Sequential and Parallel RGB Image Histogram Equalization in C++, OpenMP and CUDA

Giulio Calamai
Matricola: 7006568
E-mail Address
`giulio.calamai1@stud.unifi.it`

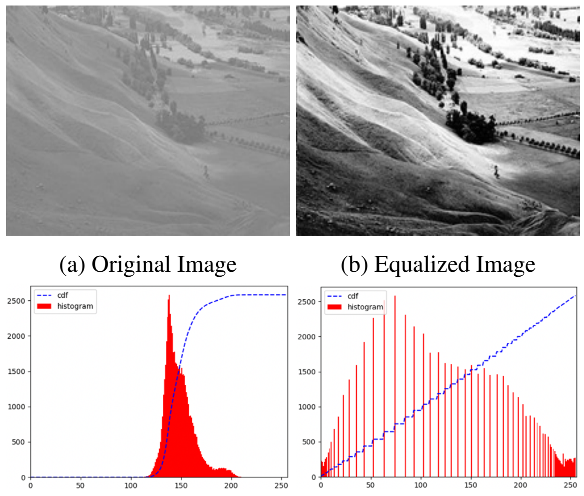
Marco Loschiavo
Matricola: 7017247
E-mail Address
`marco.loschiavo1@stud.unifi.it`

Abstract

This work is about the implementation of an important task in Image Processing that makes clearer the vision of an image and homogeneous its histogram: Histogram Equalization.This technique has been applied to RGB images.The task has been realized in both sequential and parallel version, respectively in C++ and OpenMP/CUDA languages. Sequential and OpenMP ones have been tested on a machine having Intel core i7, quad core. Instead, CUDA code has been ran on a NVDIA Titan-X GPU, having 12Gb of RAM e 3072 CUDA cores.Performances have been finally evaluated in terms of speedup and curves representing the execution times to compute equalized images.

1. Introduction

Histogram equalization is a spatial domain method that produces output image with uniform distribution of pixel intensity. This means that the histogram of the output image is flattened and extended systematically. This approach customarily works for image enhancement because of its simplicity and is relatively better than other traditional methods. This method usually increases the global contrast of many images, especially when the usable data of the image is represented by close contrast values. Through this adjustment, the intensities can be better distributed on the histogram. This allows for areas of lower local contrast to gain a higher contrast. Histogram equalization accomplishes this by effectively spreading out the most frequent intensity values. The method is useful in images with backgrounds and foregrounds that are both bright or dark. A key advantage of the method is that it is a fairly straightforward technique and an invertible operator. So in theory, if the histogram equalization function is known, then the original histogram can be recovered. A disadvantage of the method is that it is indiscriminate. It may increase the contrast of background noise, while decreasing the usable signal. Histogram equalization often produces unrealistic effects in photographs; however it is very useful for scientific images like thermal, satellite or x-ray images[1].



(a) Original Image (b) Equalized Image

(c) Original Histogram (d) Equalized Histogram

Figure 1: Histogram Equalization for a greyscale image.

1.1. Histogram Equalization

Consider a discrete grayscale image $\mathbf{x} = [x_{i,j}]$ and let n_k be the number of occurrences of gray level k. The probability of an occurrence of a pixel x of level k in the image is:

$$p_x(k) = p(x = k) = \frac{n_k}{n} \qquad 0 \leq k < L \tag{1}$$

L being the total number of gray levels in the image (typically 256), n being the total number of pixels in the image, and $p_x(k)$ being in fact the image's histogram for pixel value k, normalized to [0,1].

Let us also define the *cumulative distribution function* *(cdf)* corresponding to p_x as:

$$cdf_x(k) = \sum_{l=0}^{k} p_x(l) \tag{2}$$

which is also the image's accumulated normalized histogram.

After normalizing cdf_x such that the maximum value is 255 ($h(k)$) we are now able, using a trasformation, to replace each pixel in the new image **y** = $[y_{i,j}]$:

$$\begin{aligned} y(i,j) &= h(x(i,j)) \\ &= round\Big(\frac{cdf(x(i,j)) - cdf_{min}}{(M \cdot N) - cdf_{min}} \cdot (L-1)\Big) \end{aligned} \tag{3}$$

where $x(i,j)$ return the intensity level of pixel $x_{i,j}$ and h is the histogram.

1.2. RGB to YUV Conversion

Above we described histogram equalization on a grayscale image. However it can also be used on color images by applying the same method separately to the Red, Green and Blue components of the RGB color values of the image. However, applying the same method on the Red, Green, and Blue components of an RGB image may yield dramatic changes in the image's color balance since the relative distributions of the color channels change as a result of applying the algorithm. However, if the image is first converted to another color space such as YUV, then the algorithm can be applied to the brightness(luma) or value channel without resulting in changes to the chrominance of the image. So the first step we have made has been RGB to YUV image conversion, which can be defined as follows:

$$\begin{cases} Y = R \cdot 0.299 + G \cdot 0.587 + B \cdot 0.114 \\ U = R \cdot (-0.168736) + G \cdot (-0.331264) + B \cdot 0.5 + 128 \\ V = R \cdot 0.5 + G \cdot (-0.418688) + B \cdot (-0.081312) + 128 \end{cases} \tag{4}$$

The effect of this operation is shown in Figure 2

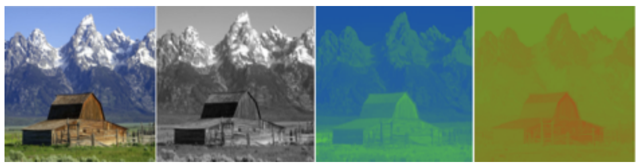
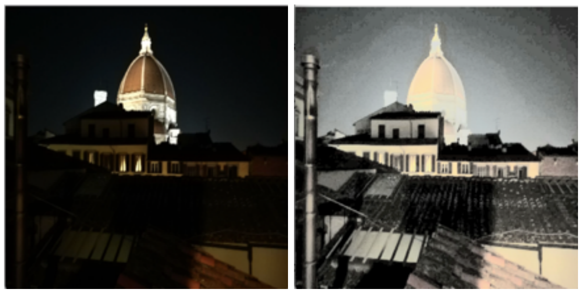


Figure 2: RGB to YUV Conversion

After this conversion the equalization has been applied to Y channel histogram, leaving U and V ones intact, and after that image has been reconverted to RGB, using the following equations:

$$\begin{cases} R = Y + 1.4075 \cdot (V - 128) \\ G = Y - 0.3455 \cdot (U - 128) - (0.7169 \cdot (V - 128)) \\ B = Y + 1.779 \cdot (U - 128) \end{cases} \tag{5}$$

The final result is shown in Figure 3



(a) Original Image (b) Equalized Image

Figure 3: RGB Equalized Image

2. Implementation

As described above, to apply histogram equalization, we first convert the image color space from RGB to YUV. Both sequential and parallel implementation are tested and timed over the equalization of the B channel.

2.1. Sequential Version

We have implemented the procedure described above through 3 functions (called in the following order): *make_histogram*, *cumulative_histogram* and *equalize*. The first one receives the image histogram as formal parameter and computes the RGB to YUV conversion, updating the Y histogram [1]. The second function [2] computes the cumulative distribution function and makes the equalization applying [1]. The third one [3] implements the conversion from YUV back to RGB image.

Complete procedure is shown in Figure 4:

2.2. Parallel versions

2.2.1 OpenMP Parallel Version

The first technology used to parallelize the procedure above is OpenMP, an API which supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran. The OpenMP parallel version of the code is implemented as the sequential one, with the main difference that, as OpenMP rules, the same functions implemented in the sequential version have been written into a particular OMP

Algorithm 1: make_histogram

```
Data: image, histogram, YUV_image
1  initialize histogram ;
2  for i=0 to image.rows do
3      for j=0 to image.cols do
4          $R, G, B = image(i, j)$ ;
5          $R = Y + 1.4075 \cdot (V - 128)$ ;
6          $G = Y - 0.3455 \cdot (U - 128) - (0.7169 \cdot (V - 128))$;
7          $B = Y + 1.779 \cdot (U - 128)$ ;
8          $histogram[Y]++$ ;
9          $YUV\_image(i, j) = Y, U, V$ ;
10     end
11 end
```

Algorithm 2: cumulative_histogram

```
Data: histogram, equalized_histogram, image.cols , image.rows
1  $cumulative\_histogram[256]$ ;
2  for i=0 to 256 do
3      $cumulative\_histogram[i] = histogram[i] + cumulative\_histogram[i-1]$;
4      $equalized\_histogram[i] = ((cumulative\_histogram[i] - histogram[0])/(image.cols \cdot image.rows - 1) \cdot 255)$ ;
5  end
```

Algorithm 3: equalize

```
Data: image, equalized_histogram, YUV_image
1  initialize histogram ;
2  for i=0 to image.rows do
3      for j=0 to image.cols do
4          $Y = equalized\_histogram[YUV\_image(i, j)]$ ;
5          $U = YUV\_image(i, j)$ ;
6          $V = YUV\_image(i, j)$
7          $R = max(0, min(255, (int)(y + 1.4075 \cdot (V - 128))))$ ;
8          $G = max(0, min(255, (int)(y + 0.3455 \cdot (U - 128) - (0.7169 \cdot (V - 128)))))$ ;
9          $R = max(0, min(255, (int)(y + 1.7790 \cdot (U - 128))))$ ;
10         $image(i, j) = R, G, B$
11     end
12 end
```

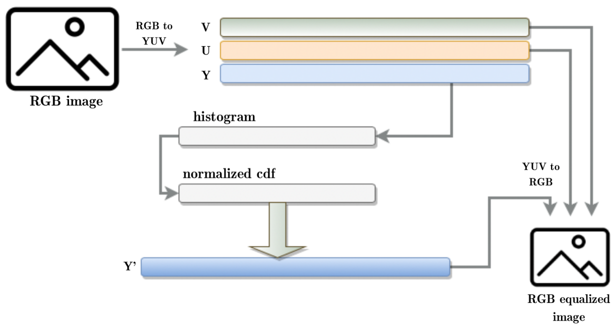


Figure 4: Diagram showing the sequantial scheme

structure that parallelizes the sequential for loops: #pragma omp parallel for. The use of this structure has been really easy and convenient: in fact it is made in such a way as to decide the number of threads to be used, depending on the task to execute and the machine which runs it.

2.2.2 CUDA Parallel Version

The second technology used to parallelize Histogram Equalization is CUDA, a parallel computing platform and application programming interface (API) model created by Nvidia. It allows to use a CUDA-enabled GPU for general purpose processing. The platform is designed to work with programming languages such as C, C++, and Fortran. For this project we ran CUDA code on a NVDIA Titan-X GPU. The main idea for task's parallelization has been the following: if possible, depending on the number of CUDA cores, pixels of the image have been processed one per thread. Otherwise what happens is illustrated in Figure 5. A special zoom has to be done on GPU management: we have allocated a char array to represent the image and three int[256] arrays in order to store image histogram, image equalized histogram and the cumulative distribution function. After that we have copied them from the host (CPU) to the device (GPU) through the function cudaMemCpy. Next, the functions illustrated in sequential version have been reproduced as three CUDA kernels: make histogram, equalize and YUV2RGB: they have been called in the same order as before, computing the equalization. Finally we have freed the GPU through the function cudaFree and saved the results.

3. Results

Tests have been performed using the same image, varying its size. All results have been achieved mediating over 10 run. They have also been evaluated when varying the number of threads until saturation (relatively to the machine used for testing).

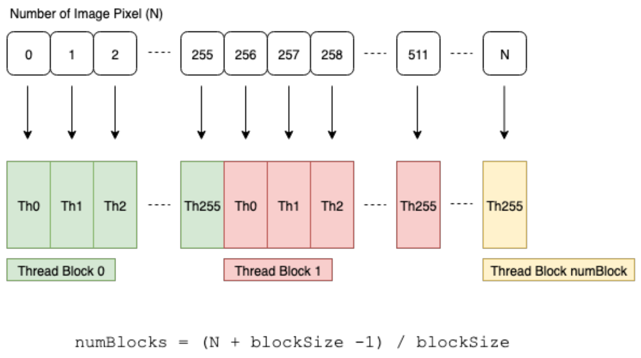


Figure 5: Management of CUDA threads and blocks on image

With OpenMP version we have tested the images above with 2, 4, 8 threads and default. The results are showed in Figure 6

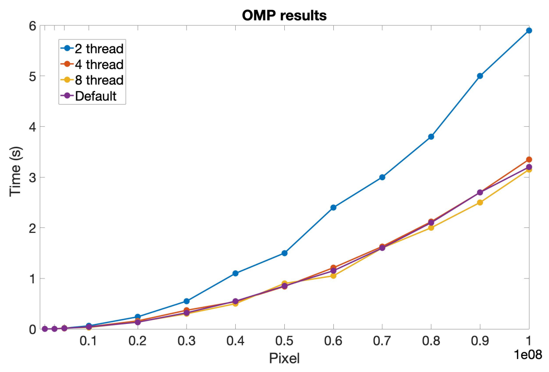


Figure 6: OpenMP results

How about CUDA code, we have evaluated that the best configuration was the one described before (1 CUDA core per pixel), and we report in Figure 7 the curve representing relative results.

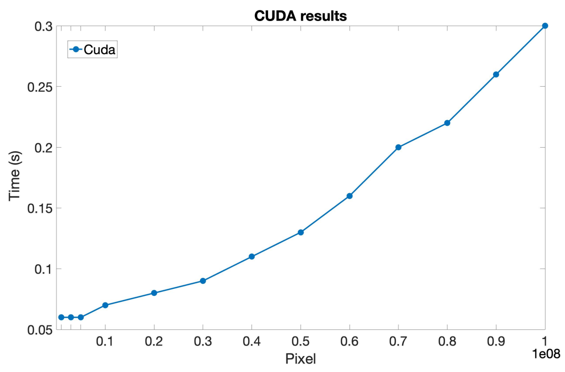


Figure 7: CUDA results

As final result we report in Figure 8 a global evaluation of each version's best configuration.

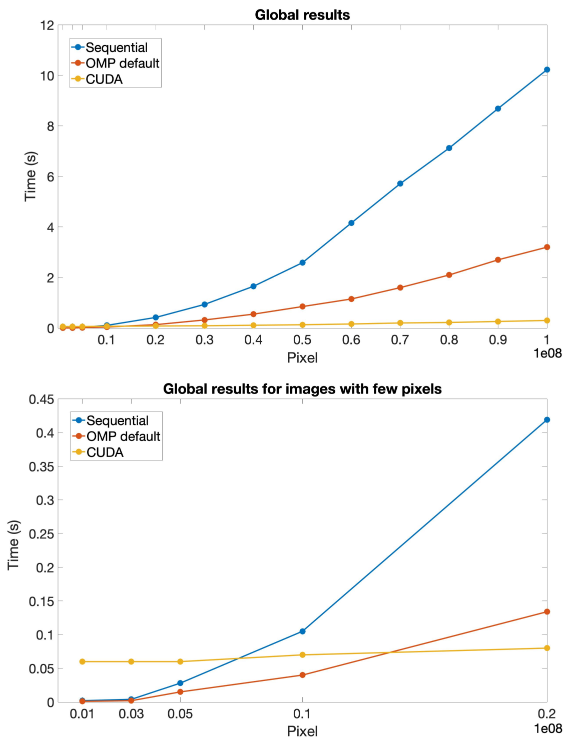


Figure 8: Global results

3.1. Speed-Up

The Figure 9 represents the speed-up obtained with OpenMP and CUDA against sequential implementation.

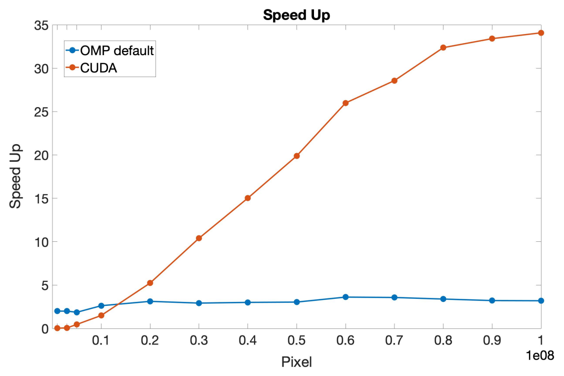


Figure 9: Speed-Up results