Parallel Programming 2023
Final assignment

Lucia Giorgi

1. Assignment description

Histogram equalization is a digital image processing method by which you can calibrate the contrast using the image histogram.
This method usually increases the overall contrast of many images, especially when the usable image data is represented by very close contrast values. Through this adaptation, the intensities can be better distributed on the histogram (fig. 1).

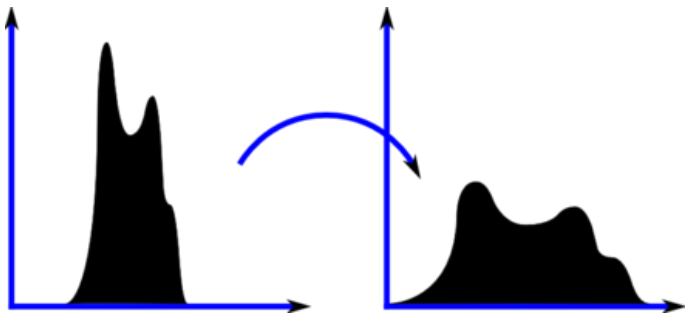

Figure 1: Histogram equalization

2. Implementation

2.1. Histogram equalization

Steps for histogram equalization of an image:

- Compute the histogram: for every possible value of a pixel (0 - 255) count how many pixels have that value (let's call it $p(v)$ for $v \in [0, L-1]$ and $L = 256$).

- Compute the CDF:

$$\text{cdf}(v) = \sum_{k=1}^{v} p(k)$$

- For every image pixel compute the new value:

$$h(v) = \frac{\text{cdf}(v) - \text{cdf}_{\text{min}}}{MN - \text{cdf}_{\text{min}}} \cdot (L-1)$$

  where M and N are the dimensions of the image and cdfmin is the smallest non-zero value of cdf(v).

For a RGB image we have to first convert the color space to HSV, apply the histogram equalization to the *value* channel and then convert it back to RGB to display the result.

2.2. CUDA

The sequential version of the program implements the above steps the way they were explained.
For the parallel version I used three kernels, each one with a specific function:

- generate the histogram from the input image
- calculate the CDF
- generate the output image

The first kernel, `histogram()`, is launched with one thread per pixel ($M \cdot N$ threads) with blocks of fixed dimension (`BLOCK_DIM_X * BLOCK_DIM_Y`). Each block uses a shared memory of `256 * sizeof(int)`. The shared memory is used to store a local histogram (an array) referring to a portion of the image of size `BLOCK_DIM_X * BLOCK_DIM_Y`. This array is initialized to zero for all positions. Each thread adds 1 in the histogram array to the position corresponding to the value of the pixel. Then, the local histograms are added together in the global memory. In this way the access to the global memory is reduced. Each addition must be an `atomicAdd()` because multiple threads write to the same address simultaneously and two `__syncthreads()` have to be used to synchronize the threads in the block.
These steps are summarized in fig. 2.

```
__global__ void histogram(const unsigned char* image, int* hist, int width, int height) {
    __shared__ int HIST[HIST_SIZE];
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int t_id = tx + ty * width;
    int t_id_block = threadIdx.x + threadIdx.y * BLOCK_DIM_X;
    if (t_id_block < HIST_SIZE) { // shared memory initialization
        HIST[t_id_block] = 0;
    }
    __syncthreads();
    if (tx < width && ty < height) {
        atomicAdd(&(HIST[image[t_id]]), 1);  // local histogram
    }
    __syncthreads();
    if (t_id_block < HIST_SIZE) {
        atomicAdd(&(hist[t_id_block]), HIST[t_id_block]); // global histogram
    }
}
```

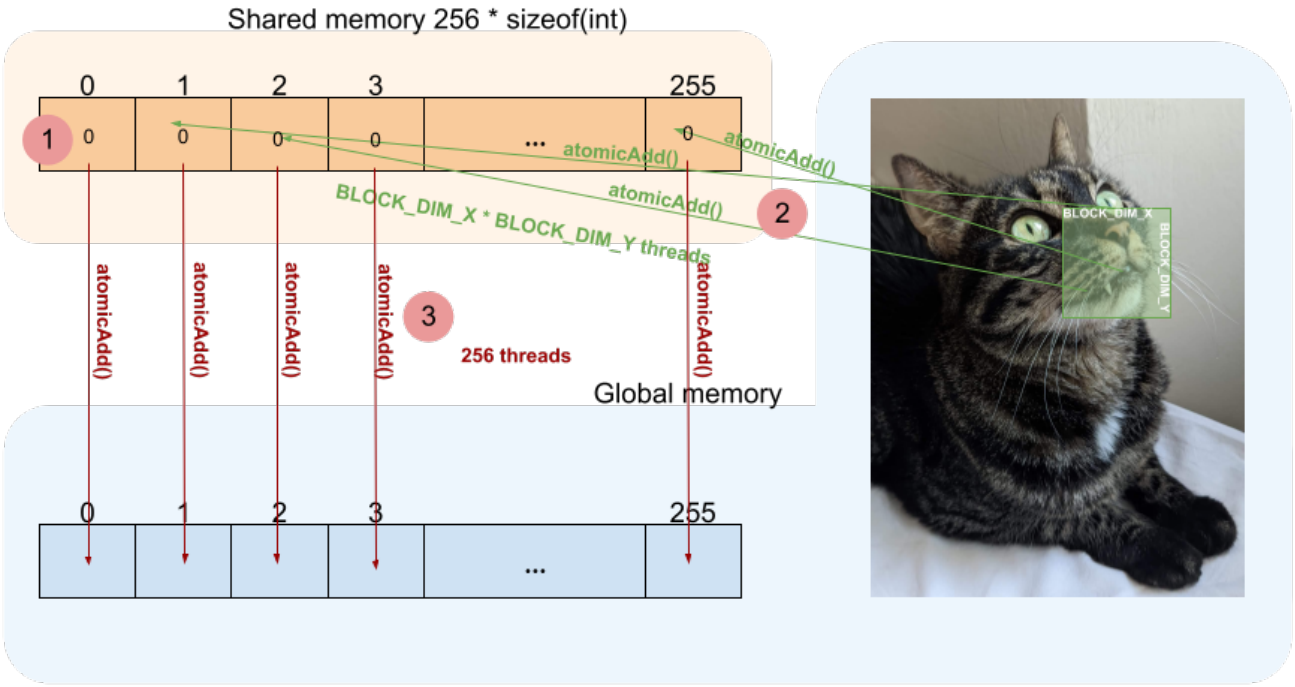

Figure 2: Steps for `histogram()` kernel

The second kernel is a scanner that calculate the CDF from the histogram. The implementation (from [**?**] is based on the Brent–Kung adder (fig 3).
The shared memory is used to write intermediate results and has size `256 * sizeof(int)`. The kernel is launched with 1 block and $256/2 = 128$ threads.

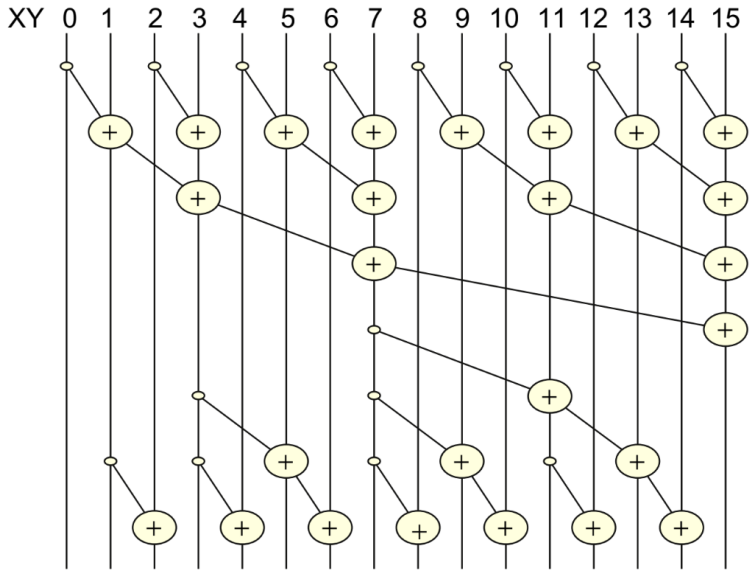

Figure 3: A parallel scan algorithm based on the Brent–Kung adder design

The value of cdf_{min} is computed sequentially because the values in the CDF are ordered.
The last kernel, `equalizer()`, calculates the new value of each pixel for the output image, and it also operates with one thread per pixel ($M \cdot N$ threads) and blocks of `BLOCK_DIM_X * BLOCK_DIM_Y` threads. Each block uses a shared memory of `BLOCK_DIM_X * BLOCK_DIM_Y * sizeof(int)` in order to store a portion of the image and reads the CDF from the constant memory. Then, every thread reads the value of its assigned pixel, applies the formula for the new value and writes the new value back. The values are then copied from the shared memory into the global memory.
These steps are summarized in fig. 4.

```
__constant__ int d_CDF[HIST_SIZE];
__global__ void equalizer(unsigned char* image, int cdf_val_min, int width, int height) {
    __shared__ unsigned char IMG[BLOCK_DIM_X * BLOCK_DIM_Y];
    int tx = blockIdx.x * blockDim.x + threadIdx.x;
    int ty = blockIdx.y * blockDim.y + threadIdx.y;
    int t_id = tx + ty * width;
    int t_id_block = threadIdx.x + threadIdx.y * BLOCK_DIM_X;
    if (tx < width && ty < height) {
        IMG[t_id_block] = image[t_id];
    }
    __syncthreads();
    if (tx < width && ty < height) {
        IMG[t_id_block] = long(d_CDF[IMG[t_id_block]] - cdf_val_min) * (HIST_SIZE - 1) /
                                            (width * height - cdf_val_min);
    }
    __syncthreads();
    if (tx < width && ty < height) {
        image[t_id] = IMG[t_id_block];
    }
}
```

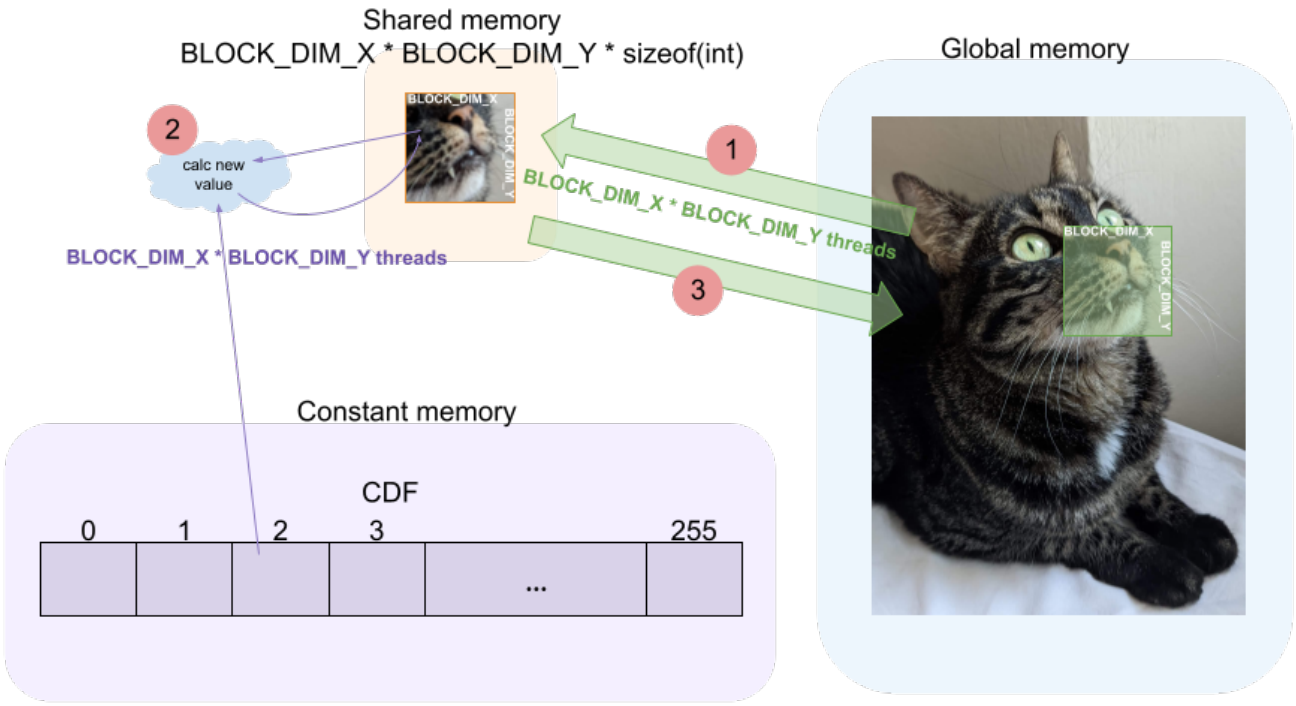

Figure 4: Steps for `equalizer()` kernel

3. Experiments and results

I ran the program with images of different sizes, 10 images for each size (5 horizontal and 5 vertical), and measured the execution time.
The start time is taken after the extraction of the value channel (before calling `histogram()` and before every CUDA instruction in the parallel version) and the end time is taken right after `equalizer()` (again, after every CUDA instruction in the parallel version).
A first "dummy" image is left out from the results because the first execution is considerably slower than the others. I checked with the profiling tool NVIDIA Nsight Systems and this overhead is due to a context initialization and the first `cudaMalloc()` (maybe there is also some caching in place). The execution time for this first image (94×125) ranges from 0.069s to 0.073s, and it's much higher than the typical execution time for the same images ($< 10^{-4}$), therefore it would have altered the average for that specific size and it was intentionally left out from the results. With a large number of images it should not impact on performance anyway.
The parallel version was compiled and executed with different block sizes up to $32 \times 32 = 1024$ by varying `BLOCK_DIM_X` and BLOCK_DIM_Y with `BLOCK_DIM_X * BLOCK_DIM_Y` multiple of 32. The smallest possible size is `BLOCK_DIM_X * BLOCK_DIM_Y` $= 256$ because it's the size of the histogram and therefore at least 256 threads per block are required.
The average execution times and speedups are reported in table 1 and figure 5. As expected, the sequential version is faster only for very small images. Among all the parallel versions, the one with a block size of $32 \times 32 = 1024$ was the slowest because only one block can be allocated for each SM. The speedup increases with the image size with a value of 12.7 for a block size of $32 \times 16 = 512$ for the biggest image size of the experiment 6780×4320.
I used the Occupancy Calculator tool of NVIDIA Nsight Compute to get some occupancy data on the kernels. In order to find the values for the required inputs I used:

- `__host__ cudaError_t cudaGetDeviceProperties (cudaDeviceProp* prop, int device)` to get information about my GPU, such as the total shared memory per multiprocessor.
- `--ptxas-options=-v` flag in `nvcc` compiler to get information about the kernels (once for every block size that was used), like the number of registers.
- NVIDIA website for the compute capability of my GPU

From the gathered data it is clear that the shared memory and registers are not a bottleneck because they are far below the limits, while the block size can be limiting. The occupancy calculated by the tool is 100% for every block size but $20 \times 16 = 320$ (83%) and $32 \times 32 = 1024$ (67%). The tool shows that the maximum number of threads per SM is 1536 (48 warps), but in the 32×32 case only one block of 1024 threads can be assigned to each SM, therefore $1024/1536 = 0.67$.
Some example of images before and after the histogram equalization are showed in fig. 6.

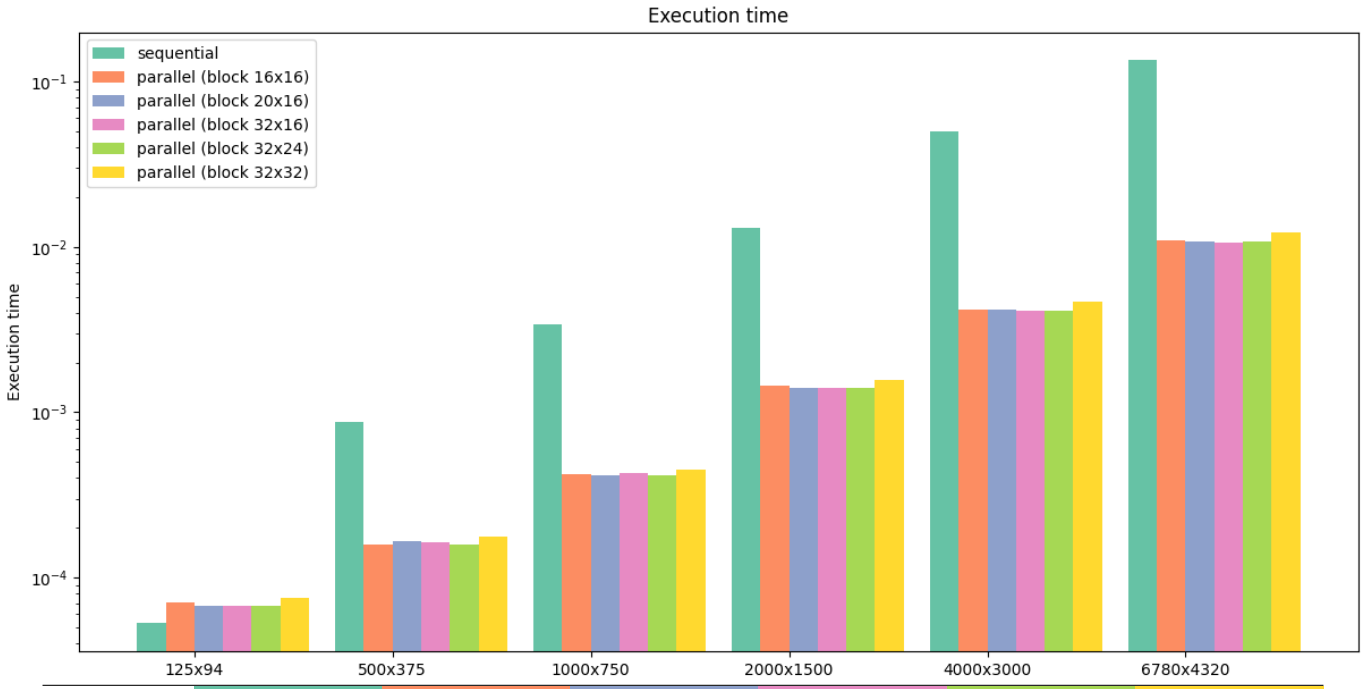

Image size	execution time sequential (s)	execution time 16x16 (s)	execution time 20x16 (s)	execution time 32x16 (s)	execution time 32x24 (s)	execution time 32x32 (s)
125x94	5.3e-05	7.1e-05	6.7e-05	6.8e-05	6.8e-05	7.6e-05
500x375	0.00087	0.000159	0.000165	0.000163	0.000158	0.000176
1000x750	0.003409	0.00042	0.000415	0.000427	0.000414	0.00045
2000x1500	0.013126	0.001457	0.001404	0.001404	0.001404	0.001556
4000x3000	0.049783	0.004177	0.00419	0.004084	0.004086	0.004694
6780x4320	0.134443	0.010917	0.010793	0.010579	0.010857	0.012192

Table 1: Average execution times on different image sizes (5 horizontal + 5 vertical) on various block dimensions

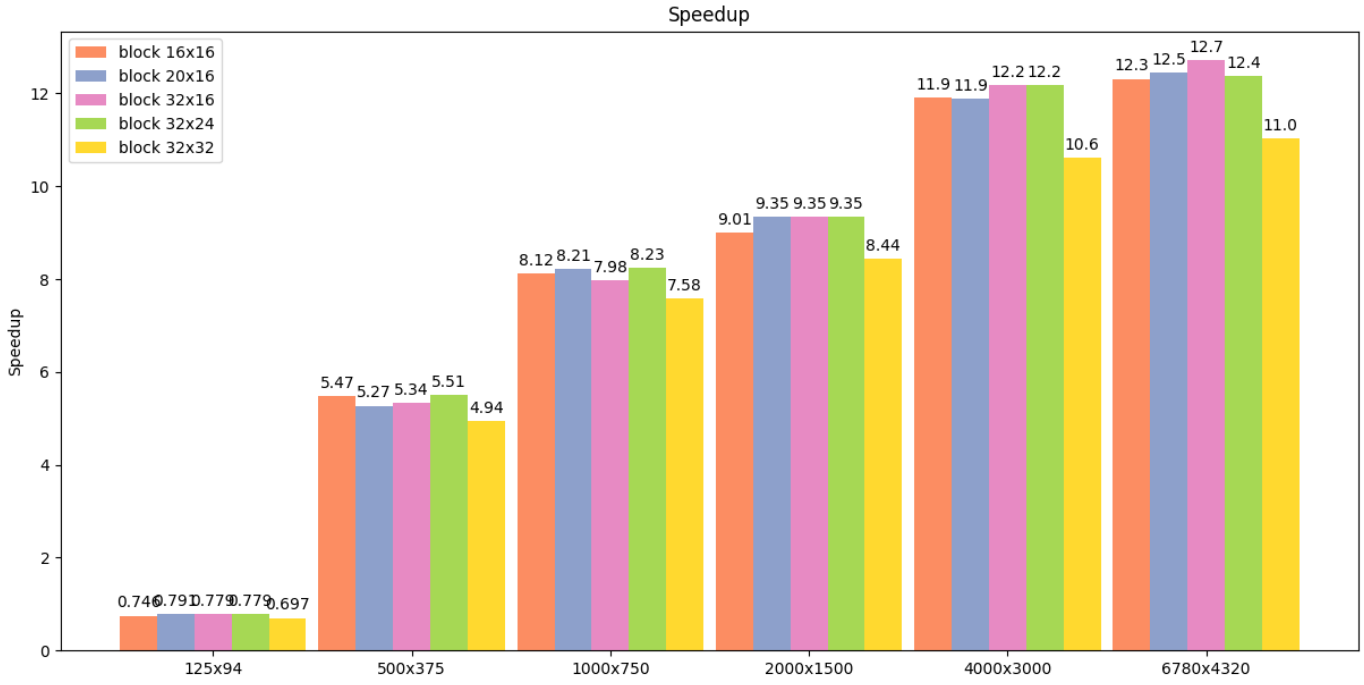

Figure 5: Average speedup measured on different image sizes (5 horizontal + 5 vertical) for various block dimensions

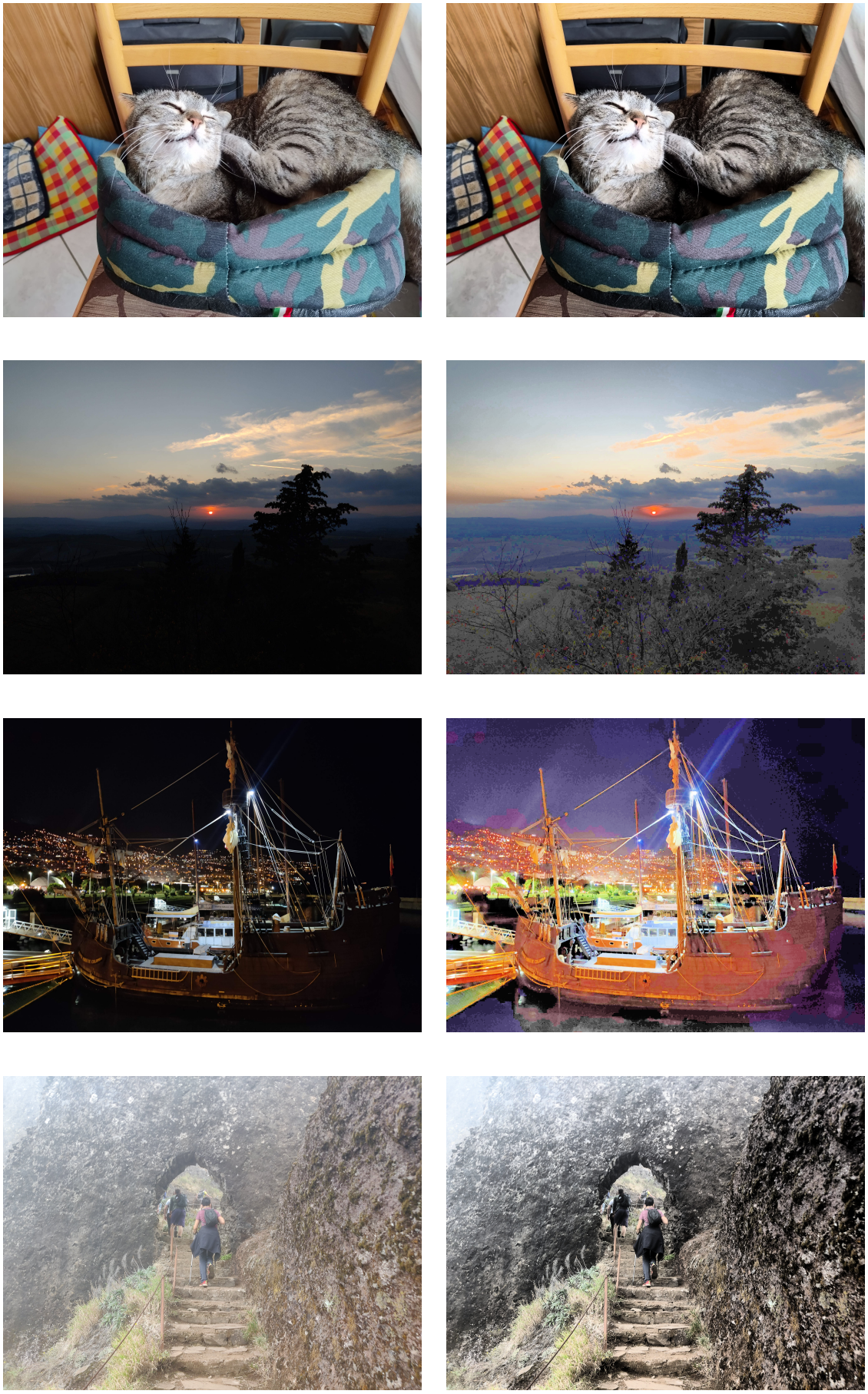

Figure 6: Images before and after histogram equalization