PC-2022/23 Sequential and parallel Histogram Equalization of a color image

Alessandro Petricci
7017385
alessandro.petricci@stud.unifi.it

Abstract

Histogram Equalization is an image processing technique to adjust the contast of an image using the image's pixel values histogram. This work focuses on comparing a sequential implementation of this technique in C++ and two parallel implementations, one again in C++ using the OpenMP library, and one in Python, using the PyCuda library.

Future Distribution Permission

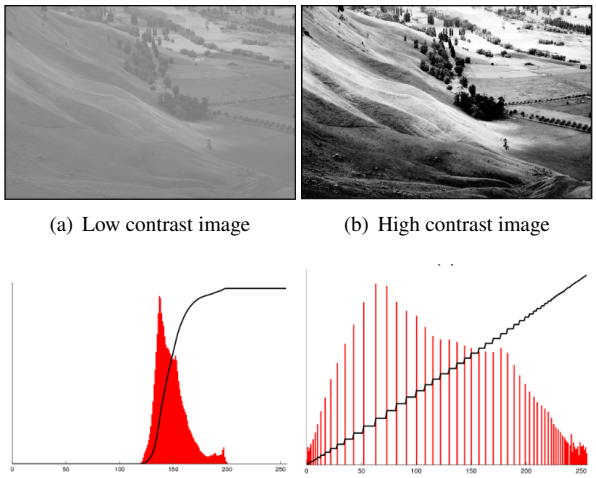
The author(s) of this report give permission for this document to be distributed to Unifi-affiliated students taking future courses.

1. Introduction

The histogram of an image is a graphical representation of the distribution of the intensity pixel of an image. In simpler terms, it's a count of how many pixel in the image have a given value. The Histogram Equalization technique is used in image processing to improve the contrast in photos and images. The underlying idea is that low contrast images tend to have histograms where values are not spread out evenly but tend to group somewhere: "stretching" the histogram helps in increasing the difference beetween the lowest and the highest value of luminance in the image, with a direct effect on the contrast.

1.1. Grayscale images

A grayscale image is a 1-channel image where pixels have values between [0,255]. The histogram of a grayscale image is therefore a function $h(p) = n$ where p is the pixel value and n is the count of the pixels with said value. As said before and shown on 1, a low contrast image will have an histogram with packed values, while the process of Histogram Equalization aims to spread out the histogram to increase the contrast. Given a grayscale image, the function to equalize its histogram is defined as:



(a) Low contrast image (b) High contrast image

(c) Low contrast histogram (d) Equalized histogram

Figure 1.

$$h_{eq}(p) = round((L-1) \times \frac{cdf(p) - cdf_{min}}{M \times N - cdf_{min}})$$

where $cdf(p)$ is the cumulative distribution function of the original histogram, M is the width of the image, N is the height of the image and cdf_{min} is the $argmin\{cdf(p)|cdf(p) \neq 0\}$. Knowing this it's possible to map old pixel values into new pixel values with:

$$p_{new} = h_{eq}(p_{old})$$

1.2. Color images

Color images, differently from grayscale ones, have 3 different channels to represent a pixel, usually in the RGB format. This means that every pixel is now a triplet of three integers. It's evident that the previous formula can't be applied as is, since even if we decided to split the 3 channels and separately apply an histogram equalization on them we would obtain a heavily distorted image in the color space. To make up for this it's possible though to convert the RGB image to a better suited color space. The color space used in these scenarios is the YUV space that, just as the RGB color space, is composed of 3 channels. The propriety of this color model is that there is a distinct channel with brightness information (luminance), Y, and 2 channels with color information (chrominance), U and V. Therefore to equalize the histogram of a color image we apply these formulas to convert the image to the YUV color space.

$$\begin{cases} Y = 0.299 * R + 0.587 * G + 0.114 * B \\ U = -0.169 * R - 0.331 * G + 0.499 * B + 128 \\ V = 0.499 * R - 0.418 * G - 0.0813 * B + 128 \end{cases}$$

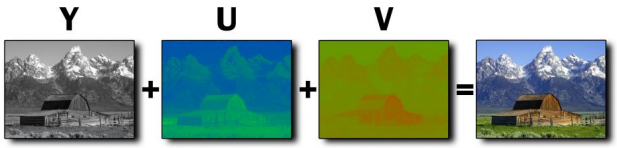


Figure 2.

2 shows the results.
After the color conversion it's possible to apply the formula seen before on the sole Y channel, which is the only one with light information, and then convert the image back to the RGB color space with the following formulas:

$$\begin{cases} R = max(0, min(255, (Y + 1.402 * (V - 128)))) \\ G = max(0, min(255, (Y - 0.344 * (U - 128) \\ \qquad -0.714 * (V - 128)))) \\ B = max(0, min(255, (Y + 1.772 * (U - 128)))) \end{cases}$$

(a) Low contrast image

(b) Equalized image

Figure 3.

The results of the equalization can be seen in 3.

2. Implementation

2.1. Sequential version

The sequential version of the program is written in C++. It can be functionally decomposed in 3 parts. The first part of code is 4. A double for loop is used to cycle over every element of $image$ and for every pixel a conversion to YUV space is applied and the values are saved to yuv. Additionally, this piece of code is responsible for creating the histogram of Y values, adding one to the proper bin on every cycle.
Subsequently, the code in 5 calculates the cumulative distribution function of the just computed histogram and then finds the index of the smallest

Figure 4.

```
// Perform the color conversion and calculate the histogram

for (int i = 0; i < num_rows; ++i) {
    for (int j = 0; j < num_cols; ++j) {
        Vec3b intensity = image.at<Vec3b>(i, j);

        uchar B = intensity.val[0];
        uchar G = intensity.val[1];
        uchar R = intensity.val[2];

        uchar Y = 0.299 * R + 0.587 * G + 0.114 * B;
        uchar U = -0.169 * R - 0.331 * G + 0.499 * B + 128;
        uchar V = 0.499 * R - 0.418 * G - 0.0813 * B + 128;

        ++histogram[Y];

        Vec3b yuv_int = yuv.at<Vec3b>(i, j);
        yuv_int[0] = Y;
        yuv_int[1] = U;
        yuv_int[2] = V;

        yuv.at<Vec3b>(i, j) = yuv_int;
    }
}
```

Figure 5.

```
//  Calculate the cumulative distribution function (CDF) and the
// and the index of the min element of CDF greather than zero
std::vector<int> cdf(256, 0);
cdf[0] = histogram[0];
int cdf_min = 0;
for (int i = 1; i < 256; ++i) {
    cdf[i] = cdf[i - 1] + histogram[i];

}
for (int i = 0; i < 256; ++i) {
    if (cdf[i] != 0) {
        cdf_min = i;
        break;
    }
}
```

value different from zero. The code in 6 does the actual histogram equalization in another double for loop, taking every element of yuv, calculating the new Y value and then converting back the values stored in yuv to the RGB color space.

Figure 6.

```
// Perform histogram equalization
for (int i = 0; i < num_rows; ++i) {
    for (int j = 0; j < num_cols; ++j) {
        Vec3b yuv_int = yuv.at<Vec3b>(i, j);

        uchar Y = static_cast<int>(round(255 * (cdf[yuv_int[0]] - cdf_min) /
                                         static_cast<float>(total_pixels - cdf_min)));
        uchar U = yuv_int[1];
        uchar V = yuv_int[2];

        uchar R = std::max(0, std::min(255,
            static_cast<int>(Y + 1.402 * (V - 128))));
        uchar G = std::max(0, std::min(255,
            static_cast<int>(Y - 0.344 * (U - 128) - 0.714 * (V - 128))));
        uchar B = std::max(0, std::min(255,
            static_cast<int>(Y + 1.772 * (U - 128))));

        Vec3b intensity = yuv.at<Vec3b>(i, j);
        intensity[0] = B;
        intensity[1] = G;
        intensity[2] = R;
        yuv.at<Vec3b>(i, j) = intensity;

    }
}
```

2.2. Parallel versions

2.2.1 OpenMP

OpenMP is a multiplatform API to create parallel applications on shared memory systems. It's supported by C, C++ and Fortran and by different operating systems. OpenMP can be used on a sequential program with little to no need for restructuring of the code. In fact when using OpenMP it's possible to use directives to instruct the compiler on how to parallelize the code, and if no library function is used the structure of the program stays the same. In this case the $pragma\ \ omp\ \ for\ \ collapse(2)$ has been used twice to parallelize the nested for loops. Since there are now more threads cooperating to compute the histogram of the image it is also necessary to make the ++ operation on the histogram atomic through the use of the directive $pragma\ \ omp\ \ atomic$. A third different directive is used on the loop calculating the cumulative distribution function: since every loop depends on an element calculated the previous loop $pragma\ \ omp\ \ single$ is used to make sure that only one thread executes that part of the code. This is in fact called a loop carried dependency and is not parallelizable.

2.2.2 PyCuda

Cuda is a parallel computing platform that allows the programmer to use Nvidia's GPU for general purpose programming. It is designed to work with C, C++ and Fortran but in this project a wrapper called PyCuda has been used to be able to call Cuda functions from a Python environment.
Even if it's meant to run on GPU, the general logic of the program is still the same. The functions doing most of the work are called kernels and are written in C. Just like in the OpenMP program, an atomic instruction is used to modify the shared array without incurring in data races. Since it's not parallelizable, the part of code that computes the cumulative distribution function runs on CPU with Python instructions. Since the work is on 2D images the kernels are called with a 2D 16x16 block and a 2D grid for a better mapping of threads to data.

3. Considerations

1. The programs will be tested on a system with an I3 8100 quadcore CPU and a GTX 960 2GB with 1024 CUDA core.

2. To measure time of execution in a C++ environment we used $std :: chrono :: system\_clock$ which measures the real time wall clock. On Python $perf\_counter$ was used from the $time$ module.

3. Every version of the program uses OpenCV to load and save the images.

4. Tests and performance

To test the application the same image is fed to the three different programs at increasingly higher dimensions, from 500x500px to 8000x8000px. Every result is averaged on 10 different computations.

Table 1. Block size: 2 characters

	Sequential C++	OpenMP	PyCuda
500x500	124.7ms	79.9	5.6ms
1000x1000	339.6ms	240.8ms	8.8ms
2000x2000	1214.3ms	768.8ms	22.2ms
4000x4000	4932.9ms	2922.6ms	76.8ms
8000x8000	19249.6ms	11745.5ms	284.9ms

As expected the PyCuda code greatly outperforms the code run on CPU, averaging a x60/x70 speedup over the sequential C++ program. The OpenMP program doesn't perform as we could expect, getting a bit less than a x2 speedup with big images on a quadcore machine.