# PC-Efficient Image Histogram Equalization

Luca Angioloni
luca.angioloni@stud.unifi.it

Francesco Pegoraro
francesco.pegoraro@stud.unifi.it

## Abstract

*We present an histogram image equalization algorithm for reducing execution time, exploiting details of the code and analyzing different results. Our system makes use of parallelisms in computation via the JAVA thread programming model. This implementation treats portions of the target image as independent input arrays for the histogram algorithm and finally merge partial results to accomplish the task. We compare the performance of the parallel approach running on the CPU with the sequential implementation across a range of image sizes.*



## 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].

### 1.1. Histogram Equalization

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:

$$p_x(i) = p(x = i) = \frac{n_i}{n}, \;\; 0 \leq i < L$$

*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 image's histogram for pixel value *i*, normalized to [0,1].

Let us also define the *cumulative distribution function* corresponding to $p_x$ as:

$$cdf_x(i) = \sum_{j=0}^{i} p_x(j)$$

After normalizing $cdf_x$ such that the maximum value is 255 we are now able to replace each pixel in the new image $x'$:

$$x'(i,j) = c\hat{df}_x(x(i,j))$$

Results on a grey scale image are shown in fig. 1 and 2.

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 HSB/HSV, then the algorithm can be applied to the brightness or value channel without resulting in changes to the hue and saturation of the image.

### 1.2. Implementation

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

#### 1.2.1 Sequential Histogram Equalization

For an image of size $w \times h$, we form the histogram using the following algorithm,

- B : input Brightness channel of dimension $w \times h$. Array elements are accessed in row major order.
- size : input size of B ($w \times h$).
- H : Histogram vector of dimension 256.
- C : *cdf* array of dimension 256.
- L : normalized *cdf* array of dimension 256, we will refer to it as *LookupTable*.
- B' : output Brightness channel of dimension $w \times h$.

**Algorithm 1** Sequential histogram equalization

```
1:  procedure EQUALIZE(B, size)
2:      for i = 0 to size do
3:          value ← round(B[i] * 255);
4:          H[value] ← H[value] + 1;
5:      C[0] = H[0];
6:      L[0] = (C[0] * 255)/size;
7:      for i = 1 to 255 do
8:          C[i] = C[i − 1] + H[i];
9:          L[i] = (C[i] * 255)/size;
10:     for i = 0 to size do
11:         value ← round(B[i] * 255);
12:         B'[i] = L[value];
13:     return B'
```

Complete procedure is shown below:

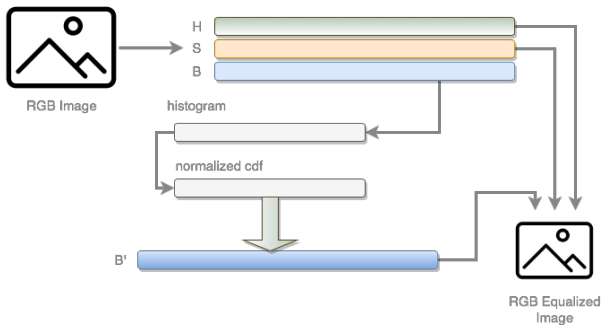


Figure 3. Diagram showing the sequantial scheme

#### 1.2.2 Parallel Histogram Equalization

We developed three different classes of threads: *PartialLookUpThread*, *LookUpAdder* and *EqualizedValueCalculator*.

*PartialLookUpThread* given a portion of the input, calculates the partial *histogram*, then the *cdf* and finally the *normalized cdf* or LookupTable. Each thread has its own portion of the image with size *image_size/number_of_threads*.

Next is shown the *run* method of this thread:

- B : input Brightness channel of dimension $w \times h$. Array elements are accessed in row major order.
- start : input index, denotes the portion of B assigned to the specific thread.

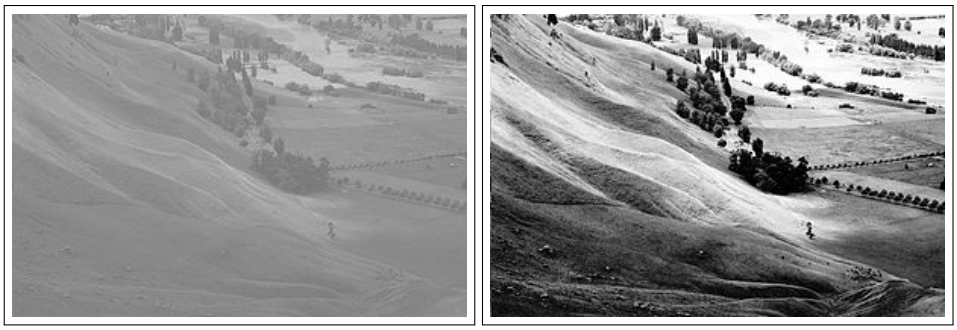

Figure 1. Image before and after equalization.

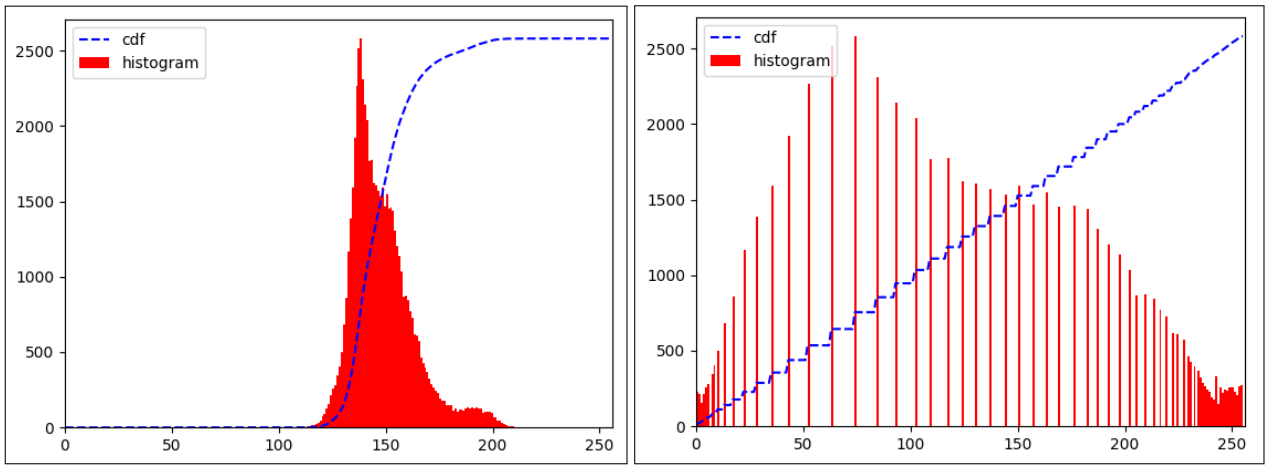


Figure 2. Histogram and *cdf* before and after equalization.

- end : input index, denotes the portion of B assigned to the specific thread.
- size : input size of B ($w \times h$).
- H : Histogram vector of dimension 256.
- C : cdf array of dimension 256.
- L : normalized cdf(LookupTable) array of dimension 256.

**Algorithm 2** Run of thread *PartialLookUpThread*

**procedure** RUN(B, start, end, size)
  **for** $i = start$ to $end$ **do**
    $value \leftarrow round(B[i] * 255)$;
    $H[value] \leftarrow H[value] + 1$;
  $C[0] = H[0]$;
  $L[0] = (C[0] * 255)/size$;
  **for** $i = 1$ to 255 **do**
    $C[i] = C[i-1] + H[i]$;
    $L[i] = (C[i] * 255)/size$;
  *return* $L$

After every thread returns its *LookupTable*, we stack them in a matrix we can refer to as *PartialLookups* of dimension $number\_of\_threads \times 256$. *LookUpAdder* threads essentially merge the result of the partial histograms computed by *PartialLookUpThread* storing values in an array we call *lookup*. Parallelization lies in the fact that each thread works with a portion of *PartialLookups* of dimension $256/number\_of\_threads$.

Next is shown the *run* method of this thread:

- PL : input matrix *PartialLookups* of dimension $number\_of\_threads \times 256$ representing results from previous *PartialLookUpThread*.
- start : input index, denotes the portion of *PartialLookups* assigned to the specific thread.
- end : input index, denotes the portion of *PartialLookups* assigned to the specific thread.
- NT : *number_of_thread*.

- L' : output *Lookup* vector of size 256. Each thread has a reference to it and fills it with values.

**Algorithm 3** Sequential histogram equalization

```
1: procedure RUN(PL, start, end, NT)
2:     for i = start to end do
3:         tot ← 0;
4:         for j = 0 to NT do
5:             tot ← tot + PL[j][i];
6:         L' ← tot;
7:     return L'
```

Now that we have a LookupTable the only thing left is to replace every pixel of the image with its new value that will result in an equalized image. This task is assigned to threads *EqualizedValuesCalculator*. As in *PartialLookUpThread* each thread works in a portion of image of size *image_size/number_of_threads*.

Here is the run method:

- B : input Brightness channel of dimension $w \times h$. Array elements are accessed in row major order.
- start : input index, denotes the portion of B assigned to the specific thread.
- end : input index, denotes the portion of B assigned to the specific thread.
- L : final *LookupTable* array of dimension 256.
- B' : output Brightness channel of dimension $w \times h$.

**Algorithm 4** Run of thread *PartialLookUpThread*

```
1: procedure RUN(B, B', start, end, L)
2:     for i = start to end do
3:         value ← round(B[i] * 255);
4:         B'[i] = L[value];
```

After every thread has finished, in B' we have the new Brightness channel that will be used to create an equalized version of the image.

Scheme below should represent the procedure with the different trheads carrying out each task:

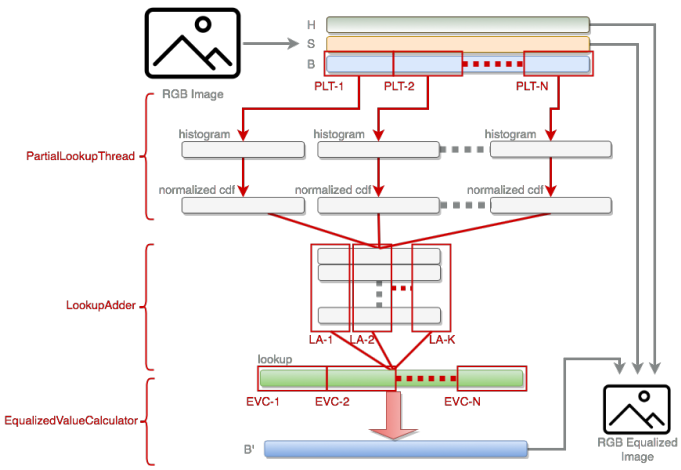


Figure 4. Diagram showing the parallel scheme

## 2. Results

All the results, in terms of optimal number of threads and execution time, are obtained on a machine with *Intel(R) Xeon(R) E5-2620 v3 @ 2.40GHz* CPU.

### 2.1. Number of thread

Using an image of size $10000 \times 16000$ we tested the execution time with different number of threads from 1 to 1000. The best result is obtained using 8 threads.

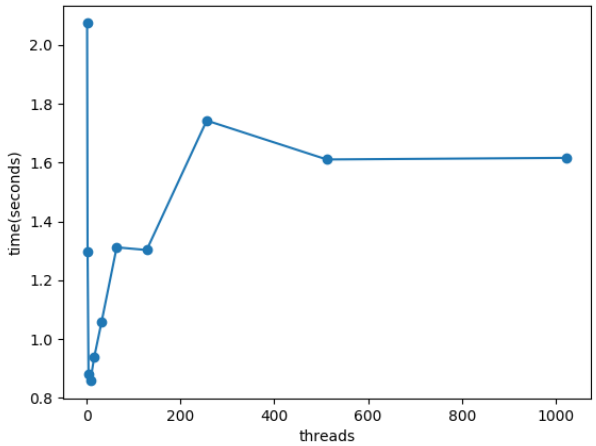


Figure 5. Execution time varying number of threads

### 2.2. Serial vs Parallel

Varying image size, we discover that with image larger than 27.800.000 pixels the parallel

algorithm is always faster of the serial. To quantify this improvement we computed the *speed up factor* between the two implementation as: $Parallel\_execution\_time/Serial\_execution\_time$. The result is observable in fig. 7

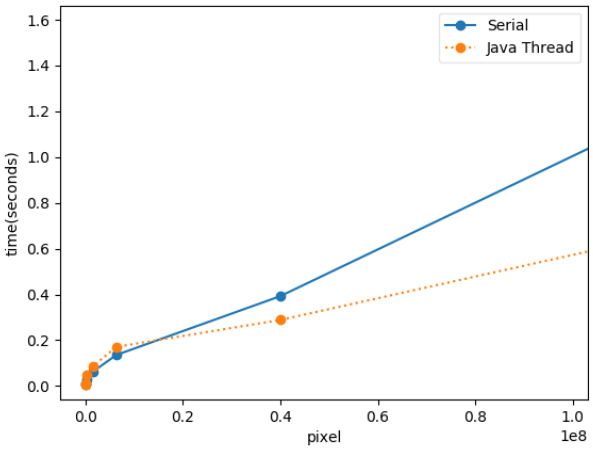


Figure 6. Seriall and Parallel execution time

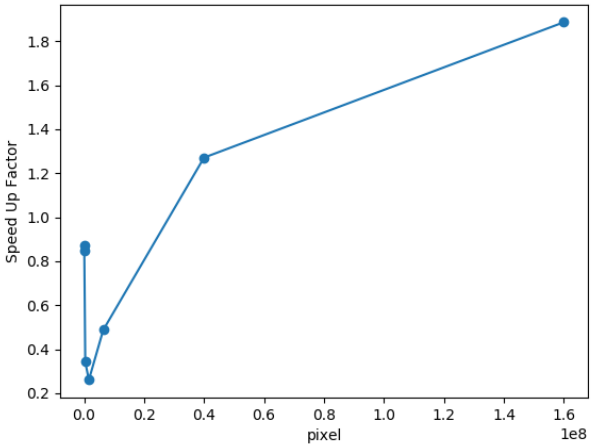


Figure 7. Speed up factor of Paralle implementation over serial

## References

[1] Yan Chai Hum, Khin Wee Lai, and Maheza Irna Mohamad Salim. “Multiobjectives bihistogram equalization for image contrast enhancement”. In: *Complexity* 20.2 (2014), pp. 22–36.