EEL3090: Embedded Systems Project

On

Implementation of Image Compression in C++ for Resource constrained environment

by

Team Members:

Priyam Gaurav (B21EE092)
Rahul Gurjar (B21EE055)

➢Abstract:

In this project, we have implemented an image compression algorithm in C++ which is tailored for a resource constrained environment. Lossy image compression methods like jpeg,heif focus on reducing the image size by discarding non critical information, getting size reduction at the cost of image quality. In case of a lossless image compression process, we reduce the size without sacrificing the quality of the image. In this project, we use the K-means method which compresses the image image by partitioning the image, and representing the image using a reduced color set. Implementation of image compression in C++, provides insight into the code execution and compares the results with JPEG compression.

➢Theory:

Image compression is a process in which we reduce the size of an image. We can compress an image either by removing bytes of information from the image or by using an image compression algorithm to reconstruct the image in such a way that it takes much less space.
Image compression is a vital part of our digital world, having a huge impact on different fields. For example it leads to improved web performance as compressed images load faster than uncompressed ones, take up less storage space, and we can transmit it faster. It plays a crucial role in many resource constrained devices, like smartphones, IoT devices, embedded systems as they have limited processing power, memory and storage.

Image compression can be divided into two major categories -

1. Lossy Image Compression
2. Lossless Image Compression

Lossy Image compression -

In lossy image compression, we only retain the most important information about the image, it removes all the redundant data from the image. Lossy image compression provides a significant reduction in the image size, but it also reduces the image quality, to the point of distortion if overcompressed.
Quality is maintained when compression is applied carefully, there is a tradeoff between image quality and size. But there are certain challenges associated with it, one of which is that it's irreversible. Once it has been applied to an image it can never be restored back to its original state.

There are many methods used for applying lossy image compression-

1. JPEG
2. HEIF

Let us take a look at how exactly our lossy compression works, taking example of the jpeg compression, in jpeg we make use of the fact that the human eye is more sensitive to brightness rather than colors. So jpeg compression converts the image color space from RGB to YCbCr. Here Y contains brightness information and Cb and Cr are chrominance blue and red channels respectively.Taking example of the jpeg compression, in jpeg we make use of the fact that the human eye is more sensitive to brightness rather than colors.

After that we downsample, we will separate the color information from the brightness information and downsample it by averaging the 4 pixels into a single color value for 4pixels. After that we further divide the samples into sizes of 8*8 pixels blocks and from here each box is processed independently. We then subtract each pixel value by 128 to make it in range -128 to +127.

Using our Discrete Cosine Transform we multiply it to our blocks of 64 pixels, doing this we obtain the weight matrix for each image. These values exactly show how much important information is stored in each block.

After this we quantize our weight matrices to remove the high frequency elements in the image, as our eye is not very good at perceiving those. So it reduces the size of the image by reducing the image quality much. Now we further encode the date to reduce the size without losing any information, this is achieved using RLE and Huffman encoding. This is achieved by multiplying the quantized weight matrix by a zigzag matrix,we get a list of integer values on which then RLE is applied to count the number and their frequency. In the end we apply huffman coding to encode the data, no data is lost during this process.

Now to decompose the image, we simply repeat the above steps in reverse order. As you can see, there are some steps where we discarded some information which was redundant, so it makes it a lossy compression technique.

Lossless Image compression -

In case of lossless image compression, we reconstruct the image and reduce its size without removing critical data or reducing the image quality and it results in a compressed image, which can be restored back to its original state.

It does not offer the same compression as a lossy image compression technique, it is generally used in places where image quality is prioritized over space or storage.

In case of lossless image compression,our image quality is almost similar to the original image.

There are many methods used for applying lossless image compression-

1. PNG
2. GIF

Let us take a look at how lossless image compression works, for this let us look at an example of how PNG works.
It basically works in two stages, prediction i.e filtering and compression.
In the case of filtering, we use the help of delta encoding, in which basically you can represent any value as different from its previous value, making the data linearly correlated. Using this transforms our dataset into very small values which are repetitive. Using this concept PNG, we basically filter all the color space values separately i.e for a single scan, it is applied separately for the red colors values, then blue and then green. We then add the values and compare the sum for each filter and choose the filter which gives the smallest value. Like this the whole image is scanned and filtered.
The second part is the compression method, once our filtering is done it is passed on to the LZ77 algorithm. It is a coding algorithm, which is similar to Huffman Coding.

Now using the above two concepts that we have discussed just now, we have applied an algorithm which does image compression using these concepts.

K Means Technique-

The K Means algorithm is basically an unsupervised algorithm, which will partition a dataset into k distinct clusters. We can initialize this algorithm in several steps. The first step is basically deciding on how many clusters we want for our dataset, which we refer to as k. We will randomly sample k different points from our data, and mark them as centroid.
In the second step, we basically assign data points to our cluster, the values closest to the centroid as assigned as part of that cluster. To calculate the closeness of the points we can use the metric of Euclidean distance.
In the third step we will update our centroids, by calculating the mean of each cluster and updating their centroid as that value. We will repeat this process until we reach a convergence point. The convergence will occur basically when our centroids do not change any further.
Now we can use this concept of k means for our image compression, by selecting k colors to represent the entire image. This helps in only utilizing the k colors, instead of the whole RGB sequence.
A color pixel is normally represented by its RGB value, which ranges from 0 to 255, as there are three colors so it will take 256^3 values to represent each color i.e 24 bits of storage for a single color. Using K means we can define our value, let

us say we only choose 64 colors to represent our image. So we can represent each color in log2(64)= 6 bits. Hence compressing our data for an overall image. We can see that it is incorporating both a lossy compression and some parts of lossless compression technique, as if we choose a k high enough, we can get the same quality as our original image but the compression will be less. We can also choose a lower value of k, which can basically increase the compression of our image as it is taking less no. of bits to represent a pixel, which can overall decrease the size of the image file.

➢Code Explanation and Analysis:

```
int main(int argc, char **argv)
{

    string inputImage = "./image.png";
    numberofcolors = 20;
    cout<<"Enter the input image address: ";
    cin>>inputImage;
    cout<<"Enter the Number of colors you want to take in image: ";
    cin>>numberofcolors;
    string outputImage ="compressed_image.jpg";
```

```
int steps=61;
while(steps<20 || steps==61){
    cout<<"Enter the number of steps you want to train the model: ";
    cin>>steps;
    if(steps==61){
        break;
    }
    else if(steps<10){
        cout<<"Note: Steps Entered must be higher for better quality of image"<<endl;
        cout<<"Re-enter the number of steps you want to train the model: ";
    }
}
Mat image1 = imread(inputImage);
```

Here the input for the image compression algorithm is taken where we start with assigning the image address of the coming image to a certain default address of some image in our directory. Also the initial values of other parameters in our algorithm are set. We also take input for the image address and the required parameters to make it convenient for the user to adjust these parameters and get the compressed image of desired quality and compression. Then we have put some constraint on the number of steps of the training model which is these steps must be at least 20. Then we read the image using the opencv function imread from the mentioned location.

```
//jpeg compression for comparison
vector<uchar> buffer;
imencode(".jpg", image1, buffer,{IMWRITE_JPEG_QUALITY, 60});
imwrite("jpeg_compressed.jpg", image1);
```

Here we have used the in-built jpeg based image compression model for comparison with our model but the jpeg based image compression model is lossless image compression model and our model is lossy type of image compression algorithm.

```
//our model compression
image = image1;
if (image1.empty()){
    cout<<"incorrect image address"<<endl;
    return -1;
}
else{
    cout<<"image uploaded successfully"<<endl;
}
int img_rows = image.rows;
int img_cols = image.cols;
labels = Mat::zeros(Size(img_cols, img_rows), CV_8UC1);

for (int i = 0; i < numberofcolors; i++){
    int randomR = random(img_rows-1);
    int randomC = random(img_cols-1);
    Vec3b img_pixel = image.at<Vec3b>(randomR, randomC);
    clusterCentres.push_back(Pixel(img_pixel[0], img_pixel[1], img_pixel[2]));
}
```

```
train_model(steps);

for(int i=0;i<img_rows;i++){
    for(int j=0;j<img_cols;j++){
        Pixel img_pixel = clusterCentres.at(labels.at<uchar>(i, j));
        Vec3b cvt_pixel(img_pixel.b, img_pixel.g, img_pixel.r);
        image.at<Vec3b>(i, j) = cvt_pixel;
    }
}
```

Our image compression model is based on the k-means clustering algorithm in machine learning. Here we firstly assigned the centroids as some pixels from the image and then stored them. Then using these random initial values we train our model and then we restore the image after performing the compression. For this we use the cluster centers and create the pixels back and form the new image.

```
//writing our compressed image
imwrite(outputImage, image);

//displaying our compressed image
imshow("Display Our Model Image", image);
waitKey(0);
return 0;
```

Finally we write the compressed image to the file and also display it when the compression is over. Hence here as soon as the compression is over, the compressed image will show up.

```
void train_model(int it){
    cout<<"Image Compression Started"<<endl;
    newClusterCenters();
    for(int i=0;i<it;i++){
        findCentroids();
        newClusterCenters();
        cout<<"Working on Compression Step: "<<i+1<<endl;
    }
}
```

This function runs for “it”number of steps making the cluster more finer with each next evaluation. It does this by firstly assigning new cluster centers in place of our randomly assigned centers and then in the next “it” steps it finds the centroids for the data points and then again finds the updated cluster centers based on centroids assigned.

```
void newClusterCenters()
{
    int rw=image.rows;
    int cl=image.cols;
    for(int i=0;i<rw;i++){
        for(int j=0;j<cl;j++){
            Vec3b img_pixel = image.at<Vec3b>(i,j);
            int centroidLabel = 0;
            int b1=img_pixel[0];
            int g1=img_pixel[1];
            int r1=img_pixel[2];
            double mindistance=DBL_MAX;
            for(int t=0;t<numberofcolors;t++){
                int val1=clusterCentres[t].b;
```

```
                int val2=clusterCentres[t].g;
                int val3=clusterCentres[t].r;
                double distance=euclideanDistance(val1,val2,val3,b1,g1,r1);
                if(distance<mindistance){
                    mindistance=distance;
                    centroidLabel=t;
                    labels.at<uchar>(i,j)=(uchar)centroidLabel;
                }
            }
        }
    }

}
```

This function assigns new cluster centers to the data points. Here we go to each and every pixel by iteration along the rows and columns. Then in each pixel we check for all cluster centers and find the cluster center with minimum distance from this point and assign the new cluster center label to this point in the image with minimum distance.

```
void findCentroids(){
    int rw=image.rows;
    int cl=image.cols;

    for(int i=0;i<numberofcolors;i++){
        double val1=0;
        double val2=0;
        double val3=0;
        int it=0;
        for(int j=0;j<rw;j++){
            for(int k=0;k<cl;k++){
                int temp = labels.at<uchar>(j,k);
                if(temp==i){
                    Vec3b img_pixel = image.at<Vec3b>(j,k);
                    val1+=img_pixel[0];
                    val2+=img_pixel[1];
                    val3+=img_pixel[2];
                    it++;
                }
            }
        }
        clusterCentres.at(i) = Pixel(val1/it,val2/it,val3/it);
    }
}
```

Now based on the labels found using the previous function the new cluster centers are found. Now to do this we go for iteration over the number of clusters and then in this we search for every pixel whether the pixel belongs to that cluster or not. If the pixel has a centroid as that cluster then these values for all such pixels are taken and averaged to find the new cluster center. The old cluster center is then replaced by the newer ones.

```
static double euclideanDistance(int x1, int y1, int c1, int x2, int y2, int c2)
{
    int tem1=pow(x1 -x2,2);
    int tem2=pow(y1 -y2,2);
    int tem3=pow(c1 -c2,2);
    return sqrt(tem1+tem2+tem3);
}
```

This function is responsible for calculating the euclidean distance between the two points which is used to calculate the distance of centroid from the pixel points. This is based on a simple euclidean formula.

```
static int random(int lim)
{
    uniform_int_distribution<int> uid{0, lim}; // uid is a uniform random no generator
    default_random_engine dre(chrono::steady_clock::now().time_since_epoch().count());
    int ret = uid(dre); // returns a random no between 0 and lim
    return ret;
}
```

This function is used to generate a random number between 0 and the assigned limit. The generated random number is from a uniform distribution and then the dre is used to get the random number. This random number is returned after generation from this function.

```
#ifndef PIXEL_H
#define PIXEL_H

struct Pixel {
    unsigned char b, g, r;

    Pixel(unsigned char b, unsigned char g, unsigned char r)
        : b(b), g(g), r(r) {}
};

#endif
```

This is in our header file where we defined the struct for the Pixel we are using. Here in this we have created three character as r, g and b which are used to represent the color of the pixel.

Other than these two files the project contains the readme file, images, makefiles and other files associated with the libraries used.

```
rahul@rahul-ASUS-TUF-Gaming-F15-FX506LH-FX506LH:~/test$ ./DisplayImage
Enter the input image address: ./image.png
Enter the Number of colors you want to take in image: 80
Enter the number of steps you want to train the model: 30
image uploaded successfully
Image Compression Started
```

These are test inputs for our program to run but the program needs to be built using the make command before running.

➢Results and Output Analysis:

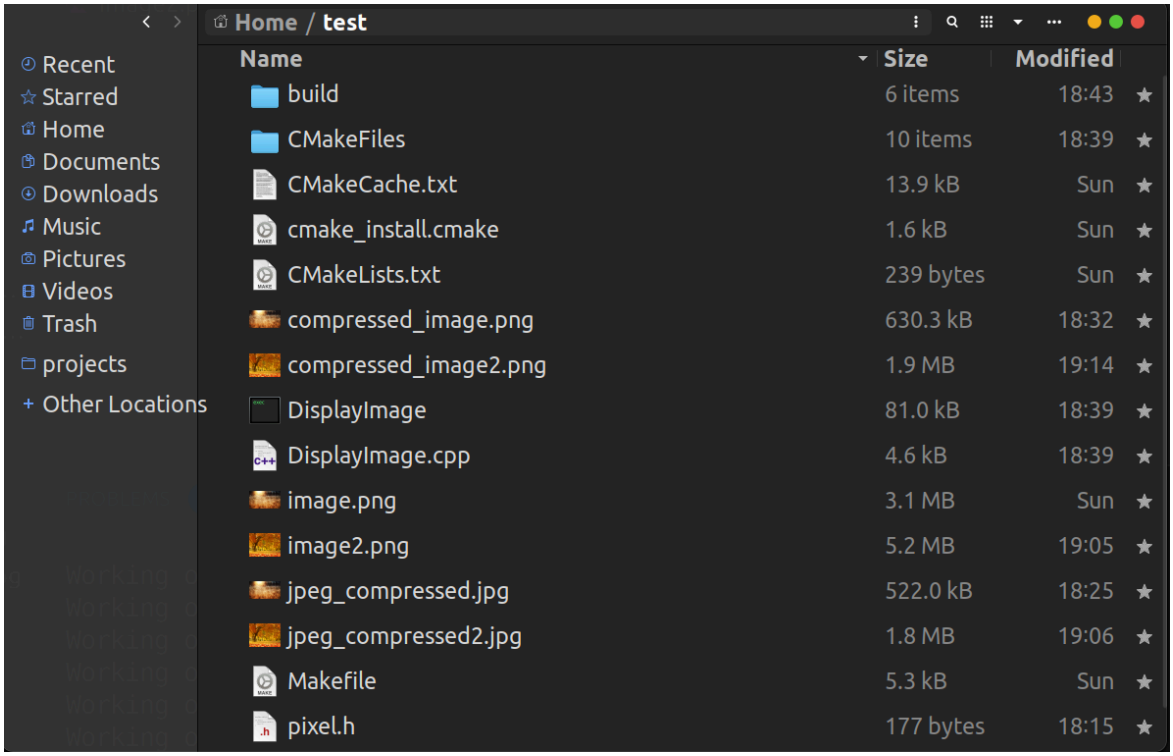
Terminal Inputs and Output :

```
rahul@rahul-ASUS-TUF-Gaming-F15-FX506LH-FX506LH:~/test$ ./DisplayImage
Enter the input image address: ./image.png
Enter the Number of colors you want to take in image: 80
Enter the number of steps you want to train the model: 30
image uploaded successfully
Image Compression Started
Working on Compression Step: 1
Working on Compression Step: 2
Working on Compression Step: 3
Working on Compression Step: 4
Working on Compression Step: 5
Working on Compression Step: 6
Working on Compression Step: 7
Working on Compression Step: 8
```

Our Model Compressed Image:

JPEG Compressed Image:

File size analysis:



S.No.	Original Image Size	Our Model Compressed Size	JPEG Compressed Size
1.	3.1MB	630.2kB	522.0kB
2.	5.2MB	1.9MB	1.8MB

➢ Conclusion:

Our implementation of the K means image compression, through experimental analysis we can see that our algorithm is in fact compressing the images. It reduces the image size of the first image 3.1MB to only 630.2KB, reducing its size to only 20%, for our second test we used another image of size 5.2MB, which was compressed and reduced to size 1.9MB offering us a compression of 36.53%.

➢ References:

1.https://www.geeksforgeeks.org/image-compression-using-k-means-clustering/
2.https://github.com/fengyurenpingsheng/Asymmetric-Learned-Image-Compression-with-MRSB-IM-and-PQF
3.https://github.com/Aftaab99/ImageColorQuantization/blob/master/KMeansCompressv2.cc
4.https://www.geeksforgeeks.org/process-of-jpeg-data-compression/
5.https://compress-or-die.com/Understanding-PNG