Parallel histogram Equalization using CUDA

Uroš Lotrič, Davor Sluga

April 2025

1 Histogram equalization

Histogram equalization is an image processing technique that transforms an image so that the colour intensity histogram of the resultant image is more uniformly distributed. Adjusting the histogram to utilize the full range of intensities improves the image's contrast. Specifically, areas of lower local contrast gain a higher contrast, possibly exposing previously unseen details in an image. The method is especially useful in images with backgrounds and foregrounds that are both bright or dark, such as X-ray images. Histogram equalization can produce unrealistic effects in photographs; however, it is very useful for scientific images like thermal, satellite or X-ray images, where one does not care about the true-colour representation of the scene but is interested in specific details.

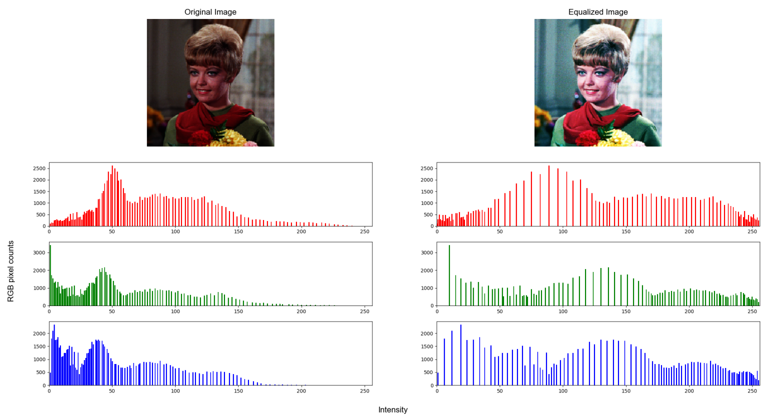

Figure 1: Histogram equalization result.

An image histogram represents the intensity distribution of a colour in the digital image. Typically digital images are represented using RGB colour space.

However, this representation is unsuitable for performing histogram equalization since it can cause colour shifts. First, we need to convert the image to a representation which separates luminosity/brightness from colour information. Commonly, YUV or YCbCr representation is used.

Conversion from RGB to YUV is performed using the following transformation:

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \tag{1}$$

To convert back to RGB, use the following transformation:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.402 \\ 1 & -0.344136 & -0.714136 \\ 1 & 1.772 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U-128 \\ V-128 \end{bmatrix} \tag{2}$$

The equalization is performed only on the Y channel (luminance). Consider the Y channel of a colour image space of size $N \times M$ pixels. The value $l \in [0, L-1]$ represents the luminance of an image pixel, where L is the number of possible values for the luminance; using the above transformations gives $L = 256$.

Let n_l^y represent the number of occurrences of luminance l. The luminance histogram is then:

$$H^y = [n_0^y, n_1^y, n_2^y, \ldots, n_{L-1}^y] \tag{3}$$

To equalize the histograms, we will need the cumulative distribution function (cdf) of the luminance. We obtain it by computing the cumulative histogram H_{cdf}^y, which tells us how many pixels in an image have a luminance lower or equal to l:

$$H_{cdf}^y(l) = \sum_{i=0}^{l} n_i^y \tag{4}$$

1.1 Histogram Equalization Algorithm

To equalize the histogram of an image, we perform the following four steps:

1. Transform the image from RGB to YUV space using transformation 1.
2. Compute the luminance histogram H^y.
3. Calculate the cumulative histogram H_{cdf}^y.
4. Calculate new pixel luminances l_{new} from original l based on the histogram equalization formula:

$$l_{new}^y = \left\lfloor \frac{H_{cdf}^y(l) - \min(H_{cdf}^y)}{N \times M - \min(H_{cdf}^y)} (L-1) \right\rfloor , \tag{5}$$

where $\min(H^y_{cdf})$ represents the **minimum non-zero value** in the cumulative histogram.

5. Assign new luminance l_{new} to each pixel.

6. Convert the image back to RGB colour space using transformation 2.

2 Parallel Histogram Equalization

Each of the Histogram Equalization steps can be more or less efficiently parallelized. Let's look into the parallelization of each step of the algorithm.

Image conversion from and to RGB: This step is embarrassingly parallel and thus trivial to parallelize. Each pixel can be processed independently. For improved efficiency, it can be integrated into the same kernel used for histogram computation and the kernel responsible for assigning new pixel values.

Parallel Histogram Computation: Performing parallel computation of a histogram involves a race condition since multiple threads access the same memory addresses when incrementing the pixel counts. To ensure correct results, atomic operation must be used, namely atomic addition/increment. This will negatively affect performance but can not be avoided. Note that the performance loss depends on the image's luminance distribution. If almost all the pixel values are the same, the threads will compete for access to the same memory location much more frequently than when values are more uniformly distributed. This issue can be alleviated using thread block-local partial histograms, where fewer memory access collisions happen when counting pixels. These partial histograms are then merged into the final histogram.

CUDA offers atomic operations like atomicAdd, which allows multiple threads to perform operations on the same memory location. To further optimize the computation of the histogram, shared memory should be employed to store partial histograms computed by each thread block. Access to shared memory is much faster than access to global memory. These partial histograms are then added to the final histogram in the GPU's global memory.

Parallel Cumulative Histogram Computation: Parallelizing the computation of the cumulative sum provides just a small benefit since the number of operations needed is rather low. Nevertheless, we will look into an approach to efficient parallelization of the computation of cumulative sum.

Computing a cumulative histogram involves the following process. Given an array A of numbers

$$A = [n_0, n_1, n_2, \ldots, n_{N-1}], \tag{6}$$

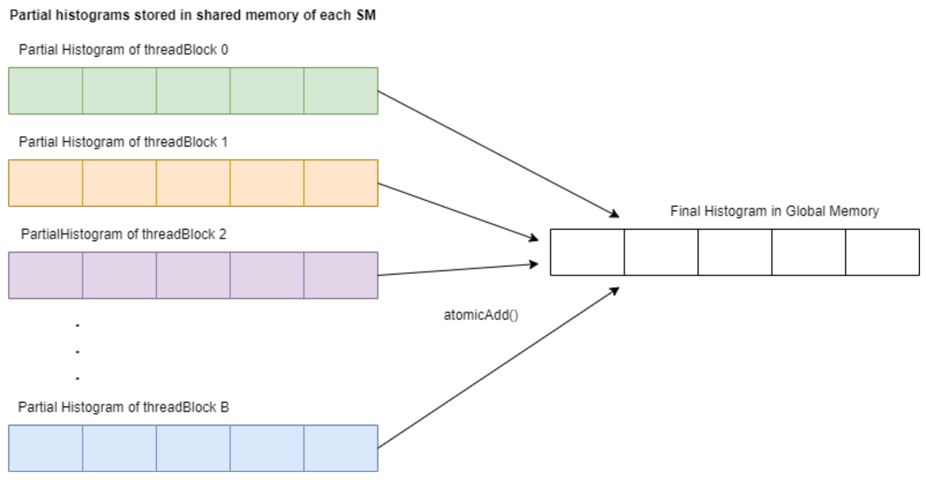
Partial histograms stored in shared memory of each SM
Partial Histogram of threadBlock 0
Partial Histogram of threadBlock 1
PartialHistogram of threadBlock 2
Partial Histogram of threadBlock B
Final Histogram in Global Memory
atomicAdd()

Figure 2: Using partial histograms in shared memory to optimize the computation of image histogram.

we want to transform its elements in the following way:

$$A = [n_0, n_0 + n_1, n_0 + n_1 + n_2, \ldots, n_0 + \cdots + n_{N-2} + n_{N-1}], \quad (7)$$

We call this operation an all-prefix-sum or scan. Computing a scan of an array using a sequential algorithm is straightforward.

Algorithm 1 Sequential scan

```
**for** $i \leftarrow 1$ to $N-1$ **do**
    $A[i] \leftarrow A[i] + A[i-1]$
**end for**
```

As we can see, the algorithm has dependencies between loop iterations and is not trivial to parallelize. A naive approach to parallelization is to compute all the sums (elements) in the array in parallel independently of each other, but this has considerable drawbacks. Firstly, it can't be done in-place, as threads will simultaneously alter the contents of the array, which will cause incorrect results. Secondly, the amount of work done will greatly increase as the same sums will need to be repeated by different threads. The sequential algorithm performs $N-1$ additions when computing the scan. A naive parallel algorithm would perform $\frac{(N-1)N}{2}$ additions. Blelloch [1] derived a work-efficient parallel scan algorithm which performs $2(N-1)$ additions in total. The scan is computed in two passes, first performing a parallel reduction and then proceeding to compute the remaining missing values.

Algorithm 2 Work-efficient parallel scan algorithm.

```
for $k = 0$ to $\log_2(N) - 1$ do
    for all $i = 0$ to $N - 1$ by $2^{k+1}$ do
        $A[i - 1 + 2^{k+1}] \leftarrow A[i - 1 + 2^k] + A[i - 1 + 2^{k+1}]$
    end for
end for
for $k = \log_2(N)$ to 1 do
    for all $i = 0$ to $N - 1$ by $2^k$ do
        $A[i - 1 + 2^{k-1} + 2^k] \leftarrow A[i - 1 + 2^{k-1} + 2^k] + A[i - 1 + 2^k]$
    end for
end for
```

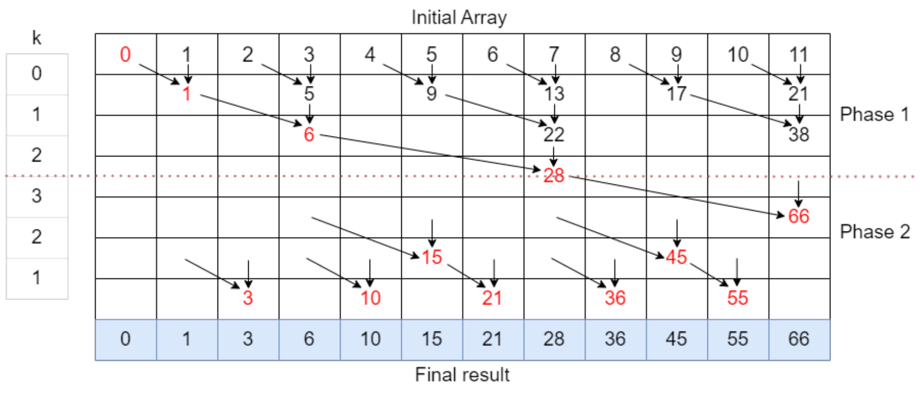

Figure 3: An illustration of the up-sweep (1) and down-sweep (2) phases of the work-efficient algorithm on a twelve-element array. All the addition operations in the same row can be done in parallel.

When implementing the above code in CUDA you need to use thread synchronization where appropriate and correctly assign element indices to threads. The scan should be done in shared memory to maximize performance. See this paper for practical ideas on how to produce a CUDA implementation of this algorithm.

Parallel Computation Of New Pixel Intensities: This step is simple to parallelize because the computation of the new intensity level based on the old level is completely independent of other operations. All of the calculations can be done in parallel. Note, however, that the benefit will not be significant since the number of operations that need to be performed is low.

Assigning New Intensities To Pixels: This step is also trivial to parallelize. All of the assignments of new pixel values are independent and can be performed in parallel. As a side note, memoization should be used to reduce the number of operations needed: The new intensity levels computed in the previous step can be stored in a look-up table and accessed based on the original colour intensity of a pixel.

3 Assignment

Implement a sequential and parallel version of the histogram equalization algorithm using C/C++ and CUDA. The algorithm should work for colour images of arbitrary size. The input image should be passed through command line arguments. You can use the STB library to read and write images from/to a file.

Basic tasks (for grades 6-8):

- Parallelize all of the steps of the algorithm using CUDA as efficiently as possible. The whole histogram equalization algorithm must be offloaded to the GPU. Assign the appropriate number of threads to each step, depending on the amount of operations that can be done in parallel. Use atomic operations where necessary. Avoid unnecessary data transfers to and from GPU memory. When computing the cumulative histogram, you can use a simple approach by computing it sequentially using one GPU thread.
- Measure the execution time of the algorithm for different image sizes. Use the next image sizes: 720x480, 1024x768, 1920x1200, 3840x2160, 7680x4320. Find the optimal number of threads and thread block size for each kernel in your code. The thread block size should be a multiple of 32 because the warp size is 32 on Nvidia GPUs. Additionally, the thread block size is limited to 1024. Run the algorithm multiple times and average the time measurements to obtain representative results. When measuring time, the data transfers to and from the GPU must also be included.
- Compute the speed-up $S = t_s/t_p$ of your algorithm for each image size; t_s is the execution time of the sequential algorithm on the CPU, and t_p is the execution time of the parallel algorithm on the GPU.
- Write a short report (1-2 pages) summarising your solution and presenting the measurements performed on the cluster. The main focus should be presenting and explaining the time measurements and speed-ups.
- Hand in your code and the report to ucilnica through the appropriate form by the specified deadline (**12. 4. 2024**) and defend your code and report during labs in the same week.

Bonus tasks (for grades 9-10):

- Use shared memory and partial histograms to reduce the impact of atomic operations when performing histogram computation.

- Parallelize the computation of cumulative histograms using the work-efficient parallel algorithm to build the final result.
- Perform additional optimizations of the algorithm where you see fit. Compare the non-optimized CUDA algorithm to the optimized CUDA algorithm in terms of execution time.

References

[1] Blelloch, Guy E. 1990. "Prefix Sums and Their Applications." Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University.