Histogram Equalization: Java and JavaThread

Alberto Ciolini
alberto.ciolini@stud.unifi.it

Alessandro Soci
alessandro.soci@stud.unifi.it

Abstract

We present the histogram equalization in two different implementations: a sequential implementation in Java and a parallel implementation using JavaThread. We compare the performances with the sequential implementation across images of various size.

1. Introduction

Histogram Equalization is an operator that modify the level of intensity of the pixels image. In particular tent to produce an uniform histogram of pixel intensity. This method is used for increasing the contrast of the images, because increase the contrast in correspondence to the peaks of the histogram and reduce the contrast in correspondence to the valleys of the histogram, 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 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 is indiscriminate. It may increase the contrast of background noise, while decreasing the usable signal [1].

2. Theoretical Formulation

Histogram Equalization use a monotonically increasing function $\tau(p) = q$. Now if we consider a set of gray levels $p \in \{p_0, ..., pl\}$ of the original image, $H(p_i)$ and $G(q_i)$ the values of the original and equalized image, with $q_i = \tau(p_i)$. Since τ is monotonically increasing is valid, therefore:

$$\sum_{q=\tau(p_0)}^{\tau(p_j)} G(q) = \sum_{q=(p_0)}^{(p_j)} H(p)$$

where p_j is a generic level. Furthermore, G(q) should approximate a uniform distribution, and for a generic $N \times M$ images, the values should be:

$$G(q) = \frac{N * M}{\tau(p_L) - \tau(p_0)}$$

and so:

$$\sum_{q=\tau(p_0)}^{\tau(p_j)} G(q) = (\tau(p_j) - \tau(p_0)) \frac{N * M}{\tau(p_L) - \tau(p_0)}$$

thus:

$$\tau(p_j) = \tau(p_0) + \frac{\tau(p_L) - \tau(p_0)}{N * M} \sum_{q=(p_0)}^{(p_j)} H(p)$$

Usually $\tau(p_L)$ is 255 and $\tau(p_0)$ is 0. $\tau(p_j)$ is the new gray level of output image for every p_j gray level of input image. [2]

An example

Figure 1. Example of Histogram Equalization

Above we described histogram equalization on a grayscale image. Obviously is possible to applied to RGB image. But the equalization applied to RGB image yield to bad and no-consistent contrast. Fortunately, like us, it's possible to convert RGB image in HSB image. This format is composed by the three channels: Hue, Saturation and Brightness. The most important for our job is the third channel of Brightness, because it gives informations on the intensity of the pixels, and we can use it for histogram equalization.

3. Implementation

We have used the theory defined above to implement the program. The most important feature is the conversion from RGB to HSB image. This operation helps us to decrease the number of loop, because we use only a channel (Brightness) instead of 3 channels.

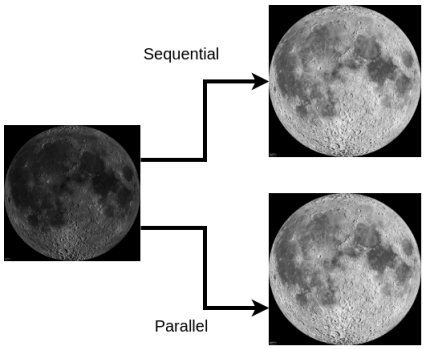

Figure 2. From orginal to Equalized image with ours algorithms

3.1. Sequential Histogram Equalization

We have implemented and organized the procedure described above through class and functions. Below we show and define the most important class and function:

- class *HSB_Image*: the constructor convert RGB channels in HSB channels
- function *equalize*: that generate the histogram and calculate the new values of pixels of output image;

Below there are the pseudo-implementation of that method[1]:

RGB_image represent the color image of input, instead hsb represent the input image but on HSB channels.

Algorithm 1 Histogram Equalization

```
1:  procedure EQUALIZE(hsb_image, lenght)
2:      histogram[256];
3:      cumulative[256];
4:      table[256];
5:      new_value[size];
6:      for i=0 to lenght do
7:          value ← HSB_image*255;
8:          histogram[value]++;
9:      cumulative[0] ← histogram[0];
10:     for i=1 to 256 do
11:         cumulative[i] ← cumulative[i-1] + histogram[i];
12:         table[i] ← cumulativesize;
13:     for i=0 to lenght do
14:         value ← max(0,min(255,HSB_image*255));
15:         new_value[i] ← table[i];
```

[1] to simplify we consider the size of the image as a vector $length \times 1$

In this function there is the core of the work. At first we calculate the histogram of the image on channel B of HSB, then we define and produce a table with the new normalized value, range [0,1]. In the end we assign the new value to the output image.

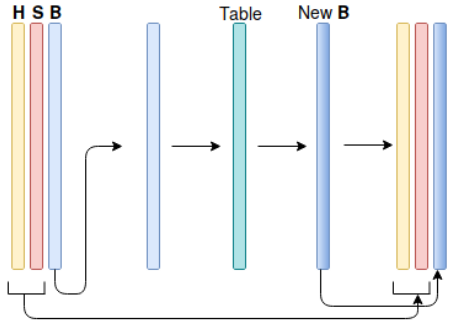

Figure 3. Simplified scheme of Parallel computation

3.2. Parallel Histogram Equalization

The parallel version is developed in JavaThread, and 3 thread classes were necessary to optimized the work. Every thread class has a specific task and is:

- *Partial_Table_Thread*: it calculates a portion of the whole histogram, and a partial of the table;
- *Table_Adder_Thread*: it recreates the whole table
- *Final_Values_Thread*: it assigns the new values to the output image

In detail:

Algorithm 2 Partial Table Thread

```
1: procedure RUN()
2:     for i=start to end do
3:         value ← HSB_image[i]*255;
4:         partial_histogram[value]++;
5:     partial_cumulative[0] ← partial_histogram[0];
6:     partial_table[0] ← partial_cumulative[0]/lenght;
7:     for j=1 to 256 do
8:         partial_cumulative[i] ← cumulative[i-1] +
   partial_histogram[i];
9:         partial_table[i] ← cumulative/lenght;
```

The variable *start* and *end* are assigned in the constructor of the class, in this case they represent only a part of whole image, for example: the image is possible to see it as a vector $N * M \times 1$ where N and M are the size of the image, the program is initialized with 8 threads; the image is divided in 8 sequences and each sequence is computed by a thread. Therefore we obtain a partial table and it's necessary to complete it with *Table_Adder_Thread*.

Algorithm 3 Table Adder Thread

```
1: procedure RUN()
2:     for i=start to end do
3:         tot ← 0;
4:         for j=0 to partial_num do
5:             tot ← tot + partial_table[j][i];
6:         table[i] ← tot;
```

In this case the values of *start* and *end* represent a part of partial table generated in the previous step, *partial_num* is equal to number of threads.

In the end we have to assign new values to output image.

Algorithm 4 Final Values Thread

```
1: procedure RUN()
2:     for i=start to end do
3:         value ← HSB_image*255;
4:         new_value ← table[value];
```

In conclusion we modify the channel B of intensity, and we add it to the other 2 channel (Hue and Saturation) e we obtain the equalized output image.

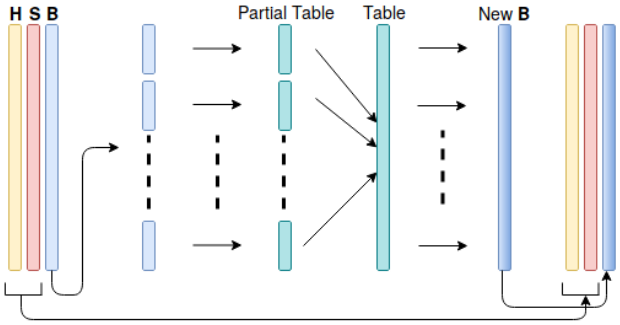

Figure 4. Simplified scheme of Parallel computation

4. Result

All the result are obtained on a machine with *Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz (total 24 cores)*.

4.1. Sequential

We have tested the *Sequential* algorithm with images of different size: minimum size 500×500 and maximum size 10000×7000.

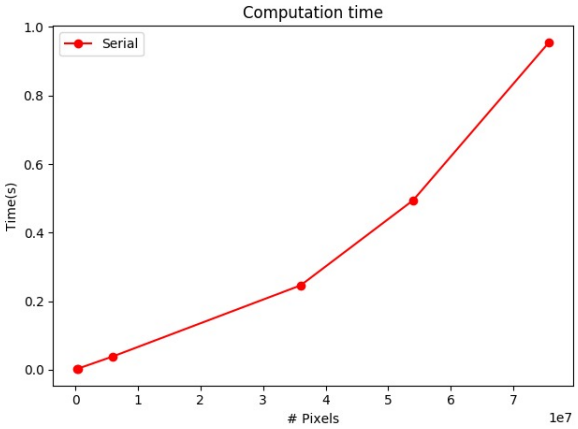

Figure 5. Sequential execution time

4.2. Parallel

We have tested the *Parallel* algorithm with images of different size: minimum size 500×500 and maximum size 10000×7000.

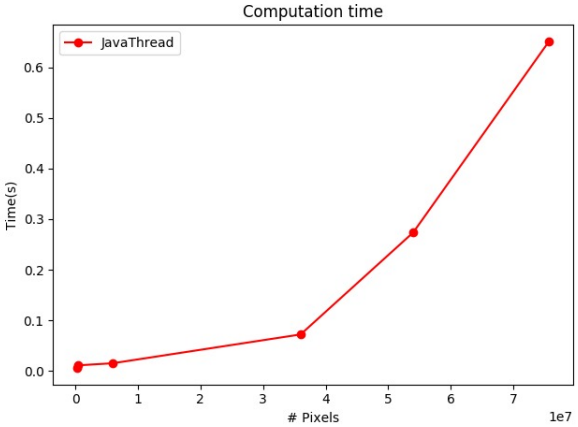

Figure 6. Parallel execution time

Furthemore, we have tested the *Parallel* version with an image of size 10000×7000 and we have calculated the time with different threads (range[1,1024]) many times, so we have be able to average the time for every thread:

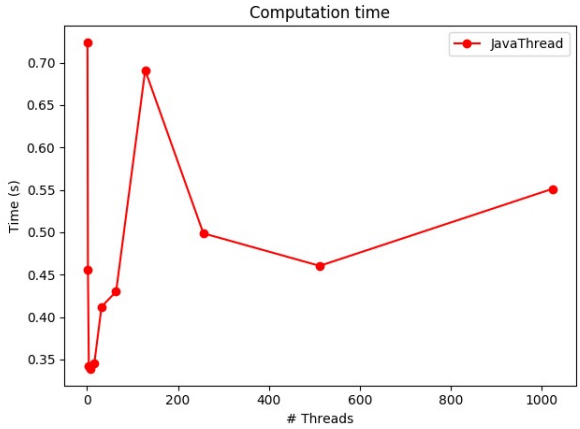

Figure 7. Execution time varying number of threads

4.3. Sequential vs Parallel

We have tested the two algorithms with images of different sizes, setting the number of thread at 8 for *Parallel* version, and compered the execution time:

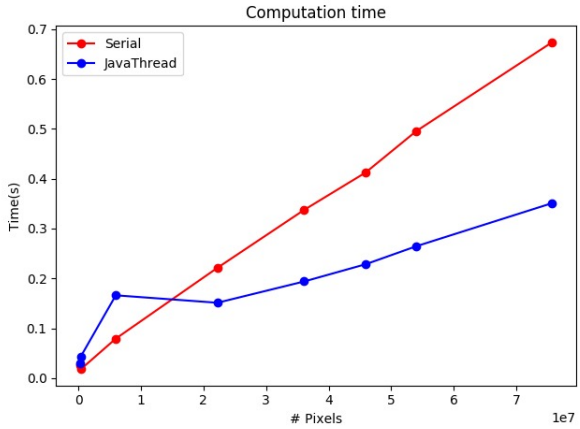

Figure 8. Sequential and Parallel execution time

The speed up factor of *Parallel* implementation over *Sequential*:

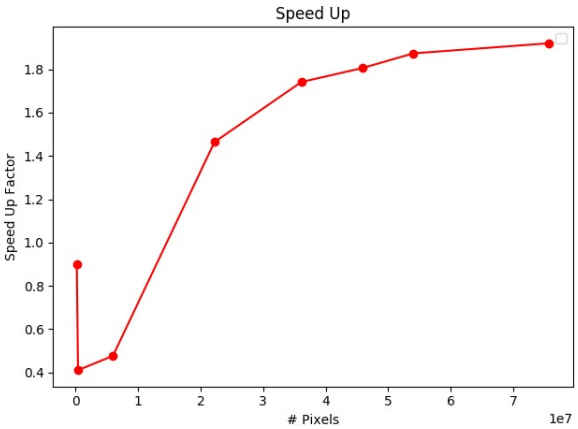

Figure 9. Speed up factor

5. Conclusion

We have processed many images of different size with sequential and parallel algorithm and we can say that if the size of image is bigger than approximately 1500000 of pixels, the parallel version is more performant than sequential version. Then we think that the initialization of the threads and the scheduling of them reduce speed of the parallel version.

References

[1] Y. C. Hum, K. W. Lai, M. Salim, and M. Irna. *Multiobjectives bihistogram equalization for image contrast enhancement*, 2014.

[2] P. Pala. *Pixel Operator*. 2016.