EE217 Project Report

Image Segmentation using K-means algorithm

Team: Avadhesh Rathi (SID: 862253309) & Sachin Sachdeva (SID: 862189180)

Overview: The project aims at studying the execution time and performance of K-means image segmentation algorithm on GPU as well as CPU. Image segmentation is a technique to reduce the complexity of the image by dividing the image into various subgroups. K-means algorithm does a decent job in segmenting the image based on the locality of the pixels relative to the randomly assigned centroids. The centroids shift as the pixels are assigned to their respective nearest centroids. As the algorithm is highly parallelizable, it is implemented on the GPU. The results are found to be as expected- the sample images performed significantly better on GPU.

Technical Description:

The project includes implementation of the image segmentation algorithm using different libraries and languages. We use the following four ways to do the segmentation:

1) *Using CUDA C on GPU:* For this, the segmentation algorithm is divided into various sub-tasks which can all be parallelized. Each task is performed in parallel using different kernels on the GPU. The functions implemented are:
   a) setInitialCentroids(): This is used to set the initial value of centroid. After trying various methods, the random number generation worked best for the sample images used.
   b) readImage() & writeImage(): This is used to read the images in raw format as it is easy to read and perform operations on.
   c) arrayReset(): Kernel function used to reset all the arrays used in the implementation.
   d) labelsReset(): Kernel function used to reset all the labels corresponding to each pixel in the image.
   e) labelAsign(): Kernel function which calculates the distance between the current pixel and the centroids and assigns its label corresponding to the nearest centroid.
   f) totalCluster(): Kernel function to calculate the sum arrays for all the clusters in all the three channels. This information is used to calculate new centroids.
   g) newCentroids(): Kernel function to calculate new centroids based on the current iteration.
   h) main(): The main function allocates the memory on CPU and GPU, does all the memory transfers and runs the K means algorithm for a set amount of iterations.

   The inputs to this program are the input file, the output file, the height and width of the image, the value of *k* and the total number of iterations to be run for K means algorithm. The program takes in a raw image. For this, the **PIL (Python Imaging Library)** is used. The program used for conversion to and from jpg to raw is from the web. In order to

2) *Using MATLAB on CPU:* Similar to previous implementation, even this implementation uses the inbuilt function *imsegkmeans()* which takes the image and the number of clusters as input and outputs the labels and centroids. The timer used is *tic toc* which gives us the time taken for the function to execute.

3) *Using Python on CPU:* For this implementation, we use the **OpenCV library** provided by python and the function used is *cv2.kmeans* which takes number of clusters (*k*), criteria for max iterations as input and returns final centroids and the label array. The output is then used to display the results. To calculate the execution time, python's *time()* library is used. The time is started just before calling the k-means function and the values are noted down as the function completes.

4) *Using NPP library on GPU:* The **NPP library** provides a limited number of image segmentation methods. The segmentation method available is the watershed algorithm. This algorithm is also used for segmentation but there is no provision to tune the number of clusters used. The function *watershedsegmentation.cpp* already implemented in the library provided at link has been used to calculate the execution time for the sample images. Only changes made to the code are an inclusion of a timer to calculate the execution of the kernel and input the sample image used for all other implementations.

Project Status: (Complete)

All the implementations mentioned above work perfectly well except the one using NPP library because the predefined implementation with minor changes has been used. The limitation of this implementation is that it has not been generalised to custom input images.

Difficulties: Following difficulties and issues were faced while implementation of the project:

1. In the first implementation, the assigning of initial centroids was a task as there are various techniques for initial centroids assignment. After trying different methods, random selection of centroid among the pixels of the image turned to be surprisingly well performing.
2. Since, the implementation was done using CUDA C, reading of jpg images became difficult as it is not operable in that format. For this, like other programming languages, there was need to convert the image to a matrix format. Looking over the internet, the algorithm using PIL to convert jpg to raw format and vice versa served the purpose.

Evaluation and Comparisons:

The results favoured the GPU as the algorithm ran way faster than on the CPU. Two different sample images are used for the comparison. First is *colosseo.jpg* and second is *parrot.jpg*. The code to convert these images to *.raw* is put in the file. The output images for different values of k and iterations along with the execution time are shown below:

1. *Using CUDA C on GPU:*

For a total 200 iterations, the following results were observed:

colosseo.jpg	
Cluster value, *k*	Execution Time in *sec*
4	0.018
6	0.281
8	0.299

parrot.jpg	
Cluster value, *k*	Execution Time in *sec*
4	0.239
6	0.393
8	0.259

For a total of 500 iterations, the following results were observed:

colosseo.jpg	
Cluster value, *k*	Execution Time in *sec*
4	0.058
6	0.662
8	0.68

parrot.jpg	
Cluster value, *k*	Execution Time in *sec*
4	0.549
6	0.56
8	0.58

The original images, *parrot.jpg* and *colosseo.jpg* respectively are:

The images obtained are for iterations = 200 as below:

k = 4 *k = 6* *k = 8*

The running program for both the sample images:

```
bender /home/eegrad/arathi/EE217/Project $ ./kmeans colosseo.raw colosseo_out.raw 1024 768 4 500
Loading the image
Image is loadedPerforming the K means algorithm
Total time taken: 0.021387 secs.
Converged in 17 iterations.

Pixels per centroids:
0 centroid: 95179 pixels
1 centroid: 422152 pixels
2 centroid: 76068 pixels
3 centroid: 193033 pixels
New centroids:
71, 48, 0
224, 148, 0
23, 27, 0
123, 106, 0
Saving the Image
Image Saved
```

```
bender /home/eegrad/arathi/EE217/Project $ ./kmeans parrot.raw parrot_out.raw 1024 768 4 500
Loading the image
Image is loadedPerforming the K means algorithm
Total time taken: 0.549488 secs.
Converged in 499 iterations.

Pixels per centroids:
0 centroid: 38424 pixels
1 centroid: 249848 pixels
2 centroid: 72224 pixels
3 centroid: 425936 pixels
New centroids:
104, 9, 0
16, 24, 0
200, 77, 0
69, 60, 0
Saving the Image
Image Saved
```

2. *Using MATLAB on CPU:*

The execution times obtained for the two sample images for different values of *k* are:

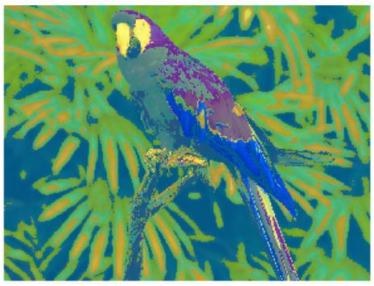
colosseo.jpg	
Cluster value, *k*	Execution Time in *sec*
4	0.561
6	1.035
8	1.53

parrot.jpg	
Cluster value, *k*	Execution Time in *sec*
4	0.425
6	0.591
8	1.06

The images are shown below:

k = 4 *k = 6* *k = 8*





Execution on MATLAB online:

```
>> kmeansmatlab
Elapsed time is 1.083248 seconds.
```

Execution on Bender server:

```
MATLAB is selecting SOFTWARE OPENGL rendering.

                          < M A T L A B (R) >
                Copyright 1984-2019 The MathWorks, Inc.
                R2019b (9.7.0.1190202) 64-bit (glnxa64)
                             August 21, 2019


To get started, type doc.
For product information, visit www.mathworks.com.

>> Image = imread('colosseo.jpg');
>>
>> tic
>> [Label,Centers] = imsegkmeans(Image,4);
>> B = labeloverlay(Image,Label);
>> toc
Elapsed time is 2.065373 seconds.
```

3. *Using Python on CPU:*

The execution times obtained for the two sample images for different values of *k* are:

colosseo.jpg	
Cluster value, *k*	Execution Time in *sec*
4	5.09
6	7.48
8	8.61

parrot.jpg	
Cluster value, *k*	Execution Time in *sec*
4	3.99
6	5.41
8	6.32

The images are shown below:



k = 4 *k = 6* *k = 8*

Execution on bender:

```
bender /home/eegrad/ssachdeva/EE217/project/ImageSegmentation_Python $ python K-means.py
('K=', 4)
('time taken for execution is=', 5.090199947357178, 'seconds')
bender /home/eegrad/ssachdeva/EE217/project/ImageSegmentation_Python $
```

```
bender /home/eegrad/ssachdeva/EE217/project/ImageSegmentation_Python $ python K-means.py
('K=', 4)
('time taken for execution is=', 3.9914000034332275, 'seconds')
bender /home/eegrad/ssachdeva/EE217/project/ImageSegmentation_Python $
```

4. *Using NPP library on GPU:*

The output file obtained from the function was in a different format of raw which could not be converted to jpg. Due to this, the output image cannot be shown here.

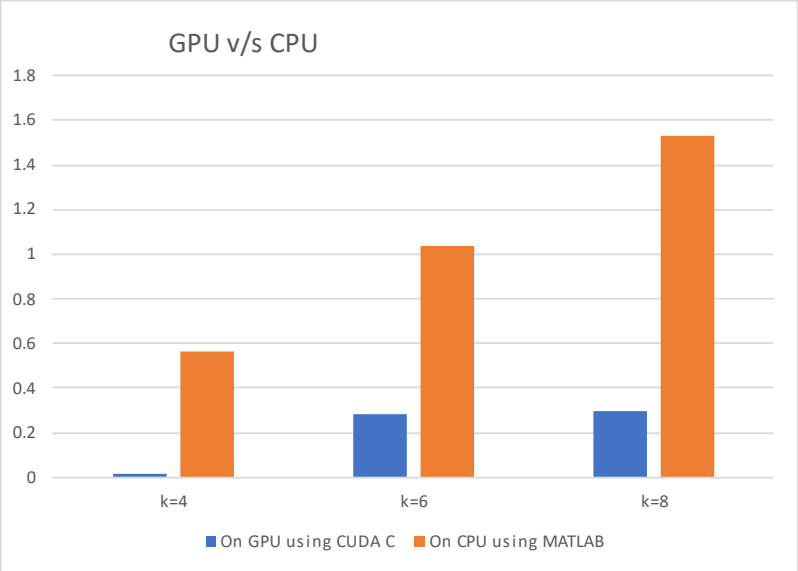
But the execution time taken by this implementation was obtained to be **0.19 seconds**

Comparison graphs for better understanding:

From the above graph, we see significant improvement in the execution time on GPU as compared to execution on CPU.

The speedup on GPU when compared to MATLAB on CPU is nearly **32 times** whereas the speedup compared to python on CPU is huge, nearly **280 times**.

Another graph below shows the comparison between the faster CPU version i.e. MATLAB with the CPU execution.



Code Compilation and run commands:

1. *Using CUDA C on GPU:*

```
nvcc -o kmeans kmeans.cu
```

(For colosseo.jpg image, k = 4 and iterations = 500)

```
./kmeans colosseo.raw colosseo_out.raw 1000 1000 4 500
```

(For parrot.jpg image, k = 4 and iterations = 500)

```
./kmeans parrot.raw parrot_out.raw 1024 768 4 500
```

Convert jpg to raw:

```
python3 convert.py colosseo.jpg
```

Convert raw to jpg:

```
python3 convert.py colosseo_out-out.raw 1000 1000
```

2. *Using MATLAB on CPU:*

```
matlab
```

Once the command line opens, file `kmeansmatlab.m`

3. *Using Python on CPU:*
   - UNZIP **final_project_Sachin.zip**
   - Go to project/ImageSegmentation_Python

- Copy the image **coloseoJPG.jpg** or **Parrot_Image_for_Segmentation.jpg** into this line of code

```
image=cv2.imread('coloseoJPG.jpg')
```

- Write the output file name into this line of code :

```
cv2.imwrite('coloseoJPG_segmented_K=8.jpg',flattened_label_reshape)
```

- Set the value of K in this line of code:

```
K = 8
```

- Then run python K-means.py

4. *Using NPP library on GPU:*
   - UNZIP **final_project_Sachin.zip**
   - Go to project/NPP_code/CUDALibrarySamples/NPP/watershedSegmentation
   - cd build
   - go to singularity by running singularity shell --nv /singularity/cs217/cs217.sif
   - cmake ..
   - make
   - ./watershedSegmentation -b 1

Task Breakdown:

TASK (Implementation)	Breakdown
Using CUDA C on GPU	Avadhesh Rathi: 100% Sachin Sachdeva: 0%
Using MATLAB on CPU	Avadhesh Rathi: 100% Sachin Sachdeva: 0%
Using python on CPU	Avadhesh Rathi: 0% Sachin Sachdeva: 100%
Using NPP library on GPU	Avadhesh Rathi: 0% Sachin Sachdeva: 100%