Implementation and Results of Image Equalization with OpenMP and CUDA

Alessandro Lemmo
E-mail address
alessandro.lemmo@stud.unifi.it

Abstract

In the following work two technologies have been studied, implemented and compared for the realization of parallel applications, such as OpenMP and CUDA. The first is a cross-platform API for the creation of parallel applications on shared memory systems. The second, acronym for Compute Unified Device Architecture is a platform for parallel processing, as well as an API, created by NVIDIA.

Two applications were created for the equalization of the histogram of images using OpenMP and CUDA respectively, which were then evaluated and compared in terms of performance, based on various factors such as the size of the images passed in input.

1. Introduction

Histogram equalization is a method in image processing of contrast adjustment using the image's histogram. 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. A disadvantage of the method is that it is indiscriminate. It may increase the contrast of background noise, while decreasing the usable signal.

Equalizing images can lead to unwanted effects if applied to images with a very low color depth. For example, in the case of 8-bit grayscale images (therefore with rather limited gray variations), the equalization could lead to unwanted artifacts as results, as it will further reduce the color depth. It works much better when applied to 16-bit grayscale images. On the other hand, if you consider color images you have two possibilities: the first is to perform equalization on the three channels of red, green and blue (RGB) indiscriminately, which however can lead to results with completely distorted color gradations compared to the original image. The second possibility is to go to convert the image into another color space, such as YCbCr, then go on to consider only the first component (Y) representing the luminance and going to perform the equalization of the histogram only on it. Then the image is converted back to RGB to obtain the final result. This is the method used in this project for both the CUDA application and the one developed with OpenMP.

CUDA (Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) model created by Nvidia. It allows to use a CUDA-enabled graphics processing unit (GPU) for general purpose processing. The CUDA platform is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels. The programming languages available in the CUDA development environment are extensions of the most common languages for writing programs. The main one is 'CUDA-C' (C with NVIDIA ex-

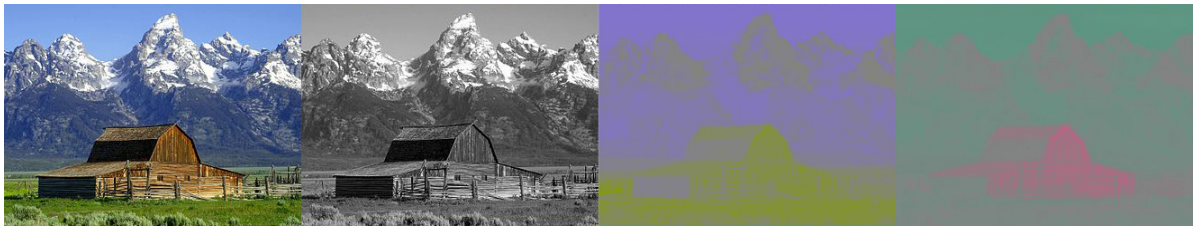

Figure 1. Scomposition of the image (at left) in the channels Y, Cb, Cr

tensions), which has also been used for the development of this project.

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

2. Histogram Equalization

On a theoretical level, considering a grayscale image, the formula for realizing the equalization of the histogram is as follows:

$$h(v) = round\left(\frac{cdf(v) - cdf_{min}}{(M \times N) - cdf_{min}} \times (L-1)\right)$$

where $cdf(v)$ is the cumulative distribution function of the histogram of the original image, M and N are the dimensions in pixels of the image and $L-1$ is the maximum value that can take a pixels.

If we consider color images, this formula will be applied only to the luminance component Y of the YCbCr decomposition. To convert an image from the RGB space to YCbCr we use the following formula:

$$\begin{cases} Y = R * 0.299000 + G * 0.587000 + B * 0.114000 \\ Cb = R * -0.168736 + G * -0.331264 + B * 0.5 + 128 \\ Cr = R * 0.5 + G * -0.418688 + B * -0.081312 + 128 \end{cases}$$

Subsequently, the equalization operation is performed on only the Y channel which will lead to the result shown in the figure:

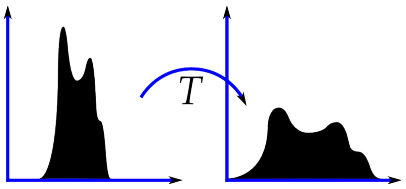

Figure 2. Histogram Equalization

Once the equalization has been performed on channel Y, it can be converted back to RGB with the following formula:

$$\begin{cases} R = Y + 1.4075 * (V - 128) \\ G = Y - 0.3455 * (U - 128) - (0.7169 * (V - 128)) \\ B = Y + 1.7790 * (U - 128) \end{cases}$$

3. Implementation

Now let's see how equalization is implemented with CUDA AND OpenMP.

3.1. CUDA

The work to be carried out can be seen as divided into three main phases: conversion of the color space from RGB to YCbCr, equalization of the Y channel and conversion back to RGB. The idea is therefore to parallelize these three operations by creating three kernels, where each of them performs one of the three operations.

The first kernel has the task of converting the color space and calculating the histogram of the Y channel which will then be passed to the second kernel for equalization. The conversion of the color space is performed through a cycle where each thread works on the single pixel according to the following scheme:

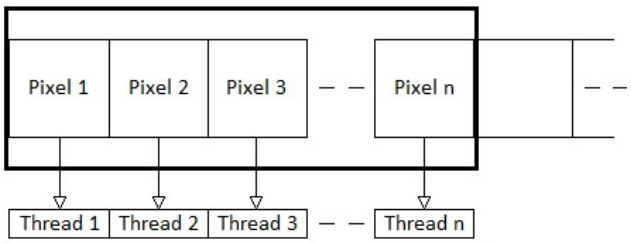

Figure 3. CUDA Threads

The window then scrolls until it covers all the pixels of the image. Following is the first kernel function in pseudocode:

```
function RGB_to_YCbCr()
{
    int idx = blockIdx.x * blockDim.x +
        threadIdx.x;

    for (int i = idx; i < width * height;
            i += blockDim.x * gridDim.x)
    {
        index = i * 3;
        int r = ptr_image[index + 0];
        int g = ptr_image[index + 1];
        int b = ptr_image[index + 2];

        //conversion to YCbCr
        int Y = ...
        int Cb = ...
        int Cr = ...

        ptr_image[index + 0] = Y;
        ptr_image[index + 1] = Cb;
        ptr_image[index + 2] = Cr;

        //Update the histogram of Y channel
        histogram[Y] += 1
    }

    __syncthreads();
}
```

The second kernel function deals with the equalization of the histogram that has been calculate for the only Y channel, taking as input the previously calculated cumulative distribution function. The pseudocode is shown below:

```
function equalize()
{
    int idx = blockIdx.x * blockDim.x
        + threadIdx.x;

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

The third kernel, which deals of the conversion back to RGB, is very similar in structure to the first kernel and therefore the pseudocode is not reported.

3.2. OpenMP

Even with the OpenMP API, the development of the histogram equalization software was based on the same logic, that is to divide the design into three phases and to parallelize them individually. The phases are similar to those implemented with CUDA, that is: conversion in the YCbCr color space and calculation of the histogram relative to the Y channel, equalization of the histogram and conversion back into RGB. In order to parallelize operations, OpenMP allows you to include the code in $\#pragma$ blocks in a very simple way. For example, to parallelize a for you use $\#pragma\ omp\ parallel\ for$. The code inside the parallelized blocks is similar to that of a sequential program. We see below the pseudocode relating to the first parallelized block:

```
function RGB_to_YCbCr()
{
    #pragma omp parallel default(shared)
    {
        #pragma omp for schedule(static)
        for each rows i in the image {
            for each cols j in the image {

                int R = image(i,j).R
                int G = image(i,j).G
                int B = image(i,j).B

                //conversion to YCbCr
                int Y = ...
                int Cb = ...
                int Cr = ...

                histogram[Y]++

                image(i,j).R = Y
                image(i, j).G = Cb
                image(i, j).B = Cr
            }
        }
    }
}
```

The second block of parallelized code deals with the equalization of the histogram, taking in input the cumulative distribution function as a parameter. The pseudocode is the following:

```
function equalize_histogram(){

#pragma omp parallel for
    for (int i = 1; i < 256; i++)
    {
        eq_histogram[i] = ((cdf[i] - cdf[0]) /
            (cols * rows - 1) * 255);
    }
}
```

As in the case of the CUDA development, the third parallel block related to the RGB conversion back is very similar in structure to the first block and therefore is not reported.

4. Results

Both programs have been tested with the same images in order to have a comparison on the same data. Seven images of different sizes were considered, ranging from about $300x200$ to $26000x9000$ pixels of resolution, all in color and all with over or under exposure, in order to have an appreciable equalization result. The table below shows the results in milliseconds of equalization on images with OpenMp and CUDA. To obtain each result, first the average is calculated between 12 executions, then the two times that deviate more from the average are removed, finally with the 10 values left the average is recalculated.

Image Dimension	OpenMP (msec)	CUDA (msec)
309 x 201	2.4	1.9
497 x 302	5.7	2.7
900 x 599	17.2	4.6
2000 x 1499	73.6	16.6
4000 x 2670	231.9	52.7
8000 x 6000	799.1	221.1
25816 x 8935	3945.2	1101,5

Table 1. Results in milliseconds with CUDA and OpenMP

It can be seen that processing on the GPU is always faster than on the CPU whatever the size of the image. More larger is the image and greater is the time advantage of processing on the GPU. This is also shown in the following graph:

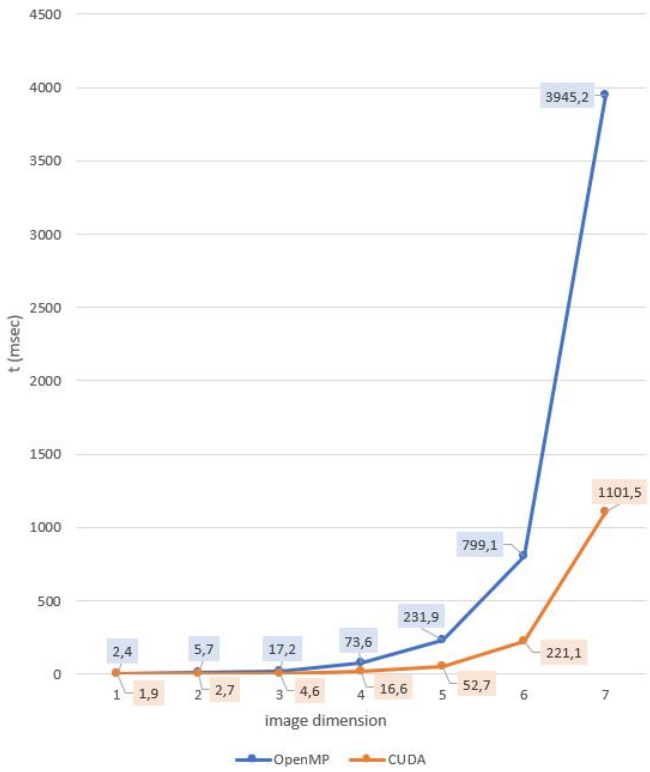

Figure 4. Time difference in execution between CUDA and OpenMP program

For this comparison, the CUDA and OpenMP executions were considered with the optimal number of threads, which is thirty-two in both cases. Further increasing the number of threads does not have a significant improvement in execution time. The variation of the times as the number of threads varies is shown in the following graphs respectively for CUDA and OpenMP.

4.1. Speedup

Speedup is a parameter for evaluating the performance of a parallel algorithm and is defined as

$$S_p = \frac{T_s}{T_p}$$

where T_s is the completation time of the sequential algorithm and T_p is the completation time of the parallel algorithm.

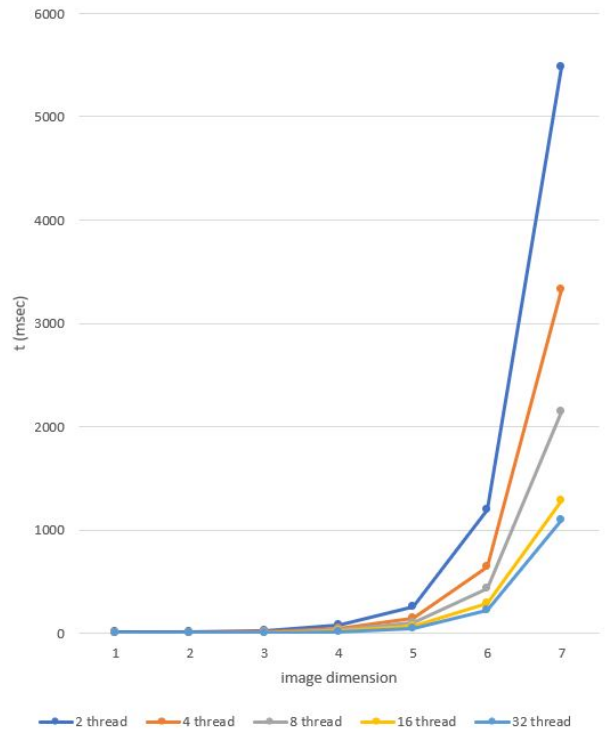

Figure 5. Time difference in execution in OpenMP program at the variation of the threads number

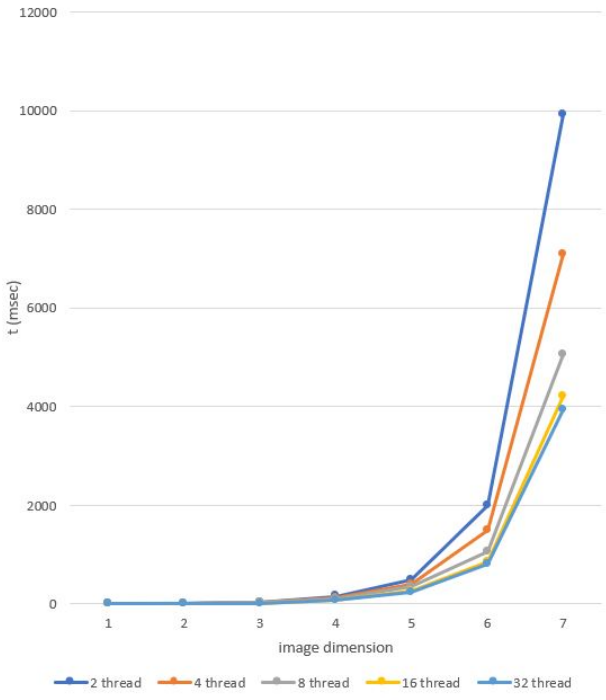

Figure 6. Time difference in execution in CUDA program at the variation of the threads number

In order to evaluate this parameter, a sequential version of the histogram equalization of a color image was also created. The functions used are similar to the parallel version implemented with OpenMP, with the only difference that you do not have the blocks marked with $\#pragma$ to parallelize the code. By comparing the execution speed of the sequential version with the parallel ones, the following result is obtained:

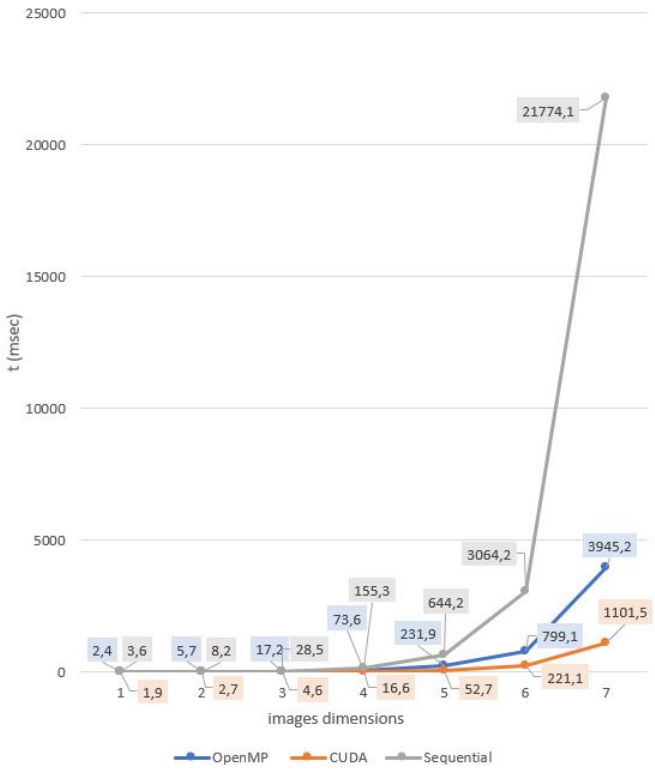

Figure 7. Time difference in execution between CUDA, OpenMP and sequential program

By calculating the speedup according to the formula written above, with respect to both parallel versions and with respect to the dimensions of the images, the results shown in the table are obtained:

The graph of the SpeedUp of the parallel versions in CUDA and in OpenMP is shown below.

It can be seen that the larger the image, the more the advantages of parallelization are important. In addition, parallel processing on the GPU achieves better performance than parallelization on the CPU.

Image Dimension	SpeedUp OpenMP (msec)	SpeedUp CUDA (msec)
309 x 201	1.5	1.89
497 x 302	1.44	3.04
900 x 599	1.66	6.2
2000 x 1499	2.11	9.36
4000 x 2670	2.78	12.22
8000 x 6000	3.83	13.86
25816 x 8935	5.52	19.77

Table 2. Results of SpeedUp in OpenMP and CUDA

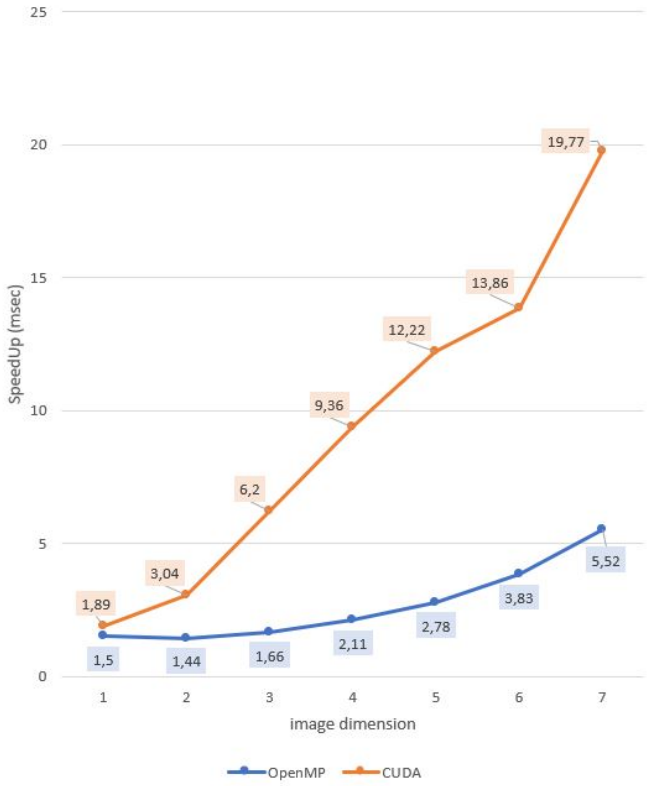

Figure 8. SpeedUp in OpenMP and CUDA

4.2. Examples

Below are some examples of equalization of images created with the software implemented. The proposed results are the same both in the calculation with both parallel modes and sequential modes. What changes is only the computation time. Two images are shown, one overexposed and one underexposed. In both cases it can be seen that the histogram is not well distributed over the whole interval. After equalization, on the other hand, a more uniform histogram is obtained and therefore a visually better result.

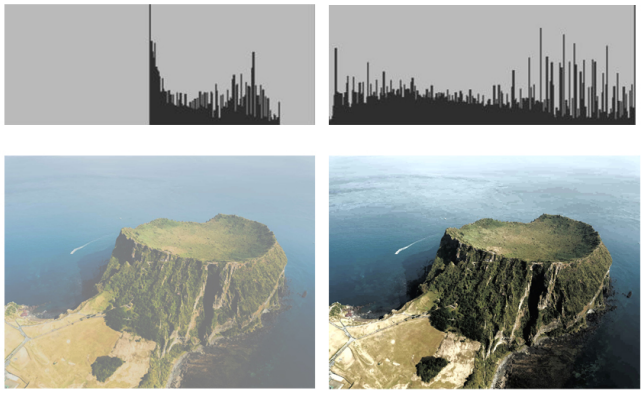

Figure 9. At the top the histograms of the image before and after equalization. At the bottom the resulting image before and after

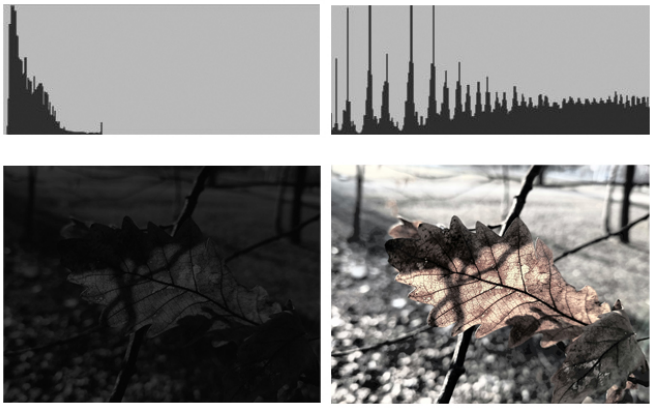

Figure 10. At the top the histograms of the image before and after equalization. At the bottom the resulting image before and after

5. Conclusion

The presented work allowed to evaluate the performance of an image processing software such as the equalization of the histogram, useful for improving the contrast of an image, comparing two types of parallel executions, on the CPU and on the GPU. The graphics card used for the CUDA implementation is an Nvidia GTX 1050 GeForce Max-Q. The results obtained show how the processing on the graphics card is the fastest and above all the more advantageous the more the image processed is larger.