Parallel Computing 2024 – 2025

Histogram equalization Assignment

Chiara Polenzani
chiara.polenzani@edu.unifi.it

Abstract

This paper presents the implementation of a system for computing and displaying the equalized histogram of an image. A function was developed to perform global histogram equalization, implemented in C++ and integrated into both a sequential and a parallel version of the program. The parallel version leverages CUDA to execute the equalization algorithms on the GPU, enabling faster processing on large images. The system supports both grayscale and color images, and generates two histograms: the original and the globally equalized, along with the corresponding processed images.

1. Introduction

An image histogram is a graphical representation of the distribution of intensity values in a digital image.
For grayscale images, the histogram is a function that maps each possible gray level to the number of pixels in the image that have that level. Histograms are essential tools for analyzing image contrast and brightness. In a well-contrasted image, the histogram tends to be spread across the full range of intensity values and often approaches a uniform distribution. Conversely, a poorly contrasted image will have a histogram concentrated in a narrow intensity range. Histogram equalization is a gray-level transformation technique that aims to redistribute intensity values to achieve a more uniform histogram. The goal is to enhance image contrast by increasing the number of gray levels assigned to frequently occurring intensities (the peaks of the histogram) and reducing those assigned to rare intensities (the valleys).
In this assignment, the user can upload a grayscale or color image from his personal computer, after which the equalized image and corresponding histogram will be generated. The entire process was implemented using the OpenCV library and then handling code parallelization with CUDA.

2. Implementation

In this project, I have developed a set of functions for image histogram processing and visualization. These functions are designed to work both for grayscale and color images, providing a versatile solution for analyzing and enhancing images through histogram equalization. Firstly, I created a function that generates the histogram of an image. This function counts the frequency of each intensity level (or color component in the case of color images) in the image and stores this information in an array.
There is a separate version of this function for grayscale images and color images. For color images, I calculate the histogram for each of the three channels, Red, Green, and Blue individually. I also created a function to visualize these histograms graphically, each color channel is represented by a different color. The next step was to implement a function that performs histogram equalization of an image. This process adjusts the intensity distribution in the image to enhance contrast, making details more visible in areas that may have been too dark or too light. For color images, each channel is processed independently to ensure that the equalization maintains a natural color balance. This generic workflow is used in both the sequential and parallel versions of the code.

2.1. Sequential version

This section describes in more detail the different functions that make up the project.
The `computeGrayHistogram` function takes the grayscale image and an array of 256 integers as input, each representing a histogram bin. A double for loop iterates over every pixel in the image. For each position it reads the pixel value using and increments the counter corresponding to that intensity. To make the distribution of values visually interpretable, the `showGrayHistogram` function creates a graphical representation of the histogram. For each bin in the histogram, a vertical black line is drawn, with its height proportional to the bin's frequency. The program can also handle color images. In this case, the `computeColorHistogram` function computes three separate histograms, one for each channel. Each image pixel

is read as a vector of three elements (Vec3b), and each component is used to increment the corresponding bin in its respective histogram. This results in three curves representing the intensity distribution of each color channel. The final result is a combined visualization of the three histograms, which helps in analyzing the separate distribution of colors and identifying any color imbalance.
The `equalizeGrayImage` function is responsible for equalizing a grayscale image. Image histogram is used to calculate the Cumulative Distribution Function (CDF), a function that accumulates pixel intensity values, providing a measure of the distribution of values in the image. Next, a mapping of intensity levels is generated and applied to each pixel in the image. The result is an equalized image in which grayscale levels are evenly distributed, improving contrast and making details that were previously hidden in areas that were too dark or bright more visible. The `equalizeColorImage` function follows a similar process, but is designed for color images. First, it divides the color image into its three channels each of which is then processed separately; for each the pixel intensity histogram is calculated, followed by the CDF calculation. After that, a mapping of new pixel values is created for each channel. This mapping is applied to all the pixels in each channel, improving the color distribution and contrast in the image. Finally, the three equalized channels are merged together, creating the final equalized color image.

2.2. Parallel version

To improve the performance of histogram computation and image equalization, a parallel version of the project was implemented using NVIDIA's CUDA programming model. CUDA enables massive parallelism by allowing the execution of many threads concurrently on the GPU. This is particularly advantageous for image processing tasks such as histogram computation and equalization, where the same operation must be applied to a large number of pixels. The parallel version mirrors the structure of the sequential implementation but introduces CUDA kernels to run computations on the GPU, efficiently distributing the workload across hundreds of threads. The CUDA implementation of the histograms closely follows the typical structure of a CUDA program. First, memory is allocated on the GPU to store the input image and the histogram array. Then, the input data is transferred from the CPU (host) memory to the GPU (device) memory. Once the data is on the device, a CUDA kernel is launched to compute the histogram in parallel. After the computation is complete, the resulting histogram is copied back from GPU memory to the host, where it can be used for further processing or visualization. Finally, all allocated memory on the device is released to avoid leaks and free up resources. One important optimization technique employed in the CUDA kernels is interleaved partitioning. This approach assigns each thread to process non-consecutive pixels in memory, which promotes memory coalescing, a key factor for efficient global memory access in CUDA. By doing so, memory bandwidth is utilized more effectively, reducing access latency and improving throughput. In this experiment, the efficiency of two parallel kernel implementations for computing the histogram of a color image was compared: one using only global memory, and another exploiting shared memory. Both versions produce correct results thanks to the use of atomic operations, but the performance difference is significant:

- In the **version without shared memory**, each thread accesses global memory directly to update the histogram. This leads to contention, as many threads attempt to update the same value simultaneously, resulting in slower execution.
- In the **version with shared memory**, each block builds a local histogram in shared memory, reducing contention. The final values are aggregated to global memory only at the end of the block's execution.

The plot shows that after just two repetitions, the shared memory version consistently outperforms the global-only version by about 3 ms, with execution time stabilizing around ~ 16 ms.

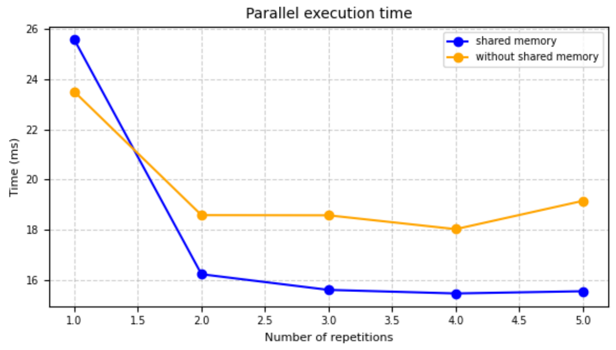


Figure 1: Execution time of the parallel version with and without shared memory.

The `computeHistogramGrayKernel` computes the histogram of a grayscale image in parallel by leveraging shared memory to reduce contention caused by atomic operations on global memory. Each block allocates a local copy of the histogram in shared memory, which is initialized in parallel by the threads within the block. Threads access a subset of the image pixels using interleaved partitioning and update the local histogram using atomic operations within shared memory. After processing, the threads combine the partial results into the global memory histogram using atomic operations. The `computeGrayHistogramCUDA` function

handles memory management and kernel execution. It allocates memory for the image and histogram on the GPU, configures the execution grid and block dimensions, launches the kernel, synchronizes the device, and finally copies the computed histogram back to the host. The function that creates a visual representation of the histogram remains unchanged from the sequential version.
The CUDA version for computing the color histogram follows the same general structure as the grayscale version but incorporates several important changes primarily due to the handling of the three color channels, each of which requires its own histogram. Each thread block allocates three shared memory arrays, one for each color channel, to store partial histograms locally within the block. Threads then process the input image using interleaved partitioning and updates the corresponding bin in the shared memory histograms using atomic operations. This shared-memory strategy significantly improves performance by reducing the number of slow atomic operations on global memory, ensuring correct and parallel updates across all three color channels. The equalization of the color image is performed channel by channel. The `equalizeColorImageCUDA` function first splits the input image into its three B, G, and R components. Each channel is then individually equalized and the results are merged back to form the final output. Four main steps are performed on the GPU: Histogram calculation, normalization, CDF calculation and equalization. After equalization, the final image is copied back to the host.

3. Profiling

The goal of the profiling is to evaluate the efficiency of the parallel implementation for histogram equalization on both color and grayscale images, by measuring the execution time of GPU and CUDA API calls. Profiling was performed using nvprof on Google Colab, and the main results are analyzed in Table 1.
In the case of the color image, the majority of the GPU time (55.72%) is spent on the `cudaMemcpy` HtoD function, which handles the transfer of data from host (CPU) memory to device (GPU) memory. This is expected, as a color image consists of three separate channels, each of which is processed and transferred individually. Another significant portion of the GPU time (14.13%) is taken up by the kernel responsible for computing the histogram of each channel. Next in terms of GPU time is the kernel that transfers data back from the GPU to the CPU after processing, followed by the `applyEqualization` function (10.92%), which applies the histogram equalization transformation to the image. With a grayscale image, the behavior changes significantly. The `computeHistogramGrayKernel` becomes the most dominant, accounting for 43.01% of the total GPU time. This kernel was designed to take advantage of the GPU's shared memory, enabling more efficient his-

Function / Kernel	**Color (RGB)**	**Grayscale**
`cudaMemcpy HtoD`	55.72% (9.82 ms)	38.10% (2.52 ms)
`computeHistogramColorKernel`	**14.13%** (2.50 ms)	–
`cudaMemcpy DtoH`	12.41% (1.88 ms)	12.86% (0.824 ms)
`applyEqualization`	10.92% (1.92 ms)	4.97% (0.328 ms)
`computeHistogramGrayKernel`	–	**43.01%** (2.84 ms)
`computeCDF`	0.74% (0.130 ms)	0.48% (0.031 ms)
`normalizeHistogram`	0.51% (0.088 ms)	0.33% (0.022 ms)
`cudaMemcpy (API)`	31.30 ms (140 calls)	12.29 ms (60 calls)
`cudaMalloc (API)`	8.38 ms	3.52 ms
`cudaEventCreate (API)`	9.35 ms	4.48 ms
`cudaFree (API)`	8.83 ms	3.18 ms

Table 1: Profiling on color and grayscale images.

togram computation. The function that copies data from the CPU to the GPU takes up 38.10% of the time, a significantly lower value compared to that observed with RGB images. This reduction is due to the fact that a grayscale image contains only a single intensity channel. The lower data volume also benefits other operations, such as memory allocation and CUDA event management.
In conclusion, the grayscale version proves to be more computationally efficient, with a higher proportion of time spent on actual processing rather than on data transfers.
The color version, on the other hand, shows a substantial overhead due to the data copy operations, caused by the presence of three separate channels, and the increased complexity in memory management.

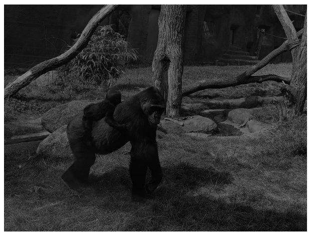
4. Experiments and Results

To evaluate the effectiveness of the developed functions, I tested them on both grayscale and color images. For each example, the complete transformation workflow is shown, step by step.

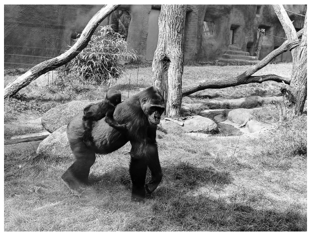
Specifically, the following are displayed:

- the original image;
- the corresponding histogram representing the distribution of pixel intensities (or RGB values);
- the equalized image;
- the histogram of the equalized image.

I verified that both the sequential and parallel implementations produce identical visual outputs, both for histograms and equalized images. By comparing the original and equalized versions, it becomes immediately evident how darker or lighter regions become more defined and balanced. The histograms help confirm this transformation at a numerical level, showing a more uniform distribution after equalization.

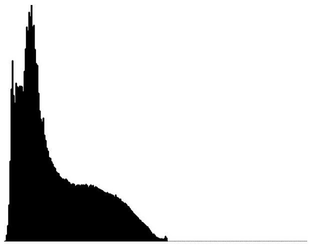


(a) Original Grayscale Image

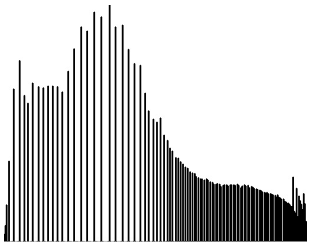


(b) Equalized Grayscale Image

Figure 2: Comparison of original and equalized grayscale image

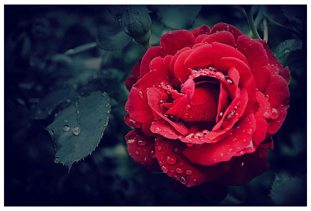


(a) Original histogram

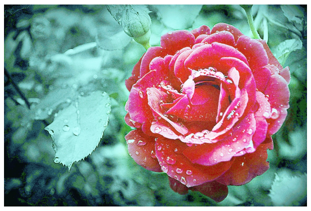


(b) Equalized histogram

Figure 3: Comparison of original and equalized grayscale histogram

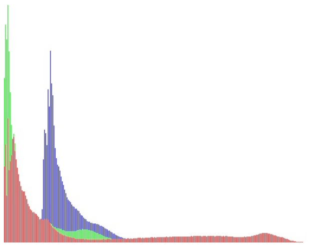


(a) Original Color Image

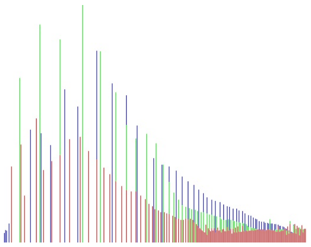


(b) Equalized Color Image

Figure 4: Comparison of original and equalized color image



(a) Original histogram



(b) Equalized histogram

Figure 5: Comparison of original and equalized color histogram

CUDA organizes threads into blocks, and blocks into grids. Each block contains a certain number of threads, and each thread executes an instance of the kernel. The goal of the first experiment was to identify the optimal number of threads per block. Specifically, several block sizes (32, 64, 128, 256, 512, 1024) were tested to observe their impact on performance in terms of execution time, both for grayscale and color images. As shown in Figure 6, for grayscale images, the execution time is very stable (~ 5 ms), with a minimum around 256 threads per block. For color images, the execution times are higher but improve significantly when increasing the block size from 32 to 128 threads. This behavior is due to better warp utilization: increasing the number of threads per block maximizes parallelism. Beyond a certain threshold, GPU occupancy is already nearly optimal, so further increases do not yield performance gains. Choosing 256 threads per block appears to be a balanced solution that ensures good occupancy and minimal execution time.

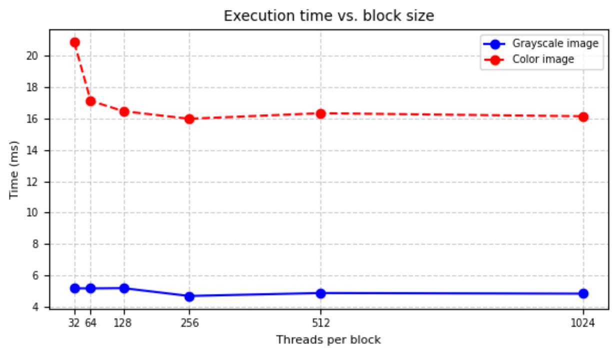


Figure 6: Execution time as a function of the number of threads per block, for both grayscale and color image processing.

In the next section, execution times of sequential and parallel implementations are reported and analyzed to evaluate the performance gain achieved by GPU acceleration.
The pipeline was executed multiple times (5 repetitions) on the same input image to reduce variability in measurements. The total duration of the process was recorded and stored in a CSV file. The collected timing data is later used to generate performance plots that compare the sequential and parallel implementations.
As can be seen from the speedup graph, parallel code is about 1.5 times faster than sequential code in the case of grayscale images, with an average execution time of about 4 ms. Also in the case of color image it is evident that the parallel version is significantly faster than the sequential one. On average, the sequential implementation takes approximately 150 milliseconds, with minimal variation between runs, while the parallel version consistently performs around 16 milliseconds. This results in a performance improvement of 9 times,

highlighting the effectiveness of GPU acceleration in color image processing, where the workload is inherently greater due to the presence of three color channels.

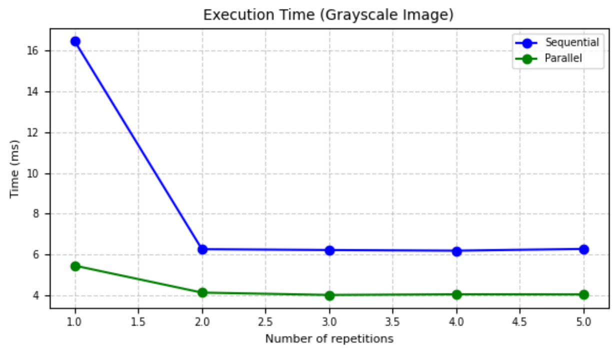


Figure 7: Comparison of execution times for grayscale image processing.

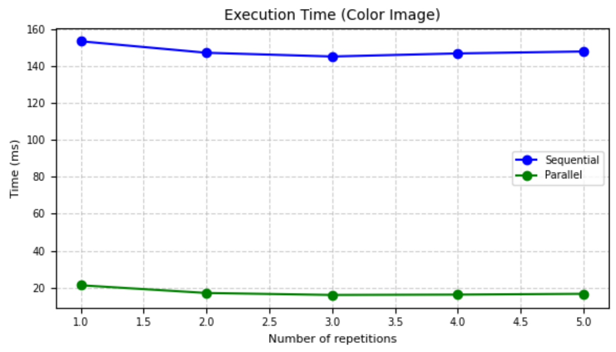


Figure 8: Comparison of execution times for color image processing.

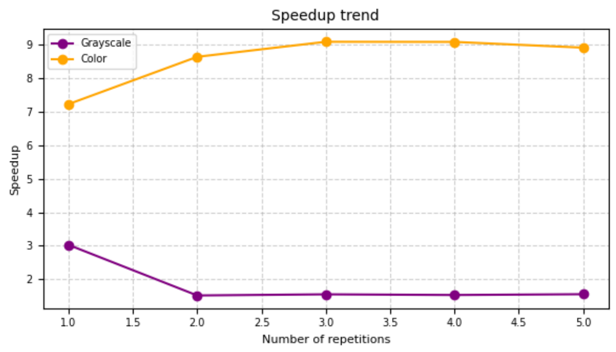


Figure 9: Speedup trend.

A further test was also performed with a large grayscale image, with the result that the sequential time increases from 6 ms to 40 ms, while the time for parallel execution increases from 4 ms to 12 ms.

5. Conclusions

In this assignment, a complete system for histogram equalization and visualization has been implemented, supporting grayscale and color images. The solution includes both sequential and parallel versions of the code, with the latter leveraging CUDA for efficient execution on the GPU. The results show that the developed functions are effective in improving image contrast as they show significantly more balanced intensity distributions. In addition, the visual and numerical consistency between sequential and parallel results confirms the correctness of the parallel implementation. Performance evaluations clearly highlight the benefits of GPU acceleration, with execution times reduced by a factor of 1.5 in small grayscale images, approximately 3 in large images and by more than 9 in color images. These results underscore the importance of parallelization in image processing applications.