IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 14, 2017, accepted October 11, 2017, date of publication October 23, 2017,
date of current version December 22, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2765380

CUDA-Based Parallelization of Power Iteration Clustering for Large Datasets

GUSTAVO RODRIGUES LACERDA SILVA[1], RAFAEL RIBEIRO DE MEDEIROS[2], BRAYAN RENE ACEVEDO JAIMES[1], CARLA CALDEIRA TAKAHASHI[1], DOUGLAS ALEXANDRE GOMES VIEIRA[2], AND ANTÔNIO DE PÁDUA BRAGA[1]

[1]Graduate Program in Electrical Engineering - Universidade Federal de Minas Gerais - Av. Antônio Carlos 6627, 31270-901, Belo Horizonte, MG, Brazil
[2]ENACOM Handcrafted Technologies, Belo Horizonte 31310-260, Brazil

Corresponding author: Gustavo Rodrigues Lacerda Silva (gustavolacerdas@ufmg.br)

This work was supported in part by the Brazilian agency CAPES, in part by CNPq, and in part by FAPEMIG.

ABSTRACT This paper presents a new clustering algorithm, the GPIC, a graphics processing unit (GPU) accelerated algorithm for power iteration clustering (PIC). Our algorithm is based on the original PIC proposal, adapted to take advantage of the GPU architecture, maintaining the algorithm's original properties. The proposed method was compared against the serial implementation, achieving a considerable speedup in tests with synthetic and real data sets. A significant volume of real data application ($>10^7$ records) was used, and we identified that GPIC implementation has good scalability to handle data sets with millions of data points. Our implementation efforts are directed towards two aspects: to process large data sets in less time and to maintain the same quality of the clusters results generated by the original PIC version.

INDEX TERMS Scalable machine learning algorithms, GPU, power iteration clustering.

I. INTRODUCTION

The implementation of clustering methods for Big Data requires scalable computing platforms that are capable of storing and processing massive and high dimensional data [1], [2]. This could drastically limit the usage of classical clustering algorithms like Spectral Clustering [3], which is data intensive and requires the calculation of the $n \times n$ distance matrix of the entire dataset and its eigenvalue decomposition.

Although they have proved to be robust, spectral clustering methods have a high order of complexity, especially for calculating the eigenvectors and eigenvalues, which can be as high as $O(n^3)$ [4]. Affinity matrix computation, which represents pairwise relations for spectral clustering, can be a strong limitation when massive data sets are used. Some authors [5], [6] used parallel implementations to solve the general problem of parallelizing similarity affinity computations. Since pairwise relations are independent, affinity matrix is particularly appropriate for parallel implementation, which paves the way for adopting end-user parallel machines, such as GPUs (Graphics Processing Units), for massive data implementations [1].

In addition, in order to overcome the $O(n^3)$ computation cost of eigenvalue decomposition, parallel implementations have also been presented in the literature [7]. Sparse implementations of the affinity matrix have shown to be a promising strategy to improve scalability [6]. The Power Iteration Clustering (PIC) [8] adopts a different approach, which is based on an approximation of the largest eigenvector, which skips the usual high dimensional matrix operations. The method is based on a normalized pairwise affinity data matrix, which turns out to be a valid clustering indicator, consistently outperforming widely used spectral methods.

This paper presents a new GPU-based version of the PIC algorithm, called here GPIC (GPU Power Iteration Clustering), which aims at parallelization of the affinity matrix calculation and at the approximation of the largest eigenvector. In order to overcome the computational bottleneck we propose the GPIC approach, which aims at creating a set of CUDA kernels that can be called one after another to perform different steps of the PIC algorithm in parallel. GPUs offer a massively parallel hardware structure, which is appropriate to overcome some of the difficulties to scale up clustering methods. In this paper, experiments were run with 39.936 GPU cores for a real dataset problem.

Affinity matrix computation may require regular memory access and abundant parallel computation according to its implementation, what is inherently appropriate for GPU implementation. GPIC's implementation proposes a novel approach to compute the large scale affinity matrix, this implementation splits the data into chunks, which are iteratively copied to the GPU board device. Using CUDA framework, a large number of dedicated processors is launched to

2169-3536 © 2017 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

accomplish the task, which represents a gain in parallelization when compared to multi-CPU implementation.

GPIC's eigenvector approximation considers the one thread per row approach, where each thread performs a dot product between one row of a matrix and an element of a vector to produce one element of the resulting vector. The main design motivation for this kernel was aimed at data reuse from GPU shared memory, particularly for vector and matrix product computations. Operations were accomplished by block-reading from GPU shared memory, followed by threads computations on the data. Again, in contrast to other parallel implementations, due to the large number of processors in GPUs, matrix-vector multiplication can be computed in a single parallel step. As presented in the results section, GPIC provides a considerable speedup when compared with the sequential version of the GPU code on a single thread.

The rest of this paper is organized as follows. In section section II, we discuss some relevant studies in this area. In section III, we describe the implementation details used to accelerate the Power Iteration Clustering method. In sections IV and V we present the results found for synthetic datasets and the real large application dataset respectively. Finally, in section VI, we present our concluding remarks and recommendations for further study.

II. CLUSTERING WITH GPUs

Recent advances in consumer computer hardware make parallel computing capability widely available to most users. Applications that make effective use of the so-called Graphics Processing Units (GPU) have reported significant performance gains [9]. In the CUDA (Compute Unified Device Architecture) model, GPU is regarded as a co-processor, which is capable of executing a large number of threads in parallel. A single-source program includes host codes running on CPU (Central Processing Unit) and also kernel codes running on GPU. Computationally-intensive and data-parallel tasks are implemented as kernel codes in such a way that they can be executed in GPUs [9].

An approach that uses k-means on GPUs is presented in [10], where the authors applied k-means to a synthetic dataset with one billion data points and two dimensions. With this test setup, the GPU-version took about 26 minutes, and the serial version of the code took six days. The GPU board used was a GeForce GTX 280 with 240 CUDA cores and 1GB of global memory.

Dong *et al.* [11] presented a GPU accelerated BIRCH (Balanced Iterative Reducing and Clustering using Hierarchies) [12], version that can be up to 154 times faster than the CPU version with good scalability and high accuracy. The hardware configuration was: Intel Core i5-2320 (CPU), 8GB RAM and NVIDIA Tesla K20 (Kepler Architecture GPU) with 2496 CUDA cores and 5GB of global memory.

Andrade *et al.* [13] presented the algorithm G-DBSCAN, a GPU parallel version of the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) [14]. Its implementation with GPUs can be over 100 times faster than its sequential version. The experiments were performed on a Intel Xeon 2.40GHz equipped with a Tesla M2050 with 448 CUDA cores and 3GB of global memory. Our approach to GPU implementation is based on the PIC algorithm [8], which will be described in the next section.

III. METHODOLOGY

The PIC algorithm is a spectral clustering method, which is aimed at computing the largest eigenvector of a matrix by the Power Iteration Method [15]. Let's consider $\mathbf{W}$ to be a diagonalizable $n \times n$ matrix with dominant eigenvalue λ_1. So, there is an eigenvector $\mathbf{x}_1$ associated to λ_1. Approximation $\hat{\mathbf{x}}_i$ of eigenvector $\mathbf{x}_i$ will be updated at each iteration during PIC execution, hence i is the iteration number and $\mathbf{x}_0$ is the initial vector in the following iteration equation 1.

$$\hat{\mathbf{x}}_i = \mathbf{W}^i \mathbf{x}_0, \quad i = 1, 2, \ldots \tag{1}$$

Let $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ be the eigenvectors and $\lambda_1, \lambda_2, \ldots, \lambda_n$ be the corresponding eigenvalues of $\mathbf{W}$. Since $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n$ are linearly independent, they form a basis of $\mathbb{R}^n$. Consequently, we can write $\mathbf{x}_0$ as a linear combination of these eigenvectors:

$$\begin{aligned}\hat{\mathbf{x}}_i &= \mathbf{W}^i \mathbf{x}_0 \\ &= c_1 \lambda_1^i \mathbf{x}_1 + c_2 \lambda_2^i \mathbf{x}_2 + \ldots + c_n \lambda_n^i \mathbf{x}_n \\ &= \lambda_1^i \left[c_1 \mathbf{x}_1 + c_2 \left(\frac{\lambda_2}{\lambda_1}\right)^i \mathbf{x}_2 + \ldots + c_n \left(\frac{\lambda_n}{\lambda_1}\right)^i \mathbf{x}_n \right]\end{aligned} \tag{2}$$

where $c_1, c_2, \ldots c_n$ are constant coefficients. Since λ_1 is the dominant eigenvalue, it means that each of the ratios $\frac{\lambda_i}{\lambda_1}$ is less than 1 in absolute value,

$$\lim_{i \to \infty} \left(\frac{\lambda_j}{\lambda_1}\right)^i = 0, \quad j = 2, \ldots, n \tag{3}$$

so that:

$$\lim_{i \to \infty} \hat{\mathbf{x}}_i = \lambda_1^i c_1 \mathbf{x}_1 \tag{4}$$

So, if $\lambda_1 \neq 0$ and $\mathbf{x}_1 \neq 0$, $\hat{\mathbf{x}}_i$ approximates a multiple of $\mathbf{x}_1$ as i increases, that is an eigenvector corresponding to λ_1, if $c_1 \neq 0$. In the PIC [8] algorithm the Power Iteration Method [15] performs the update step:

$$\hat{\mathbf{x}}_{i+1} = \frac{\mathbf{W}\hat{\mathbf{x}}_i}{\|\mathbf{W}\hat{\mathbf{x}}_i\|_1} \tag{5}$$

where $\|\mathbf{W}\hat{\mathbf{x}}_i\|_1$ is the L_1 norm of $\mathbf{W}\hat{\mathbf{x}}_i$. A serial version of PIC [8] is presented in Algorithm 1.

A. OUR APPROACH GPIC - GPU POWER ITERATION CLUSTERING

An experiment was conducted to evaluate the PIC behavior for a moderate amount of data using MATLAB, according to [8]. We used two synthetic datasets, two moons and three circles with $N = 15,000$, 30,000 and 45,000 data points. Some notations are defined: N is the number of samples of the dataset, $\mathbf{A}$ is the affinity matrix and m is the input space dimension.

Algorithm 1 Power Iteration Clustering [8]

Input:
- $\mathbf{W}$ Row-normalized affinity matrix
- k Number of clusters
- $\mathbf{x}_0$ Initial vector
- ϵ Precision

Output:
- $\mathbf{C}$ Clusters $C_1, C_2, \ldots, C_k$

1: $\boldsymbol{\delta}_0 \leftarrow \mathbf{v}_0$ ▷ Initialize the variation of eigenvector
2: **for** $i = 0, 1, 2, \ldots$ **do**
3: $\hat{\mathbf{x}}_{i+1} \leftarrow \frac{\mathbf{W}\hat{\mathbf{x}}_i}{\|\mathbf{W}\hat{\mathbf{x}}_i\|_1}$ ▷ Update the eigenvectors of $\mathbf{W}$
4: $\boldsymbol{\delta}_{i+1} \leftarrow |\hat{\mathbf{x}}_{i+1} - \hat{\mathbf{x}}_i|$ ▷ Update the variation of the previous and current eigenvector
5: **if** $|\boldsymbol{\delta}_{i+1} - \boldsymbol{\delta}_i| \leq \epsilon$ **then** ▷ Stop criteria
6: **break** ▷ Terminate the estimation of eigenvectors
7: **end if**
8: **end for**
9: $\mathbf{C} \leftarrow k$-means of $\hat{\mathbf{x}}_i$ ▷ Cluster eigenvalues
10: **return C**

The results of the experiment indicate that the main bottleneck of the PIC algorithm [8] is to compute the pairwise distance/similarity for all data points, which is $O(n^2)$ in the worst case. On average, it consumes 88.61% of PIC's total time. The results of the experiment are presented in Table 1.

TABLE 1. Runtime (seconds) of PIC algorithm ($m = 2$).

Dataset	N	**A** size[GB]	**A** time [s]	PIC total time [s]	**A** time from PIC total time [%]
2 moons	15k	3.85	2.534	2.548	99.45%
3 circles			2194.27	2363.64	92.83%
2 moons	30k	16	18666.39	21680.05	86.09%
3 circles			18064.53	21091.17	85.64%
2 moons	45k	36.5	97966.50	103778.10	94.39%
3 circles			62228.62	84977.15	73.22%

In order to overcome this bottleneck we propose the GPIC (GPU Power Iteration Clustering) approach, which aims at creating a set of CUDA kernels that can be called one after another to perform different steps of the PIC algorithm. According to the growth of the data volume, a larger amount of memory must be available to store the entire affinity matrix, however the GPU memory is limited. Aiming to solve these challenge, our approach splits the data into chunks, which are iteratively copied to the device as presented in Figure 1.

The iteratively affinity matrix generation steps are described bellow:

1) Read the input dataset and store in CPU memory.
2) Allocate memory space in the GPU for the input dataset. The chunk size is dynamically set according to the memory size of the GPU board. The input data that is maintained in the CPU memory is copied to the GPU's global memory.

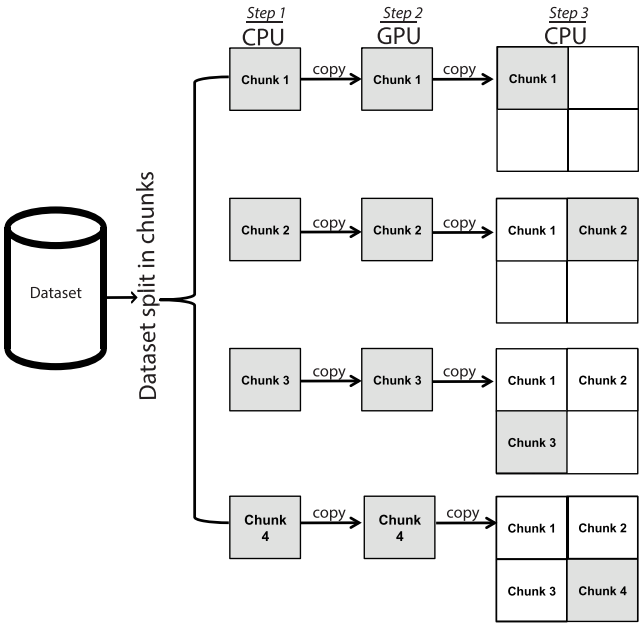

FIGURE 1. GPIC split data into chunks and iteratively copied to the device.

3) Launch a kernel on the GPU board to calculate a chunk of the affinity matrix. It is considered only a portion of the affinity matrix. Then the result is copied back to the CPU memory.

GPIC is described in Algorithm 2 while Figure 2 presents all steps of the execution flow. The two main focuses of GPIC are the Affinity matrix calculation and the Power Iteration step.

The first kernel, AffinityMatrix($\mathbf{S}, p$), is launched to evaluate the Affinity matrix $\mathbf{A}$, where $\mathbf{S}$ is the input matrix and p is the number of threads. Matrix $\mathbf{A}$ is symmetrical with dimensions $n \times n$, where n is the number of samples. When launched, this kernel has p threads and each thread is responsible for calculating a set of $\left\lceil \frac{n}{p} \right\rceil$ rows of matrix $\mathbf{A}$. Thus, each element ij of matrix $\mathbf{A}$ is indexed according to the identifier j of each thread and the current index i. This step has order of complexity $O(n^2/p)$, and each CUDA core calculates parts of matrix $\mathbf{A}$.

Figure 3 presents an example of an affinity matrix $\mathbf{A}$ and this representation on GPU. In this scenario, each thread calculates a row of a matrix $\mathbf{A}$, and each thread performs a control flow structure for each of the elements in each row.

The second kernel RowSum($\mathbf{A}, p$) is launched to sum the rows of matrix $\mathbf{A}$ and the result of this operation is stored in vector $\mathbf{d}$. This step has order of complexity $O(n^2/p)$.

The third kernel, NormMatrix($\mathbf{A}, \mathbf{d}, p$), is launched to normalize the affinity matrix $\mathbf{A}$, with the result stored in matrix $\mathbf{W}$. This step has order of complexity $O(n^3/p)$.

The fourth kernel, Reduction($\mathbf{d}, p$), with order of complexity $O(n/p + log(p))$, is launched to calculate the sum of vector $\mathbf{d}$. In step 9 of the GPIC Algorithm 2, this operation is called again and the result is stored in τ. Sequential reduction

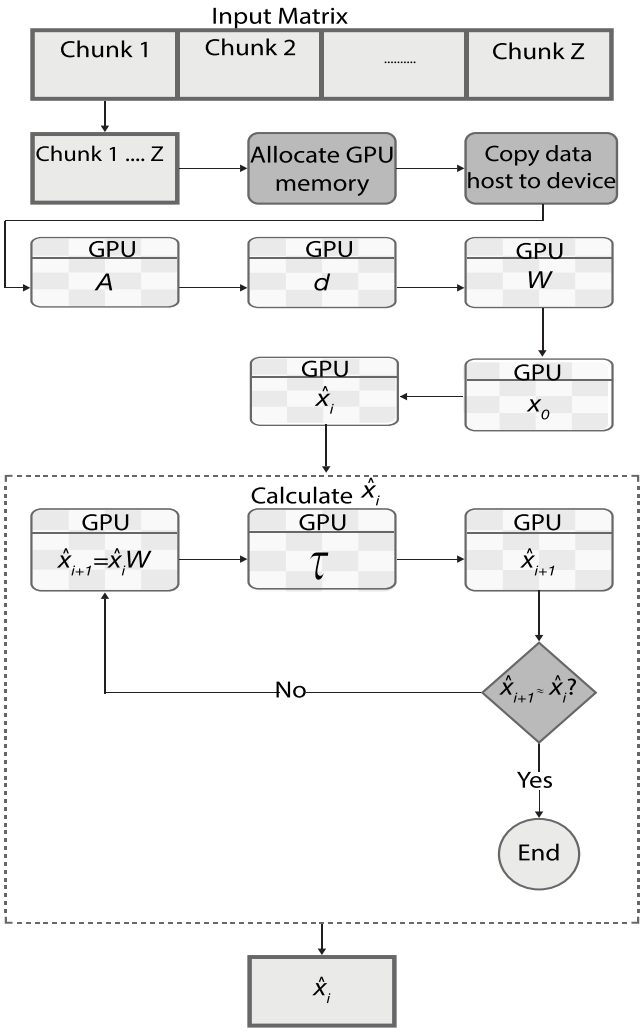

FIGURE 2. GPIC execution flow.

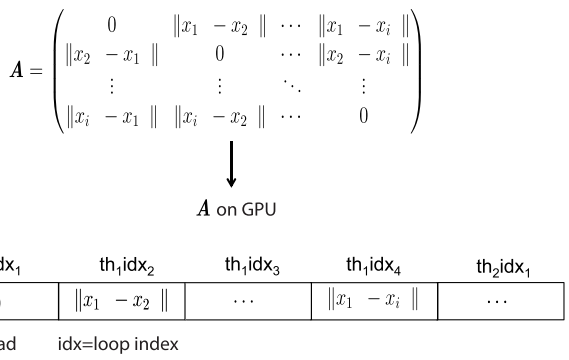

FIGURE 3. Affinity matrix representation on GPU.

occurs in $O(n)$ and in a thread block where $p < n$ we have $O(n/p + log(p))$ complexity. This pattern is suggested by Nvidia [16] since it is conflict free.

The fifth kernel Norm($\hat{\mathbf{x}}_i$, τ, p) is launched to normalize vector $\mathbf{d}$, with the result stored in $\hat{\mathbf{x}}_i$. The step 10 of the GPIC Algorithm 2 invokes the Norm($\hat{\mathbf{x}}_i$, τ, p) again in order to update values of $\hat{\mathbf{x}}_i$. The order of complexity of this kernel is $O(n/p)$. The last kernel, Multiply($\hat{\mathbf{x}}_i$, $\mathbf{W}$), with order of complexity $O(n^2/p)$ is launched to multiply matrix $\mathbf{W}$ by vector $\hat{\mathbf{x}}_i$.

This version of PIC implemented in GPU converges to exactly the same result as the original sequential method, since the multi-thread exploits the fact that the operations are independent.

Algorithm 2 GPU Power Iteration Clustering

Input:
- $\mathbf{S}$ Input matrix
- $\mathbf{A}$ Affinity matrix
- k Numbers of clusters
- $\mathbf{x}_0$ Initial vector
- p Number of threads
- ϵ Precision

Output:
- $\mathbf{C}$ Clusters $C_1, C_2, \ldots, C_k$

1: $\mathbf{A} \leftarrow$ AffinityMatrix($\mathbf{S}$, p) ▷ Affinity matrix kernel
2: $\mathbf{d} \leftarrow$ RowSum($\mathbf{A}$, p) ▷ Sum lines kernel
3: $\mathbf{W} \leftarrow$ NormMat($\mathbf{A}$, $\mathbf{d}$, p) ▷ Normalize kernel
4: $\mathbf{x}_0 \leftarrow$ Reduction($\mathbf{d}$, p) ▷ Sum kernel
5: $\hat{\mathbf{x}}_i \leftarrow$ Norm($\mathbf{d}$, $\mathbf{x}_0$, p) ▷ Normalize kernel
6: $\boldsymbol{\delta}_0 \leftarrow \mathbf{x}_0$ ▷ Initialize the variation of eigenvector
7: **for** $i = 0, 1, 2, \ldots$ **do**
8: $\hat{\mathbf{x}}_{i+1} \leftarrow$ Multiply($\hat{\mathbf{x}}_i$, $\mathbf{W}$, p) ▷ Multiply kernel
9: $\tau \leftarrow$ Reduction($\hat{\mathbf{x}}_{i+1}$, p) ▷ Sum kernel
10: $\hat{\mathbf{x}}_{i+1} \leftarrow$ Norm($\hat{\mathbf{x}}_{i+1}$, τ, p) ▷ Update the eigenvectors kernel
11: $\boldsymbol{\delta}_{i+1} \leftarrow |\hat{\mathbf{x}}_{i+1} - \hat{\mathbf{x}}_i|$ ▷ Update the variation of the previous and current eigenvector
12: **if** $|\boldsymbol{\delta}_{i+1} - \boldsymbol{\delta}_i| \leq \epsilon$ **then** ▷ Stop criteria
13: **break** ▷ Terminate the estimation of eigenvectors
14: **end if**
15: **end for**
16: $\mathbf{C} \leftarrow k$-means of $\hat{\mathbf{x}}_i$ ▷ Cluster eigenvalues
17: **return C**

IV. EXPERIMENTS

The experiments performed in this work were divided in two steps. First experiment shows the runtime and speedup of the two versions of the PIC Algorithm (sequential version of the GPU code, running it on just one thread and GPIC). The second one evaluated a profiling experiment in GPIC Algorithm which resulted in a new version of them.

A. EXPERIMENT I - SPEEDUP COMPARISON WITH SERIAL AND PARALLELS VERSIONS

In this section the results of the proposed GPIC algorithm are compared with the sequential version of the GPU code, running it on just one thread version of PIC. Experiments were conducted on a server containing two Intel Xeon E5-2620 (2 GHz, totalling 24 cores) with 64 GB RAM and

one k40m model NVIDIA card with 12 GB GDDR5 SDRAM and 2880 CUDA cores running at 745 MHz. Thereby, performance was evaluated using two synthetic datasets (two moons and three circles) with the following matrix dimensions: 15.000 × 15.000 (3.85GB), 30.000×30.000 (16GB) and 45.000×45.000 (36GB). The results presented are the average of ten trials in all experiments.

Table 2 shows running times and speedup of PIC and GPIC. Results show a significant speedup of the GPIC Algorithm which, on average, was 188.17 times faster than the original PIC (sequential version of the GPU code running it on just one thread).

TABLE 2. Runtime (in seconds) and speedup comparison of PIC (sequential version of the GPU code running it on just one thread) and GPIC method on two synthetic datasets. The parameters for all experiments are maxiterations=3, $\epsilon = 0.00001/N$, $m = 2$ (dimension), and euclidean distance similarity function.

Dataset	N	A size [GB]	PIC Sequential time [s]	GPIC time [s]	GPIC x PIC Sequential Speedup
2 moons	15k	3.85	838	3.96	211.61
3 circles			837	4.01	208.72
2 moons	30k	16	3336	17.57	189.86
3 circles			3336	17.75	187.94
2 moons	45k	36.5	7499	44.97	166.75
3 circles			7500	45.69	164.14

B. EXPERIMENT II - PROFILING GPIC ALGORITHM

In order to analyze GPIC algorithm performance, a *profile* process was performed with the NVIDIA Visual Profiler tool [17]. This tool provides a graphical interface where you can get metrics about the code that was run on the GPU. These metrics present the runtime of each kernel, the amount of global and shared memory used, and processors usage. This tool also includes an automated analysis engine to identify code optimization opportunities.

The NVIDIA Visual Profiler tool needs a graphical user interface to present results of profile process in a Graphical User Interface (GUI). However, the server where the k40m GPU card is installed does not support a GUI, thus it was necessary to change the simulations to another machine, and consequently another GPU card. This new machine has an Intel Xeon 2.4 GHz, 96 GB of RAM and a graphics card NVIDIA Quadro 4000 with 2 GB of GDDR5 memory and 256 CUDA cores.

Figure 4 presents all kernels launched in the GPU and the duration of each of them. The computational cost of each kernel was evaluated, and, although the result of the profile shows the Multiply($\hat{\mathbf{x}}_i$, $\mathbf{W}$, p) kernel with 46.6% of the total cost, in reality this is not the most expensive one. This kernel actually runs three times because of the iterative nature of the GPIC algorithm. The cost of running this kernel when evaluated individually is only 15.53% of the total time. The results indicate that the most expensive kernel was the NormMat($\mathbf{A}$, $\mathbf{d}$, p) since it consumes 31% of the total time. This occurs because this kernel runs the affinity matrix by

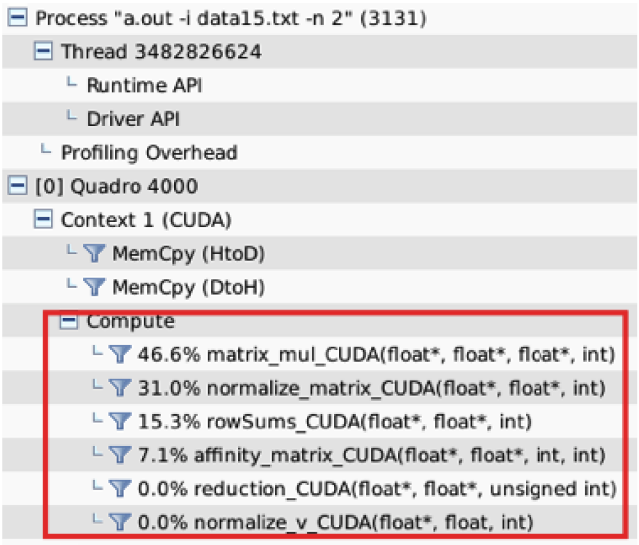

FIGURE 4. Kernels launched on GPU and runtime.

performing reading, updating, and writing operations on each of the array elements.

Figure 5 shows that there was not an efficient use of global and shared memory on GPIC code. As a result, a small change in GPIC code was made. The NormMat($\mathbf{A}$, $\mathbf{d}$, p) kernel was rewritten to use shared memory instead of global memory. After this modification in the code, another experiment was executed and the corresponding mean execution times are presented in Table 3.

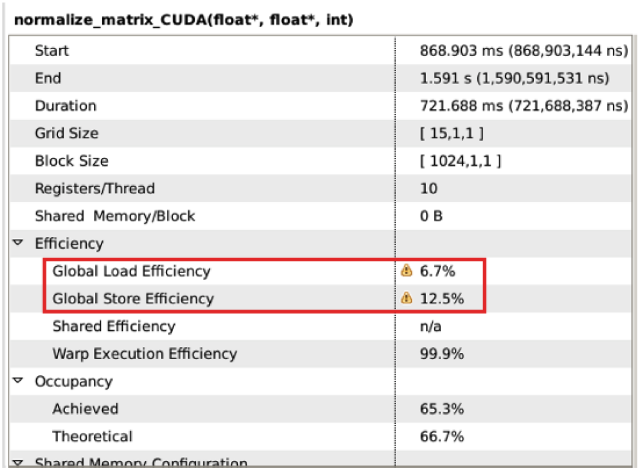

FIGURE 5. NormMat(A, d, *p*) kernel distilled.

An analysis of the stall reasons on the GPIC code are presented in Figure 6. A stall condition occurs when a core cannot run the next instruction because of a dependency on a previous operation. As might be expected, the main reason for processor idleness was memory dependency (47.77%), which corresponds to the time when the GPU is waiting for the completion of a data transfer between the GPU and the CPU. The second reason for idleness was the dependency between

TABLE 3. Runtime in seconds and the percentage gain of GPIC modified version.

Dataset	N	GPIC time [s]	GPIC modified time [s]	Gain GPIC modified [%]
2 moons	15k	3.96	0.76	19.19 %
3 circles		4.01	0.77	19.20 %
2 moons	30k	17.57	5.57	31.70 %
3 circles		17.75	5.51	31.04 %
2 moons	45k	44.97	20.84	46.34 %
3 circles		45.69	20.77	45.45 %

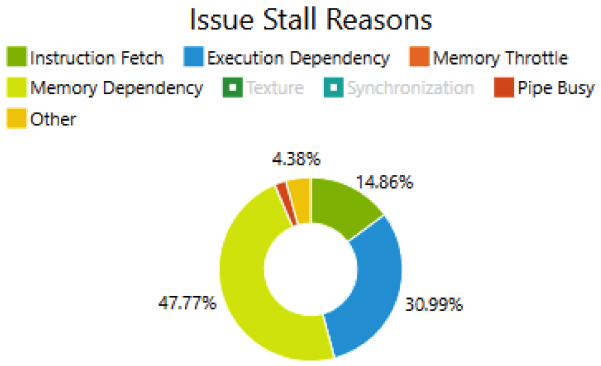

FIGURE 6. GPIC stall reasons.

instructions (30.99%). This behavior was also expected due to the iterative nature of the GPIC algorithm. The profiling experiment on the GPIC algorithm was important to identify bottlenecks and their causes. Table 3 presents the performance gain on a modified version of the GPIC code over the first version, where the average runtime decreased 32.15%. Finally, Table 4 presents the new runtime and speedup of the modified GPIC compared with PIC (sequential version of the GPU code running it on just one thread). Results show a significant speedup of the GPIC Algorithm which, on average, was 685.82 times faster than the original PIC (sequential version of the GPU code running it on just one thread).

TABLE 4. Runtime (in seconds) and speedup comparison of PIC (sequential version of the GPU code running it on just one thread) and GPIC modified version.

Dataset	N	A size [GB]	PIC Sequential time [s]	GPIC time [s]	GPIC x PIC Sequential Speedup
2 moons	15k	3.85	838	0.76	1102.63
3 circles			837	0.77	1087.01
2 moons	30k	16	3336	5.57	598.92
3 circles			3336	5.51	605.44
2 moons	45k	36.5	7499	20.84	359.83
3 circles			7500	20.77	361.09

C. GPU MEMORY TRANSFER ANALYSIS

The execution of a kernel in the GPU requires all data used by this kernel to be copied from the CPU to the GPU memory before processing begins. After executing the kernel, the data produced needs to be moved back to CPU memory. Usually, the function used in this type of procedure is a CUDA memory copy operation (*cudaMemcpy*).

An experiment was conducted to analyse the GPU communication/memory access overhead specifically the *cudaMemcpy* operation. A profile process was performed again in GPIC algorithm focused in this metric. This experiment considers the two moons dataset with 23 different size of data volume (5,000, 10,000, 15, 000, . . . 115, 000).

Figure 7 presents the GPIC memory access analysis. The percentage of total execution time used in a CUDA memory copy operation is highlighted with a black line and the percentage of a total execution time of all the other CUDA operations combined is presented in gray. Due to the increase of data volume in the GPU board, the operation of data transfer between GPU and CPU presents itself as the step that consumes most of the time.

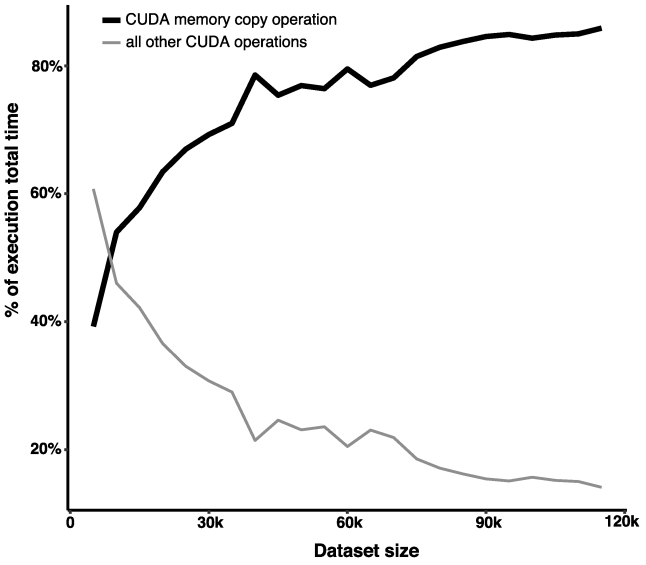

FIGURE 7. GPIC memory access overhead analysis.

V. REAL APPLICATION

Many current real-world problems may involve large datasets with thousands of variables that tend to rise continuously if the current growth rate is maintained. In order to test GPIC in a real dataset, an experiment with GPIC was conducted with a very large data set, which turns out to be unfeasible to the original version of serial PIC algorithm.

In this section, results are presented for an image segmentation problem. Experiments were conducted in an Amazon cloud environment that contains a particular GPU instance `p2.16xlarge`. This instance provides general-purpose GPUs along with high CPU performance, large memory and high network speed for applications.

The operational system is Linux, and the hardware configuration is composed of 64 CPU cores, 720 GB RAM, 16 GPUs cards (NVIDIA K80 model) with 12 GB GDDR5 SDRAM with 2496 CUDA cores each board. A new version of the GPIC code was developed to use multiples GPUs on the same server.

Image segmentation relates to partitioning a digital image into multiple regions, based on some criteria. Formally it is defined as the process of partitioning an image into non-intersected regions, where the pixels of these areas share similar properties. In the experiments performed using the GPIC method, only the colour information was explored during image segmentation process.

A. RUNTIME

To evaluate the performance of the GPIC method in the image segmentation application, 157 aerial images were selected with dimensions of 350×290 resulting in a 101,500 pixels vector. The whole dataset contained 15,935,500 pixels for the 157 images. These images were captured by a drone during a real flight over the city of São Carlos, Brazil. The entire dataset was provided by the Instituto de Estudos Avançados da Aeronáutica IEAV, São José dos Campos, Brazil.

In order to better understand how multiple GPUs implementation of GPIC can be useful to process the whole dataset (15,935,500 pixels), an experiment was conducted with three multiple GPUs configurations:

- Scenario I - 8 GPUs cards (NVIDIA K80 model) and 8 images running at the same time one on each GPU board;
- Scenario II - 12 GPUs cards (NVIDIA K80 model) and 12 images running at the same time one on each GPU board;
- Scenario III - 16 GPUs: cards (NVIDIA K80 model) and 16 images running at the same time one on each GPU board.

In this experiment, each image was copied directly to one of the available GPUs and the GPIC method was immediately applied. After processing in the GPUs, the result was returned to the CPU memory. Since the multiple GPUs have the same computational resources, the images were evenly distributed. Figure 8 presents a schematic diagram of the GPIC running in multiple GPUs in the Amazon cloud environment.

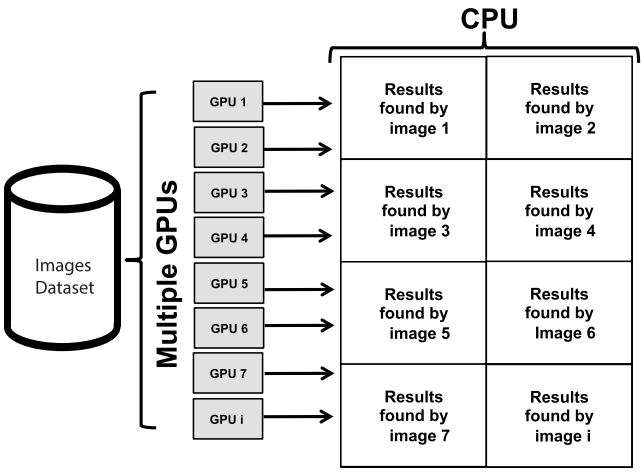

FIGURE 8. GPIC method running in multiple GPUs enviroment.

The results of this experiment are presented in Table 5. As we can see in Scenario I, the average time to process each image in each GPU board is 26.19% faster than Scenario II and 52.59% than Scenario III. This situation can be explained due to the GPU bandwidth saturation of the bus, since the data volume used in Scenario II and III are much bigger than the Scenario I.

TABLE 5. Three multiple GPUs scenarios to analyze GPIC algorithm performance in AWS cloud environment for aerial images dataset. The parameters for all experiments are maxiterations=3, $\epsilon = 0.00001/N$, $k = 2$, and euclidean distance similarity function.

Scenarios	Time in seconds spent for each GPU board			Average total time for each scenario [s]
	Min [s]	Max [s]	Mean [s]	
8 Gpus	161.70	223.30	200.40	3931.84
12 Gpus	187.80	285.70	252.90	3307.93
16 Gpus	249.50	357.90	305.80	2996.84

To complement the analysis, the total time spent to process the whole dataset was evaluated. Scenarios II and III have lower total run-times than scenario I. It can be seen in Table 5 that the total time spent in Scenario I is 18.86% bigger than Scenario II and 31.19% bigger than Scenario III. It can be concluded that although the runtime of each GPU is lower in the scenario with a small number of GPU boards, the scenarios that used a larger number of GPU boards presented an expressive reduction in the total runtime, because it has the capacity to process more images at the same time.

B. CLUSTER QUALITY EVALUATION

In order to compare cluster quality outcomes, the images were pre-processed with two different filters, Gaussian [18] and Median [19]. These filters were applied to reduce the details and texture, smoothing the pictures and under certain conditions, they preserve edges. This required because the aerial images have texture and changes in luminosity that must be reduced to obtain better results when the labeling of the pictures is done.

The criteria adopted for assessing the quality of the yielded clusters partition was based on a cluster quality index [20]. Two types of tests are commonly used to assess the quality of clusters: external and internal [21]. We chose external indexes to validate our experiments once we have a reference partition to compare the results. In the external criteria, the quality index is calculated according to a reference partition obtained from the dataset. Typically, a reference partition P is used to perform the validation. External indexes are computed in the interval [0,1], and the closer the result is to 1, the better the cluster quality is.

The chosen cluster quality indexes were RI (Rand Index) [22], Jaccard [23] and Meil [24]. The above-mentioned experiments were performed on the dataset 30 times for statistical significance and the results for all trials is presented in Table 6. In all experiments the significance level of the statistical tests used was 5%. To analyze the cluster quality results we applied two statistical tests. The first test applied was a Shapiro-Wilk normality test [25],

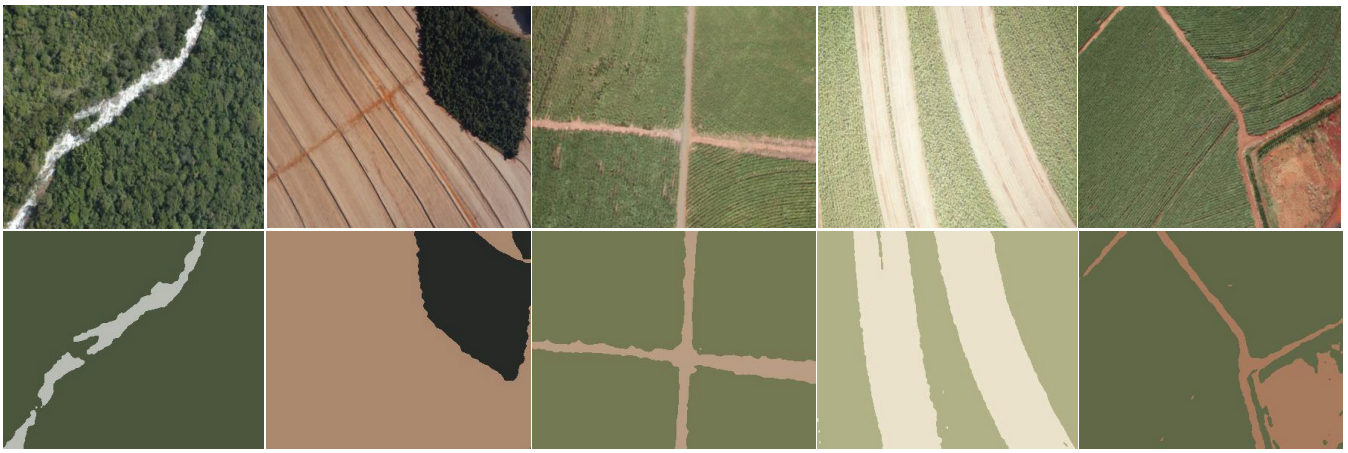

FIGURE 9. GPIC segmentation images result for two clusters using Median filter. The entire dataset was provided by the Instituto de Estudos Avançados da Aeronáutica IEAV, São José dos Campos, Brazil.

and the result found was that for all two filters and all metrics (RI, Jaccard and Fowlkes-Mallows) the data results do not follow a normal distribution.

The second test applied was Wilcoxon Mann-Whitney [26] to check if there is a significant difference between the scenarios and to determine which was the best scenario. It was detected that different between scenarios exists. The best result found was obtained by the Median filter and the best external cluster quality metric is Fowlkes-Mallows 0.8438.

According to Table 6, it can be concluded that for the two filters (Gaussian and Median) applied in the aerial images dataset the GPIC algorithm can generate good cluster results for an image segmentation problem.

TABLE 6. Cluster quality metrics for aerial images dataset.

Filter	External indexes		
	Rand Index	Jaccard	Fowlkes-Mallows
Gaussian	0.7093 $\pm$ 0.1271	0.7093 $\pm$ 0.1271	0.8388 $\pm$ 0.0759
Median	0.7178 $\pm$ 0.1277	0.7178 $\pm$ 0.1277	0.8438 $\pm$ 0.0760

Finally, Figure 9 presents the GPIC results in the image segmentation process for five images of the dataset. It can be observed that GPIC was able to segment the images very consistently.

VI. CONCLUSION

This paper presented a new clustering algorithm, the GPIC, a CUDA-based parallelization of Power Iteration Clustering. Our algorithm is based on the original PIC proposal, adapted to take advantage of the GPU architecture. The proposed method was compared with the sequential implementation, achieving a considerable speedup in synthetic and real large datasets. To the extent of our knowledge, this is the first paper that provides a CUDA-based parallelization of Power Iteration Clustering that is able to reduce runtime maintaining cluster quality of the original PIC algorithm.

We understand the speedup occurred for two reasons. Firstly the use of parallelism techniques and secondly the availability of a hardware that is suitable to run parallel applications. Matrix operations, which are computationally costly, have been replaced by simpler, similar operations that did not impact on the final result of the algorithm.

NVIDIA Visual Profiler tool helped to find bottlenecks in the first version of GPIC code and also estimated the potential performance benefits from removing those bottlenecks. With a small modified version of GPIC code, the average runtime decreased 32.15%.

Finally, a real-world application from computer vision was explored. Results pointed out that GPIC implementation has good scalability, without suffering performance degradation with large datasets.

REFERENCES

[1] R. Bekkerman, M. Bilenko, and J. Langford, Eds., *Scaling Up Machine Learning: Parallel and Distributed Approaches*. Cambridge, U.K.: Cambridge Univ. Press, 2011.

[2] A. S. Shirkhorshidi, S. Aghabozorgi, T. Y. Wah, and T. Herawan, "Big data clustering: A review," in *Computational Science and Its Applications*. Cham, Switzerland: Springer, 2014, pp. 707–720. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-09156-3_49

[3] U. von Luxburg, "A tutorial on spectral clustering," *Statist. Comput.*, vol. 17, no. 4, pp. 395–416, 2007.

[4] M. Maila and J. Shi, "A random walks view of spectral segmentation," in *AI and STATISTICS (AISTATS)*. Jan. 2001.

[5] W. Y. Chen, Y. Song, H. Bai, C. J. Lin, and E. Chang. (2008). *PSC: Parallel Spectral Clustering*. [Online]. Available: http://www.cs.ucsb.edu/~wychen/sc

[6] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, and E. Y. Chang, "Parallel spectral clustering in distributed systems," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 33, no. 3, pp. 568–586, Mar. 2011.

[7] L. Li, Y. Hu, and X. Wang, "A parallel way for computing eigenvector sensitivity of asymmetric damped systems with distinct and repeated eigenvalues," *Mech. Syst. Signal Process.*, vol. 30, pp. 61–77, Jul. 2012.

[8] F. Lin and W. W. Cohen, "Power iteration clustering," in *Proc. 27th Int. Conf. Mach. Learn. (ICML)*, Haifa, Israel, Jun. 2010, pp. 655–662.

[9] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: A Hands-On Approach*. Amsterdam, The Netherlands: Elsevier, 2016. [Online]. Available: https://books.google.com.br/books?id=wcS_DAAAQBAJ

[10] R. Wu, B. Zhang, and M. Hsu, "Clustering billions of data points using GPUs," in *Proc. Combined Workshops UnConventional High Perform. Comput. Workshop Plus Memory Access Workshop*, 2009, pp. 1–6.

[11] J. Dong, F. Wang,, and B. Yuan, "Accelerating BIRCH for clustering large scale streaming data using CUDA dynamic parallelism," in *Proc. 14th Int. Conf. Intell. Data Eng. Autom. Learn. (IDEAL)*, Hefei, China, Oct. 2013, pp. 409–416. [Online]. Available: https://doi.org/10.1007/978-3-642-41278-3_50, doi: 10.1007/978-3-642-41278-3_50.

[12] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An efficient data clustering method for very large databases," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Montreal, QC, Canada, 1996, pp. 103–114.

[13] G. Andrade, G. Ramos, D. Madeira, R. Sachetto, R. Ferreira, and L. Rocha, "G-DBSCAN: A GPU accelerated algorithm for density-based clustering," *Procedia Comput. Sci.*, vol. 18, no. Supplement C, pp. 369–378, 2013. [Online]. Available: https://doi.org/10.1016/j.procs.2013.05.200 and http://www.sciencedirect.com/science/article/pii/S1877050913003438

[14] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proc. KDD*, vol. 96. 1996, pp. 226–231.

[15] C. Lanczos, *An Iteration Method for the Solution of the Eigenvalue Problem of Linear Differential and Integral Operators*. Washington, DC, USA: GPO, 1950.

[16] M. Harris. (2015). Optimizing parallel reduction in CUDA. Nvidia. [Online]. Available: https://docs.nvidia.com/cuda/samples/6_Advanced/reduction/doc/reduction.pdf

[17] NVIDIA. (Sep. 2016). *Nvidia Visual Profiler Tool*. [Online]. Available: https://developer.nvidia.com/nvidia-visual-profiler

[18] K. Ito and K. Xiong, "Gaussian filters for nonlinear filtering problems," *IEEE Trans. Autom. Control*, vol. 45, no. 5, pp. 910–927, May 2000.

[19] J. S. Lim, *Two-Dimensional Signal and Image Processing*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1990, p. 710.

[20] M. Halkidi, Y. Batistakis, and M. Vazirgiannis, "Cluster validity methods: Part I," *ACM Sigmod Rec.*, vol. 31, no. 2, pp. 40–45, 2002.

[21] E. Rendón *et al.*, "A comparison of internal and external cluster validation indexes," in *Proc. Amer. Conf.*, vol. 29. San Francisco, CA, USA, 2011, pp. 158–163.

[22] L. Hubert and P. Arabie, "Comparing partitions," *J. Classification*, vol. 2, no. 1, pp. 193–218, 1985.

[23] P. Jaccard, "Nouvelles recherches sur la distribution florale," *Bull. Soc. Vaudoise Sci. Naturelles*, vol. 44, pp. 223–270, 1908.

[24] M. Meil , "Comparing clusterings by the variation of information," in *Proc. 16th Annu. Conf. Learn. Theory, 7th Kernel Workshop Learn. Theory Kernel Mach. COLT/Kernel*, Washington, DC, USA, Aug. 2003, pp. 173–187, [Online]. Available: https://doi.org/10.1007/978-3-540-45167-9_14, doi: 10.1007/978-3-540-45167-9_14.

[25] P. Royston, "Approximating the Shapiro–Wilk W-test for non-normality," *Statist. Comput.*, vol. 2, no. 3, pp. 117–119, 1992.

[26] V. DePuy, V. W. Berger, and Y. Zhou, "Wilcoxon–Mann–Whitney test," in *Encyclopedia of Statistics in Behavioral Science*. Wiley, 2005. [Online]. http://dx.doi.org/10.1002/0470013192.bsa712, doi: 10.1002/0470013192.bsa712.

GUSTAVO RODRIGUES LACERDA SILVA received the B.Sc. degree in information system and the master's degree in electrical engineering from Federal University Minas Gerais (UFMG), Brazil, in 2006 and 2011, respectively. He is currently pursuing the Ph.D. degree in electrical engineering, UFMG. His research interest are scalable machine learning and parallel and heterogeneous computing, including GPU.

RAFAEL RIBEIRO DE MEDEIROS received the B.Sc. degree in computer engineering from the Federal University of Minas Gerais in 2016, where he is currently pursuing the master's degree in computer science.

BRAYAN RENE ACEVEDO JAIMES received the B.Sc. degree in electronic engineering from the Francisco de Paula Santander University, Colombia, in 2014, the master's degree in electrical engineering from the Federal University of Minas Gerais, Brazil, in 2016, where he is currently pursuing the Ph.D. degree in electrical engineering. His current research interests include position estimation and autonomous navigation in UAVs, pattern recognition, digital image processing, neural networks and remote sensing.

CARLA CALDEIRA TAKAHASHI received the bachelor's degree in electrical engineering from UFMG in 2012, when she was able to participate in an exchange program to Lund University, with an Erasmus Mundus Scholarship, in 2011, and the master's degree in electrical engineering in computer intelligence field from Federal University of Minas Gerais, in 2014, granted at the time a CNPq Scholarship. She is currently pursuing the Ph.D. degree in electrical engineering with UFMG, funded by a Capes Scholarship.

DOUGLAS ALEXANDRE GOMES VIEIRA received the B.Sc. and Ph.D. degrees in electrical engineering from the Universidade Federal of Minas Gerais, Brazil, in 2003 and 2006, respectively. He was a Visiting Researcher with Oxford University, U.K., in 2005, and an Associate Researcher with the Imperial College London, U.K., in 2007. He has authored over 50 refereed papers. His main interests are in multiobjective optimization theory and applications, machine learning, and their interfaces.

ANTÔNIO DE PÁDUA BRAGA received the B.Sc. degree in electrical engineering and the master's degree in computer science from Federal University Minas Gerais, Brazil, in 1987 and 1991, respectively, and the Ph.D. degree in electrical engineering in recurrent neural networks from the Imperial College, University of London, in 1995. Since 1991, he has been with the Electronics Engineering Department, Federal University of Minas Gerais, where he is currently a Full Professor and the Head of the Computational Intelligence Laboratory. He is also an Associate Researcher with the Brazilian National Research Council. As a Professor and a Researcher, he has co-authored many books, book-chapters, journal, and conference papers.