CS 214 Extra Credit 2:
Advanced C Kernel (100 points)

The goal of this assignment is to continue becoming familiar with C kernels as well as perform a small performance analysis study of your kernel. Recall from assignment 2 that a kernel is a function or library you call from a higher level language, such Python or Java. Your task is to write a C kernel that is equivalent to a Python function and measure the performance differences between them. In addition to handing in programming code, you will hand in a write up that documents the performance difference between the C kernel and the Python version.

The Python code, minus the function you must fill-in, and two test cases are included in the assignment. See the folder at the link below:

https://rutgers.instructure.com/courses/329001/files/folder/Extra%20Credit%202%20Files

Images using Colormaps

Recall that uncompressed color information in images requires 3 bytes per pixel, either in RGB or YCbCr format. One method of compression is to convert from RGB to YCbCr and reduce the number of bits in the Chroma (CbCr) components.

An alternative method of compression is to use a color map. In this approach, each pixel is reduced in bits and does not encode color information, but rather is an index to a table. The table, or colormap, contains the actual RGB color. For example, an 8 bit color map would encode each pixel in 8 bits (one byte) giving up to 256 possible 24-bit colors for an image. A 16 bit color map would index the table using 16 bit integers, thus allowing for 2^16 or 65536 possible colors.

An image file would contain both the pixel information and the color map, as both are needed to reproduce the image.

Using K-means clustering to reduce the color Space from 24 bits/pixel to 8 bits/pixel.

Given an image array encoded as a flattened 1D array of 24 bit pixels we need to reduce the number of colors from a possible 2^24 while maintaining image quality. K-means clustering is one strategy for reducing the number of possible colors in an image.

For images, K-means clustering begins with K means (or centroids); each mean corresponds to color in the color map. Thus, for an 8-bit colormap, there will be 256 means/centroids. Each mean is a position in the colorspace; we can think of the color space as a 3 dimensional space with axis in the red, green and blue components. The 256 means are points in the 3D colorspace which hopefully capture “clusters” of many points’ colors with a similar color. Because each color in the colormap is “in the middle” of a set of colors in the original image, we call these the values in the colormap centroids of the color space.

The K-means algorithm begins by selecting 256 pixels at random to select the starting means, which is a 24 bit color. The algorithm runs for a number of iterations adjusting the means as follows:

For each point in the image, find the color in the colormap (current set of K-means) which is closest in the colorspace. Typically, euclidean distance is the metric used (i.e. sqrt((red1-red2)^2 + (green1-green2) + (blue1-blue2)^2). Recall this is the distance in the 3D colorspace, not the 2D image plane. Note that as an optimization you could remove the sqrt() function as taking the square root does not change the ordering of distances.

Next in the iteration, for each color (centroid) in the colormap, compute the new mean by finding the mean of the pixels assigned to that color. The mean per-dimension in the color space, so there will be a separate mean for red, green and blue components. That is, for each color, sum the R,G,B components individually and divide by the number of pixels assigned to that color.

The number of iterations is a parameter to the program. You can find more description of using K-means clustering for colormap reduction at this link:

C Kernel:

You will write your kernel in a separate file called "kernel.c". It must have a function with the following signature:

```
void kmeans_clustering(float* pixels, int num_pixels, int
num_centroids, int max_iters, int seed, float* centroids, int* labels)
```

The inputs are:

```
    float* pixels: The input array of pixels, each pixel is a 3 byte
RGB value
  int num_pixels: the number of pixels in the above array
  int num_centroids: the number of centroids, which is the same as the
number of pixels in the colormap.
  int max_iters: maximum number of iterations for the K-means
algorithm
  int seed: see to use for the srand() function. Use srand()/rand()
  float* centroids: array of 3 byte centroids.
  int* labels: The color in the map for each pixel - one byte for each
pixel
```

To create a shared object (.so) for your C code, you will need to use the flags "-shared". You should also compile with the highest level of optimization, so add the -O3 flag to the compile line in your makefile as well. You may need to include the math library in your code to get the sqrt() function to work.

Example Usage

To take as input an image file called “input.png” and output the Python and C versions of the reduced colormap images and colormaps:

```
python3 24Bit.py input.png output_image_Py.png output_image_C.png
colormap_Py.txt colormap_C.txt --seed 214 --num_centroids 256
--max_iters 3
```

Submission Requirements:

If you are working in a group, you **must submit as a group in Gradescope**. See this link for how to submit as a group:
https://help.gradescope.com/article/m5qz2xsnjy-student-add-group-members

Your submission must include 3 components: (1) Code, (2) Images, and (3) a Write-up. These should all be in the same directory as a zip file.

Code (60 points):

Submit your C source code in a single `` `kernel.c` `` file along with a makefile called `` `makefile` `` that compiles `kernel.c` to a shared object (.so file) called `` `libkernel.so` ``. The C code must include a function `kmeans_clustering()` as described above, which is callable from Python using the **You will receive a 0 if these instructions are not followed.**

You must also include the python3 file called 24Bit.py which is provided.

Images (10 points):

Include 5 images of varying color complexity in png or jpg formats where the maximum image dimensions are 1920x1080. The images can range from cartoons with a few colors, to landscapes, to synthetic datasets that maximize the color usage. Include at least 1 low-color variance image (<300 unique colors) and 1 high variance image (> 128K colors). The grade depends on getting images with a range of color pallets.

Write Up (30 points):

For the writeup, you will document the performance of Python program vs your C kernel on 5 images. Run the code with a seed of 214 and for 3 iterations.

Your writeup must be in a PDF file called “writeup.pdf” and must contain these sections. See the attachment for an example writeup file.

1. Title and Date

2. Name of the group’s members and their netIDs.

3. Table 1: Images Used. Show the 5 images, their names, and a side-by-side comparison of the original image and the one with an 8-bit color map generated by the C kernel.

4. Computer Specification you ran the test on:
    a. Processor (the “model name” in /proc/cpuinfo in linux)
    b. Memory: (the memory total field in /proc/meminfo in linux)
    c. Operating System (the operating system in the hostnamectl command in linux)

    If you use Windows or MacOS use equivalent values for the above.

5. Table 2: Language Performance: Include the image sizes, Python and C times with the following columns: Image name, width, height, total pixels, seed, iterations, time in Python (sec), time in C (sec)

6. The average processing time per pixel, averaged over the entire dataset, in Python and in C, expressed in microseconds.

    Note: for background on comparing computer systems’ performance, see “Notes on Calculating Computer Performance” by Bruce Jacob and Trevor Mudge, UMich Tech report CSE-TR-231-95: https://tnm.engin.umich.edu/wp-content/uploads/sites/353/2021/06/1995_Notes_on_calculating_computer_performance.pdf