PC-2019/20 Final Project: Histogram equalization: Java threads and CUDA, parallel approach comparison

Marioemanuele Ghianni
E-mail address
marioemanuele.ghianni@stud.unifi.it

Abstract

This project is the last part of the exam of the course Parallel Computing at University of Florence.
The aim of this project is to introduce and analyze a sequential and two parallel version of a program that implements an image processing technique called histogram equalization.
The project is developed in C++ for the sequential version and in CUDA and Java for the parallel one.

Future Distribution Permission

1. Introduction and basic concepts

1.1. Histogram equalization

An histogram is a graphical representation of the intensity distribution of an image.
In simple terms, it represents the number of pixels for each intensity value considered.
As for grayscale images, [2] the histogram shows the number of pixels for each brightness level (from black to white), and when there are more pixels, the peak at the certain brightness level is higher.
Histogram equalization is a computer image processing technique used to improve contrast in image for better detection and features extraction. It accomplishes this by effectively spreading out the most frequent intensity values, i.e. stretching out the intensity range of the image.
This method usually increases the global contrast of images when its usable data is represented by close contrast values. This allows for areas of lower local contrast to gain a higher contrast.

1.2. Formulation

Let imm an image represented by a NxM matrix of pixel intensities ranging from 0 to $I-1$ (255) and $hist$ the relative histogram.
The function $histEq$ is defined as below:

$$histEq(hist) = round(\frac{cdf(hist) - cdf(0)}{(N*M)-1} * (L-1)).$$

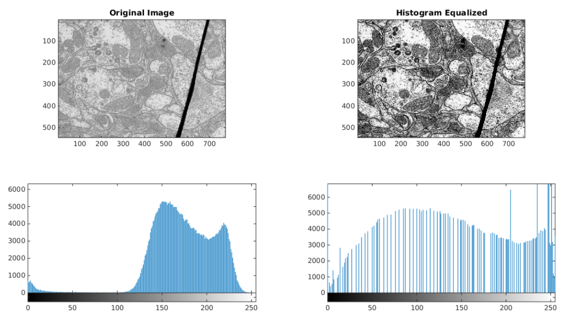

Figure 1: An example of an histogram equalization process.

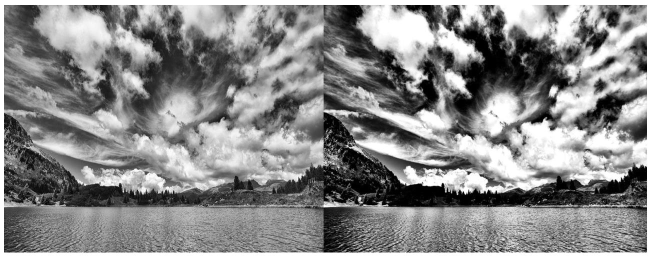

Figure 2: Another example of grayscale histogram equalization.

1.3. RGB Histogram equalization

Histogram equalization can also be applied to RGB images, of course, but we need some changes as we now have 3 colors channel.
There are various methods, one of them is to convert the RGB image to the **YCbCr** color space, do the equalization on the luma component Y and reconvert back to RGB.
The conversion to YCbCr can be defined as below:

$Y = 0.257R + 0.504G + 0.98B + 16$
$Cb = -0.148R - 0.291G + 0.439B + 128$
$Cr = 0.439R - 0.368G - 0.071B + 128$

while the reconversion to RGB:

$R = (Y - 16) * 1.164 + 1.596 * (Cr - 128)$
$G = (Y - 16) * 1.164 - 0.813 * (Cr - 128) - (0.392 * (Cb - 128))$
$B = (Y - 16) * 1.164 + 2.017 * (Cb - 128)$

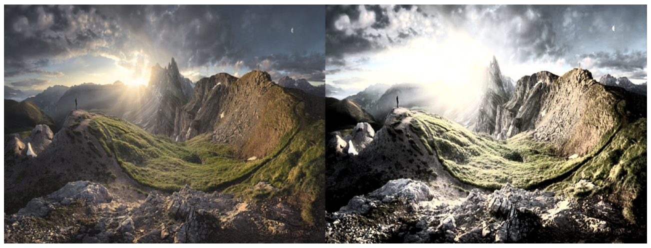

Figure 3: An example of RGB histogram equalization.

2. Structure

Summing up, the structure of the proposed program is divided into 3 phases:

- image acquisition, conversion to YCbCr color space and histogram construction

- effective equalization of the histogram

- reconversion to an RGB image

2.1. Functions

In this project, as pointed out in the structure, there are three basic functions: $convertToYCbCr$ takes the image and an histogram and convert the rgb image to YCbCr while assign values to the histogram; $equalizeHist$ takes the histogram and actually implements the equalization and $revertToRGB$ simply, as the name suggests, revert the image to RGB.

2.2. Main

The main file simply read the images in a directory, istantiate an histogram, invokes the functions and calculates the execution times.

3. Sequential Histogram equalization

The sequential version is implemented in both C++ and Java and, in the first case, the opencv library was used for reading and editing images while in Java is used the BufferedImage class.
Apart from this, the two sequential version have been written as similar as possible.
Detailed C++ code of the three fundamentals function can be seen below.

```
void convertToYCbCr(int width, int height,
    unsigned char* im_ptr, int histogram[]) {

    for (int i = 0; i < height * width * 3; i +=
        3) {

        int R = im_ptr[i + 0];
        int G = im_ptr[i + 1];
        int B = im_ptr[i + 2];

        int Y = R * .257000 + G * .504000 + B *
            .098000 + 16;
        int Cb = R * -.148000 + G * -.291000 + B
            * .439000 + 128;
        int Cr = R * .439000 + G * -.368000 + B *
            -.071000 + 128;

        im_ptr[i + 0] = Y;
        im_ptr[i + 1] = Cb;
        im_ptr[i + 2] = Cr;

        histogram[Y] ++;
    }
}

void equalizeHist(int histogram[], int
    equalizedHist[], int cols, int rows)
{
    int cumulative_histogram[256];

    cumulative_histogram[0] = histogram[0];

    for (int i = 1; i < 256; i++)
    {
        cumulative_histogram[i] = histogram[i] +
            cumulative_histogram[i - 1];
        equalizedHist[i] = (int)(((float)
            cumulative_histogram[i] - histogram
            [0]) / ((float)cols * rows - 1) *
            255);
    }
}

void revertToRGB(unsigned char* im_ptr, int width
    , int height, int equalizedHist[]) {
    for (int i = 0; i < height * width * 3; i +=
        3) {

        int value_before = im_ptr[i];
        int value_equalized = equalizedHist[
            value_before];

        im_ptr[i] = value_equalized;

        int Y = im_ptr[i + 0];
        int Cb = im_ptr[i + 1];
        int Cr = im_ptr[i + 2];

        unsigned char R = (unsigned char)max(0,
            min(255, (int)((Y - 16) * 1.164 +
            1.596 * (Cr - 128))));
        unsigned char G = (unsigned char)max(0,
            min(255, (int)((Y - 16) * 1.164 -
            0.813 * (Cr - 128) - (0.392 * (Cb -
            128)))));
        unsigned char B = (unsigned char)max(0,
            min(255, (int)((Y - 16) * 1.164 +
            2.017 * (Cb - 128))));

        im_ptr[i + 0] = R;
        im_ptr[i + 1] = G;
        im_ptr[i + 2] = B;
    }
}
```

4. Parallel approach

These three functions have therefore been parallelized via Cuda and Java Threads but we can observe that most of the computational load lies in the two conversion and reversion functions while the equalize is simply a loop on 256 elements with a not very expensive calculation.
In particular, the function $revertToRGB$ will be the one that will have longer calculation times due to the various max and min operations.

4.1. CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by Nvidia.
It allows software developers and software engineers to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing through CUDA Development Toolkit [1].
GPUs are extremely effective in parallel calculations thanks to their large number of cores, a much larger bandwidth in memory access and not too much control logic.
In particular, for this specific problem, GPUs computing it is ideal for performing the 3 fundamental functions extremely quickly.
The idea for the parallel approach is that every pixel of the image is processed by a different thread so the grid dimension is determined by the number of image's pixels and 256 that is the number of threads per block (See Figure 4).
Therefore, three different kernels are set for each function and the appropriate allocations and copies are made from / to cpu and gpu.
Below an example of block and grid definition and iteration through.

```
// block and grid definition

int block_size = 256;
int grid_size = (width * height + (block_size -
    1)) / block_size;

//an example of iteration

int idx = blockIdx.x * blockDim.x + threadIdx.x;
for (int i = idx; i < width * height; i +=
    blockDim.x * gridDim.x)
```

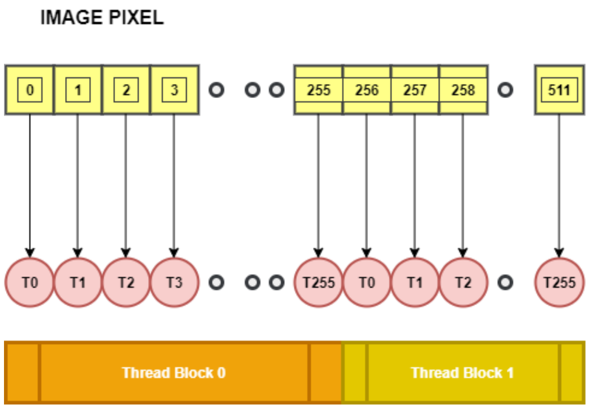

Figure 4: image's pixel management.

4.2. Java

Regarding parallel implementation in Java, a fixed (4) thread pool is instantiated and are managed by an executor service.
Then, the image is divided horizontally into equal parts each managed by a thread which deals with the conversion to YCbCr and the conversion to RGB (as we can see in Figure 5).
As for the equalizing function, the histogram is simply divided between threads.
Future interface is used to take the results of the asynchronous computation performed by threads who uses the Java interfaces Runnable and Callable.
An extract of the java code showing this procedure in the case of the first function can be seen below.

```
//allocate an array of future to take results of
    parallel execution through callable function

ArrayList<Future> futures = new ArrayList<>();

//we use a fixed pool of threads equal to the
    number of cores/threads

ExecutorService exec = Executors.
    newFixedThreadPool(cores);

//divide the image according to the number of
    threads

int yStart = subdivision((double)height, cores);
int subHeight = subdivision((double)height, cores
    );

for (int i = 0; i < cores; i++) {

    if(i == cores-1 && height%cores != 0){
            subHeight = height - i*subdivision((
                double)height, cores);
                    }
    BufferedImage subImg = img.getSubimage(0,(i*
        yStart),(width),(subHeight));

    //start a thread that run the callable
        function on the subImg to convert RGB->
        YCbCr

    futures.add(exec.submit(new ConvertToYCbCr(
        subImg)));

 }
 //Wait for the pool to finish and reconstruct
     the histogram

for(Future<int[]> future : futures){

    int[] localHist = future.get();

     for(int j = 0; j < 256; j++) {
            histogram[j] += localHist[j];
                    }
                }
```

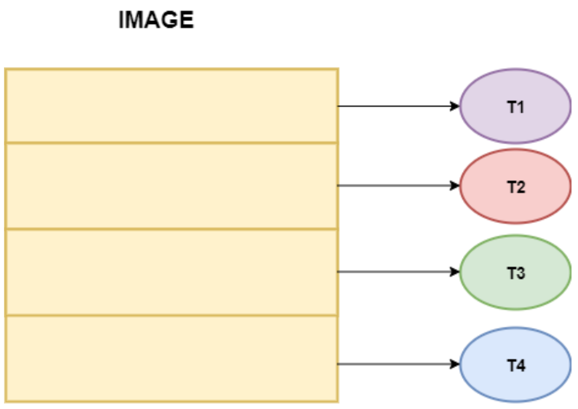

Figure 5: image subdivision scheme.

5. Tests and comparison

The tests are basically based on increasing the size of the images to be processed and calculating an average of the execution times.
A little optimization was done in the CUDA code, in particular, in the first kernel ($convertToRGB$), a shared histogram has been defined with the overhead of manage it by synchronizing and reconstructing the histogram.
That, in an image of size 7680 x 4800, produces an improvement of 5 ms on a total of 56ms.

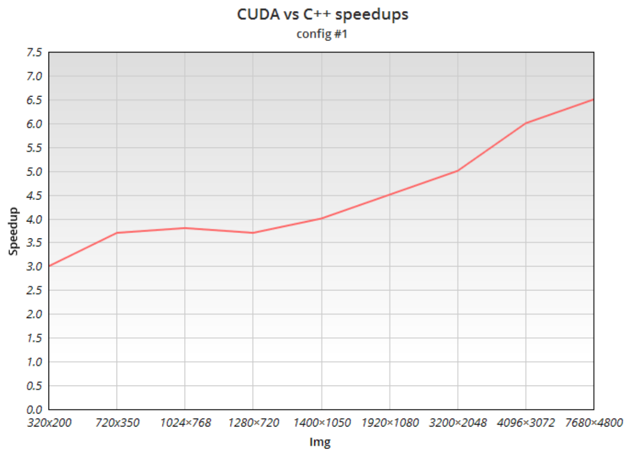

Figure 6: Speedup graph of CUDA version on Nvidia MX150 vs C++ sequential.

5.1. Test configuration

The tests were performed mainly on an Intel i57200U clocked at 2.50GHz system with 2 physical core and 2 virtual core with a Nvidia MX150 dedicated GPU.
As regards the tests on CUDA, they were also carried out on another GPU-oriented configuration with a Ryzen 5 3600 clocked at 3.60 Ghz and a Nvidia RTX 2060 Super dedicated GPU.
This was done to actually evaluate the impact of a much better graphics card.

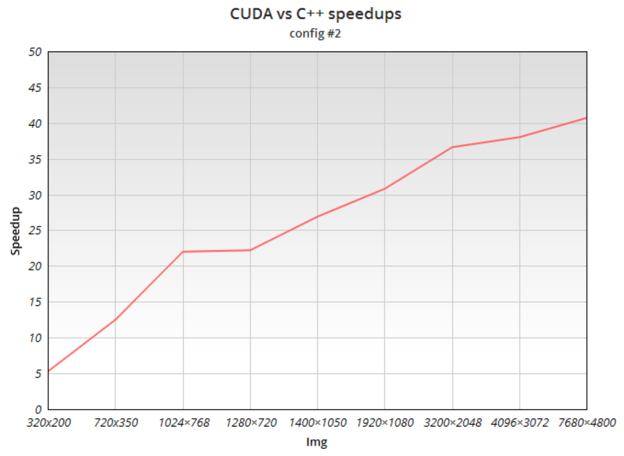

Figure 7: Speedup graph of CUDA version on Nvidia RTX2060 vs C++ sequential.

5.2. C++ vs CUDA

As expected CUDA implementation is way better than the sequential C++ code.
Even with a modest graphics card for laptops, the Nvidia MX150, over 6 speedup value is achieved on large images (7680x4800).
Much better situation with the RTX 2060 super graphics card where we even reach a speedup above 40.

5.3. Java Threads vs Java sequential

The classic parallel approach on the CPU also brings benefits even if in a more contained way.
As we can see in the above graph (Figure 8), for relatively small images the overhead given by the creation and management of threads makes the sequential approach give better results but in the case of moderate-sized images we begin to see advantages in the parallel approach.
For very large images we can see how the speedup reaches close to 3.

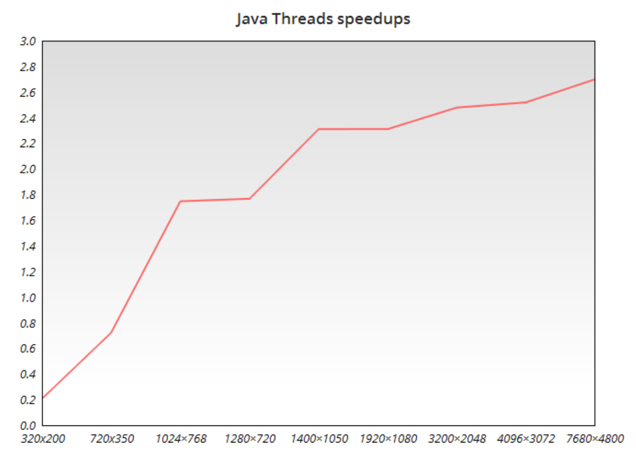

Figure 8: Speedups graph of Java parallel vs sequential approach.

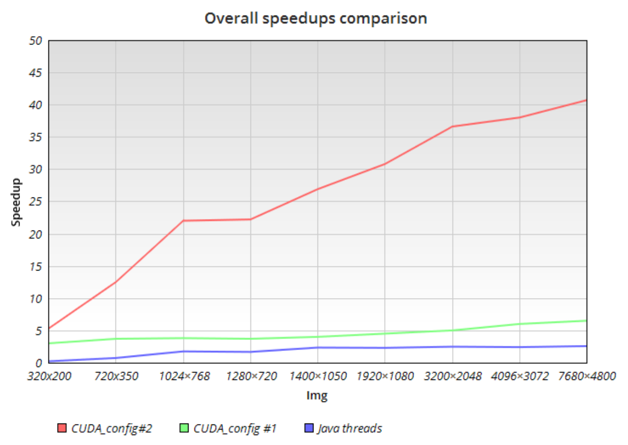

Figure 9: Overall speedup comparison.

6. Conclusions

Concluding, classic CPU parallel approach for this problem it's widely surpassed by a GPU parallel approach which shows a really low overhead for managing threads even with small images and a great gain in speedup that scales very well with gradually better performing graphics cards.
On a large image (7680 x 4800) we need 672 ms for Java Threads and 54 ms on CUDA with config 2 that's approx 12 value of speedup.

References

[1] *CUDA Toolkit Documentation v10.2.89*. 2019. URL: `https://docs.nvidia.com/cuda/`.

[2] Shreenidhi Sudhakar. *Histogram Equalization*. 2017. URL: `https://towardsdatascience.com/histogram-equalization-5d1013626e64`.