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 two sequential and two parallel version of a program that implements an image processing technique called histogram equalization.
The project is developed in C++ and Java for the sequential version and in CUDA and Java for the parallel one.

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, [5] 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:

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

where cdf is the cumulative distribution function and $round$ normalizes values between 0 and 255.

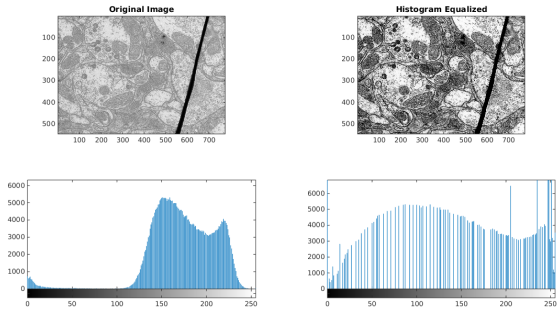


Figure 1: An example of an histogram equalization process.

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.
[2]So naturally, you can just split them into three separate channels, apply histogram equalization on each and then combine them back together but it doesnt exactly works very well.
Histogram equalization is a non-linear process.
Channel splitting and equalizing each channel separately is incorrect. Equalization involves intensity values of the image, not the color components.
So it needs to be applied in such a way that the intensity values are equalized without disturbing the color balance of the image.
So, the first step is to convert the color space of the image from RGB into one of the color spaces that separates intensity values from color components.
Some of the possible options are HSV/HLS, YUV, YCbCr, etc. YCbCr is preferred as it is designed for digital images.
Perform histogram equalization on the intensity plane Y then, convert the resultant YCbCr image back to RGB.
The conversion formula to YCbCr from RGB is defined as:

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

while the reconversion to RGB:

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

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

- cumulative 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 four basic functions: $convertToYCbCr$ takes the image and an histogram and convert the rgb image to YCbCr while assign values to the histogram; $calcCumulativeHist$ build the cumulative histogram, $equalizeHist$ takes the histogram and actually implements the equalization and $revertToRGB$ simply, as the name suggests, revert the image to RGB.

Figure 3: An example of RGB histogram equalization.

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.
In the sequential version the cumulative histogram it is calculated in the same function where equalization is done.
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 (plus a function to calculate cumulative histogram) have therefore been parallelized via Cuda and Java Threads .
We can observe that most of the computational load lies in the two conversion and reversion functions while the equalize and cumulative histogram calculation are simply a loop on 256 elements with a not very expensive calculation.
The function to calculate the cumulative histogram is also not very parallelizable because it's a logical sequential operation where the calculation of a term depends on the previous ones.
For this reason, it requires a lot of synchronization between threads and a parallel version is sometimes worse than a sequential version.
It was therefore implemented in a parallel version

only in the CUDA code using the parallel pattern scan.

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 conversion and the reconversion functions extremely quickly.
The idea for the parallel approach is that every pixel (or every couple/ 3 / 4 pixels) 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, four different kernels are set for each function and the appropriate allocations and copies are made from / to cpu and gpu.

Kernel execution time	
$convertToYCbCr$	$12.78ms$
$calcCumulativeHist$	$< 1ms$
$equalize$	$< 1ms$
$revertToRGB$	$9.98ms$

Table 1: kernel execution times on image of 7680 x 4800 pixels measured by Nsight Compute profiler .

We can consider the kernel $equalize$ and $revertToRGB$ as map functions wich take an array as input, apply a function for each element and write the results to an output array.
$convertToYCbCr$ is also a map but, in addition to the conversion, it also builds the histogram while $calcCumulativeHist$ is a scan function.
Below an example of block and grid definition.

```
// block and grid definition to define a thread
    for each image pixel

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

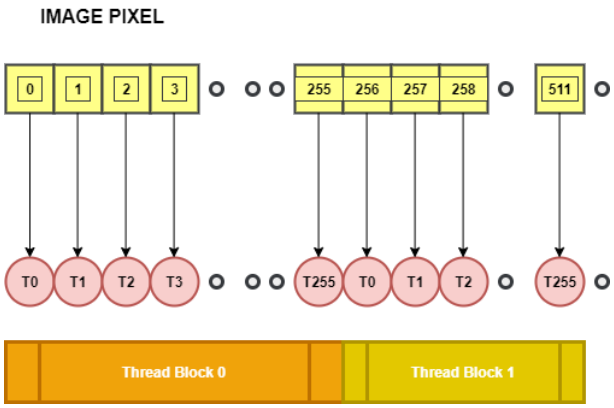


Figure 4: image's pixel management. case : 1 thread x 1 pixel

4.2. Java

Regarding parallel implementation in Java, a more classic approach has obviously been used.
First, 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 reconversion to RGB (as we can see in Figure 5).
As for the equalizing function, the histogram is simply divided (1D, vertically) 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.

The only shared resource accessed and modified by multiple threads is the histogram and, similar to the implementation in CUDA, each thread calculates its own local histogram and in the end they are added up to form the global histogram.
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];
                }
            }
```

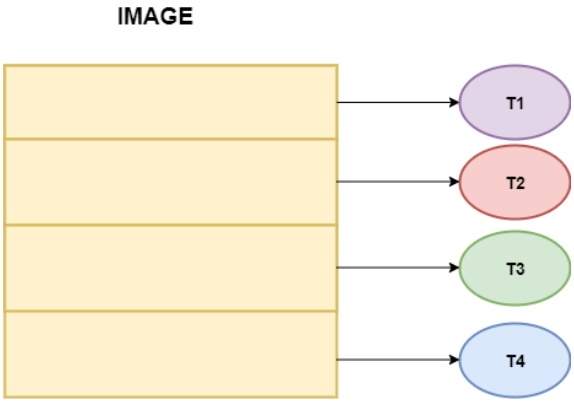


Figure 5: image subdivision scheme.

5. Tests and comparison

The tests are basically based on increasing the size of a pool of different images to be processed and calculating an average of the execution times.

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 CUDA code tests and profiling, 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 with a more recent architecture and many more computational resources.

5.2. CUDA code profiling

On configuration no.2, a simple profiling and performance analysis was carried out thanks to the Nvidia Nsight Compute tool.
[3]NVIDIA Nsight Compute is an interactive

kernel profiler for CUDA applications.
It provides detailed performance metrics and API debugging via a user interface and command line tool.
One parameter that has been monitored is **occupancy**; occupancy is the ratio of the number of active warps per multiprocessor.
Higher occupancy does not always result in higher performance, howewer low occupancy always result in overall performance degradation because reduce the ability to hide latencies.
As we can see in Figure 6, the developed code reaches a high occupancy (98.5% and 31.56 active warps per SM) and we can clearly see thanks to the profiling that with 256 threads per block we are in one of the maximum for warp occupancy so we have no reason to change the size of the blocks.
The performance impact, in the first kernel, of the shared memory histogram, was also investigated.
In fact, in $convertToYCbCr$, both a global histogram and a shared histogram shared between the threads of the block (as long as the global histogram is rebuilt in the end) can be used.
In particular, **atomic** add operations are performed on the histogram; atomic operations serialize simultaneous updates to a location, thus to improve performance the serialization should be as fast as possible and changing location on global memory is generally slow.

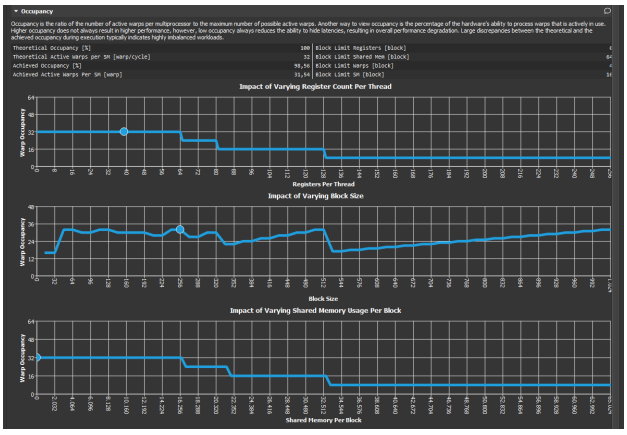
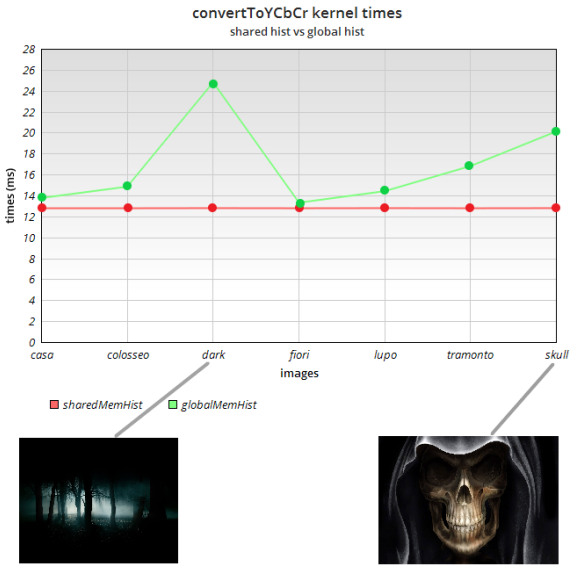


Figure 6: Occupancy stats executing kernels on 7680 x 4800 image.

The benfits of using private (shared) histogram is that the private copies can be accessed with much less contention and often at much lower latency.
Recent GPUs allow atomic operation to be performed in the last level cache, which is shared among all SMs, in fact, Maxwell architecture's shared atomics outperforms Kepler architecture's global atomics [4].
In our case, always through Nsight compute, the kernel execution times and statistics were measured on various images both in shared and global configuration.
The characteristic that is immediately noticeable in the image below, is that the shared version has a constant execution time when the images change while the global version is very variable.
It is no coincidence that images where execution times are much longer are images characterized by many dark pixels, in fact, as already introduced, we have similar intensities and threads must access the same positions in memory leading to a contention which, in the case of global memory, even leads to double the execution times.



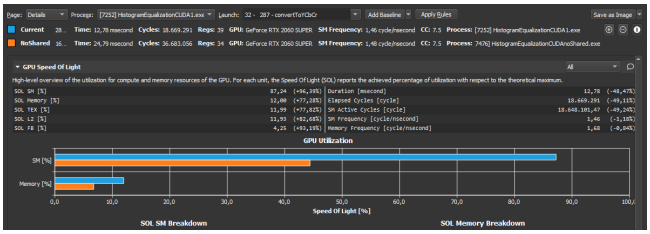


Figure 7: Profiling comparison of $convertToYCbCr$ kernel on dark image.

5.3. 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 with values that tend to increase as the size of the image increases (larger parallelizable part).

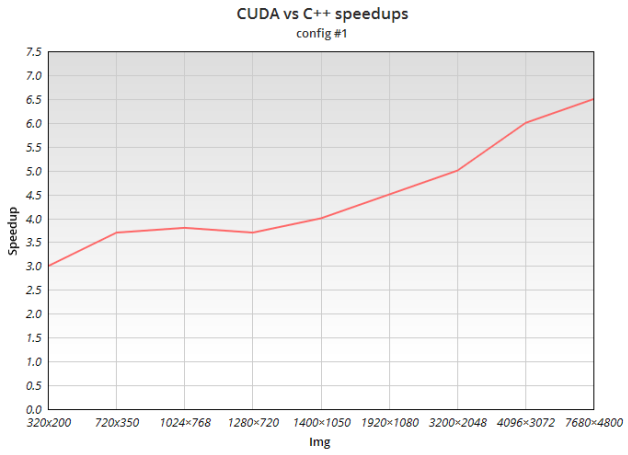


Figure 8: Speedup graph of CUDA version on Nvidia MX150 vs C++ sequential.

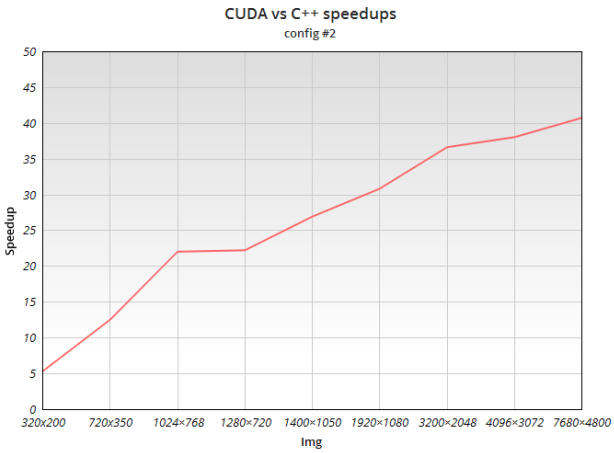


Figure 9: Speedup graph of CUDA version on Nvidia RTX2060 vs C++ sequential.

5.4. 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 11), 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.

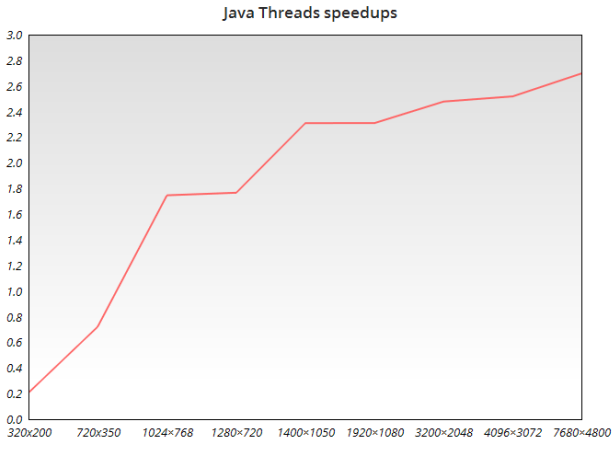


Figure 10: Speedups graph of Java parallel vs sequential approach.

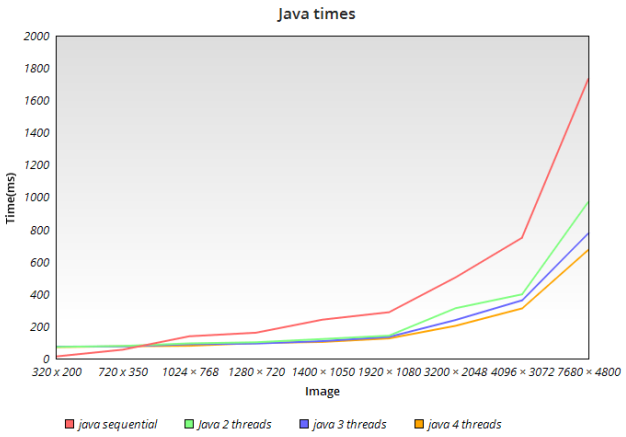


Figure 11: Java execution times varying no. threads.

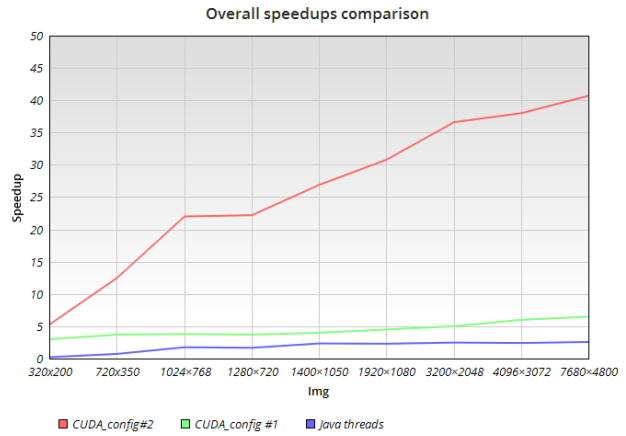


Figure 12: Overall speedup comparison.

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] PERPETUAL ENIGMA. *Histogram Equalization*. 2013. URL: https://prateekvjoshi.com/2013/11/22/histogram-equalization-of-rgb-images/.

[3] *NVIDIA Nsight Compute*. 2019. URL: https://developer.nvidia.com/nsight-compute-2019_5.

[4] Nikolay Sakharnykh. *GPU Pro Tip: Fast Histograms Using Shared Atomics on Maxwell*. 2015. URL: https://devblogs.nvidia.com/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/.

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