Histogram Equalization: Sequential and Parallel

Harjinder Singh Sandhu
E-mail address:
harjinder.sandhu@stud.unifi.it

Leonardo Casini
E-mail address:
leonardo.casini1@stud.unifi.it

Abstract

We present a work about the implementation of Histogram Equalization, which is a computer image processing technique used to improve contrast in images. This work has been done in both sequential and parallel version. The chosen languages are C++ for the first version and OpenMP and CUDA for the second one. This technique has been applied to RGB images. All the results, in terms of optimal number of threads and execution time, are obtained on a machine with Intel(R) Core(TM) i7 @ 2.80GHz CPU.

1. Introduction

Image quality improvement is a very important task in Image Processing, which can be often achieved combining various techniques, as the Histogram Equalization. The histogram represents the number of pixels for each intensity value considered.The equalization is a technique for adjusting image intensities to enhance contrast. It accomplishes this by effectively spreading out the most frequent intensity values. It modifies the dynamic range of an image by altering the pixel values, guided by the intensity histogram of that image. Through this adjustment, the intensities can be better distributed on the histogram. This allows for area of lower local contrast to gain a higher contrast. The method is useful in images with backgrounds and foregrounds that are both bright or both dark. In particular, the method can lead to better views of bone structure in x-ray images, and to better detail in photographs that are over or under exposed. 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. The calculation is not computationally intensive. A disadvantage of the method is that it's indiscriminate. It may increase the contrast of background noise, while decreasing the usable signal.

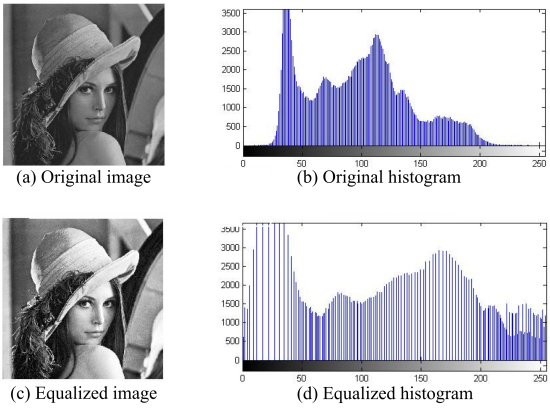


Figure 1: Histogram equalization

1.1. Theoretical formulation

We'll consider the case of a grayscale image, anyhow if we have an RGB image, we just need to convert it into YUV color space and then use this method only on the Y channel. Consider a discrete grayscale image x and let n_i be the number of occurrences of gray level i. The

probability of an occurrence of a pixel of level i in the image is:

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

L being the total number of gray levels in the image (typically 256), n being the total number of pixels in the image, and $p_x(i)$ being in fact the images 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 images 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 $y = [y_{ij}]$:

$$y(i,j) = h(x(i,j)) = \\ round(\frac{cdf(x(i,j) - cdf_{min})}{(M \times N - cdf_{min})}) \tag{3}$$

where $x(i, j)$ return the intensity level of pixel $x_{i,j}$ and h is the histogram. MN gives the image's number of pixels

1.2. RGB to YUV

The equalization used for gray scale image, can also be applied to color images by using the same method separately to the Red, Green and Blue components of an 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 images color balance since the relative distributions of the color channels change as a result of applying the algorithm. So what we first do is convert the image to another color space, that is YUV, and then the algorithm is ready to be applied to the brightness or value channelwithout resulting in changes to the hue and saturation of the image. The YUV model defines a color space in terms of one luma component (Y) and two chrominance components, called U (blue projection) and V (red projection) respectively. This is how we convert from RGB to YUV:

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

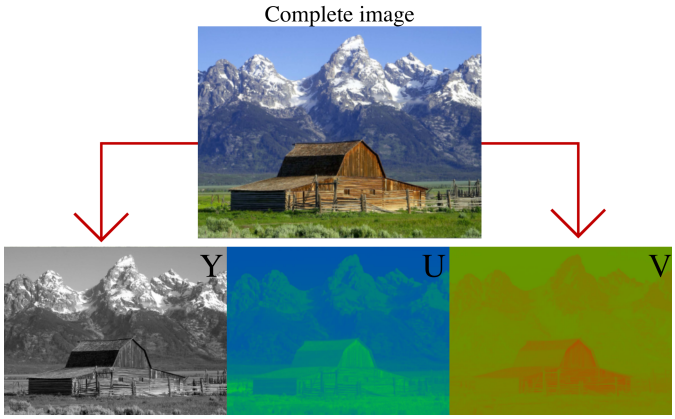
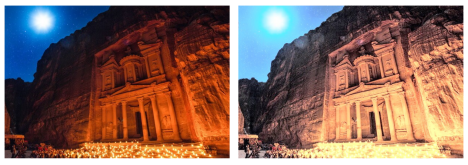


Figure 2: RGB to YUV Conversion

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:

$$\begin{cases} R = Y + 1.4075 \times (V - 128) \\ G = Y - 0.3455 \times (U - 128)(0.7169 \times (V128)) \\ B = Y + 1.779 \times (U - 128) \end{cases} \tag{5}$$



(a) Original image (b) Equalized image

Figure 3: RGB Equalized Image

2. Implementation

The implementation is done in both sequential and parallel version, is tested and timed

over the equalization of the B channel.

2.1. Sequential

We have implemented the procedure described above through 3 functions (called in the following order): *hist_maker*, *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. The second function computes the cumulative distribution function and makes the equalization applying. The third one implements the conversion from YUV back to RGB image.

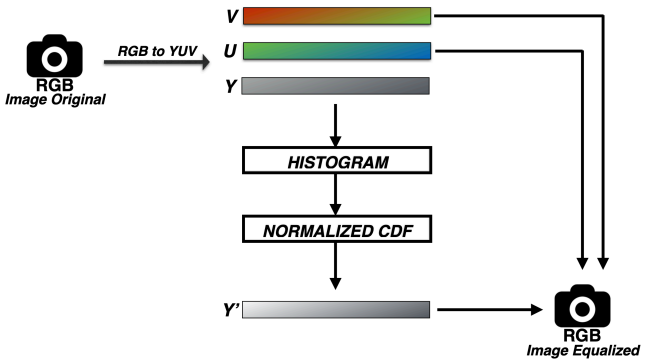


Figure 4: Sequential scheme

Algorithm 1: hist_maker

Data: $image, hist, YUVimage$
for $i = 0$ *to* $image.rows$ **do**
  for $j = 0$ *to* $image.cols$ **do**
    $R, G, B = image(i, j)$;
    $R = Y + 1.4075 \times (V - 128)$;
    $G = Y - 0.3455 \times (U - 128) - (0.7169 \times (V - 128))$;
    $B = Y + 1.779 \times (U - 128)$;
    $hist[Y++]$;
    $YUVimage(i, j) = Y, U, V$;
  end
end

2.2. Parallel

2.3. Parallel version: OpenMP

Substantially this parallel version is really similar to the sequential one descrived before.

Algorithm 2: cumulative_histogram

Data: $hist, equalized\_histogram, image.cols, image.rows$
$cumulative\_histogram[256]$;
initiliaze histogram;
for $i = 0$ *to* 256 **do**
  $cumulative\_histogram[i] = hist[i] + cumulative\_histogram[i1]$;
  $equalized\_histogram[i] = ((cumulative\_histogram[i] - hist[0])/(image.cols \times image.rows - 1) \times 255)$
end

Algorithm 3: equalize

Data: $image, equalized\_histogram, YUVimage$
for $i = 0$ *to* $image.rows$ **do**
  for $j = 0$ *to* $image.cols$ **do**
    $Y = equalized\_histogram[YUVimage(i, j)]$;
    $U = YUVimage(i, j)$;
    $V = YUVimage(i, j)$;
    $R = max(0, min(255, (int)(y + 1.4075 \times (V - 128))))$;
    $G = max(0, min(255, (int)(y + 0.3455 \times (U - 128) - (0.7169 \times (v - 128)))))$;
    $R = max(0, min(255, (int)(y + 1.7790 \times (U - 128))))$;
    $image(i, j) = R, G, B$;
  end
end

OpenMP(Open Multi-Processing) is an API that provides C/C++/Fortran the ability to run certain parts of the code in parallel, without explicitly managing (creating, destroying, assigning) threads. OpenMP parallel version 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 structure that parallelizes the sequential for loops: #pragma omp parallel for. The OpenMP parallel for flag tells the OpenMP system to split this task among its working threads. In addition to providing directives for parallelization, OpenMP allows developers

to choose among several levels of parallelism. E.g., they can set the number of threads manually. It also allows us to identify whether data are shared between threads or are private to a thread. OpenMP is available on several open-source and commercial compilers for Linux, Windows, and Mac OS X systems.

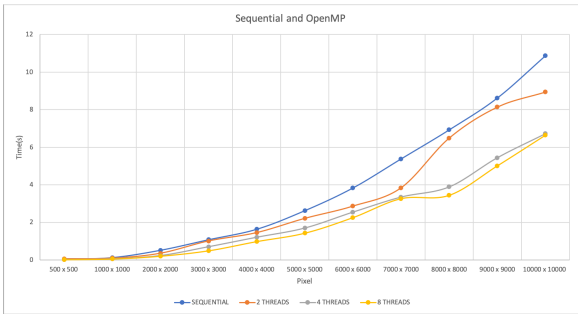


Figure 5: Sequential and OpenMP results

2.4. Parallel version: CUDA

We used CUDA as the second technology for the parallel version. CUDA is 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 tasks parallelization has been the following: if possible, depending on the number of CUDA cores, pixels of the image have been processed one per thread. A special focus has been done on GPU management: we've 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 cudaMem-Cpy. Next, the functions illustrated in sequential version have been reproduced as three CUDA kernels: *hist_maker*, *normalizeCdf* and *equalize*: 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.

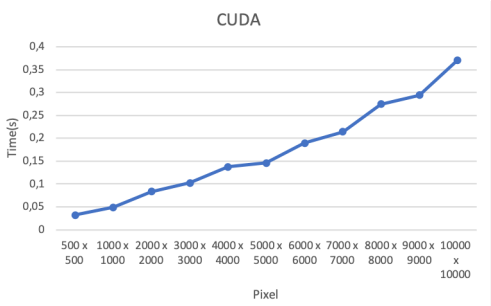


Figure 6: CUDA results

3. Results

The results has been calculated running some tests on the same image, varyinge its own size from 100×100 to 10.000×10.000.
Every result is the calculated on the average of 10 runs. With OpenMP version we have tested the images above with 2, 4, 8 threads.

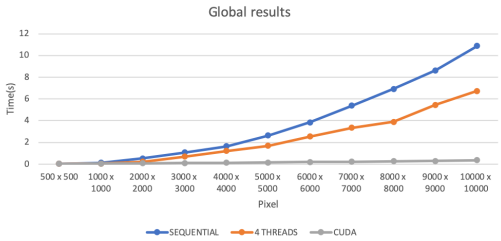


Figure 7: Global results

3.1. Speed-Up

Speed-up obtained with OpenMP and CUDA against sequential implementation.

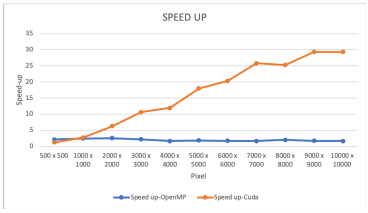


Figure 8: Speed up results

References

[1] Wikipedia - Histogram Equalization,
`https://en.wikipedia.org/wiki/Histogram_equalization`

[2] NVIDIA docs,
`https://docs.nvidia.com/cuda/index.html`

[3] Materiale didattico - Parallel Computing