Article

Parallel K-means Clustering for Brain Cancer Detection using Hyperspectral Images

Emanuele Torti [1]*, Giordana Florimbi [1], Francesca Castelli [1], Samuel Ortega [2], Himar Fabelo [2], Gustavo Marrero Callicó [2], Margarita Marrero-Martin [2] and Francesco Leporati [1]

[1] Department of Electrical, Computer and Biomedical Engineering, University of Pavia, Pavia I-27100, Italy; emanuele.torti@unipv.it (E.T.), giordana.florimbi01@ateneopv.it (G.F.), francesca.castelli02@ateneopv.it (F.C.), francesco.leporati@unipv.it (F.L.)

[2] Institute for Applied Microelectronics (IUMA), University of Las Palmas de Gran Canaria (ULPGC), 35017 Las Palmas de Gran Canaria, Spain; sortega@iuma.ulpgc.es (S.O.), hfabelo@iuma.ulpgc.es (H.F.), gustavo@iuma.ulpgc.es (G.M.C.), margarita.martin@ulpgc.es (M.M.)

* Correspondence: emanuele.torti@unipv.it; Tel.: +39-0382-985678

Version October 8, 2018 submitted to Electronics

Abstract: The precise delineation of brain cancer is a crucial task during surgery. There are several techniques employed during surgical procedures to guide neurosurgeons in the tumor resection. However, hyperspectral imaging (HSI) is a promising non-invasive and non-ionizing imaging technique that could improve and complement the currently used methods. The HypErspectraL Imaging Cancer Detection (HELICoiD) European project has addressed the development of a methodology for tumor tissue detection and delineation exploiting HSI techniques. This paper describes the development of the K-means clustering algorithm on different parallel architectures, in order to provide real-time processing during surgical procedures. This algorithm will generate an unsupervised segmentation map that, combined wiht a supervised classification map, will offer guidance to the neurosurgeon during the tumor resection task. We present parallel K-means clustering based on OpenMP, CUDA and OpenCL paradigms. These algorithms have been validated through an *in-vivo* hyperspectral human brain image database.. Experimental results show that the CUDA version is capable of achieving a speed-up of ~ 150 x with respect to a sequential processing.

Keywords: Graphics Processing Units (GPUs); CUDA; OpenMP; OpenCL; K-means; Brain cancer detection; Hyperspectral imaging; Unsupervised clustering

---

1. Introduction

One of the most diffused types of cancer is the brain tumor, which has an estimated incidence of 3.4 per 100000 subjects [1]. There are different types of brain tumors; the most common one concerns the *glial* cells of the brain and is called *glioma*. It accounts from the 30 % to the 50 % of the cases. In particular, in the 85 % of these cases, it is a malignant tumor called *glioblastoma*. Moreover, these kind of *gliomas* are characterized by fast-growing invasiveness, which is locally very aggressive, in most cases unicentric and rarely metastasizing [2].

Typically, the first diagnosis is performed through the Magnetic Resonance Imaging (MRI) and the Computed Tomography (CT). Those techniques are capable to highlight possible lesions. However, it is not always possible to use them, since they can, for example, make interference with pacemakers or other implantable devices. Moreover, the certainty of the diagnosis only comes from the histological and pathological analyses, which require samples of the tissue. In order to obtain this tissue, an *excisional biopsy* is necessary, which consists in the removal of tissue from the living body through surgical cutting. It is important to notice that all those approaches have some disadvantages; in

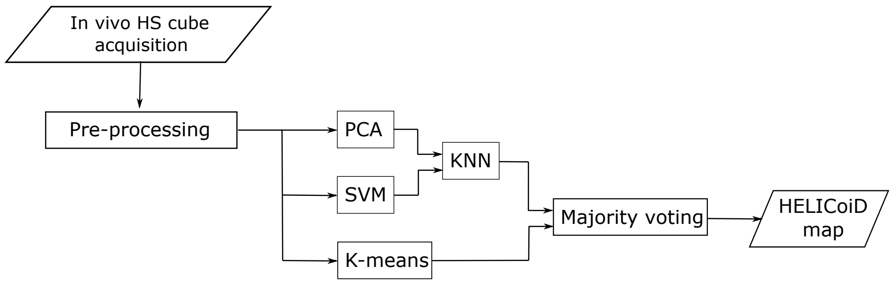

Figure 1. Hyperspectral brain cancer detection algorithm proposed in [3].

particular they are not capable of providing a real-time response and, most important, they are invasive and/or ionizing.

The clinical practice for brain cancers is the tumor resection, which can cure the lowest grade tumors and prolongs the life of the patient in the most aggressive cases. The main issue about this approach is the inaccuracy of the human eye in distinguishing between healthy tissue and cancer. This is because the cancer often infiltrates and diffuses into the surrounding healthy tissue and this is particularly critical for brain cancers. As a consequence, the surgeon can unintentionally left behind tumor tissue during a surgery routine potentially causing tumor recurrence. On the other hand, if the surgeon removes too much healthy tissue, a permanent disability to the patient can be provoked [4].

The HELICoiD European project aims at providing to the surgeon a system which can accurately discriminate between tumor and healthy tissue in real-time during surgery routines [5,6]. Traditional imaging techniques feature a low grade of sensitivity and often cannot clearly determine the tumor region and its boundaries. Therefore, the HELICoiD project exploits Hyperspectral Imaging (HSI) techniques in order to solve this critical issue. Hyperspectral images (HS) can be acquired over a wide range of the electromagnetic spectrum, from visible to near-infrared frequencies and beyond. Hyperspectral sensors acquire the so-called *HS cube* where the spatial information are in the *x-axis* and in the *y-axis*, while the spectral information is in the *z-axis*. Thus, a single hyperspectral pixel can be seen as a mono-dimensional vector, which contains the spectral response across the different wavelengths. Moreover, it is important to notice that the spectral information is strictly correlated with the chemical composition of the specific material. It is possible to say that each hyperspectral pixel contains the so-called *spectral signature* of a certain substance. Thus, different substances can be distinguished by properly analyzing those images [7].

A previous study [3,5] proposed a processing chain for hyperspectral image analysis acquired during brain surgery. The framework developed in this work is depicted in Fig. 1.

First, the acquired HS cube is pre-processed in order to perform a radiometric calibration, reduce the noise and the dimensionality of the HS image and normalize it. After this preparatory step, the image is given as input to three different algorithms: the Principal Component Analysis (PCA), the Support Vector Machine (SVM) and the K-means. The outputs of the first two serve as inputs for the k-nearest neighbor (KNN) filter. The result of this filter is combined with the output produced by the K-means through a majority voting method. This allows to obtain the final classification map. Whilst the PCA, the SVM and the KNN filter are executed through a fixed number of steps, the K-means algorithm iterates until a certain condition is satisfied. In order to provide the real-time classification during surgery, parallel computing is required, since the computational load of the algorithms is extremely high. Parts of this framework have been already developed in parallel, in particular the SVM [8] and the KNN filtering [9] have been recently proposed in the literature. Those works target Graphics Processing Units (GPUs) technology since the considered algorithms have an intrinsically parallel structure. Moreover, GPUs are going to be more and more used for real-time image processing [10–12], together with other scientific applications related to simulation and modeling or machine learning in biomedical applications [13,14].

In this paper, we present the parallelization of the K-means algorithm on different parallel architectures in order to evaluate which one is more suitable for real-time processing. In particular, we consider multi-core CPUs through the OpenMP API and the GPU tehcnology using NVIDIA CUDA framework. We also propose OpenCL based implementations in order to address code portability.

In other words, the work performed allows identifying the best suitable parallel approach between one that could be more appealing since it requires low programming effort and another one more efficient but also more demanding in terms of optimisation and tuning. A tool that allows intra-architectures portability (OpenCL) was also considered but due to its lower performance it is not competitive with the other two approaches.

The paper is organized as follows: Section 2 describes the K-means algorithm for hyperspectral images, while Section 3 details the different parallel versions. Section 4 contains the experimental results and their discussion, making comparisons between the different approaches described in this paper. Section 5 concludes the papers and addresses some possible future research lines.

2. K-means algorithm for hyperspectral images

As already said, the K-means algorithm, unlike the other ones of the hyperspectral brain cancer detection algorithm, is not performed through a fixed number of steps. It performs an unsupervised learning since no previous knowledge of the data is needed. The algorithm separates the input data into K different clusters with a K value fixed a priori. Data are grouped together on the basis of feature similarity. The first step of the algorithm is the definition of K *centroids*, one for each cluster. Using those centroids, a first grouping is performed on the basis of the distance of each point to the centroids. A point is associate to the cluster represented by the nearest centroid. At this moment, each k *centroid* is updated as the baricenter of the group it represents. This process iterates until the difference between the centroids of two consecutive iterations are smaller than a fixed threshold or if the maximum number of iteration is reached.

The pseudo-code of the K-means algorithm is shown in Alg. 1, where Y indicates an hyperspectral image made up of N pixels and L bands. Therefore, the hyperspectral image can be seen as an NxL matrix. The number of clusters to produce is determined by K, the threshold error by min_error and the maximum number of iterations by max_iter. The K-means algorithm produces as a results a KxL array containing the centroids, which will be referred as cluster_centroids in Alg. 1 and an N-dimensional array containing the label of the cluster assigned to each pixel. This array is denoted by assigned_cluster.

In Alg. 1, lines 1 and 2 contain the initialization of the variables. In particular, cluster_centroids is initialized with K different hyperspectral pixels pseudo-randomly chosen from the input image Y. The variable actual_error is initialized with a huge value in order to ensure that the main loop of the algorithm (from line 5 to 17) is performed at least one time. Inside this main loop there are two *for* loops that iterate over the number of pixels N and the number of clusters K (lines from 6 to 12). For each pixel, a temporary array centroid_distances is set to 0, used for storing the distances between the considered hyperspectral pixel and the centroids. The distance metric used for hyperspectral pixels is usually the *Spectral Angle* (SA) which is defined as:

$$SA = \theta(x,y) = \cos^{-1}\left(\frac{\sum_{h=1}^{L} x_h y_h}{\left(\sum_{h=1}^{L} x_h^2\right)^{1/2}\left(\sum_{h=1}^{L} y_h^2\right)^{1/2}}\right) \tag{1}$$

where x and y are the spectral vectors and x_h and y_h represent the response of the h-th band of x and y respectively, being L the number of bands.

The label assigned to the i-th pixel corresponds to the group represented by the centroid with the minimum SA value, as shown in line 11. This phase is repeated for each pixel.

After these steps, the centroids used for the SAs computation are stored in the previous_centroids array. Successively, the centroids are updated by computing the barycenter of each group that is computing the mean value, for each band, of the pixels belonging to the group. Using the updated

Algorithm 1 K-means

```
Input: $Y$, $K$, min_error, max_iter
1: Pseudo-random initialization of cluster_centroids
2: Initialize previous_centroids at 0          ▷ previous_centroids is an $K$ x $L$ array
3: n_iter $\leftarrow 0$                             ▷ initialize the iteration counter to 0
4: Initialize actual_error with a huge value
5: while actual_error $>$ min_error and n_iter $<$ max_iter do
6:     for i:=1 to $N$ do
7:         Initialize centroid_distances to 0      ▷ centroid_distances is a $K$-dimensional array
8:         for j:=1 to $K$ do
9:             centroid_distance$_j$ $\leftarrow$ distance between the $j$-th centroid and the $i$-th pixel
10:        end for
11:        assigend_cluster$_i$ $\leftarrow$ index of min centroid_distance
12:    end for
13:    previous_centroids $\leftarrow$ cluster_centroids
14:    update cluster_centroids
15:    actual_error $\leftarrow \frac{\sum_{i=1}^{K}\sum_{j=1}^{L}|\text{previous\_centroids}_{i,j}-\text{cluster\_centroids}_{i,j}|}{K \cdot L}$
16:    n_iter $\leftarrow$ n_iter $+ 1$
17: end while
Output: assigned_cluster, cluster_centroids
```

centroids and the previous ones, it is possible to evaluate the variation from the previous iteration. It represents how much the centroids have changed and it can be used as a stopping criteria when these variations become small (line 15). The last step of the *while* loop is the increment of the n_iter variable, used for controlling the maximum number of iterations performed by the algorithm.

The next section describes the serial and the parallel versions of this algorithm that we developed using different parallel approaches, together with a code profiling carried out in order to identify the heaviest code parts from the computational point of view.

3. Parallel K-means implementations

First, we developed a serial version of the K-mean algorithm written in C code. It serves both as reference for validating the results of the parallel implementations and for performing a careful code profiling needed to identify the most complex code parts. The numerical representation used is the IEEE-754 floating-point single precision.

3.1. Serial code profiling

The code profiling was performed using a dataset formed by real HS images and assuming $K = 24$, min_error $= 10^{-3}$ and max_iter $= 50$. This K value was stablished during the development of the HS brain cancer algorithm presented in [3]. Using this configuration, the execution of the algorithm never reached the maximum number of iterations. The characteristics of the dataset are shown in Table 1.

The profiling highlighted that the heaviest code parts are the computation of distances, which are evaluated between each hyperspectral pixel and each centroid. In the considered cases, the *for* loops of lines 6-12 (Alg. 1) take from 94 to 98 % of the time for the smallest and the biggest image, respectively. Notice that these computations can be performed in parallel, since there is no dependency between the evaluations needed by a single pixel and the others.

Table 1. Dataset characteristics

Image ID	# of rows	# of columns	Total # of pixels	# of bands	Size (MB)
Image 1	329	379	124,691	128	60.88
Image 2	493	376	185,368	128	90.51
Image 3	402	472	189,744	128	92.65
Image 4	496	442	219,232	128	107.05
Image 5	548	459	251,532	128	122.82
Image 6	552	479	264,408	128	129.11

3.2. OpenMP algorithms

OpenMP[1] is a parallel programming framework capable of exploiting multi-core architectures. It is based on a set of simple *#pragma* statements used for code annotations that indicates to the compiler which parts should be parallelized. An example is the *#pragma omp parallel for* statement, which generates a set of parallel threads and assigns to each one a group of iterations. It is also possible to indicate to the compiler which variables should be shared among the threads and which ones are private through the *shared* and *private* clauses, respectively. Finally, it is possible to choose the scheduling algorithm to use through the *schedule* option. The supported scheduling algorithms are *static*, *dynamic* and *guided*. In the first case, the number of iterations are equally or as equal as possible subdivided among the threads. Thus, each thread performs the same number of iterations. The *dynamic* scheduling uses the internal work queue to give a chunk-sized block of loop iterations to each thread. When a thread finishes, it retrieves the next block of loop iterations from the top of the work queue. The default value of the chunk size is 1, but it is possible to change it by a proper command. Finally, the *guided* scheduling is similar to the *dynamic* one, but the chunk size starts from a big value and then decreases in order to manage load imbalance between different iterations.

We developed two different OpenMP versions of the K-means algorithm. The first one parallelizes the *for* loop which iterates over the hyperspectral pixels (line 6, Alg. 1). In this way, at each iteration of the main *while* loop, a set of parallel threads are generated and each one computes the SA between a certain group of pixels and the centroids. All the other operations are performed in a serial way. The shared arrays are the cluster_centroids and the input image Y, while all the other variables are private. In this version, the parallel region is created and destroyed at each iteration of the main *while* loop.

Concerning the second implementation, the majority of the operations are performed in parallel. The operations that continue to be performed sequentially are the actual_error computation and the increment of n_iter, at lines 15 and 16 of Alg. 1, respectively. A barrier must be placed after the actual_error computation, in order to prevent the other threads to evaluate the *while* condition with an inconsistent old value. In this case, also the centroid_distance is declared as shared. Notice that, in this version, the parallel region is created and destroyed only once, at the beginning and at the end of the main *while* loop. However, in this case, it is necessary to introduce a barrier in order to ensure the correct execution of the program.

3.3. CUDA algorithms

CUDA[2] is a parallel programming framework developed by NVIDIA to exploit GPU computing power. In this framework, the GPU, also called *device*, is seen as a parallel co-processor, with separated address space with respect to the CPU, also called *host*. The execution of a CUDA program always begins from the host, using a serial thread. When it is necessary to perform a parallel operation, the host allocates memory on the GPU and transfers the data to that memory. Those two operations

[1] https://www.openmp.org/
[2] https://developer.nvidia.com/cuda-zone

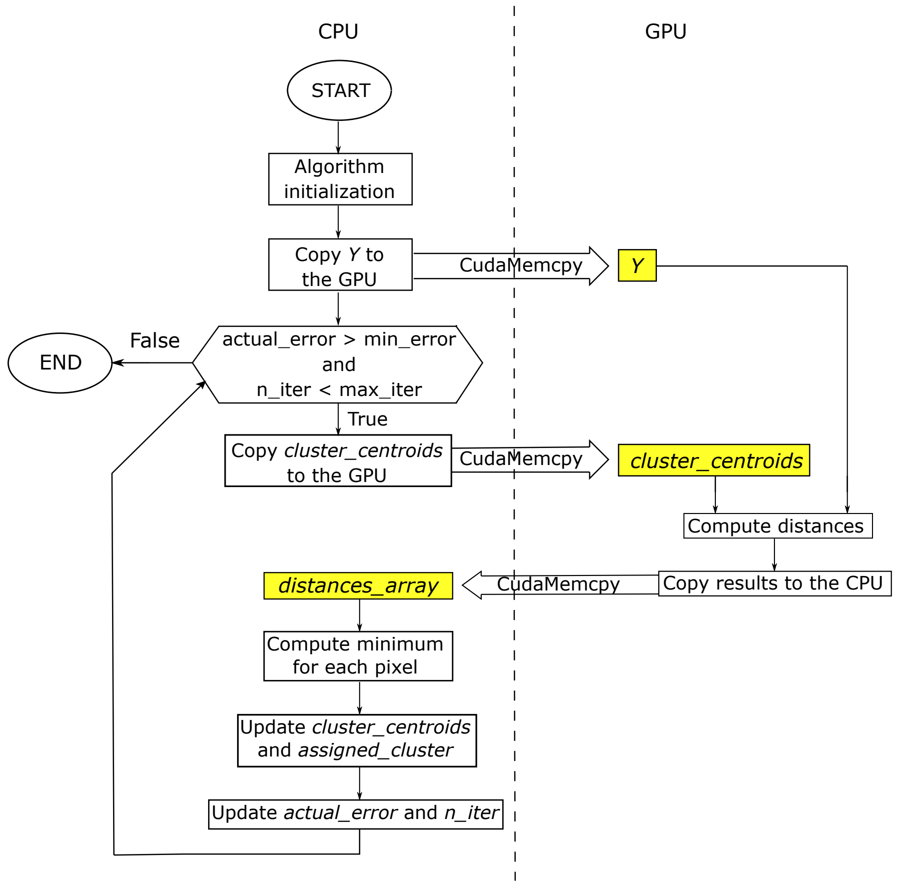
CPU
GPU
START
Algorithm initialization
Copy *Y* to the GPU
CudaMemcpy
Y
actual_error > min_error and n_iter < max_iter
False
END
True
Copy *cluster_centroids* to the GPU
CudaMemcpy
cluster_centroids
Compute distances
Copy results to the CPU
CudaMemcpy
distances_array
Compute minimum for each pixel
Update *cluster_centroids* and *assigned_cluster*
Update *actual_error* and *n_iter*

Figure 2. Schematization of the first GPU implementation. The operations are in white boxes, while data are in yellow boxes.

are performed through the *cudaFree* and *cudaMemcpy* routines. At this point, the GPU generates thousands of parallel threads, which cooperate in order to perform the desired computation. The function performed by the GPU is called *kernel*. When the kernel execution ends, the CPU retrieves the results from the GPU memory through memory transfer (*cudaMemcpy* routine). The GPU memory is then deallocated by the *cudaFree* routine. The execution proceeds then in a serial way.

The threads generated by the GPU are grouped into *blocks*, which form the *grid*. The blocks can be mono-dimensional, bi-dimensional or three-dimensional and the number of threads within a block can be chosen by the programmer.

The typical bottleneck of GPU applications is represented by memory transfers. Therefore, it is necessary to properly manage them in order to achieve the best performance.

In this work, we present three different parallel versions of the K-means algorithm. The first one is based on the parallelization of the distance computation. In this case, the thread performs the computation of the distance between the assigned pixel and the K centroids. This kernel takes as inputs the hyperspectral image Y and the K centroids stored in the cluster_centroids variable. It produces as output a NxK array which contains the distances between each pixel and each centroid. In particular, the i-th row and the j-th column of this array store the distance between the i-th pixel and the j-th centroid. Therefore, it is necessary to add a supplementary temporary array (NxK) with respect to the serial implementation. The schematization of this implementation is shown in Fig. 2.

The hyperspectral image Y is copied to the GPU memory only once, before the beginning of the main *while* loop. This has been done since the image is not modified by the algorithm. At every iteration, the only data transferred to the GPU is the matrix containing the centroids, that is used, together with the image, to compute the distances, stored in a temporary matrix (distances_array in Fig.

2). Data are sent back to the host, which computes the minimum distance for each pixel (i.e. each row of this matrix) and then updates the centroids and computes the error in order to evaluate convergence.

The second CUDA version has been developed in order to avoid the limit of the amount of data transferred during each iteration of the main *while* loop. Therefore, the minimum distance computations, the centroids update and the error evaluation have been performed on the GPU side. Since the distances are stored in an $N \times K$ array, the kernel used to find the index of the minimum distance is executed by N threads. The i-th thread performs a *for* loop in order to evaluate the minimum distance of the i-th centroid. In other words, this task has been parallelized by assigning to each thread the computation of the minimum distance for one pixel. The index of the minimum distance is stored in the *assigned_cluster* array, which contains the classification obtained at the current iteration of the main loop. The update of the centroids has been performed by a simple kernel where the i-th thread computes the update of the i-th centroid. Concerning the error evaluation, it is possible to use the highly optimized routines offered by the CUBLAS library. In particular, it is possible to use the *cublasSasum* routine, which calculates the sum of the absolute values stored in the input array. Before activating this kernel the element-wise difference between the values stored in the *actual_centroids* and *previous_centroids* arrays must be computed. This is done by a kernel in which a single thread computes the difference between two elements. The sum computed by the *cublasSasum* routine is returned to the host, which performs the final division needed for error evaluation and increments the number of iteration. The schematization of this CUDA version is shown in Fig. 3.

It is important to notice that, in this version, only a single precision floating-point value is transferred at each iteration of the main loop. However, an additional data transfer at the end of the main loop must be performed, since it is necessary to retrieve the *assigned_cluster* that contains the hyperspectral pixel classification.

The last CUDA version developed in this work exploits the *dynamic parallelism* introduced by CUDA 6.0. This allows to use a thread inside a kernel in order to generate a grid which executes another kernel. In the proposed case, it is possible to take advantage of dynamic parallelism by moving the main *while* loop inside the kernel. In other words, this version is made up of a single kernel executed on the GPU by a single thread, which manages the activation of the kernels already described for the second CUDA version. In this case, the only memory transfers are performed before (the hyperspectral image Y) and after the main loop (the classified pixels *assigned_cluster*). However, it is worth noting that the activation of a kernel from another kernel requires a launching overhead, which will be discussed in Section 4.

3.4. OpenCL algorithms

OpenCL[3] is a parallel programming framework maintained by the Khronos Group which addresses the issue of portability between devices from different vendors. It assumes a model similar to CUDA, with the difference that the blocks are called *working groups* and the threads are called *working items*. The computing platforms that can be programmed using OpenCL range from multi-core CPUs to many-core GPU and finally to FPGAs. Similarly to CUDA, this paradigm assumes that the computing platform is made up of a serial processor, called *host*, and one or more parallel *devices*. At the beginning of an OpenCL application, it is important to correctly initialize the execution context. This has the effect of pointing out to the OpenCL environment which devices will be used in the case that there are more than one OpenCL compatible boards installed on the same machine. Data that must be processed by the devices are stored into *buffers*, which can be of different types, depending on the targeting devices of the implementation. In particular, considering a generic GPU as a device, a buffer is a portion of the device memory where data are copied from the host memory. On the other hand, if we consider as a device a multi-core CPU or an integrated GPU which shared the RAM

[3] https://www.khronos.org/opencl/

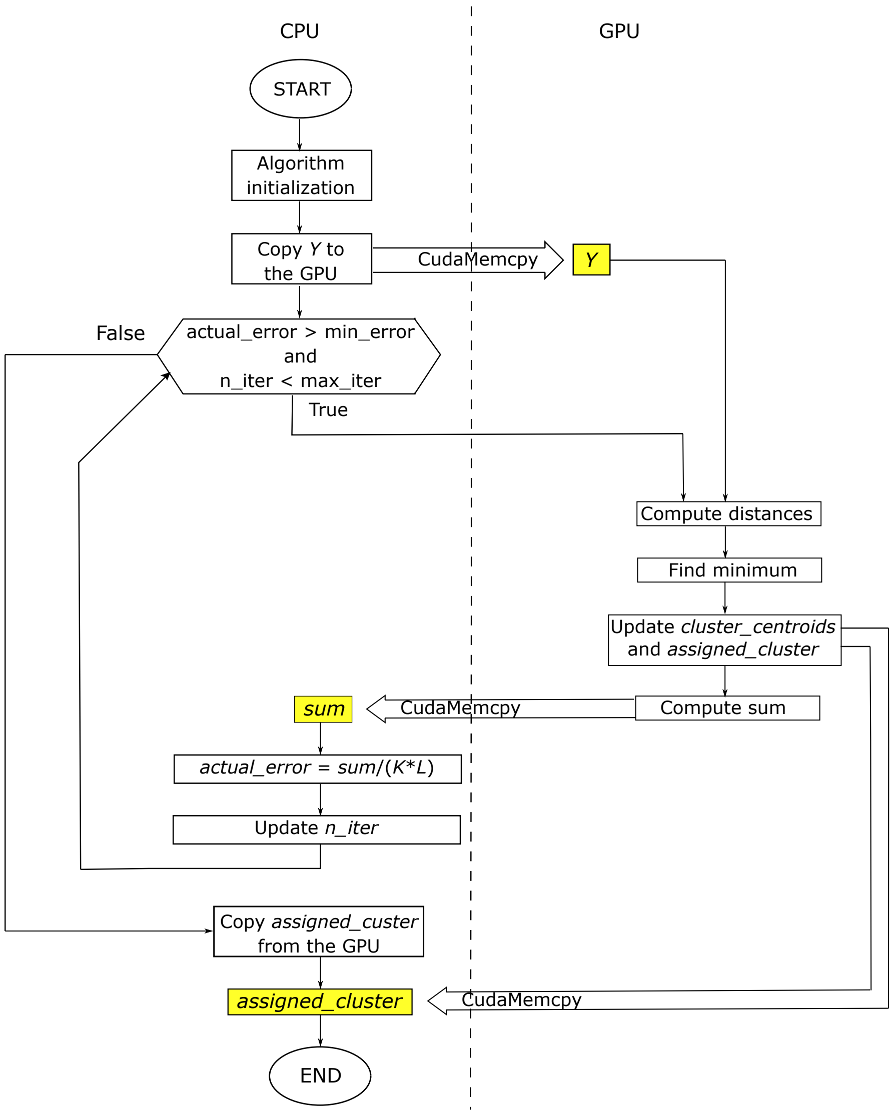

Figure 3. Schematization of the second GPU implementation. The operations are in white boxes, while data are in yellow boxes.

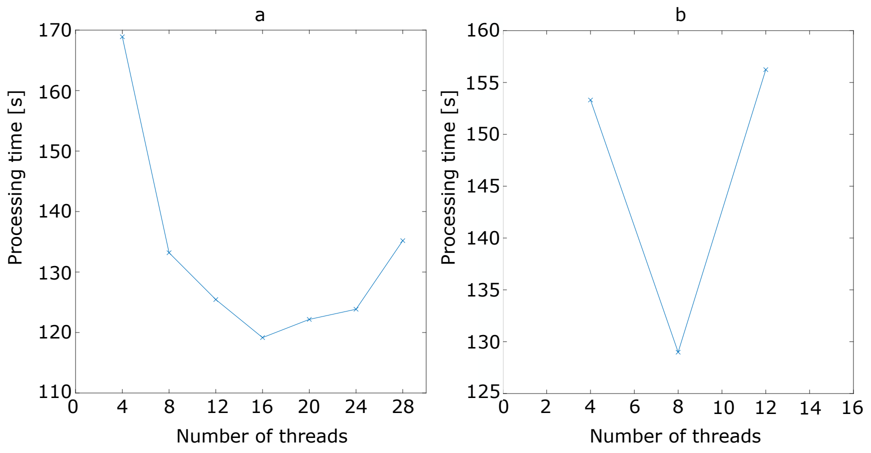

Figure 4. Processing time for Image 6 with respect to the number of threads for the first (a) and for the second (b) OpenMP version.

memory with the host, the buffer is only a reference to the RAM portion where data are stored. Notice that, in this last case, there will be no data transfers between host and device since the RAM address space is shared. An important difference when compared with CUDA is the absence of dynamic parallelism, thus, it is not possible to activate a kernel from another one. Therefore, we only developed two versions based on OpenCL. The first one performs the parallel computation of the distances on the device while the other operations are performed on the host. The second version performs all the operations inside the main *while* loop on the device, as we implemented in the second CUDA version already described. However, it is not possible to exploit the CUBLAS library, since it is strictly correlated with the adoption of NVIDIA devices. Therefore, we exploit the *clBLAS* library which is very similar to CUBLAS. In particular, we use the *clblasSasum* routine which performs the summation of all the elements stored in a given array. All the other operations have been performed as described for the second CUDA version.

4. Experimental results and discussion

All the parallel implementations have been tested with the hyperspectral dataset already used for serial code profiling and shown in Table 1. The images have been employed both to evaluate the processing times of the different versions and to validate the results. All the parallel versions have obtained the same results than the serial one when removing the random initialization. In other words, if the serial and the parallel versions have the same initialization, they perform the same number of iterations and produce the same outputs.

4.1. OpenMP performance evaluation

First, for each OpenMP implementation, several tests have been conducted in order to establish the optimal number of threads to be generated, using the biggest image (Image 6). These tests have been performed on an Intel i7 6700 processor working at 3.40 GHz equipped with 32 GB of RAM. The codes have been compiled with the *vc140* compiler, using compilation options to indicate the target architecutre (i.e. x64 processor) and to maximize the processing speed. The processing times have been measured through the *omp_get_wtime* routine. The obtained results are shown in Fig. 4.

It is important to highlight that the experiments have been conducted with the same initialization, and they provide the same number of iterations and the same classification results. For the first OpenMP implementation, we measured the processing time from 4 to 28 threads. We did not test the application with more threads since the processing time begins to significantly grow after 28 threads. By analyzing Fig. 4a, it is possible to see that the optimal number of threads for the first OpenMP version is 16. These measures have also been performed for the second OpenMP version, but in this

Table 2. Comparison between the serial and the OpenMP versions of the algorithm. The speed-up is reported between brackets.

Image ID	# of iterations	Serial [s]	OpenMPv1 [s]	OpenMPv2 [s]
Image 1	32	272.20	73.18 (3.72x)	75.88 (3.59x)
Image 2	13	162.37	44.42 (3.65x)	45.80 (3.55x)
Image 3	23	289.64	77.81 (3.72x)	81.35 (3.56x)
Image 4	10	151.50	40.98 (3.70x)	43.42 (3.49x)
Image 5	13	214.52	59.00 (3.64x)	63.09 (3.40x)
Image 6	25	465.51	118.31 (3.93x)	136.99 (3.40x)

Table 3. Comparison between the serial and the CUDA versions of the algorithm on a Tesla K40 GPU. The speed-up is reported between brackets.

Image ID	# of iterations	Serial [s]	CUDAv1 [s]	CUDAv2 [s]	CUDAv3 [s]
Image 1	32	272.20	87.48 (3.11x)	4.52 (60.22x)	4.87 (55.89x)
Image 2	13	162.37	54.51 (2.98x)	2.97 (54.67x)	3.34 (48.61x)
Image 3	23	289.64	100.67 (2.88x)	4.99 (58.04x)	5.12 (56.57x)
Image 4	10	151.50	56.41 (2.69x)	2.96 (51.18x)	3.47 (43.66x)
Image 5	13	214.52	79.74 (2.69x)	3.99 (53.76x)	4.14 (51.82x)
Image 6	25	465.51	159.77 (2.91x)	7.45 (62.48x)	7.83 (59.45x)

case the maximum number of threads tested was 12 since the processing times begins to grow. In this case, as highlighted by Fig. 4b, the optimal number of threads is 8.

After establishing the optimal number of threads, we tested the two OpenMP versions on the entire dataset presented in Table 1. In order to allow a direct comparison between the serial and the parallel versions, we initialized the algorithm with the same values for a given image. The obtained results, together with the speed-up values, are reported in Table 2, where OpenMPv1 indicates the first version (the parallelization of the distance metric computation), while OpenMPv2 indicates the second one (the whole main loop parallelized). Those results, together with the others obtained by the CUDA and OpenCL versions, will be discussed in Section 4.4.

4.2. CUDA performance evaluation

The three CUDA versions have been compiled using the NVIDIA *nvcc* compiler, which is part of the CUDA 9.0 environment. Compilation options have been chosen in order to maximize the execution speed. The tests have been conducted using two different GPUs. The first one is a NVIDIA Tesla K40 GPU equipped with 2880 CUDA cores working at 750 MHz and with 12 GB of DDR5 RAM. It is based on the Kepler architecture that does not have a graphical output port since it is optimized for scientific computations. The second GPU is a NVIDIA GTX 1060 equipped with 1152 CUDA cores working at 1.75 GHz and with 3 GB of DDR5 RAM. This GPU is more recent than the first one and it is based on the Pascal architecture, having a graphical output port. In order to take full advantage of the specific architecture of each GPU, we indicate to the compiler which is the target micro-architecture. Specifically, we used the options *sm_35* and *compute_35* for the Tesla K40 GPU and the options *sm_60* and *compute_60* for the GTX 1060, where the values 35 and 60 represent the Kepler and the Pascal architecture, respectively. The results obtained using the Tesla K40 GPU are reported in Table 3, while the results obtained by the GTX 1060 GPU are reported in Table 4.

In both tables, CUDAv1 indicates the version where only the distance computation is computed on the GPU, CUDAv2 indicates the version where all the operations are performed in parallel and, finally, CUDAv3 indicates the version exploiting dynamic parallelism.

Concerning the GPU implementation, we also conducted a profiling using the NVIDIA Visual Profiler. This tool allows to profile the code execution on GPU together with memory transfers, in order to evaluate the efficiency of the implementation. Figure 5 shows the results obtained by profiling

Table 4. Comparison between the serial and the CUDA versions of the algorithm on a GTX 1060 GPU. The speed-up is reported between brackets.

Image ID	# of iterations	Serial [s]	CUDAv1 [s]	CUDAv2 [s]	CUDAv3 [s]
Image 1	32	272.20	80.84 (3.37x)	2.37 (114.85x)	4.21 (64.66x)
Image 2	13	162.37	44.54 (3.65x)	1.94 (83.70x)	2.93 (55.42x)
Image 3	23	289.64	98.61 (2.94x)	2.66 (108.89x)	2.96 (97.85x)
Image 4	10	151.50	52.47 (2.89x)	2.02 (75.00x)	3.25 (46.62x)
Image 5	13	214.52	75.00 (2.86x)	2.41 (89.01x)	3.48 (61.64x)
Image 6	25	465.61	147.50 (3.16x)	3.16 (147.34x)	3.69 (126.18x)

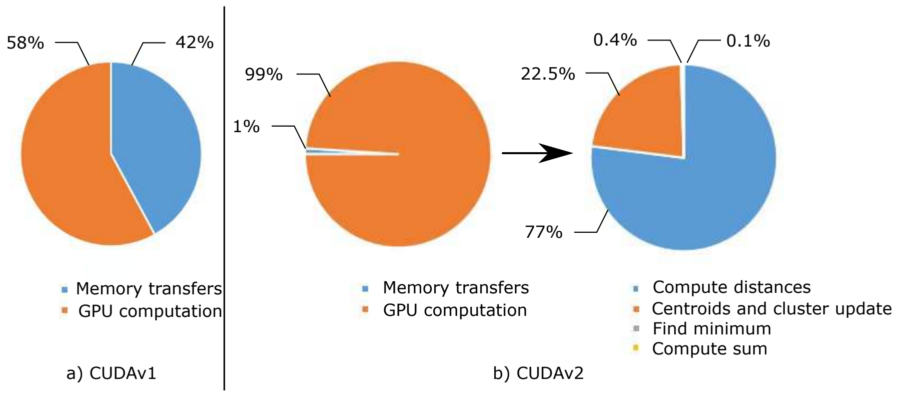

Figure 5. Profiling of GPU versions on the NVIDIA GTX 1060 board for the CUDA v1 (a) and CUDA v2 (b) versions.

the Image 1 (the most demanding one) processing on the GTX 1060 with the CUDAv1 (a) and CUDAv2 (b) codes. Concerning the CUDAv2, in Fig 5b the different kernels executions percentage on the GPU are detailed. Profiling of CUDAv3 is not shown since it is very similar to CUDAv2 as well as the code profiling on the NVIDIA Tesla K40 GPU.

4.3. OpenCL performance evaluation

OpenCL codes have been compiled using vendor-specific compilers. In particular, the OpenCL version without memory transfers have been tested on an Intel i7 6700 processor working at 3.40 GHz, equipped with 32 GB of RAM and on an Intel HD Graphics 530 integrated GPU with 16 cores working at 350 MHz. The integrated board shares the RAM with the CPU. Concerning the OpenCL version which performs memory transfers, it has been tested on the NVIDIA GTX 1060 GPU. Results obtained by the OpenCL versions are reported in Table 5.

Table 5. Comparison between the serial and the OpenCL versions of the algorithm. The speed-up is reported between brackets.

Image ID	# of iterations	Serial [s]	Intel i7 [s]	Intel HD 530 [s]	GTX 1060 [s]
Image 1	32	272.20	74.13 (3.67x)	183.96 (1.48x)	57.61 (4.72x)
Image 2	13	162.37	44.32 (3.66x)	114.24 (1.42x)	35.61 (4.56x)
Image 3	23	289.64	79.52 (3.64x)	203.62 (1.42x)	63.75 (4.54x)
Image 4	10	151.50	40.52 (3.74x)	113.82 (1.33x)	35.36 (4.28x)
Image 5	13	214.52	59.92 (3.58x)	156.56 (1.37x)	47.03 (4.56x)
Image 6	25	465.51	121.59 (3.83x)	312.68 (1.49x)	93.65 (4.97x)

4.4. Comparisons and Discussion

The OpenMP version that offers the better results is the one where only the distance evaluations are processed in parallel. In this version, a parallel region is created and then destroyed at every iteration of the main loop, while in the second version the parallel region is created only once before the beginning of the main loop and is destroyed after the end of the main loop. However, the second version requires synchronization barriers between the threads, since there are operations that should be performed sequentially in order to obtain correct results. As an example, the increment of the number of iterations and the check of the conditions for repeating or not the main loop should be performed by a single thread. This is a critical issue since the advantage of creating and destroying the parallel region only once is thwarted by the synchronization bottleneck. The result analysis shows that the processing times are very similar, but the processor manages better the first version (OpenMPv1). Moreover, the speed-up values of the two versions are similar. Finally, it is important to highlight that the considered processor is equipped with 4 physical cores and the obtained speed-up is always greater than 3.5x. This means that the parallelization efficiency is close to the theoretical value.

Concerning the CUDA versions, by analyzing Tables 3 and 4, it is possible to observe that, for both GPU boards, the first CUDA version (CUDAv1) performs worst than the OpenMP ones. The reason is highlighted in Fig. 5a, where the profiling results show that the memory transfers take about 42% of the time and only the remaining 58% is used for the computation. This is the typical bottleneck of GPU computing since the memory transfers are performed by the PCI-express external bus. The second CUDA version (CUDAv2) is not affected by this issue since the amount of data transferred at each main iteration is significantly lower than in the previous case. It is possible to parameterize the amount of transferred data at each iteration for this two version. In the first case (CUDAv1), the data transferred is the `distance_array` matrix, which is made up of $N \times K$ elements represented in single precision floating-point arithmetic, while in the second case (CUDAv2) only one single precision floating-point value is copied back to the host. In the third case (CUDAv3), when dynamic parallelism is used, there are no data transfers inside the main loop. Therefore the third CUDA version is the one which transfers the minimum possible amount of data, but it does not perform better than the second version. This is because the dynamic parallelism produces an overhead due to the GPU switch between the main kernel and the subroutine kernel. This overhead affects every sub-kernel activation. In this specific case, four different sub-kernels are activated at every iteration. This overhead is not negligible and, as it can be seen form the results of Tables 3 and 4, it takes longer than the copy of a single float value from device to host. In other words, the time needed by the GPU to manage the generation of four sub-kernels (CUDAv3) is comparable with the time taken by a single value copy from the host to the device memory (CUDAv2). Finally, for all the CUDA versions, the GTX 1060 board performs better than the Tesla K40, even if the last board is optimized for scientific computations. This is because the first board is equipped with a more recent architecture which has better CUDA cores working at a higher frequency than the Tesla GPU.

The analysis of the OpenCL versions highlight that, considering the Intel i7, the processing times are close to the OpenMP ones. For what concerns the Intel HD 530 integrated GPU, the performance is very poor and the speed-ups are negligible. This is probably due to the low-end integrated GPU with a working frequency of 350 MHz and only 16 parallel processing elements. Therefore, it is not possible to obtain a significant speed-up compared to the serial version. Comparing the OpenCL version and the CUDA versions running on the GTX 1060 GPU, it is possible to notice that the OpenCL version performs better than the CUDAv1, but is significantly slower than the other two CUDA versions. These CUDA versions (CUDAv2 and CUDAv3) employ highly optimized routines, which exploits all the hardware features of a GPU. Moreover, the CUDA versions have been compiled using compilation options in order to produce an executable code which fully exploits the specific target architecture. This is not possible in OpenCL since it targets portability between different devices as main feature.

We also performed a comparative study between the three best performing versions of the three considered technologies in order to characterize how the speed-up varies with respect to the number

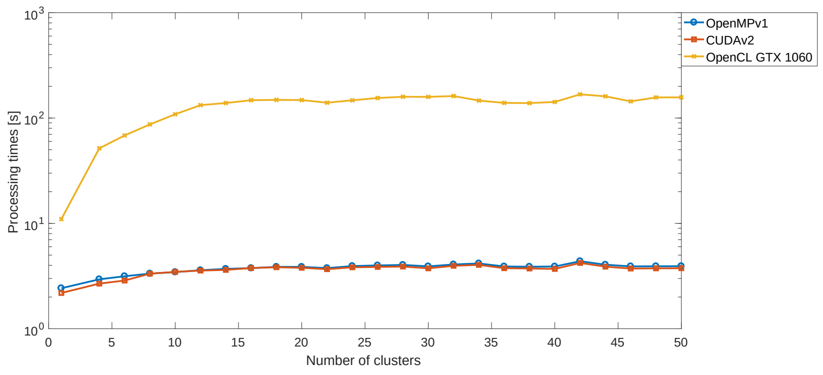

Figure 6. Speed-ups achieved by the three best parallel implementation (one for each evaluated parallel technology) with respect to the number of clusters.

of clusters. In particular, we performed experiments using Image 6 and K values variying from 2 to 50. The speed-ups of the OpenMP, CUDA and OpenCL best versions with respect to the serial implementation are shown in Fig. 6 using a semi-logarithmic scale. It is possible to see that the CUDA version has a speed-up that ranges from 10x to ~ 150 x and from 12 clusters on it becomes nearly constant. On the other hand, the solutions based on a multi-core processor have speed-ups that are close to 4x.

In the literature, there are different works about parallel K-means.
Baramkar *et al.* [15] perfomed a review of different parallel GPU based K-means, but the considered works where focused only on general classification, without considering high data dimensionality, which is the case explored in our work. Therefore it is hard to perform direct comparisons with these works, which achieve very different speed-ups ranging from 11x to 220x.

Zechner *et al.* [16] proposed a parallel implementation of this algorithm using both CPU and GPU. In particular, the GPU was only employed for distance computation, while centroids update was left to the CPU. They classified an artificial dataset with two-dimensional elements ranging from 500 to 500, 000. The maximum speed-up achieved was 14x, lower than the one obtained in our work. This is because the optimization proposed in [16] is only valid for low-dimensional data and cannot be employed for classifying high-dimensional data like hyperspectral images.

A similar approach is shown in [17,18], with the difference that also the clusters update has been performed on the GPU. However, between the distance computation and centroids update they performed host computation for updating each pixel label. This choice leads to a maximum speed-up of 60 in both works, lower than our one, since we moved all the computation on the device side.

In [19], a GPU based K-means algorithm is proposed, with a distance computation that is evaluated through a simple Cartesian distance. Under this assumption, they classify 1, 000, 000 pixels with 32 features in 1.15 s. In our case, the bigger image has 264, 408 pixels, the features (i.e. the bands) are 128 and it is processed in ~ 3.56 s. Moreover, the distance metric that we adopt (the spectral angle) is more complex than the one proposed in [19].

Baydoun *et al.* [20] developed a parallel K-means for RGB images classification. They adopted as metric a simple Cartesian distance and they parallelize only this computation, achieving a maximum speed-up of ~ 25x. In this case, the metric and the data dimensionality are very different compared to this work.

In [21], the K-means algorithm was modified in order to further reduce the distance computation. The speed-up varies from 4x to 386x, but also, in this case, it is not possible to perform a direct comparison since there are not sufficient details about the dataset composition. Finally, in [22], the K-means algorithm was developed on GPU with the Cartesian distance. They adopt a modern GPU with 1536 CUDA cores obtaining a maximum speed-up of 88x, which is very similar to our results.

Table 6. Comparison between the proposed work and the literature.

Paper	Maximum image size	Data dimensionality	Technology	Speed-up
[15]	2,000,000	8	GPU NVIDIA GTX 280	220
[16]	500,000	2	GPU NVIDIA 9600 GT	14
[17]	1,000,000	2	GPU NVIDIA 8800 GTX	60
[18]	15,052,800	3	4 x GPU NVIDIA GTX 750Ti	60
[19]	1,000,000	32	GPU NVIDIA GTX 280	N. A.
[20]	16,777,216	3	GPU NVIDIA Tesla C2050	25
[21]	245,057	4	GPU NVIDIA GeForce 210	386
[22]	500,000	16	GPU NVIDIA Quadro K5000	88
[23]	N. A.	N. A.	GPU NVIDIA GTX 1080	18.5
[24]	20,000	10	2 x AMD Opteron quad-core	8
[24]	65,536	10	GPU NVIDIA Tesla 2050	60
[24]	17,692	9	Mitrion MVP FPGA Simulator	N. A.
Our work	264,408	128	GPU NVIDIA GTX 1060	126

Lutz *et al.* [23] proposed a parallel K-means implementation using an NVIDIA GTX 1080 GPU. They perfomed only experiments producing four groups and no further details are given in the paper about the dataset. They achieved a maximum speed-up of 18.5x which is nearly an order of magnitude smaller than the one of our implementation.

A comparative analysis similar to the one we conducted is reported in [24]. Authors exploited GPUs, OpenMP, Message Passing Interface (MPI) and Field Programmable Gate Arrays (FPGAs). However, also in this case, they considered only a dataset made up of 10-dimensional points, therefore the computational complexity of the distance computation is lower than our one. On the other hand, the results were not as good as our ones, since the maximum GPU speed-up achieved is ~ 60x. They also demonstrated that the speed-up could reach a value up to 200x if the number of clusters to produce was significantly increased (i.e. more than 2000 clusters), but a study of how this speed-up varied also with respect to the data dimensionality were not carryed out. Concerning OpenMP, the classification of $20,000$ 10-dimensionality points took ~ 3 s. Our smallest image is 6 times bigger than this one and with a dimensionality 18 times grater than the one considered. Keeping this in mind, the performance of our best OpenMP version is quite similar to this one. Finally, concerning the FPGA implementation, experiments were reported only with a $17,692$ 9-dimensionality dataset. Classification time is ~ 100 ms, but, as stated in the paper, the FPGA resources, expecially memory banks, were not enough to process bigger datasets.

The comparison between our work and the literature is summarized in Table 6.

5. Conclusion

In this paper, we presented different parallel implementations of the K-means algorithm for hyperspectral medical image clustering. In particular, we evaluated multi-core CPUs and many-core GPUs through the OpenMP and CUDA frameworks, respectively. Moreover, we also addressed the problem of code portability by developing OpenCL based versions. We performed experiments with a dataset made-up of *in-vivo* hyperspectral human brain images. Those experiments validated the results of all the proposed parallel implementations. Among them, CUDA achieved the better performance, outperforming OpenMP implementations. The cost of the better performance is the parallelization effort, which is significantly greater when working with CUDA. In fact, the development of the CUDA versions required the development of custom kernels and ad-hoc memory transfer management, while OpenMP only required code annotations with suitable pragmas. Code portability has also been addressed with OpenCL. However, this technology is not yet competitive with OpenMP or CUDA, achieving the worst results among the developed parallel applications. Moreover, OpenCL guarantees portability among different devices, but, for obtaining the best perfomance from a given device, it is necessary to tune the code with respect to specific hardware features. The comparison of the proposed implementations shows that the best one is based on CUDA and executed on the GTX 1060 board, achieving a maximum speed up of ~ 125x. In particular, the best CUDA version performs all the computations on the GPU without exploiting dynamic parallelism.

We also made comparisons with other recent works in the literature, that only in one case achieved results comparable but not better than ours, except for the FPGA solution proposed in [24]. However, FPGA memory constraint does not allow to process images with more than 17, 692 pixels. This limits the use of this technology and, in particular, it makes FPGAs not suitable for our target application.

Summarizing, the proposed work confirms that the GPU technology is the best solution for these class of problems, even when considering a data dimensionality bigger than the ones considered before. It also highlights that the GPU algorithm has a good scalability with respect to the number of clusters (K). Moreover, when considering high data dimensionality, the parallelization of the distance computation is not enough, since also the centroids update and the error computation can be parallelized. This ensures a suplementary speed-up. Finally, the technological evolution of GPUs offers increasing computing power at relatively low cost. In our case, a consumer GPU sold at about $200 outperforms a more expensive Tesla K40 GPU (~$5,000) of a previous generation, but optimized for scientific computations.

Future research will be focused on integrating this parallel algorithm in more complicated classification frameworks, such as the one proposed in [3,6].

Author Contributions: E.T. performed the GPU implementations, the algorithms optimizations, designed and performed experiments and wrote the manuscript. G.F. designed experiments and edited the manuscript. F.C. performed the GPU implementations and experiments. S.O., H.F. performed the serial algorithm implementation and edited the manuscript. G.M.C., M.M., F.L. supervised the project and edited the manuscript.

Funding: This work has been supported in part by the Canary Islands Government through the ACIISI (Canarian Agency for Research, Innovation and the Information Society), ITHACA project "Hyperspectral Identification of Brain Tumors" under Grant Agreement ProID2017010164 and it has been partially supported also by the Spanish Government and European Union (FEDER funds) as part of support program in the context of Distributed HW/SW Platform for Intelligent Processing of Heterogeneous Sensor Data in Large Open Areas Surveillance Applications (PLATINO) project, under contract TEC2017-86722-C4-1-R. Additionally, this work has been supported in part by the 2016 PhD Training Program for Research Staff of the University of Las Palmas de Gran Canaria.

Acknowledgments: The authors would like to thank NVIDIA Corporation for the donation of the NVIDIA Tesla K40 GPU used for this research.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Ferlay, J., Soerjomataram, I., Ervik, M., Dikshit, R., Eser, S., Mathers, C., Rebelo, M., Parkin, D. M., Forman, D., Bray, F., Cancer Incidence and Mortality Worldwide: IARC CancerBase No. 11, **2013**.
2. Louis, D. N., Perry, A., Reifenberger, G., von Deimling, A., Figarella-Branger, D. C., Webster, K., Ohgaki, H., Wiestler, O. D., Kleihues, P., Ellison, D. W., The 2016 World Health Organization Classification of Tumors of the Central Nervous System: a summary. *Acta Neuropathologica* **2016**, *131(6)*, 803-820, doi: 10.1007/s00401-016-1545-1.
3. Fabelo, H., Ortega, S., Ravi, D., Kiran, B. R., Sosa, C., Bulters, D., Callicó, G. M., Bulstrode, H., Szolna, A., Pineiro, J. F., Kabwama, S., Madronal, D., Lazcano, R., J-O'Shanahan, A., Bisshopp, S., Hernández, M., Báez, A., Yang, G.-Z., Stanciulescu, B., Salvador, R., Juárez, E., Sarmiento, R., Spatio-spectral classification of hyperspectral images for brain cancer detection during surgical operations. *PLoS One* **2018**, *13*, 1–27, doi:10.1371/journal.pone.0193721.
4. Sanai, M., Berger, M. S., Operative techniques for gliomas and the value of extent of resection. *Neurotherapeutics* **2009**, *6 (3)*, 478-486, doi: 10.1016/j.nurt.2009.04.005.
5. Fabelo, H., Ortega, S., Kabwama, S., Callicó, G. M., Bulters, D., Szolna, A., Pineiro, J. F., Sarmiento, R., HELICoiD project: a new use of hyperspectral imaging for brain cancer detection in real-time during neurosurgical operations. *Proceedings of SPIE-The International Society for Optical Engineering* **2016**, *12*, 9860 - 9860, doi: 10.1117/12.2223075.

6. Fabelo, H., Ortega, S., Lazcano, R., Madronal, D., Callicó, G. M., Juárez, E., Salvador, R., Bulters, D., Bulstrode, H., Szolna, A., Pineiro, J. F., Sosa, C., J. O'Shanahan, A., Bisshopp, S., Hernández, M., Morera, J., Ravi, D., Kiran, B. R., Vega, A., Báez-Quevedo, A., Yang, G.-Z., Stanciulescu, B., Sarmiento, R. An intraoperative visualization system using hyperspectral imaging to aid in brain tumor delineation. *Sensors* **2018**, *18*, 430, doi:10.3390/s18020430.
7. Chang C-I. Hyperspectral data processing: algorithm design and analysis. John Wiley & Sons, 2013, 978-0-471-69056-6.
8. Torti, E., Fontanella, A., Florimbi, G., Leporati, F., Fabelo, H., Ortega, S., Callicó, G. M., Acceleration of brain cancer detection algorithms during surgery procedures using GPUs. *Microprocessors and Microsystems* **2018**, *61*, 171-178, doi: 10.1016/j.micpro.2018.06.005.
9. Florimbi, G., Fabelo, H., Torti, E., Lazcano, R., Madronal, D., Ortega, S., Salvador, R., Leporati, F., Danese G., Báez-Quevedo, A., Callicó, G. M., Juárez, E., Sanz, C., Sarmiento, R., Accelerating the K-Nearest Neighbors Filtering Algorithm to Optimize the Real-Time Classification of Human Brain Tumor in Hyperspectral Images. *Sensors* **2018**, *18(7)*, 2314, doi:10.3390/s18072314.
10. Fontanella, A., Marenzi, E., Torti, E., Danese, G., Plaza, A., Leporati, F., A suite of parallel algorithms for efficient band selection from hyperspectral images. *Journal of Real-Time Image Processing* **2018**, 1-17, doi: 10.1007/s11554-018-0765-0.
11. Marenzi, E., Carrus, A., Danese, G., Leporati, F., Callicó, G. M., Efficient Parallelization of Motion Estimation for Super-Resolution. *25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing* **2017**, St. Petersburg (Russian Federation) 6-8 March 2017, 274-277, doi: 10.1109/PDP.2017.64.
12. Lopez-Fandino, J., Heras, D. B., Arguello, F., Dalla Mura, M., GPU Framework for Change Detection in Multitemporal Hyperspectral Images. *International Journal of Parallel Programming* **2017**, 1-21, doi: 10.1007/s10766-017-0547-5.
13. Florimbi, G., Torti, E., Danese, G., Leporati, F., High Performant Simulations of Cerebellar Golgi Cells Activity. *25th Euromicro International Conference on Parallel, Distributed and Network-Based Processing* **2017**, St. Petersburg (Russian Federation) 6-8 March 2017, 527-534, doi: 10.1109/PDP.2017.91.
14. Feng, X., Jin, H., Zheng, R., Zhu, L., Dai, W., Accelerating Smith-Waterman Alignment of Species-Based Protein Sequences on GPU. *International Journal of Parallel Programming* **2015**, *43 (3)*, 359-380, doi: 10.1007/s10766-013-0284-3.
15. Baramkar, P. P., Kulkarni, D. B., Review for K-Means On Graphics Processing Units (GPU). *International Journal of Engineering Research & Technology* **2014**, *3 (6)*, 1911-1914.
16. Zechner, M., Granitzer, M., K-Means on the Graphics Processor: Design and Experimental Analysis. *International Journal On Advances in Systems and Measurements* **2009**, *2 (2&3)*, 224-235, doi: 10.1016/j.jcss.2012.05.004.
17. Hong-tao, B., Li-li, H., Dan-tong, O., Zhan-shan, L., He, L., K-Means on Commodity GPUs with CUDA. *WRI World Congress on Computer Science and Information Engineering* **2009**, Los Angeles (USA), 31 March-2 April 2009, 651-655, doi: 10.1109/CSIE.2009.491.
18. Fakhi, H., Bouattane, O., Youssfi, M. , Hassan, O., New optimized GPU version of the k-means algorithm for large-sized image segmentation. *Intelligent Systems and Computer Vision* **2017**, Fez (Morocco), 17-19 April 2017, 1-6, doi: 10.1109/ISACV.2017.8054924.
19. Li, Y., Zhao, K., Chu, X., Liu, J., Speeding up k-Means algorithm by GPUs. *Journal of Computer and System Sciences* **2013**, *79(2)*, 216-229, doi: 10.1016/j.jcss.2012.05.004.
20. Baydoun, M., Dawi, M., Ghaziri, H., Enhanced parallel implementation of the K-Means clustering algorithm. *3rd International Conference on Advances in Computational Tools for Engineering Applications (ACTEA)* **2016**, Beirut (Lebanon) 13-15 July 2016, 7-11, doi: 10.1109/ACTEA.2016.7560102.
21. Saveetha, V., Sophia, S., Optimal Tabu K-Means Clustering Using Massively Parallel Architecture. *Journal of Circuits, Systems and Computers* **2018**, In press, doi: 10.1142/S0218126618501992.
22. Cuomo, S., De Angelis, V., Farina, G., Marcellino, L., Toraldo, G., A GPU-accelerated parallel K-means algorithm. *Computers & Electrical Engineering* **2017**, 1-13, doi: 10.1016/j.compeleceng.2017.12.002.
23. Lutz, C, Bress, S., Rabl, T., Zeuch, S., Markl, V., Efficient k-means on GPUs. *14th International Workshop on Data Management on New Hardware* **2018**, Huston (USA), 11 June 2018, Article No. 3, doi: 10.1145/3211922.3211925.

24. Yang, L., Chiu, S. C., Liao, W. K., Thomas, M. A., High Performance Data Clustering: A Comparative Analysis of Performance for GPU, RASC, MPI, and OpenMP Implementations. *The Journal of supercomputing* **2014**, *70(1)*, 284-300, doi: 10.1007/s11227-013-0906-y.