Accelerating Fuzzy C-Mean for Medical Image Segmentation

Project Report: Parallel Processing Systems

15 June, 2019

Ahmed Gouda
Patricia Cabanillas
Jaime Simarro
Ro'a Khaled

MSc. Medical Imaging and Applications (MAIA)
Università degli Studi di Cassino e del Lazio Meridionale

Contents

1 Abstract

In the medical field, the process of diagnosis and treatment has been empowered by the rise of medical imaging and medical image processing. Segmentation, by which we can partition different regions representing different types of tissues, is one of the most important processing steps in many medical applications. Hence, many works have been devoted to improve both the accuracy and the speed of segmentation. Fuzzy C-means is one of the most effective segmentation algorithms and thus it has been extensively used for this task. However, due to its long processing time this approach is not suitable for many purposes. In this work, we found the main reasons which slow down the algorithm; the image size and the number of clusters. In consequence, two types of improvements are proposed.

The first optimization step tackles the image size issue, by changing the algorithm itself. Instead of using the intensity value for each pixel in the image, the histogram has been used as input for the algorithm.

In the second step, we focus on solving the number of cluster problem, introducing parallel computation in the most time consuming steps of the algorithm, thus a hybrid parallel algorithm has been used.

Finally, the optimized algorithm was tested on different image sizes and different number of clusters and the obtained results show significant improvements when using number of clusters larger than 20. The execution speed of our proposed algorithm went even to a value of 8.87 times faster than the serial version in our experiments.

2 Introduction

Image segmentation is the process of dividing a digital image into a set of regions, this is an important step in many applications of computer vision [1]. Within the medical field, image segmentation is typically used to separate different regions according to their tissues. Providing essential information not only for for visualization but also for surgical planning and early disease detection.

Segmentation can be the main performance bottleneck in many applications of medical image analysis. Thus, great efforts have been invested to optimize the segmentation algorithms, modifying how it behaviours and employing different parallel programming technologies [2].

The use of Graphics Processing Unit (GPU) has emerged as a competitive solution for computing massively parallel problems. Nowadays, there are two types of parallel implementation. On one hand, the pure parallel implementation, in which the execution of the entire algorithm is only done on the GPU side. On the other hand, the hybrid implementation where sections of the algorithm are executed on the GPU while the remaining are executed on the Central Processing Unit (CPU). [3]

In this paper, a hybrid system has been developed in order to accelerate the Fuzzy C-Mean clustering algorithm.

2.1 Fuzzy C-Mean

Fuzzy C-Mean (FCM) clustering is an unsupervised technique that has been successfully applied to image segmentation, feature analysis, clustering and classifier, in a wide fields of sciences such as astronomy, geology, medical imaging and target recognition [4]. The FCM is a soft segmentation algorithm, where each point can have a certain membership to all the clusters. In other words, this algorithm come from the Fuzzy Logic where binary classification (does or does not belong to a certain cluster) is replaced by probabilities of belonging to each of the clusters. The strongest membership determines the best group for a point, in contrast with the well-know K-Means it is possible to define a minimum threshold for a point to become member of a cluster, adding the option of does not belong to any cluster. The measure of distance used by the literature is the Euclidean distance [5].

Three main steps are need to apply FCM algorithm.

The first step is to calculate the memberships for each data point by using the Equation 1.

$$\mu_{ij} = \frac{1}{\left(\sum_{L=1}^{C} \frac{|x_i - c_j|}{|x_i - c_L|}\right)^p} \tag{1}$$

$$p = \frac{2}{f-1} \tag{2}$$

where C is the number of clusters, f is the fuzziness factor and x_i is a data point. In this step each data point will have a percentage of belonging to all clusters.

Secondly, updating the centroids on each iteration applying the Equation 3.

$$c_j = \frac{\sum_{i=1}^{N} \mu_{ij}^{f} x_j}{\sum_{i=1}^{N} \mu_{ij}^{f}} \tag{3}$$

where N is the number of points and c_j is the center of cluster j.

The last step is calculate the cost function which can be calculated by the Equation 4.

$$J_f = \sum_{i=1}^{N} \sum_{j=1}^{C} \left[\mu_{ij}^{m} \left\| x_i - c_i \right\|^2\right] \tag{4}$$

In each iteration the current cost value and the previous one are compared. If the difference between these two values is less than a certain threshold value, FCM stops and outputs the segmented data. The following pseudo code simplify the FCM algorithm.

Algorithm 1 FCM steps

```
Random initialization of the centers
while $J_k - J_{k-1} < threshold$ do
    Calculate the membership matrix according to 1.
    Calculate the cluster centers vectors according to 3
    Calculate the cost according to 4
end
Output segmented data
```

3 Optimization

Fuzzy C-Means is a very powerful segmentation tool. However, this algorithm requires long computational time which makes this approach not suitable for many purposes. Optimization of FCM is a key and required factor for spreading this segmentation algorithm. We found that the image size and the number of clusters are the main factor which slow down the algorithm. Indeed, two optimization steps are proposed to speed up FCM:

3.1 Algorithm Optimization

The first optimization step tackles the problem of image size by changing the input of the FCM algorithm. We have realized that it is more efficient to calculate FCM based on the histogram intensity levels instead of the intensity in each pixel. Consequently, the data points are now smaller, corresponding to the frequency in each histogram bins (usually 256) instead of a large number of data points which represent the intensity value in each pixel of the image. This optimization step is not trivial because it reduces the order of magnitude of problem. This improvement can be done due to the fact that spacial information is not used in FCM. Therefore, any pixels with the same intensity value will have the same membership so there will be many repeated calculations.

Due to this change it is required to reformulate the Equation 3 to:

$$c_j = \frac{\sum_{i=1}^{N} \mu_{ij}^{f} x_j h_i}{\sum_{i=1}^{N} \mu_{ij}^{f}} \tag{5}$$

where h_i represents the number of pixels that have the intensity i (frequency in the value i in the histogram).

In addition, the Equation 4 needs to be change to:

$$J_f = \sum_{i=1}^{N} \sum_{j=1}^{C} \left[\mu_{ij}^{m} h_i \left\| x_i - c_i \right\|^2 \right] \tag{6}$$

Note that spite of the notable improvement of this approach, it has not been used in the literature [3].

3.2 Software Optimization

3.2.1 Computation of Histogram in parallel

The first kernel of the parallel FCM algorithm calculates the histogram of 256 histogram bins (intensity levels) for a gray 8-bit medical image. The kernel is initialized in the host-side as shown in the following steps:

- The medical image loaded on the host-side by using OpenCV library.
- Then, a global allocation is performed on the device side, with the same dimensions of the loaded medical image.
- Since the histogram bins size is not changing, it is initialized as a *constant memory* type on the GPU, in order to accelerate the computational speed.
- Next, the image is copied from the host to the device.
- In the device-side, the number of allocated 2D blocks per grid is as same as the dimensions of the medical image, and each block contains only one thread as shown in Figure 1.
- Then, the histogram kernel function *device_hist()* is executed.

As shown in the code block in Listing 1, the kernel function *"device_hist"* optimizes the histogram computation by counting the numbers of occurrence of each pixel per each thread. Since the number of pixels for each histogram bin are different, each thread requires a different time to count the number of pixel according to its bin value. Furthermore, each allocated memory value for the histogram bins is required to be incremented and overwritten by a different threads at once. Therefore, an *"atomicAdd()"* function is used in order to avoid the race condition between threads and organize the memory access for the histogram bins memory. Note that this kernel is not the bottleneck of the algorithm, therefore, we focus our optimization effort in the second kernel.

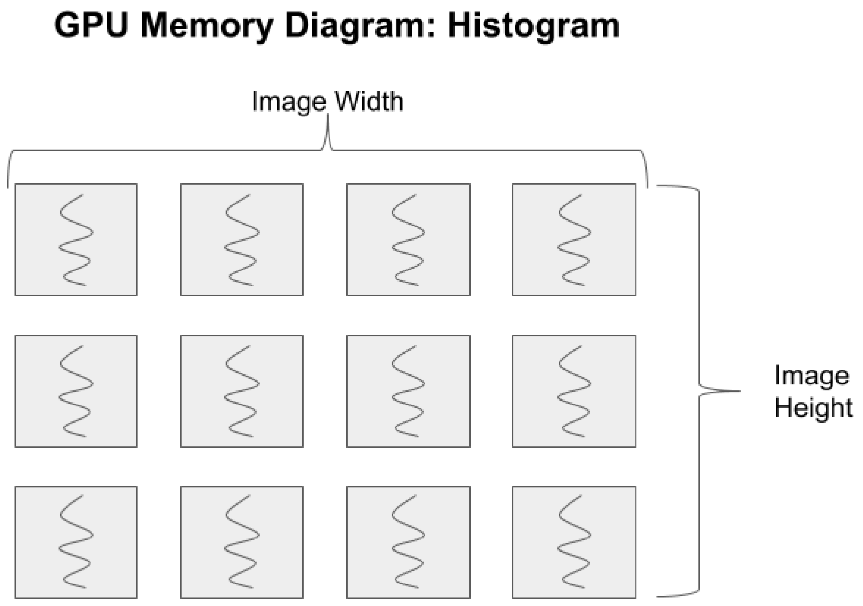

Figure 1: GPU Memory Diagram in Histogram kernel. Note that in thread represent a pixel in the input image

Listing 1: Histogram kernel

```
    __constant__ int const_hist_d[256];
    __global__ void device_hist(const int *img_d_in, const int BIN_COUNT)
    {
        int x = blockIdx.x * blockDim.x + threadIdx.x;
        int y = blockIdx.y * blockDim.y + threadIdx.y;

        int myItem = img_d_in[y * gridDim.y + x];
        int myBin = myItem % BIN_COUNT;
        atomicAdd(&(const_hist_d[myBin]), 1);
    }
```

3.2.2 Computation of Membership and Cost Function in parallel

In the second kernel, the membership between each histogram bin is calculated as is shown in Equation 1. Then, the cost function in Equation 4 is computed. Firstly, the kernel memory size and the number of blocks and threads are initialize on the host-side as shown in the following steps:

- In the first step, an array of centers point are randomly generated from 0 to the maximum value of histogram, without repetition.
- Then two dynamic memory allocated on the device side. The first memory with the dimensions of histogram bins by the number of center points. The second memory size

is one to save the cost value.

- The number of allocated blocks is one dimension that is equal to the number of histogram bins, while the number of initialized threads per block is equal to the number of center points as in Figure 2.
- In the next step the membership and cost kernel function *"device_ membership_ cost()"* is executed.

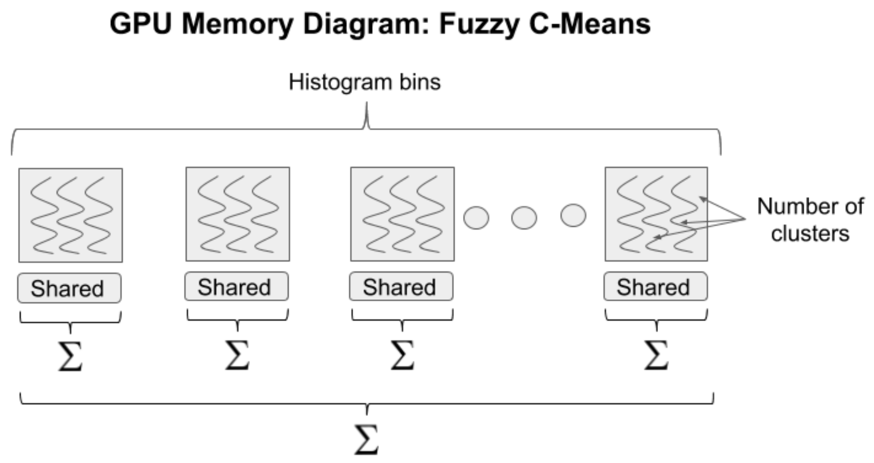

Figure 2: GPU Memory Diagram in Membership and Cost kernel. Note that each thread represent the relation between a histogram bin and a center

From code block in Listing 2, the membership for each histogram bin is firstly calculated in parallel as in Equation 1. Then, the output value of memberships is the used for computing the cost function as in Equation 4. To decrease the number of atomic addition operation, it is divided into two sub atomic addition operations (creating a Hierarchical Atomics strategy), following with the idea of Divide and Conquer. Each thread calculated the cost of relation center-histogram bin. Then, in the first layer of the additions, all the thread from the same block are added computing the cost of a histogram bin. Communication between thread of the same block is needed, consequently *shared memory* is required. Then output from the shared memory is atomically added to *constant memory* array with the same size of histogram bins, which contains the costs of each histogram bin.In the second layer of cost function calculation, the cost array of each histogram bin coming from the first sub operation are added parallel sum reduction strategy to optimize the number of sum operations from 2^n to n, where n is the bit depth of histogram bins as shown in Figure 3. From the sum reduction operation, the first location of the histogram cost array will contain the total cost value of all histogram bins.

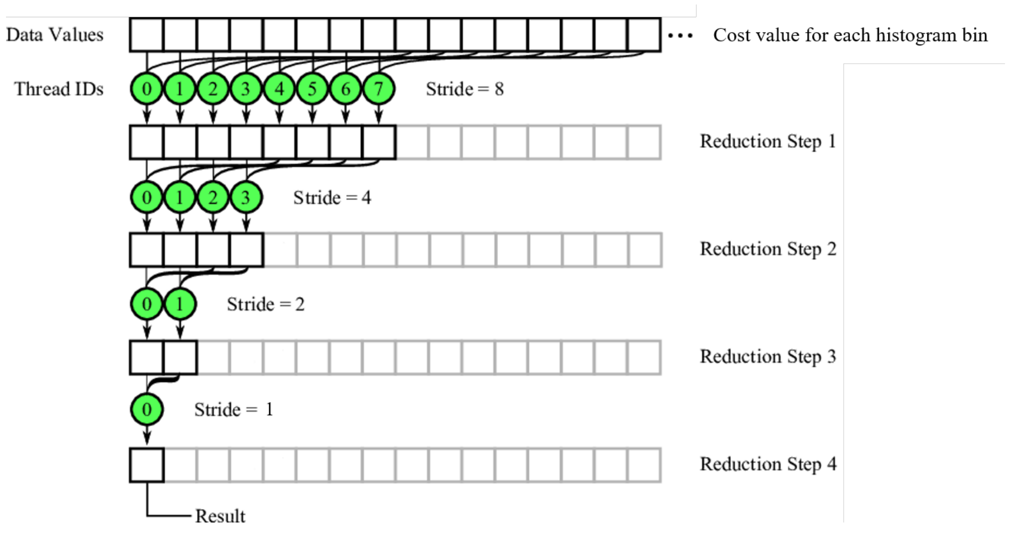

Figure 3: Sum reduction algorithm architecture (in the case of 8 bins histogram cost values)

Listing 2: Membership and cost function kernel

```
__constant__ float sum_arr[256];
__global__ void device_membership_cost(const int BIN_COUNT, int ...
    CLUSTERS_NUMBER, float * membership_d, float * cost_d, float * ...
    randcenters_d)
{
__shared__ float shared_cost_pixel;

int x = blockIdx.x; //Histogram Bins
int y = threadIdx.x; // Realtion btw center and bin
float sum = 0;
const int threath_position=x * blockDim.x  + y;
// 1- Calculate membership
float p = 2 / (FUZZINESS - 1);
float numerator = fabs(float(x - randcenters_d[y]));
    if (numerator < 0.00001)
        membership_d[threath_position]= 1;
    else {
        for (int k = 0; k < CLUSTERS_NUMBER; k++) {
            float denominator = fabs(float(x - randcenters_d[k]));
            if (denominator < 0.00001) {
                denominator = 0.000001;
            }
            float t = numerator / denominator;
            t = pow(t, p);
            sum += t;
        }
        membership_d[threath_position] = 1.0 / sum;
    }

    // 2. Calculate distance
```

```
        if (y == 0) {
            sum_arr[x] = 0;
        }
        __syncthreads();

        shared_cost_pixel = pow(x - randcenters_d[y], 2) * ...
           membership_d[x * blockDim.x + y] * const_hist_d[x];
        atomicAdd(&(sum_arr[x]), shared_cost_pixel);
        if (y == 0) {

            for (int offset = blockDim.x / 2; offset > 0; offset >≥ 1)
            {
                if (x < offset)
                {
                    // add a partial sum upstream to our own
                    sum_arr[x] += sum_arr[x + offset];
                }
                __syncthreads();
            }
            __syncthreads();
            cost_d[0] = sum_arr[0];

        }
    }
```

4 Experimental Works and Results

In this section we present our results obtained from different comparison experiments we implemented. Our goal is to observe the quality of segmentation as well as the difference in performance between both the serial and the hybrid parallel versions of the algorithm and how does it change with different sizes of images and number of clusters. Then we present our analysis of the obtained results.

4.1 Hardware Specifications

For experimental works implementation, the developed algorithms are applied on Lenovo Legion Y530 laptop with the following device-side and host-side specifications shown in the Table 1 and Table 2 .

Graphic card model	NVIDIA Geforce GTX 1050 Ti
NVIDIA CUDA Cores	768
Base Clock (MHz)	1290
Boost Clock (MHz)	1392
Memory Speed	7 Gbps
Standard Memory Config.	4 GB GDDR5
Size of a thread block (x,y,z)	(1024, 1024, 64)
Size of a grid size (x,y,z)	(2147483647, 65535, 65535)

Table 1: Device-side hardware specifications

Processor	intel Core i5-8300H (2.30GHz, up to 4.00GHz, 4 Cores, 8MB Cache)
RAM	16.0GB PC4-21300 DDR4

Table 2: Host-side hardware specifications

4.2 Segmentation with different number of clusters

In order to observe the quality of the segmentation algorithm we have run our algorithm on a brain MRI image with a tumor. As it is shown in the Figure 4 an accurate segmentation is achieved. Note the used of a minimum membership threshold is required to belong to a cluster, otherwise this pixels do not belong to any cluster (marked in the image with the color white). In contrast with the well-known K-Means, this minimum membership ensure an precise segmentation based on high relation within the members of the same cluster.

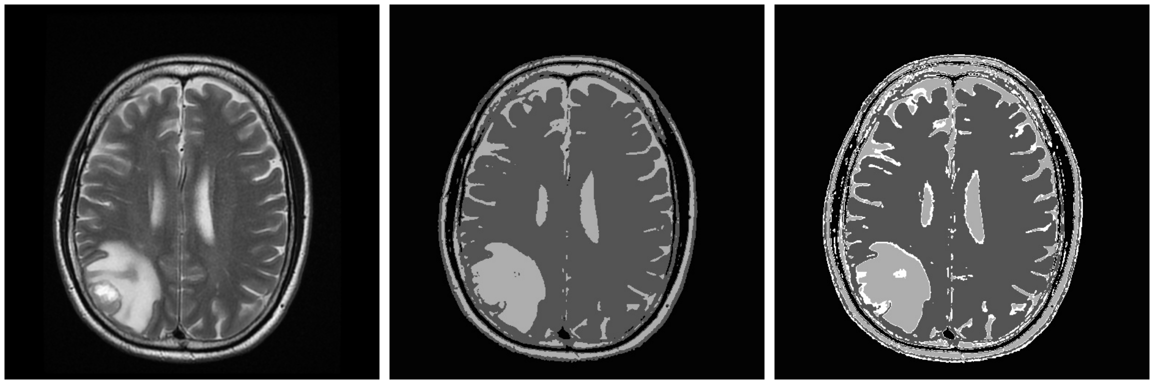

Figure 4: (a) Image before segmentation. (b) Segmented image with 2 clusters. (c) Segmented image with 3 clusters.

4.3 Serial and hybrid parallel implementations with different image sizes and number of clusters

To experiment our algorithm performance we run it on a brain MRI image and changed the size by resizing it to obtain three different sizes, hence each of the three images was segmented with both the serial and the hybrid parallel versions. For each image segmentation, we changed the number of clusters from 10-70 clusters with both the serial and the hybrid parallel versions. The three sizes of image we used are: 320X320 pixels, 640X640 pixels and 1280X1280 pixels. Figure 5 shows the performance results in terms of the execution time of these experiments.

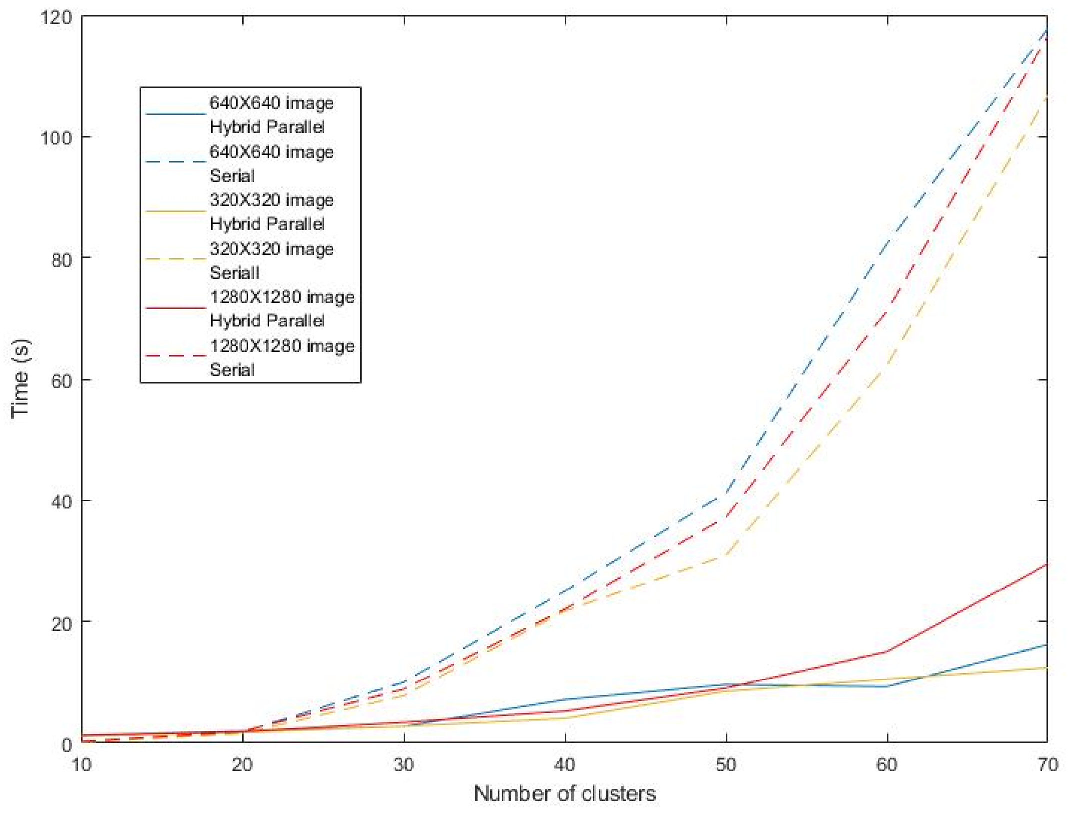

Figure 5: Performance of both serial and hybrid parallel versions with different image sizes and clusters number

4.4 Analysis

Based on the results obtained in 4.2 and 4.3 we can observe several findings.

In terms of segmentation quality, figure 4 shows that our optimized FCM algorithm provides the same segmentation quality of the original FCM algorithm. Thus it is very useful since it accelerates the performance without affecting the segmentation quality.

In terms of execution time, we can observe the following findings from Figure 5.

First, it is observed that in both implementations (serial and hybrid) when changing the image size, the performance is similar and no relatively big difference in the execution time when the image size is increased. This is due to the optimization we made in the way how FCM is implemented on images which was discussed earlier in Section 3.1. Our results prove the efficiency of this optimization technique.

Second, it is observed that at low number of clusters the employment of parallel computation did not improve the performance. In fact the serial version had a slightly faster performance. Figure 6 gives a more clear view of the performance results at lower number of clusters.

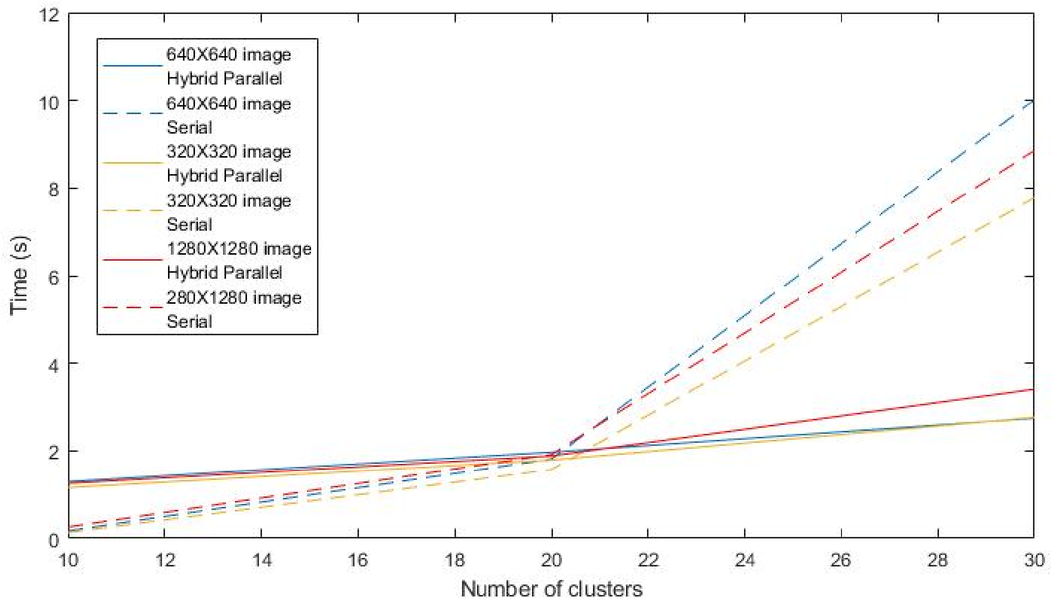

Figure 6: Performance of both serial and hybrid parallel versions with different image sizes at low number of clusters

As seen in Figure 6, using number of clusters lower than 20 (in our case) is not meaningful. This can be explained by the fact that the time needed for sending data between the CPU and GPU will be longer than just implementing the serial version since the number of clusters is relatively low and the FCM is implemented on an optimized way (on image histogram instead of image pixels pixels).

Third, it is observed that at larger number of clusters the hybrid parallel version outperforms the serial version. Figure 7 shows the improvement in performance at each number of clusters for the three image sizes.

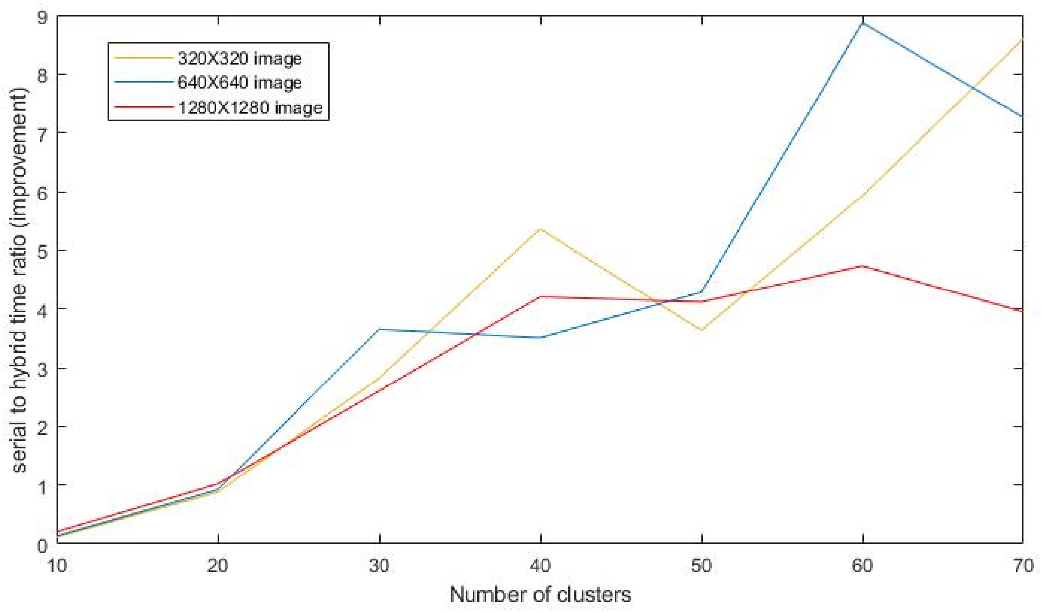

Figure 7: Improvement in performance (serial to hybrid execution time ratio) at each number of clusters for the three image sizes

These results prove that the way we are utilizing parallel programming is effective for specific applications (such as large clusters number).

5 Conclusion and Further Works

In this work we have presented an optimized method to implement medical image segmentation using FCM. Our proposed method consists of two types of optimization; one is optimizing the FCM algorithm by applying it to the intensity values instead of all pixels of the image, using the image histogram, and the other is utilizing parallel programming to execute the most crucial steps of the algorithm.

Our results show that the optimization technique of the FCM algorithm is effective and improves the performance without any limitations by the size of the image. Moreover, it has the same segmentation quality as the original FCM algorithm.

The results also prove the efficiency of the way we utilized parallel computing, especially for applications in which large number of clusters is needed.

Finally, this algorithm can be further improved to be used on 16-bit images and 3D medical images.

6 Bibliography

References

[1] Chen Ch, Pau LF, Wang PSp Handbook of pattern recognition and computer vision, vol 27. World Scientific.

[2] Murugavalli S, Rajamani V (2006) A high speed parallel fuzzy c-mean algorithm for brain tumor segmentation. BIME journal 6(1):29–33.

[3] Al-Ayyoub, M., AlZu'bi, S., Jararweh, Y., Shehab, M. A., Gupta, B. B. (2018). Accelerating 3D medical volume segmentation using GPUs. Multimedia Tools and Applications, 77(4), 4939-4958.

[4] Chuang, K. S., Tzeng, H. L., Chen, S., Wu, J., Chen, T. J. (2006). Fuzzy c-means clustering with spatial information for image segmentation. computerized medical imaging and graphics, 30(1), 9-15.

[5] Dunn JC (1973) A fuzzy relative of the ISODATA process and its use in detecting compact well-separated clusters. Taylor Francis.