


# Histogram Equalization

## for color images

Implementation of sequential and parallel version in Java and CUDA

**Alessandro Sestini - 6226094**

15-01-2018

# Introduction

## Histogram Equalization

The histogram equalization is a process **to enhance** an image based on **its histogram**.

Suppose we have a greylevel image, we compute the histogram of grey value of each pixel:

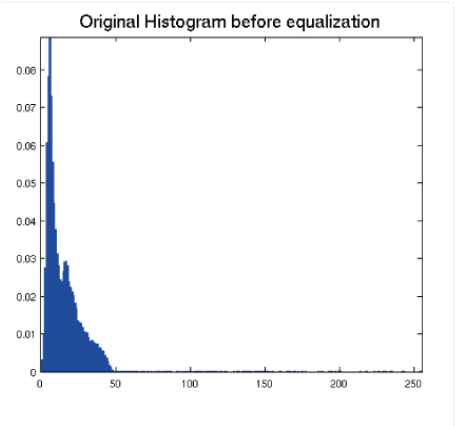
$$h(v_k) = n_k$$

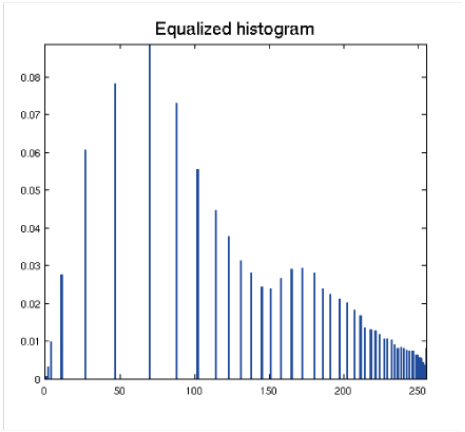
And then we can apply the **equalization** to make the histogram more uniform using:

$$h_{eq}(v) = round\left(\frac{cdf(v) - cdf(0)}{(M * N) - 1} * L - 1\right)$$

Where $cdf(v)$ is the cumulative distribution function, $M * N$ is the size of the image and $L - 1$ is the maximum value of a pixel (tipically 255). Then the new values are defined by:

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




Result of the equalization

# Equalization of a color image

The same process of greylevel equalization can be applied to a **RGB image**:

- First we apply **color space conversion** from RGB to YCbCr;
- Then we compute the **histogram of the Y channel values**;
- We equalize the Y histogram and map the old values to the new ones;
- Finally we invert the color space from YCbCr to RGB;

This color space is suitable for us because the Y values are the **luminance values**, and in order to enhance the contrast of a figure we have to equalize them.

Result of the equalization on color image

# Implementation

## Sequential version

For the implementation were used **Java and C/CUDA**. The sequential one is very similar for both languages:

- We load a image file and **for each** pixel we convert it from RGB to YCbCr while we compute the Y channel histogram;
- We make the **histogram equalization**;
- **For each** pixel of the image, we map the old values to the new ones and we make the conversion from YCbCr to RGB.

In Java we use a *BufferedImage* object, while in C the image is treated like an array of char.

# Parallel version

## Java

In java we use **Java threads**. The task is to parallelize the 3 main cycles, with 3 different type of threads.

### First Thread

- It provides the color space conversion from RGB to YCbCr while computes **locally** the histogram. Each thread takes in input a **non overlapping sub-part of the image**;
- To do this, it must implement the Callable<int[]> interface so that the result can be take in the sequential part by some Future<int[]> objects;
- At the end, all the **local histograms are added** together to make the global histogram;
- The thread execution is managed by a Thread Pool.

```
public int[] call(BufferedImage sub_image) {

  //ITakes in input only a sub-part of the image
  int width = sub_image.getWidth();
  int height = sub_image.getHeight();
  for (int y = 0; y < height; y++) {

    sub_image.getRaster().getPixels(0,y,width,1,iarray);
    for (int x = 0; x < iarray.length; x+=3) {
      //Convert and compute the histogram
      int r = iarray[x+0];
      int g = iarray[x+1];
      int b = iarray[x+2];

      int Y = (int)(0.299*r+0.587*g+0.114*b);
      int Cb = (int)(128-0.168736*r-0.331264*g+0.5*b);
      int Cr = (int)(128+0.5*r-0.418688*g-0.081312*b);

      iarray[x+0] = Y;
      iarray[x+1] = Cb;
      iarray[x+2] = Cr;

      this.histogram[Y]++;
    }
    sub_image.getRaster().setPixels(0,y,width,1,iarray);
  }
  return histogram;
}
```

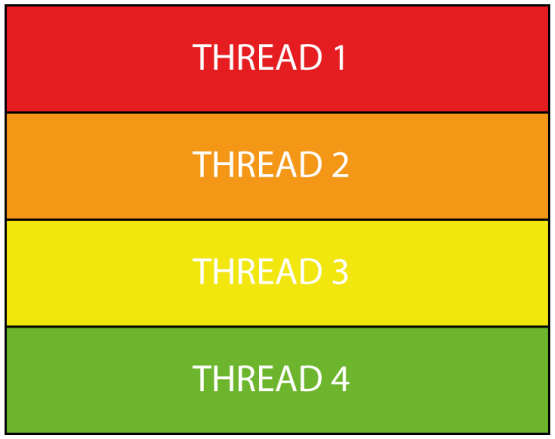



Image suddivision by thread

# Second Thread

- The second thread computes the **equalized histogram**. It takes the **cdf**, the histogram and it works only on a **sub-part of the equalized histogram**;
- There are no possibile race conditions.

Equalized histogram


Histogram suddivision by threads

# Third Thread

- Like the first type, each thread works on a **sub-part** of the image;
- It **Maps** the new value based on the new histogram ;
- It **Converts** from YCbCr to RGB.

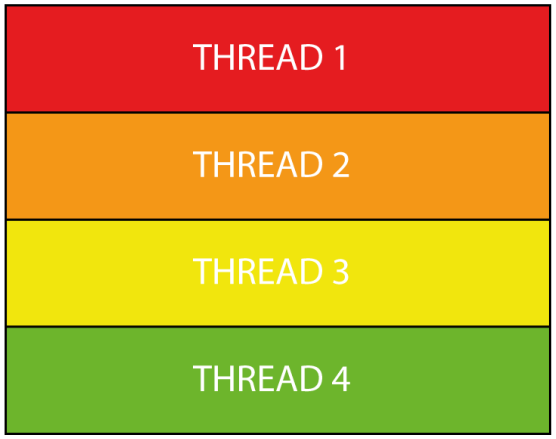



Image suddivision by thread

# Parallel version

## CUDA

In CUDA we use 3 different **kernels**.

### Memory Management

We allocate some objects with ***cudaMalloc*** in the global memory:

- A buffer for the input image, and then we do a ***cudaMemCpy*** to move it from host to the decive;
- An array of int of size 256 for the histogram;
- An array of int of size 256 for the cdf;
- An array of int of size 256 for the equalized histogram.

# First Kernel

- We first create a ***__shared__ int histogram [256]*** to compute the histogram locally in a block;
- Then the kernel converts the image from RGB to YCbCr;
- If $num_{threads} \geq num_{pixels}$ then each thread will process only **one pixel**, else each thread will process **contiguous elements of different sections** of the image;

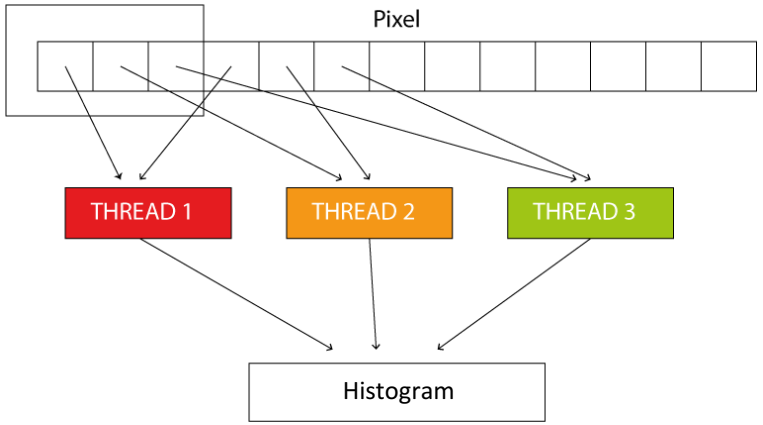



Image suddivision by CUDA threads

## Coalesced access and shared histogram

- If the threads inside a warp access contiguous sections of an array (like our image) that are in the same burst, then **only one request** will be made to the DRAM of the GPU. The **coalesced access** will speedup the performance against the global memory access;

- For the same reason, we compute the histogram of the Y channel locally within a block, beacuse the shared memory is private to each SM and has very **short access latency**; this directly translates into increase throughput of atomic operations. The down side is that the **private copies need to be merged into the original data** structure after the computation completes.

```
__shared__ int hist_priv[256];
... Initialize hist_priv ...

__synchtreads();

//Thread index
int idx = blockIdx.x*blockDim.x + threadIdx.x;
//Total number of threads
int stride = blockDim.x*gridDim.x;
for(int i = idx; i < width*height; i += stride){
  int point_index = i*3;
  int r = input[point_index];
  int g = input[point_index+1];
  int b = input[point_index+2];

  int Y = (int)(0.299*r+0.587*g+0.114*b);
  int Cb = (int)(128-0.168736*r-0.331264*g+0.5*b);
  int Cr = (int)(128+0.5*r-0.418688*g-0.081312*b);

  //Atomic operation on private histogram
  atomicAdd(&(hist_priv[Y]), 1);
}

__synchtreads();


for(int bin_idx = threadIdx.x; bin_idx < 256;
          bin_idx += blockDim.x){
  atomicAdd(&(hist[bin_idx]), hist_priv[bin_idx]);
}
```

# Second and third kernel

- Once the first kernel completes, the histogram is copied from device to host and the **cdf is calculated in the CPU**, then it's copied from host to device;
- The second kernel **equalizes the histogram**:

```
for(int i = idx; i < 256; i += blockDim.x*gridDim.x){
  hist[i] = (int)(((((float)cdf[i] - cdf[0]))/(((float)width*height - 1)))*255);
}
```

- The third maps the old values to the new ones and **converts** the image from YCbCr to RGB, exactly like the first one.

# Experiments and results

| | Java | | Cuda | |
|---|---|---|---|---|
| | Sequential | Parallel | Sequential | Parallel |
| 800x600 | 0.059 s | 0.080 s | 0.018 s | 0.002 s |
| 1200x1600 | 0.150 s | 0.125 s | 0.080 s | 0.007 s |
| 3840x2160 | 0.340 s | 0.250 s | 0.321 s | 0.025 s |
| 7680x4320 | 0.950 s | 0.530 s | 1.100 s | 0.100 s |

The main difference is that in Java we spend **much more time on the creation of the threads**, so the speedup is only visible on big images. Instead, in CUDA we spend much less time on the creation, so the time gain is visible also for **smaller images**.