GPU accelerated image segmentation using k-means clustering

Akber Raza
araza008@ucr.edu

March 21, 2019

1 Introduction

Image segmentation is an important part of many computer vision problems. *k-means* clustering is one of the widely used methods in this area. It involves clustering pixels into disjoint groups so that pixels in each group are similar with respect to their intensities.

2 Algorithm

2.1 Problem definition

A set of data $\mathbf{X} \in \mathbb{R}^{n \times d}$, where n is the number of and data points and d is the size of feature space. For our problem specifically, $d = 3$ (RGB channels) and $n = image\ width \times image\ height$. Number of clusters, $k \in \mathbb{N} < n$ is a user-defined parameter. Each data point (pixel) is randomly assigned a cluster $C_j, j = 1, \ldots, k$ with centroid $c_j \in \mathbb{R}^d$. Minimum euclidean distance between centroid and data point is used for cluster assignment. Optimum set of centroids can be found by minimizing function [1]:

$$\Theta = \sum_{i=1}^{n} min_{\mathrm{c}_j \in C} \mathcal{D}(\mathrm{x}_i, \mathrm{c}_i)^2 \tag{1}$$

where $\mathcal{D}$ is euclidean distance between two input arguments. Following steps sum up the working of $k - means$ algorithm:

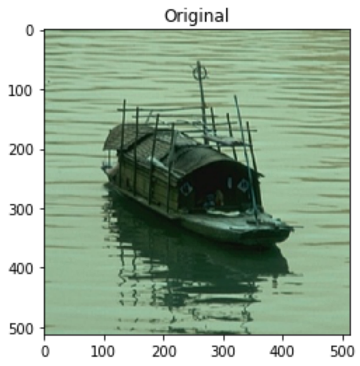
1. Initialize k random centroids, c_j
2. **Label** - assign each point, x_i, a cluster which satisfies the criterion given in (1)
3. **Update** - recalculate centroids as mean of all assigned points $\mathrm{x} \in C_j$

2.2 Program flow

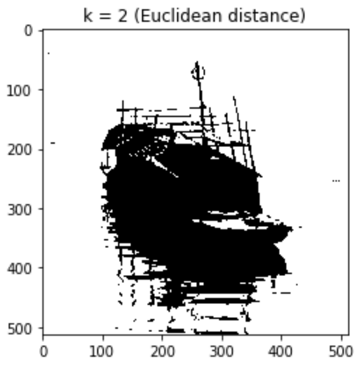
The algorithm is executed by both CPU and GPU. CPU is reponsible for fetchine the image, perform initialization and update phase of the algorithm. Labelling phase is carried out by CUDA kernel in GPU because there is no dependency in cluster assignment of pixels in the image. Both phases are iteratively repeated.

3 Results

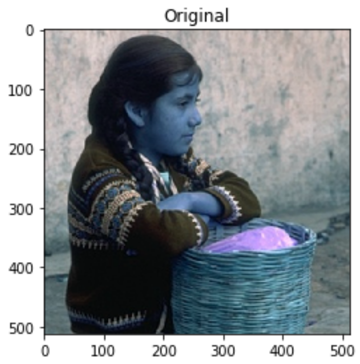
Parallelized implementation of the algorithm has been done in *python*. I have tried for a number of values of k and euclidean distance is used to evaluate similarity of pixel with each centroid. The method is repeated over 100 iterations. Fig. 1 shows the results obtained.



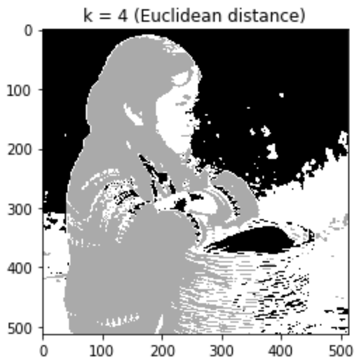
(a) Input image



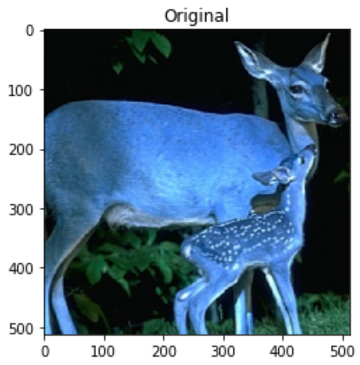
(b) Segmentated image



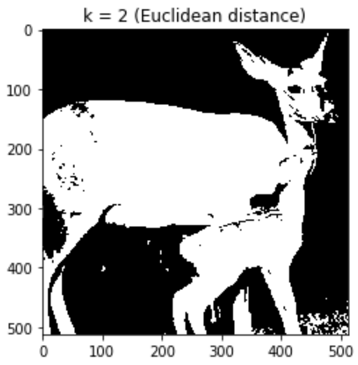
(c) Segmentated image



(d) Segmentated image



(e) Segmentated image



(f) Segmentated image

Figure 1: Original and segmented images for different values of k

Code

```
import numpy as np
from PIL import Image
import pycuda.driver as cuda
from pycuda.compiler import SourceModule

h_in = np.array(Image.open("lenna.png"),dtype=np.uint8)
h_out = np.empty(h_in.shape[:-1],dtype=np.uint8)
d_in = cuda.mem_alloc(h_in.nbytes)
d_out = cuda.mem_alloc(h_out.nbytes)
cuda.memcpy_htod(d_in, h_in)

k = np.int32(10)
BLOCK_SIZE = 16
height = np.int32(h_in.shape[0])
width = np.int32(h_in.shape[1])
means = np.random.randint(low=0,high=256,
                          size=(k,3),dtype = np.uint8)

mod = SourceModule("""
    __constant__ unsigned char centroids[48];
    __global__ void rgb2grey(const unsigned char *in,
            unsigned char *out, int height, int width, int k){
        int y = threadIdx.y + blockIdx.y * blockDim.y;
        int x = threadIdx.x + blockIdx.x * blockDim.x;
        if (x < width && y < height){
            int idx = (width*y+x)*3;
            float d[16];
            for (int h=0; h<k; h++){
                d[h]=0;
                for (int i=0; i<3; i++){
                    float in_f = in[idx+i];
                    float c_f = centroids[3*h+i];
                    d[h] += (in_f-c_f)*(in_f-c_f);
                 }
            }
            float min = d[0];
            int loc = 0;
            for (int c=1; c<k; c++){
                if (d[c] < min){
                    min = d[c];
                    loc = c;
                }
            }
            out[width*y+x] = loc;
        }
    }
    """)
func = mod.get_function("rgb2grey")
cen = mod.get_global("centroids")[0]
for i in range(100):
    cuda.memcpy_htod(cen, means)
    func(d_in, d_out, height, width,k,
         grid=((height-1)/BLOCK_SIZE+1,(width-1)/BLOCK_SIZE+1,1),
              block=(BLOCK_SIZE,BLOCK_SIZE,1))
    cuda.memcpy_dtoh(h_out, d_out)
    for j in range(k):
        loc = np.array(np.where(h_out==j)).T
        means[j,:] = np.mean(h_in[loc[:,0],loc[:,1],:],axis=0)
im = Image.fromarray(h_out,'L')
```

4 References

[1] Mario Zechner, Michael Granitzer: “Accelerating K-Means on the Graphics Processor via CUDA”, First International Conference on Intensive Applications and Services, 2009