart grid integration. *Neural Computing and Applications, 35*(11), 8625–8634. https://doi.org/10.1007/s00521-022-08160-x

Shen, W., Babushkin, V., Aung, Z., & Woon, W. L. (2013). An ensemble model for day-ahead electricity demand time series forecasting. In *Proceedings of the Fourth International Conference on Future Energy Systems* (pp. 51–62). https://doi.org/10.1145/2487166.2487173

Tian, Y., Sehovac, L., & Grolinger, K. (2019). Similarity-Based Chained Transfer Learning for Energy Forecasting With Big Data. *IEEE Access, 7*, 139895–139908. https://doi.org/10.1109/ACCESS.2019.2943752

Wen, Z., Shi, J., Li, Q., He, B., & Chen, J. (2018). ThunderSVM: A fast SVM library on GPUs and CPUs. *The Journal of Machine Learning Research, 19*(1), 797–801.

Zheng, J., Gao, D. W., & Lin, L. (2013). Smart Meters in Smart Grid: An Overview. *IEEE Green Technologies Conference (GreenTech), 2013*, 57–64. https://doi.org/10.1109/GreenTech.2013.17