University of Naples "Parthenope"
Department of Science and Technology
Master's Degree Course in Computer Science
(Machine Learning and Big Data)

High Performance Computing

Line Detection with Hough Transform: Parallel implementation of equalization with CUDA

Candidate
Renato Esposito
Matr. 0120000278

Academic Year 2023/2024

CONTENTS

1. Problem definition and analysis . . . 2
1.1 RGB to grayscale conversion . . . 2
1.2 Image resize . . . 2
1.3 Histogram Equalization . . . 3

2. Input and Output . . . 4
2.1 Input . . . 4
2.2 Output . . . 5

3. Algorithm description . . . 6
3.1 Hough transform . . . 6
3.1.1 Briefly . . . 6
3.1.2 In detail . . . 6
3.1.3 Pseudo code . . . 9
3.2 Equalization . . . 10
3.2.1 Pseudo code . . . 10
3.2.2 Equalization on GPU and Kernel settings . . . 10
3.2.2.1 GPU pseudo code without SM . . . 12
3.2.2.2 GPU pseudo code with SM . . . 13
3.2.2.3 Pseudo code for Kernel settings . . . 14

4. Implemented routines . . . 15
4.1 Conversion from RGB to grayscale . . . 15
4.1.1 CPU . . . 15
4.1.2 GPU . . . 15
4.2 Resize of the image . . . 16
4.2.1 CPU . . . 16

4.2.2 GPU 16
4.3 Histogram calculation 16
4.4 Equalization 17
4.4.1 Library equalization 17
4.4.2 Equalization on CPU 18
4.4.3 Kernel setting function 18
4.4.4 Equalization on GPU without shared memory 19
4.4.5 Equalization on GPU with shared memory 19
4.5 Hough transform 20
4.5.1 Hough transform on CPU 20
4.5.2 Hough transform on GPU 21

5. Performance analysis 22
5.1 Results 22

6. Usage examples 24

Appendix 25

A. Open Source Computer Vision Library 26

B. CUDA 27

C. OpenCV CUDA module 28

D. Code 29

1. PROBLEM DEFINITION AND ANALYSIS

The goal of this project is to develop software that performs the Hough transform to detect lines within an image. Like any operation on images, it requires the use of a series of preprocessing operations, which play a key role in preparing images for line detection algorithms such as Hough. The preprocessing operations are:

- RGB to grayscale conversion.
- Image resize.
- Histogram equalization.

1.1 RGB to grayscale conversion

The first important step is the conversion from RGB to grayscale, this because color images contain more complex, and often unnecessary, information for detecting lines, the main reasons behind the conversion are:

- **Computational complexity**: processing color images requires more computational resources than grayscale images.
- **Redundant information**: additional color channels can introduce redundant information that can create a lot of problems for Hough's algorithm, this could lead to inaccurate results in line detection.

1.2 Image resize

This reduces computational complexity and improves the efficiency of line detection. High-resolution images require more time and resources for processing, slowing down the entire line detection process. *The problem* with image resizing,

however, is that, in some cases, resizing inappropriately can lead to the loss of details that are useful to Hough's algorithm, this cannot be addressed in a general way, but we need to have knowledge of the input images.

1.3 Histogram Equalization

The histogram of an image is a graphical representation of the distribution of pixel light intensities within an image, in other words, shows how many times each intensity level appears in the image. Histogram equalization is a process used to improve the contrast of an image. This is done by redistributing pixel intensities more evenly over the entire range of available intensities. The goal is to obtain an image with a wider dynamic range so that regions of different light intensities are more easily distinguishable. This operation is useful to improve the visibility of lines in the image by increasing the contrast between different gray intensities. The problems without equalization are:

- **Insufficient contrast**: images with insufficient contrast can make line detection difficult, especially with complex backgrounds or details.
- **Sensitivity to lighting changes**: Hough's algorithm can be sensitive to lighting changes, compromising the robustness of the application.

All operations were done on CPU and GPU in order to compare performance and understand what the advantages are in using GPUs. To be able to work on the images on the CPU the **OpenCV library** was used, while for the GPU the **OpenCV CUDA** module was used, and in addition, a **CUDA Kernel** code was proposed for histogram equalization. For more details on OpenCV and CUDA see appendix.

2. INPUT AND OUTPUT

In this section I will describe the software input and parameters required during execution, I will also explain the expected output and the form in which the data must be provided.

2.1 Input

The proposed software requires a **color image as input** and nothing else, in fact during execution no interactivity with the software is required, but it will autonomously obtain all the information that it needs. In fact, the algorithm will calculate the best kernel configuration based on the dimensions of the image, which are modified and brought to a size of 600x600 pixels.

Fig. 2.1: Example of input image of size 868x600

2.2 Output

The software returns two files as output:

1. **The original image**, but in grayscale and with a size of 600x600 pixels, with the identified lines highlighted.

2. A **.txt file** that summarizes the algorithm's performance.

Fig. 2.2: Example of output image of size 600x600

3. ALGORITHM DESCRIPTION

In this chapter I will explain **Hough's algorithm**[1] for finding lines within the image, I will also provide a description of the **calculation and equalization** of the histogram. I will conclude with the analysis of the **CUDA Kernel** configuration.

3.1 Hough transform

3.1.1 Briefly

The Hough Transform was introduced by Paul Hough in 1962 as a tool for automatically identifying straight lines in images. It is also possible to use the Hough transform to find other shapes. A so-called generalized Hough transform has been proposed because it can be applied whenever the shape we are looking for can be expressed with a **parametric function**, i.e. characterized by parameters. The main idea is to exploit the edges that we find with other algorithms like **Canny** and understand if the edge pixels that have been detected are arranged along the shape we are looking for.

3.1.2 In detail

If we consider the **canonical form** of a straight line its equation will be:

$$y = mx + b \tag{3.1}$$

where:

- m represents the slope of the straight line.
- b represents the ordinate at the origin that is, the point at which the line intersects the ordinate axis (y) when x is equal to zero.

Any straight line is completely specified by the value of **parameters (m,b)**.
It is now possible to consider a *transformation from the image space to the parameter space* (m,b), that is, the Hough's idea is to introduce parameter space where:

- The image became a point in this new space.
- A point in image space corresponds to a line in parameter space.

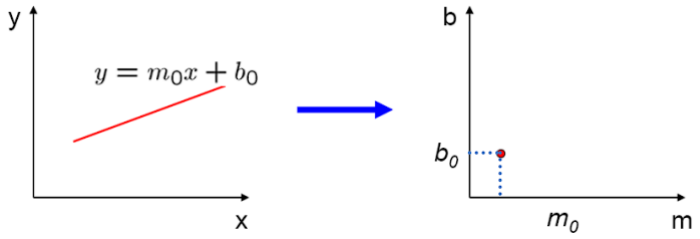

Fig. 3.1: Transformation from image space to parameter space

Here we can write the equation of the line in function of b and m:

$$b = -mx + y \tag{3.2}$$

Let's consider two points in image space, since infinite lines pass through a point we use the values of m,b of these to describe the line in the parameter space that represents the point x,y.
Applying this reasoning for both points we obtain two lines in the parameter space, these intersect at a point with coordinates m_0, b_0 which describes a line in the image space, **this is the line on which the points lie**.

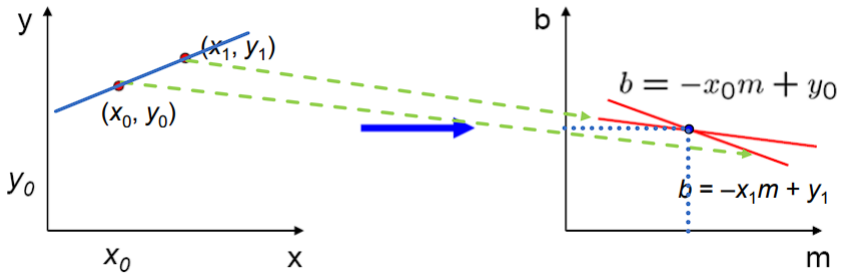

Fig. 3.2: Transformation from image space to parameter space

To find the edge points, i.e. the *(x,y)* pairs, we can use an edge detection algorithm such as the Canny algorithm. The **problem** is that using the canonical representation we can have that the vertical lines have $m = \inf$ and this could be a problem to handle in parameter space, which is why we use **Hesse normal form representation**:

$$\rho = x \cos\theta + y \sin\theta \tag{3.3}$$

In this case the parameter space will be in a function of ρ and θ, but the reasoning remains the same. However, the combinations of ρ and θ no longer describe a straight line but a sinusoid, but this doesn't change the effectiveness of the algorithm.

It is important to remember that real images are affected by noise so the edge points may not be perfectly aligned, therefore after finding the edge points, it's necessary to quantize the parameter space so as to consider only a range of values ρ and θ. The idea is to overlap a grid on the parameter space obtaining a third space called vote space, in which each cell is an accumulator. For each point in the image space we have a sinusoid that adds a "vote" in the cells corresponding to the parameters of the lines that pass through that point When all the points have "voted", the cells containing a number of votes above a certain threshold correspond to the lines in the image space. We find the maximum rating at the intersection point and therefore **with a high probability these parameters correspond to the most rated line and therefore to a line within the image space**, but but I could still find more peaks (if 3 lines pass through a cell I give a score of 3 but it doesn't mean that they pass through the same point of the cell, perhaps 1 in the center, one on the right and one on the left etc.), we can solve this problem of **votes spurious** using a threshold.

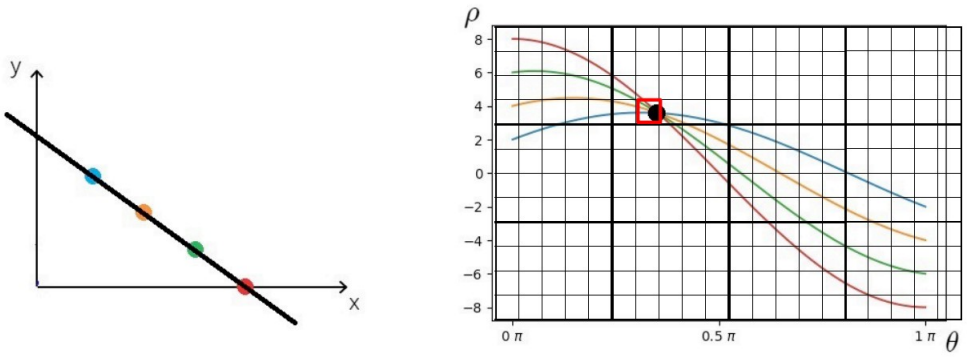

Fig. 3.3: Transformation from image space to parameter space

3.1.3 Pseudo code

Algorithm 1 Hough Algorithm for Line Detection

1: *Initialize the accumulator* H
2: *Apply Canny's algorithm to locate edge points*
3: **for all** *edge points* (x, y) **do**
4: **for** *each angle* $\theta = 0$ **to** 180 **do**
5: $p \leftarrow x \cdot \cos(\theta) + y \cdot \sin(\theta)$
6: $H(\rho, \theta) \leftarrow H(\rho, \theta) + 1$
7: **end for**
8: **end for**
9: *Cells* $H(\rho, \theta)$ *with a value greater than a threshold correspond to straight lines in the image*

3.2 Equalization

Histogram equalization is an image processing process whose goal is to improve the contrast and distribution of pixel intensities in an image.
The main goal is to make the image histogram more uniform, distributing the pixel intensities over a wider range of values. This sharpens details and improves the visibility of image features.
The idea is that for each pixel in the input image, its intensity value is used to obtain the corresponding normalized value from the cumulative histogram.

3.2.1 Pseudo code

Algorithm 2 Histogram calculation

```
1: int area ← image.rows × image.cols
2: int ngraylevel ← 256
3: for i da 0 a image.rows - 1 do
4:     for j da 0 a image.cols - 1 do
5:         pixel_value ← image.at<uchar>(i, j)
6:         equalizedImage.at<uchar>(i, j) ← (double(ngraylevel)/area) × cumulative_hist[pixel_value]
7:     end for
8: end for
9: return equalizedImage
```

3.2.2 Equalization on GPU and Kernel settings

For equalization on the GPU the idea was to have the two-dimensional blocks of the GPU with threads organized in a two-dimensional grid, in this way the image is divided into blocks of similar size to those of the thread grid, finally each block can be assigned to a specific portion of the image. This arrangement is commonly used for image operations and offers several advantages because images are naturally two-dimensional, and organizing threads in a two-dimensional grid allows them to parallelize processing across both dimensions of the image simultaneously. This maximizes the efficiency of parallelism.
Two versions of the CUDA Kernel code are available, one in which the histogram

is placed in global memory, another in which it is placed in shared memory. From the tests it appears that for images of size 600x600 without shared memory we obtain a better result even if only slightly, probably for very large images the shared memory could lead to better results.

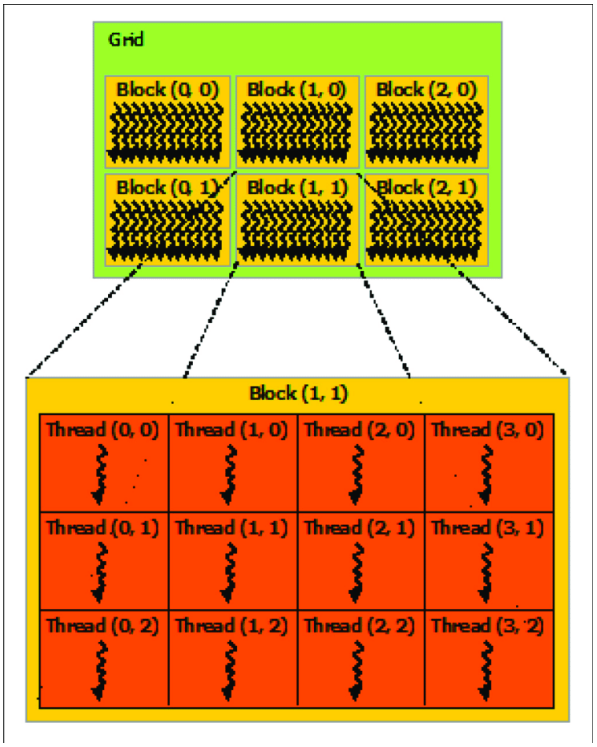

Fig. 3.4: CUDA programming: Grid of thread blocks

3.2.2.1 GPU pseudo code without SM

Algorithm 3 CUDA Kernel for Histogram Equalization without SM

procedure EQUALIZEHISTCUDA(input, output, cumulative_hist, cols, rows)
    $nGrayLevels \leftarrow 256$
    $area \leftarrow \text{cols} \times \text{rows}$
    $i \leftarrow \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y}$
    $j \leftarrow \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$
    if $i < \text{rows}$ **and** $j < \text{cols}$ **then**
        $\text{index} \leftarrow i \times \text{cols} + j$
        $\text{pixelValue} \leftarrow \text{input[index]}$
        $\text{output[index]} \leftarrow \text{uchar}\left(\frac{nGrayLevels}{\text{area}} \times \text{cumulative\_hist[pixelValue]}\right)$
    end if
end procedure

3.2.2.2 GPU pseudo code with SM

To copy data into shared memory, the idea is to exploit 256 threads for each block, if available, in such a way as to assign each thread an element of the histogram (255 gray level). If not available, we divide the histogram into portions and assign them to the maximum number of threads for each block.

Algorithm 4 CUDA Kernel with Shared Memory Histogram Equalization

1: **procedure** EQUALIZEHISTCUDASM(input, output, cumulative_hist, cols, rows)
2: $nGrayLevels \leftarrow 256$
3: $area \leftarrow \text{cols} \times \text{rows}$
4: $i \leftarrow \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y}$
5: $j \leftarrow \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}$
6: $\text{InBlockThreadID} \leftarrow \text{threadIdx.x} + \text{blockDim.x} \times \text{threadIdx.y}$
7: $elements\_per\_thread \leftarrow (\frac{256}{\text{blockDim.x} \times \text{blockDim.y}} > 1)?(\frac{256}{\text{blockDim.x} \times \text{blockDim.y}}) : 1$
8: $start\_index \leftarrow \text{InBlockThreadID} \times \text{elements\_per\_thread}$
9: **for** $i \leftarrow 0$ to $elements\_per\_thread - 1$ **do**
10: $index \leftarrow \text{start\_index} + i$
11: **if** $index < 256$ **then**
12: $\text{shared\_cumulative\_hist}[index] \leftarrow \text{cumulative\_hist}[index]$
13: **end if**
14: **end for**
15: **if** $i < \text{rows}$ **and** $j < \text{cols}$ **then**
16: $index \leftarrow i \times \text{cols} + j$
17: $\text{pixelValue} \leftarrow \text{input}[index]$
18: $\text{output}[index] \leftarrow \text{uchar}\left(\frac{nGrayLevels}{\text{area}} \times \text{shared\_cumulative\_hist}[\text{pixelValue}]\right)$
19: **end if**
20: **end procedure**

3.2.2.3 Pseudo code for Kernel settings

This function calculates the configuration of the CUDA grid and the block sizes to effectively parallelize the processing of an image on a GPU based on the size of the image and the number of threads available on GPU, trying to exploit the maximum number of threads available .

Algorithm 5 Calculate CUDA Grid Configuration

```
1: procedure CalcCudaGrid(numBlocks, nThreadPerBlocco, rows, cols)
2:     cudaDeviceProp prop;
3:     cudaGetDeviceProperties(&prop, 0);
4:     // Max thread's num. x block of the GPU
5:     int maxThreadsPerBlock ← prop.maxThreadsPerBlock;
6:     nThreadPerBlocco.x ← min(cols, int(√maxThreadsPerBlock));
7:     nThreadPerBlocco.y ← min(rows, maxThreadsPerBlock/nThreadPerBlocco.x);
8:     numBlocks.x ← (cols + nThreadPerBlocco.x − 1)/nThreadPerBlocco.x;
9:     numBlocks.y ← (rows + nThreadPerBlocco.y − 1)/nThreadPerBlocco.y;
10: end procedure
```

4. IMPLEMENTED ROUTINES

In this chapter I will list and explain the main routines that I have implemented.

4.1 Conversion from RGB to grayscale

For grayscale conversion I used the functions of OpenCV and the CUDA module for running on GPU.

4.1.1 CPU

The function `cv::Mat cpu_RGBtoGRAYSCALE(cv::Mat, float*)` takes care of performing the conversion on the CPU. The function takes as input a color image of type cv::Mat, which is the OpenCV image type, and a pointer to a float that indicates the execution time of the routine. The output is the grayscale image.
The OpenCV `cv::cvtColor(in,out,cv::COLOR_BGR2GRAY)` function is used to perform the conversion operation in this routine. The function converts "*in*" to grayscale and the result will be stored in "*out*".

4.1.2 GPU

The function
`cv::cuda::GpuMat gpu_RGBtoGRAYSCALE(cv::cuda::GpuMat, cudaEvent_t*, float&)`
takes care of performing the conversion on the GPU. The function takes as input a color image of type cv::cuda::GpuMat, which is the OpenCV::cuda image type, a pointer to a cudaEvenet which is the cuda event to track to store time. The last argument is a reference to the memory area in which to store the elapsed time. The outputs is the grayscale image on GPU, if it is necessary to view or store it,

it's necessary to download it with the *.download()* method of the *GpuMat* class. The OpenCV::CUDA `cv::cuda::cvtColor(gpuImage,out,cv::COLOR_BGR2GRAY)` function is used to perform the conversion operation in this routine. The function converts "*gpuImage*" to grayscale and the result will be stored in "*out*".

4.2 Resize of the image

4.2.1 CPU

The function `cv::Mat cpu_resizeImage(cv::Mat,cv::Size, float*)` takes care of performing the resize on the CPU, this operation makes the image 600x600. The function takes as input a color image of type cv::Mat, a cv::Size that is the size that we want and, finally, a pointer to a float that indicates the execution time of the routine. The output is the resized grayscale image.
The OpenCV `cv::resize(in, out, cv::Size size)` function is used to perform the operation. This function transforms the image "*in*" into 600x600 and stores it in "*out*".

4.2.2 GPU

To do the same operation on GPU we use the function
`cv::cuda::GpuMat gpu_resizeImage(cv::cuda::GpuMat, cv::Size size, cudaEvent_t*, float&)`.
As input we provide the image to resize, the size and like all other functions on GPU, the event to track and the reference to the time elapsed. The output is the resized grayscale image on GPU.
This function to perform the resize operation uses the CUDA module function `cv::cuda::resize(gpuImage, out, outputSize)`, that scales the image "*gpuImage*" into 600x600, given by *outputSize*, and stores it in "*out*".

4.3 Histogram calculation

Histogram calculation involves reading and updating counters for each pixel intensity value in the image. This can create dependencies between iterations, especially if we use a shared data structure to store the histogram. For this reason I chose to calculate it only on the CPU by using the function:

`void calcCumHist(cv::Mat image, int *cumHist)`, and then moved to GPU.
The function takes two parameters as input: *cv::Mat image* and *int *cumHist*. The first represents the grayscale image on which to calculate the cumulative histogram, while the second is a pointer to a vector in which the cumulative histogram will be stored.
Inside the function, some local variables are declared. *nBins* represents the number of bins in the histogram, usually 256 for a grayscale image. *sum* is a variable used to accumulate the histogram values as it is calculated. A local vector *hist* is also declared which will be used to store the histogram of the image.
Before starting the calculation, the histogram is initialized with zeros using the *memset* function. Next, a double for loop loops through each pixel of the image. Inside the loop, the grayscale pixel value is obtained, and the histogram is updated by incrementing the counter corresponding to the current pixel value.
After calculating the image histogram, another for loop passes through the histogram vector and calculates the **cumulative histogram**. The *sum* variable is updated with the current value of the histogram, and this cumulative sum is stored in the *cumHist* vector, that is the output of the function. The code is in the appendix

4.4 Equalization

Both CPU and GPU code were written for equalization, and a function that uses the OpenCV and OpenCV::CUDA library routines was also provided to compare performance. For GPUs, **CUDA Kernel code** has been implemented both with and without shared memories.

4.4.1 Library equalization

For equalization using OpenCV library and CUDA module was written the function:
`void EqualizationByRoutine(cv::cuda::GpuMat, cv::Mat, cudaEvent_t*, float&, float *)`.
This function takes as input the GPU image, the CPU image and their respective timing arguments and calculates the equalization using:

- `cv::equalizeHist(cpuImg, opencvEqualizedImg)`: this function takes the image as input, performs the equalization on the CPU and stores the equalized output in *"opencvEqualizedImg"*.
  The histogram calculation is included in this function.

- `cv::cuda::equalizeHist(gpuImg, gpuEqualizedImage)`: this function takes the image as input, performs the equalization on the GPU and stores the equalized output in *"gpuEqualizedImage"*.
  The histogram calculation is included in this function.

4.4.2 Equalization on CPU

My CPU equalization implementation is represented by the function:
`cv::Mat cpu_equalization(cv::Mat, int*, float*)` which takes as input an image, cumulative histogram and returns a pointer to the time necessary for the operation and the equalized image.
The function has a double for loop, inside the loop, the pixel value of the original image is used to calculate the new pixel value in the equalized image, following the cumulative histogram transformation, that, in pseudo code, is:

1: pixelValue $\leftarrow$ originalImage[$index$]
2: output[$index$] $\leftarrow$ uchar $\left(\frac{nGrayLevels}{\text{area}} \times \text{cumulative\_hist}[\text{pixelValue}]\right)$

4.4.3 Kernel setting function

To be able to define the configuration of the blocks and the kernel based on the characteristics of the GPU and the image I created the function:
`void CalcCudaGrid(dim3 &numBlocks, dim3 &nThreadPerBlocco, int rows, int cols)`
The function starts by getting the properties of the CUDA device, these properties include information about the capacity of the device, such as the maximum number of threads per block. Next, the function calculates the maximum number of threads per block, this value is used to define the maximum size of the thread grid on each block.
The number of threads per block, along the x- and y-axis, is set as the minimum

of the number of columns (or rows for y) and the square root of the GPU's maximum number of threads per block. In this way we either exploit the maximum number of GPU threads or assign exactly one thread to each pixel of the image. Finally the total number of blocks needed to cover the entire image are calculated, this is done by dividing the size of the image by the size of the thread grid in order to completely cover the image.

4.4.4 Equalization on GPU without shared memory

My GPU equalization CUDA Kernel is represented by the function:

```
__global__ void equalizeHistCUDA(unsigned char* in, unsigned char* out,
int* hist, int cols, int rows)
```

where we use the *unsigned char* type to work with images on the GPU since it's not possible to work with complex types. The function takes as input an input image, the output image, a pointer to a vector of integers which is the cumulative histogram, the number of columns and rows. Stores the equalized image in *out*.
Equalization on GPU without SM follows exactly the same logic as equalization on CPU, obviously the routine does not have for loops but calculates the threads indexes, verifies that the indexes work on correct portions of the image and then applies the equalization formula.

4.4.5 Equalization on GPU with shared memory

My GPU equalization with SM CUDA Kernel is represented by the function:

```
__global__ void equalizeHistCUDASM(unsigned char* in, unsigned char* out,
int* hist, int cols, int rows)
```

which is the same to the function without shared memory, but in this case adds the management of the copy of the cumulative histogram in shared memory.
A shared array is declared in the shared memory of the block, this array will be used to temporarily store the cumulative histogram shared between threads within the same block.
The number of cumulative histogram elements assigned to each thread is calculated, this division is designed to ensure equal distribution of elements between

threads. The index of the histogram from which each thread must start working is calculated based on its identifier and the number of elements assigned.
Begins a loop that goes through the items assigned to each thread. Inside the loop, the absolute index of the cumulative histogram that the thread must process is calculated, if the calculated index is within the range of the cumulative histogram, the thread copies the corresponding value from the global cumulative histogram into the shared memory of the block.

4.5 Hough transform

The Hough transform was implemented using the functions made available by the OpenCV library.

4.5.1 Hough transform on CPU

The routine that performs the operation on CPU is:

```
cv::Mat cpu_HoughTransformLine(cv::Mat ,float *)
```

It takes an image as input and returns its Hough transform and a pointer to the time elapsed.
My routine uses the OpenCV functions in order to do the trasform:

- `cv::GaussianBlur(input, gass, cv::Size(5, 5), 0, 0)`: to be able to apply a Gaussian filter and clean the image from noise. *Input* is the input image, *gass* is the output image and the third argument is the size of the Kernel, *0* and *0* are the *sigma X* and *sigma Y* parameters, that will be automatically calculated by OpenCV.

- `cv::Canny(gass, can, 80, 140)`: to perform Canny so as to return the edge points of the image. *Gass* is the input image, *can* is the output image, *80* and *140* are the thresholds used by Canny to determine which pixels should be considered edges. They must be chosen based on the image.

- `cv::HoughLines(can, lines, 1, CV_PI / 180, 147)`: locates straight lines in a grayscale image previously obtained by Canny edge detection. *Can* is the input image, *lines* is a vector that will contain the parameters of the lines detected by the Hough transform algorithm, *1* is the accumulator radius

resolution, in this case, every pixel in the image will have a match in the accumulator, *CV PI / 180* is the accumulator angle resolution, in this case, the angle is represented in radians and *147* that is a theshold. Only lines that have a number of votes greater than this threshold will be accepted as detected lines.

At the end, the `cv::Mat DrawLines(cv::Mat HoughImage, std::vector<cv::Vec2f> lines)` function is called to be able to draw the identified lines (the informations are in the lines vector). Finally, the output image is returned with the lines highlighted.

4.5.2 Hough transform on GPU

For the Hough transform on GPU there is the function:

```
cv::Mat GPU_HoughTransformLine(cv::cuda::GpuMat input,
udaEvent_t* timer, float& elapsedTime)
```

it takes an image as input, the event to be tracked and a reference to the time elapsed. Returns the Hough transform done by GPU and a pointer to the time elapsed, the output this time is also downloaded from the GPU and transferred to the CPU.
The function on GPU is exactly like its CPU counterpart, but obviously in this case it makes use of the functions of the OpenCV CUDA module, which are:

- `cv::Ptr<cv::cuda::Filter> gaussianFilter = cv::cuda::createGaussianFilter(CV_8U, CV_8U, cv::Size(5,5),0)`: to create a gaussian filter and after we apply the filter by using `gaussianFilter->apply(input,gass)`.
- `cv::Ptr<cv::cuda::CannyEdgeDetector> canny = cv::cuda::createCannyEdgeDetector(80, 140, 3, false)`: to create a Canny edge detector and after we can apply the algorithm by using `canny->detect(gass, can)`.
- `cv::Ptr<cv::cuda::HoughLinesDetector> hough = cv::cuda::createHoughLinesDetector(1.0, CV_PI / 180, 147)`: to create a Hough detector for line and after we can apply the algorithm by using `hough->detect(can,lines)`.

Once this is done, we can download the image of the GPU and draw the lines with the same method used for the CPU:
`cv::Mat DrawLines(cv::Mat HoughImage, std::vector<cv::Vec2f> lines)`.

5. PERFORMANCE ANALYSIS

About the performance of the software, after each execution a *.txt* file is generated which contains all the execution times of the routines and a summary of the kernel settings.
In this section I will show the output of an execution accompanied by a small explanation of the results.

5.1 Results

The table 5.1 shows the number of blocks, the number of threads and the size of the input image.
The number of blocks and threads refers to the configuration of the kernel that runs the CUDA kernel and in any case is based on the size of the resized image. Regarding the configuration used by the CUDA module in OpenCV, the approach that the library uses is **unknown** for who uses it.

	Value
Image size	868x600
Number of blocks	19x19
Number of threads x blocks	32x32

Tab. 5.1: Kernel settings

The table 5.2 shows execution times based on table parameters 5.1, where the "-" means that it isn't possible to obtain a time because the execution isn't on that "device" (CPU or GPU).
If we focus on the results, we can see that for the operations of: conversion from RGB to grayscale, resize, equalization with and without shared memory, we get significantly better results with the GPU.
The table row "*Equalization using OpenCV and Cuda routines*" results are altered due to the calculation of the histogram which is not considered in the functions that I've implemented, however in the library routines it is calculated autonomously and there is no way to indicate a ready one.
About the comparison between the use of shared memory we can see that in this test we obtain results slightly in favor of shared memory-free, but probably by increasing the size of the image the approach with shared memory could be the winner.

Action	**CPU Time (msec)**	**GPU Time (msec)**
RGB to Grayscale	453.998962	43.228065
Resize	38.692657	12.916992
Equalization using OpenCV and Cuda routines	18.760418	213.928223
Equalization on CPU by myself	13.152274	-
Equalization on GPU without SM by myself	-	0.153696
Equalization on GPU with SM by myself	-	0.164576
Hough transform for lines	362.820129	93.103554

Tab. 5.2: Results

The times of the CUDA routines also contain the cost of passing data between the CPU and GPU.

6. USAGE EXAMPLES

To use the software, you will need to use the terminal to go to the folder where the project files are located and execute the command:

```
$ ./compile.sh image.cu out
```

where

- *compile.sh* is the shell script that allows you to compile the program correctly.
- *image.cu* is the source code to compile.
- *out* is the executable.

It is not possible to indicate the input photo, but it must be changed from the source code. A possible solution is to delete the photo from the project and save another one as "*foto.jpg*". Below is an example of input and output photos.
The dimensions of the images in this report do not respect the actual dimensions, in fact the input image is larger, in all directions, than the output one.

Fig. 6.1: Input and output image

APPENDIX

A. OPEN SOURCE COMPUTER VISION LIBRARY

Open Source Computer Vision Library (**OpenCV**) [2] is an open source computer vision and machine learning software library written natively in C++. The library has more than 2500 optimized algorithms that can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, extract 3D models of objects etc. It has C++, Python, Java and MATLAB interfaces and supports Windows, Linux, Android and Mac OS. Currently the CUDA and OpenCL interfaces are already available, but still under development.

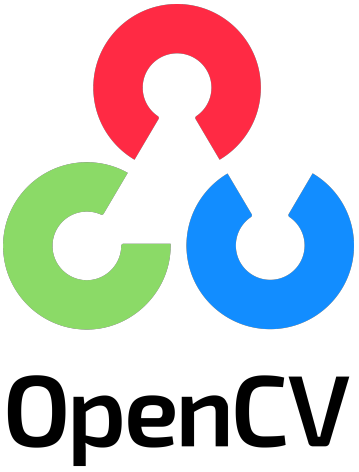

Fig. A.1: OpenCV logo

B. CUDA

Compute Unified Device Architecture (**CUDA**) [3] is a parallel computing platform and programming model created by NVIDIA that helps developers speed up their applications by harnessing the power of GPU accelerators. In other words, CUDA extends the functionality of GPUs beyond traditional use for graphics, allowing them to perform general computing tasks. CUDA is written primarily in C and C++ and uses an extended version of the C programming language, called CUDA C, which includes extensions to take advantage of GPU features. CUDA introduces a parallel programming model where a set of threads can run simultaneously on the GPU. These threads are organized into blocks, and the blocks are organized into grids.

Fig. B.1: CUDA logo

C. OPENCV CUDA MODULE

The **OpenCV CUDA module** is a set of classes and functions to utilize CUDA computational capabilities. It is implemented using NVIDIA* CUDA* Runtime API and supports only NVIDIA GPUs. The OpenCV CUDA module includes utility functions, low-level vision primitives, and high-level algorithms. The utility functions and low-level primitives provide a powerful infrastructure for developing fast vision algorithms taking advantage of CUDA whereas the high-level functionality includes some state-of-the-art algorithms (such as stereo correspondence, face and people detectors, and others) ready to be used by the application developers. [4]

D. CODE

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <cuda.h>
#include <opencv2/imgproc.hpp>
#include <opencv2/opencv.hpp>
#include <opencv2/core/cuda.hpp>
#include <opencv2/cudaimgproc.hpp>
#include <opencv2/cudaarithm.hpp>
#include <opencv2/highgui/highgui.hpp>

void calcCumHist(cv::Mat, int*);
void CalcCudaGrid(dim3&, dim3&, int, int);
void EqualizationByRoutine(cv::cuda::GpuMat, cv::Mat, cudaEvent_t*, float&, float *);
void writeToFile(FILE*, const char*, float, float) ;
std::vector<cv::Vec2f> fromIMGtoVec2f(cv::Mat);

cv::Mat DrawLines(cv::Mat, std::vector<cv::Vec2f>);
cv::Mat cpu_RGBtoGRAYSCALE(cv::Mat, float*);
cv::Mat cpu_resizeImage(cv::Mat,cv::Size, float*);
cv::Mat cpu_equalization(cv::Mat, int*, float*);
cv::Mat cpu_HoughTransformLine(cv::Mat ,float *);

cv::cuda::GpuMat gpu_RGBtoGRAYSCALE(cv::cuda::GpuMat, cudaEvent_t*, float&);
cv::cuda::GpuMat gpu_resizeImage(cv::cuda::GpuMat, cv::Size size, cudaEvent_t*, float&);
__global__ void equalizeHistCUDA(unsigned char*, unsigned char*,int* , int, int);
__global__ void equalizeHistCUDASM(unsigned char*, unsigned char*, int *, int , int );
cv::Mat GPU_HoughTransformLine(cv::cuda::GpuMat, cudaEvent_t*, float&);

int main(int argn, char *argv[]){
    //Variables
    cv::Mat cpu_grayscaleImage, cpu_resizedImage, cpu_equalizedImage;
    cv::cuda::GpuMat gpuImage, gpu_grayscaleImage, gpu_resizedImage;
    int cumHist[256]={0};
    int *cumHist_device;
    dim3 nThreadPerBlocco, numBlocks;
    cudaEvent_t timer[2];
    float GPUelapsedTime, CPUelapsedTime;
    cv::Size size(600,600);

    //Performance files
    FILE *file = fopen("Performance.txt", "wb");
```

```
43
44      //
45      if (file == NULL) {
46          fprintf(stderr, "Error: impossible to open.\n");
47          return 1;
48      }
49
50      //Kernel settings
51      CalcCudaGrid(numBlocks,nThreadPerBlocco, size.height,size.width);
52
53      //Header
54      fprintf(file, "\t***Kernel settings***:\nNumber of blocks: %dx%d\
55      nNumber of threads x blocks: %dx%d.\n",
56      numBlocks.y,numBlocks.x,nThreadPerBlocco.y,
57      nThreadPerBlocco.x);
58
59      fprintf(file,"\t***Warning***:\nThe Cuda and OpenCV functions
60      for equalization also calculate the histogram so the times are also
61      influenced by this calculation.\nThe functions
62      that I implemented use a ready-made histogram so the time is
63      without histogram calculation.\n\n\n");
64
65      fprintf(file, "%-60s %-20s %-20s\n", "Action",
66      "CPU Time (msec)", "GPU Time (msec)");
67
68      //Read the input image
69      cv::Mat input = cv::imread("foto.jpg");
70
71      if (input.empty()) {
72          fprintf(stderr, "Unable to load image\n");
73          return -1;
74      }
75
76      //Loading of the image from the cpu to gpu
77      gpuImage.upload(input);
78
79      //Timer Evenet creation
80      cudaEventCreate(&timer[0]);
81      cudaEventCreate(&timer[1]);
82
83      //RGB to Grayscale function (CPU)
84      cpu_grayscaleImage = cpu_RGBtoGRAYSCALE(input, &CPUelapsedTime);
85
86      //RGB to Grayscale function (GPU)
87      gpu_grayscaleImage = gpu_RGBtoGRAYSCALE(gpuImage, timer, GPUelapsedTime);
88      writeToFile(file, "RGB to Grayscale", CPUelapsedTime,GPUelapsedTime);
89      //Resize on CPU with CPU image as input
90      cpu_resizedImage=cpu_resizeImage(cpu_grayscaleImage,size, &CPUelapsedTime);
91
92      //Resize on GPU
93      gpu_resizedImage = gpu_resizeImage(gpu_grayscaleImage,size,
```

```
94      timer, GPUelapsedTime);
95      writeToFile(file, "Resize", CPUelapsedTime,GPUelapsedTime);
96
97      //CPU Equalization by OpenCV and Cuda
98      EqualizationByRoutine(gpu_resizedImage,cpu_resizedImage, timer,
99      GPUelapsedTime, &CPUelapsedTime);
100
101     writeToFile(file, "Equalization using OpenCV and Cuda routines",
102     CPUelapsedTime,GPUelapsedTime);
103
104     //CPU Equalization by myself
105     calcCumHist(cpu_resizedImage,cumHist);
106     cpu_equalizedImage =
107     cpu_equalization(cpu_resizedImage,cumHist,&CPUelapsedTime);
108     writeToFile(file, "Equalization on CPU by myself", CPUelapsedTime,0);
109
110 //Equalization on GPU - NO SM
111
112     //Mem. allocation on GPU for cumHist
113     cudaMalloc((void**)&cumHist_device,256*sizeof(int));
114     cudaMemcpy(cumHist_device, cumHist, 256*sizeof(int), cudaMemcpyHostToDevice);
115
116     cv::cuda::GpuMat gpu_equalizedImage =
117     cv::cuda::createContinuous(gpu_resizedImage.rows,gpu_resizedImage.cols,CV_8UC1);
118
119     //Timer's start
120     cudaEventRecord(timer[0], 0);
121
122     equalizeHistCUDA<<<numBlocks,nThreadPerBlocco>>>(gpu_resizedImage.data,
123     gpu_equalizedImage.data,
124     cumHist_device,gpu_resizedImage.cols,gpu_resizedImage.rows);
125     cudaError_t cudaErr = cudaGetLastError();
126     if (cudaErr != cudaSuccess)
127         fprintf(stderr, "CUDA Error: %s\n", cudaGetErrorString(cudaErr));
128
129     cudaDeviceSynchronize();
130     //Timer's end
131     cudaEventRecord(timer[1], 0);
132     cudaEventSynchronize(timer[1]);
133     //Elapsed time calculation
134     cudaEventElapsedTime(&GPUelapsedTime, timer[0], timer[1]);
135     writeToFile(file, "Equalization on GPU without SM by myself",0,GPUelapsedTime);
136
137 // END Equalization - NO SM
138
139 //Start Equalization with SM
140
141     cv::cuda::GpuMat gpu_equalizedImageSM =
142     cv::cuda::createContinuous(gpu_resizedImage.rows,gpu_resizedImage.cols,CV_8UC1);
143
144     //Timer's start
```

```
145     cudaEventRecord(timer[0], 0);
146     equalizeHistCUDASM<<<numBlocks,nThreadPerBlocco>>>(gpu_resizedImage.data,
147     gpu_equalizedImageSM.data,
148     cumHist_device,gpu_resizedImage.cols,gpu_resizedImage.rows);
149     cudaErr = cudaGetLastError();
150     if (cudaErr != cudaSuccess)
151         fprintf(stderr, "CUDA Error: %s\n", cudaGetErrorString(cudaErr));
152     cudaThreadSynchronize();
153     //Timer's end
154     cudaEventRecord(timer[1], 0);
155     cudaEventSynchronize(timer[1]);
156     //Elapsed time calculation
157     cudaEventElapsedTime(&GPUelapsedTime, timer[0], timer[1]);
158     writeToFile(file, "Equalization on GPU with SM by myself",0,GPUelapsedTime);
159
160 //END Equalization with SM
161
162     //Hough on CPU
163     cv::Mat cpu_Hough;
164     cpu_Hough = cpu_HoughTransformLine(cpu_equalizedImage, &CPUelapsedTime);
165     cv::imwrite("CPU Hough.jpg", cpu_Hough);
166
167     //Hough on GPU
168     cv::Mat gpu_Hough;
169     gpu_Hough = GPU_HoughTransformLine(gpu_equalizedImage,timer,GPUelapsedTime);
170     cv::imwrite("GPU Hough.jpg", gpu_Hough);
171     writeToFile(file, "Hough transform for lines",CPUelapsedTime,GPUelapsedTime);
172
173
174     //The memory of cv::cuda::GpuMat and cv::Mat
175     //objects is automatically deallocated by the library.
176     //But to avoid any problem I do it manually.
177     cpu_grayscaleImage.release();
178     cpu_resizedImage.release();
179     cpu_equalizedImage.release();
180     cpu_Hough.release();
181     gpu_Hough.release();
182     gpuImage.release();
183     gpu_grayscaleImage.release();
184     gpu_resizedImage.release();
185     gpu_equalizedImage.release();
186     gpu_equalizedImageSM.release();
187     cudaFree(cumHist_device);
188     cudaEventDestroy(timer[0]);
189     cudaEventDestroy(timer[1]);
190     fclose(file);
191     return 0;
192 }
193
194 void writeToFile(FILE *file, const char *string, float CPUTime, float GPUTime) {
195     if (file == NULL) {
```

```
196         fprintf(stderr, "File Error.\n");
197         return;
198     }
199 
200     fprintf(file, "%-60s", string);
201     fprintf(file, "%-20f", CPUTime);
202     fprintf(file, "%-20f", GPUTime);
203     fprintf(file, "\n");
204 }
205 
206 //Convert an image to a Vec2f
207 std::vector<cv::Vec2f> fromIMGtoVec2f(cv::Mat input){
208     std::vector<cv::Vec2f> h_lines;
209     if (input.rows == 2 && input.cols > 0) {
210         h_lines.resize(input.cols);
211         for (int i = 0; i < input.cols; ++i) {
212             h_lines[i] = input.at<cv::Vec2f>(0, i);
213         }
214     } else if (input.rows > 0 && input.cols == 2) {
215         h_lines.resize(input.rows);
216         for (int i = 0; i < input.rows; ++i) {
217             h_lines[i] = input.at<cv::Vec2f>(i, 0);
218         }
219     } else {
220         fprintf(stderr,"ERROR: Incorrect size\n");
221     }
222 
223     return h_lines;
224 }
225 
226 //Equalization by Cuda and OpenCV routines
227 void EqualizationByRoutine(cv::cuda::GpuMat gpuImg, cv::Mat cpuImg,
228 cudaEvent_t* timer, float& GPUelapsedTime, float *CPUelapsedTime){
229     //CPU
230     struct timespec start_time, end_time;
231     cv::Mat opencvEqualizedImg;
232     clock_gettime(CLOCK_MONOTONIC, &start_time);
233     cv::equalizeHist(cpuImg, opencvEqualizedImg);
234     clock_gettime(CLOCK_MONOTONIC, &end_time);
235     *CPUelapsedTime = (end_time.tv_sec - start_time.tv_sec) * 1000.0
236     + (end_time.tv_nsec - start_time.tv_nsec) / 1000000.0;
237 
238     //GPU
239     cv::cuda::GpuMat gpuEqualizedImage;
240     //Timer's start
241     cudaEventRecord(timer[0], 0);
242     cv::cuda::equalizeHist(gpuImg, gpuEqualizedImage);
243     //Timer's end
244     cudaEventRecord(timer[1], 0);
245     cudaEventSynchronize(timer[1]);
246     //Elapsed time calculation
```

```
247     cudaEventElapsedTime(&GPUelapsedTime, timer[0], timer[1]);
248 }
249
250 //Cumulative Histogram computation
251 void calcCumHist(cv::Mat image, int *cumHist){
252     int nBins = 256, sum=0;
253     int hist[nBins];
254     memset(hist,0,sizeof(hist));
255     //Histogram
256     for (int i = 0; i<image.rows; i++){
257         for(int j = 0; j<image.cols; j++){
258             unsigned char pixel_value= image.at<unsigned char>(i, j);
259             hist[pixel_value]++;
260         }
261     }
262
263     for (int i = 0; i<nBins; i++){
264         sum+=hist[i];
265         cumHist[i]=sum;
266     }
267 }
268
269 //Draws the detected lines on the original image
270 cv::Mat DrawLines(cv::Mat originalImage, std::vector<cv::Vec2f> lines){
271     cv::Mat output = originalImage.clone();
272     for (int i = 0; i<lines.size(); i++) {
273         float rho = lines[i][0];
274         float theta = lines[i][1];
275         double a = std::cos(theta), b = std::sin(theta);
276         double x0 = a * rho, y0 = b * rho;
277         cv::Point pt1(cvRound(x0 + 1000 * (-b)), cvRound(y0 + 1000 * (a)));
278         cv::Point pt2(cvRound(x0 - 1000 * (-b)), cvRound(y0 - 1000 * (a)));
279         cv::line(output, pt1, pt2, cv::Scalar(0), 3);
280     }
281     return output;
282 }
283
284 //Histogram equalization on CPU
285 cv::Mat cpu_equalization(cv::Mat image,int *cumulative_hist, float *elapsedTime){
286     struct timespec start_time, end_time;
287     cv::Mat equalizedImage(cv::Size(image.rows,image.cols),
288     CV_8UC1,cv::Scalar(255));
289     int area = image.rows*image.cols, ngraylevel=256;
290     uchar pixel_value;
291     clock_gettime(CLOCK_MONOTONIC, &start_time);
292     //Equalization
293     for (int i =0; i<image.rows; i++){
294         for(int j = 0; j<image.cols; j++){
295             pixel_value = image.at<uchar>(i,j);
296             equalizedImage.at<uchar>(i,j) =
297             ((double)ngraylevel/area)*cumulative_hist[pixel_value];
```

```
298         }
299     }
300     clock_gettime(CLOCK_MONOTONIC, &end_time);
301     *elapsedTime = (end_time.tv_sec - start_time.tv_sec) * 1000.0
302     + (end_time.tv_nsec - start_time.tv_nsec) / 1000000.0;
303 
304     return equalizedImage;
305 }
306 
307 //Resize of the image using OpenCV (CPU)
308 cv::Mat cpu_resizeImage(cv::Mat in,cv::Size size, float *elapsedTime){
309     struct timespec start_time, end_time;
310     cv::Mat out;
311     clock_gettime(CLOCK_MONOTONIC, &start_time);
312     cv::resize(in, out, size);
313     clock_gettime(CLOCK_MONOTONIC, &end_time);
314     *elapsedTime = (end_time.tv_sec - start_time.tv_sec) * 1000.0
315     + (end_time.tv_nsec - start_time.tv_nsec) / 1000000.0;
316     return out;
317 }
318 
319 //Converting RGB to Grayscale using OpenCV (CPU)
320 cv::Mat cpu_RGBtoGRAYSCALE(cv::Mat in, float *elapsedTime){
321     struct timespec start_time, end_time;
322     //Output image
323     cv::Mat out;
324     clock_gettime(CLOCK_MONOTONIC, &start_time);
325     //BGR to Grayscale
326     cv::cvtColor(in,out,cv::COLOR_BGR2GRAY);
327     clock_gettime(CLOCK_MONOTONIC, &end_time);
328     *elapsedTime = (end_time.tv_sec - start_time.tv_sec) * 1000.0
329     + (end_time.tv_nsec - start_time.tv_nsec) / 1000000.0;
330     return out;
331 }
332 
333 //HoughTransform for line on CPU
334 cv::Mat cpu_HoughTransformLine(cv::Mat input, float *elapsedTime){
335     struct timespec start_time, end_time;
336     cv::Mat gass, can;
337     cv::Mat output = input.clone();
338     std::vector<cv::Vec2f> lines;
339 
340     clock_gettime(CLOCK_MONOTONIC, &start_time);
341 
342     //Clean the image of any noise so as to reduce it the problem of spurious votes
343     cv::GaussianBlur(input, gass, cv::Size(5, 5), 0, 0);
344 
345    //Perform Canny so as to return the edge points of the image
346     cv::Canny(gass, can, 80, 140);
347     cv::HoughLines(can, lines, 1, CV_PI / 180, 147);
348 
```

```
349     clock_gettime(CLOCK_MONOTONIC, &end_time);
350     *elapsedTime = (end_time.tv_sec - start_time.tv_sec) * 1000.0
351     + (end_time.tv_nsec - start_time.tv_nsec) / 1000000.0;
352 
353     return DrawLines(input,lines);
354 }
355 
356 //Converting RGB to Grayscale using OpenCV for CUDA (GPU)
357 cv::cuda::GpuMat gpu_RGBtoGRAYSCALE(cv::cuda::GpuMat gpuImage,
358 cudaEvent_t* timer, float& elapsedTime){
359     cv::cuda::GpuMat out =
360     cv::cuda::createContinuous(gpuImage.size(),gpuImage.type());
361     //Timer's start
362     cudaEventRecord(timer[0], 0);
363     //BGR to Grayscale
364     cv::cuda::cvtColor(gpuImage,out,cv::COLOR_BGR2GRAY);
365     //Timer's end
366     cudaEventRecord(timer[1], 0);
367     cudaEventSynchronize(timer[1]);
368     //Elapsed time calculation
369     cudaEventElapsedTime(&elapsedTime, timer[0], timer[1]);
370 
371     return out;
372 }
373 
374 //Resize of the image using OpenCV for CUDA (GPU)
375 cv::cuda::GpuMat gpu_resizeImage(cv::cuda::GpuMat gpuImage,
376 cv::Size outputSize, cudaEvent_t* timer, float& elapsedTime){
377 
378     cv::cuda::GpuMat out = cv::cuda::createContinuous(outputSize,gpuImage.type());
379     //Timer's start
380     cudaEventRecord(timer[0], 0);
381     cv::cuda::resize(gpuImage, out, outputSize);
382     //Timer's end
383     cudaEventRecord(timer[1], 0);
384     cudaEventSynchronize(timer[1]);
385     //Elapsed time calculation
386     cudaEventElapsedTime(&elapsedTime, timer[0], timer[1]);
387     return out;
388 }
389 
390 //Compute the kernel configuration
391 void CalcCudaGrid(dim3 &numBlocks, dim3 &nThreadPerBlocco, int rows, int cols){
392     cudaDeviceProp prop;
393     cudaGetDeviceProperties(&prop, 0); // 0 device's index
394     //Max thread's num. x block of the gpu
395     int maxThreadsPerBlock = prop.maxThreadsPerBlock;
396     nThreadPerBlocco.x =
397     min(cols, int(sqrt(maxThreadsPerBlock))); // Max for x
398     nThreadPerBlocco.y =
399     min(rows, maxThreadsPerBlock / nThreadPerBlocco.x); // Max for y
```

```
400     numBlocks.x = (cols + nThreadPerBlocco.x - 1) / nThreadPerBlocco.x;
401     numBlocks.y = (rows + nThreadPerBlocco.y - 1) / nThreadPerBlocco.y;
402 }
403
404 //CUDA Kernel code for SM-free equalization
405 __global__ void equalizeHistCUDA(unsigned char* input, unsigned char* output,
406 int *cumulative_hist, int cols, int rows) {
407     int nGrayLevels = 256, area = cols*rows;
408     int i = blockIdx.y * blockDim.y + threadIdx.y;
409     int j = blockIdx.x * blockDim.x + threadIdx.x;
410     if (i < rows && j < cols){
411         int index = i * cols + j;
412         int pixelValue = input[index];
413         output[index] = static_cast<uchar>((static_cast<double>(nGrayLevels) / area)
414         * cumulative_hist[pixelValue]);
415     }
416 }
417
418 //CUDA Kernel code for equalization with SHARED MEMORY
419 __global__ void equalizeHistCUDASM(unsigned char* input, unsigned char* output,
420 int *cumulative_hist, int cols, int rows) {
421     int nGrayLevels = 256, area = cols * rows;
422     __shared__ int shared_cumulative_hist[256];
423     int elements_per_thread = ( 256/(blockDim.x*blockDim.y) > 1 )
424     ? (256/blockDim.x*blockDim.y) : 1;
425     int InBlockThreadID = threadIdx.x + blockDim.x * threadIdx.y; //from 0 to 1023 x block of 32x32
426     int start_index = InBlockThreadID * elements_per_thread;
427     for (int i = 0; i < elements_per_thread; i++){
428         int index = start_index + i;
429         if (index < 256)
430             shared_cumulative_hist[index] = cumulative_hist[index];
431     }
432     __syncthreads();
433
434     int i = blockIdx.y * blockDim.y + threadIdx.y;
435     int j = blockIdx.x * blockDim.x + threadIdx.x;
436     if (i < rows && j < cols) {
437         int index = i * cols + j;
438         int pixelValue = input[index];
439         output[index] = static_cast<unsigned char>((static_cast<double>(nGrayLevels) / area)
440         *shared_cumulative_hist[pixelValue]);
441     }
442 }
443
444 //HoughTransform for line on GPU
445 cv::Mat GPU_HoughTransformLine(cv::cuda::GpuMat input, cudaEvent_t* timer, float&elapsedTime){
446
447     cv::cuda::GpuMat gass, can, lines;
448     std::vector<cv::Vec2f> cpu_lines;
449     //Timer's start
450     cudaEventRecord(timer[0], 0);
```

```
451     cv::Ptr<cv::cuda::Filter> gaussianFilter =
452     cv::cuda::createGaussianFilter(CV_8U, CV_8U, cv::Size(5,5),0);
453     gaussianFilter->apply(input,gass);
454 
455     cv::Ptr<cv::cuda::CannyEdgeDetector> canny =
456     cv::cuda::createCannyEdgeDetector(80, 140, 3, false);
457     canny->detect(gass, can);
458 
459     cv::Ptr<cv::cuda::HoughLinesDetector> hough =
460     cv::cuda::createHoughLinesDetector(1.0, CV_PI / 180, 147);
461     hough->detect(can,lines);
462 
463     //Timer's end
464     cudaEventRecord(timer[1], 0);
465     cudaEventSynchronize(timer[1]);
466     //Elapsed time calculation
467     cudaEventElapsedTime(&elapsedTime, timer[0], timer[1]);
468     cv::Mat tmp, img;
469     lines.download(tmp);
470     cpu_lines = fromIMGtoVec2f(tmp);
471     input.download(img);
472 
473     //draws the lines contained in the vector inside the original image.
474     return DrawLines(img,cpu_lines);
475 }
476 }
```

code.cu

BIBLIOGRAPHY

[1] Rafael C Gonzalez. *Digital image processing*. Pearson Education International, 2009. Chap. 10.

[2] OpenCV Team. *About - OpenCV*. URL: https://opencv.org/about/.

[3] FRED OH. *what-is-cuda*. URL: https://blogs.nvidia.com/blog/what-is-cuda-2/.

[4] OpenCV Team. *CUDA Module Introduction*. URL: https://docs.opencv.org/4.x/d2/dbc/cuda_intro.html.