Histogram Equalization

VPSA - Final Project Assignment

Anže Šavli, 63200281

February 2023

Table of contents

1. Introduction ... 3
2. Parallel implementation of histogram equalization in CUDA ... 4
   2.1. Computing image histogram ... 4
   2.2 Computing the cumulative distribution ... 4
   2.3 Transform function ... 4
3. Experiment ... 5
   3.1 Hardware specifications ... 5
   3.2 Images specifications ... 5
   3.3 Measuring methodology ... 5
4. Results and discussion ... 6
5. Conclusion ... 11
6. Sources ... 11

Table of Images

Image 1: Alps before eq ... 6
Image 2: Alps after eq ... 6
Image 3: Baby before eq ... 6
Image 4: Baby after eq ... 6
Image 5: Car before eq ... 7
Image 6: Car after eq ... 7
Image 7: Cats before eq ... 7
Image 8: Cats after eq ... 7
Image 9: Kolesar before eq ... 7
Image 10: Kolesar after eq ... 7
Image 11: Nature before eq ... 8
Image 12: Nature after eq ... 8
Image 13: Panda before eq ... 8
Image 14: Panda after eq ... 8
Image 15: Shoes before eq ... 8
Image 16: Shoes after eq ... 8
Image 17: Space before eq ... 9
Image 18: Space after eq ... 9
Image 19: Window before eq ... 9
Image 20: Window after eq ... 9

Table of figures

Figure 1: Chart of average ms spent running. ... 10
Figure 2: Chart representing time increase in CUDA. ... 10

1. Introduction

In this assignment I will implement parallel algorithm using CUDA to equalize a histogram. Histogram represents an 8-bit grayscale image. To achieve the CUDA parallelization I will parallelize three steps (image histogram computing, cumulative distribution computing, and image transform function). I will achieve that by using methods like binary tree reduction, atomic operations and shared memory. I will then test the algorithms on 10 different images of various sizes and graylevel distributions. I will then compute the average time needed for 10 iterations of equalization for both CUDA and sequential implementation and compare the results. To achieve the best results possible, I will run the code on NSC.

2. Parallel implementation of histogram equalization in CUDA

In this section I will describe which steps I managed to implement in CUDA and how that was achieved.

2.1. Computing image histogram

First step I managed to implement in CUDA was the step of computing the image histogram. I achieved that using atomic operation. Firstly, I created three variables (dev buffer, dev histogram and histogram of size of amount of graylevels). I then allocated needed memory and set histogram values to 0 using cudaMemset(). After that I defined global kernel function that checks image pixel graylevel and adds plus 1 to count. After successfully implementing CUDA I added shared memory to eliminate competition for memory addresses. The drawback with my method is that the thread count must be the same as colour range, so if we had more that 1byte colour range, the thread count would have to be greatly increased and the method would become unusable for anyone trying to calculate a colour range greater than 1024 since that is the most threads most graphic cards have available per block.

2.2 Computing the cumulative distribution

After histogram computation I moved on to parallelizing CDF calculation. Firstly, I filled shared memory with data from histogram calculated in the first step. I then perform binary tree reduction to calculate the prefix sum of the histogram. Which in other words means I performed parallel reduction where each value was the sum of all previous values and current value. I managed to parallelize that by dividing histogram into smaller parts and saving intermediate results into shared memory and repeating this process until it was no longer possible to split it more. The problem with this method is that banking conflicts can occur. Which occur because multiple threads access the same bank of shared memory.

After parallelizing CDF, I started working on parallelizing find minimum value in CDF. I did that by using reduction algorithm. I filled CDF values into shared memory and compared values in for loop until I had a single value stored in thread 0. I achieved that by using min function and I eliminated zeros by checking if value equals to 0 and in that case took the max value of given numbers. I could implement this step inside of kernel that computed CDF, but I decided that I would make it separate for better clarity, since it didn’t impact performance that much due to pictures only having 256 graylevels.

2.3 Transform function

I parallelized this step using a very simple and non-complex solution. I passed all pointer to output image, input image data, image size and CDF data into my function and calculated new value for each pixel calculated by combining block id and thread id and by using scale function, which took CDF minus minimum and divided it by image size minus minimum and multiplied that with graylevels-1.

3. Experiment

In this section I will describe my working environment, I will give short description of images used and steps taken to measure the performance.

3.1 Hardware specifications

For running the code, I used supercomputer provided by the faculty. NSC is composed of 1984 cores (Intel Xeon E5-2640 and AMD Opteron 6376) and 16 Nvidia Tesla Kepler 40 graphics cards. It also includes 9216 GB of RAM and runs using Centos 8 operating system.

3.2 Images specifications

For the test I ran the code 10 times on 10 different images. I tried to capture variety of different sizes and levels of grays, to try and get different results and stress test the code.

- alps-neq.jpg - image of Europe mountains with a lake house and boats. Dimensions: 3840x2160px
- baby-neq.jpg - image of a baby in a spaghetti costume in kitchen. Dimensions: 1080x1080px
- car-neq.jpg - image of a car on driveway with a lot of greens. Dimensions: 6000x4000px
- cats-neq.jpg - image of two cats hugging. Dimensions: 564x564px
- kolesar-neq.jpg - image of a cycler going up a mountain. Dimensions: 1024x640px
- nature-neq.jpg - image of trees and a lake. Dimensions: 2560x1440px
- panda-neq.jpg - image of a superhero panda. Dimensions: 128x128px
- shoes-neq.jpg - image of a girl wearing Nike socks and shoes. Dimensions: 828x1001px
- space-neq.jpg - image of an astronaut in space. Dimensions: 45x80px
- window-neq.jpg - image of an airplane window looking at clouds. Dimensions: 300x533px

3.3 Measuring methodology

For measuring performances, I ran code 10 times and compared average running time. For calculating CUDA performance I used CudaEvent_t variables start and stop that I created with cudaEventCreate and started recording it with CudaEventRecord, before calling kernels. For calculating sequential performance I used clock() function from <time.h> library. I set start before calling functions for image manipulation. I then subtracted current clock() with start and divided it with clocks_per_sec and then multiplied it with 1000 to get milliseconds. I choose to put measurements around functions, because I only wanted to get the difference in processing performance while neglecting time needed for memory allocation.

4. Results and discussion

In this section I will present results and comparisons in time needed with both methods. I will also present the image before the equalization and after equalization.

1. Alps-neq.jpg (3840x2160)
    - Average time CUDA: 3.70309 ms
    - Average time sequential: 183.1 ms

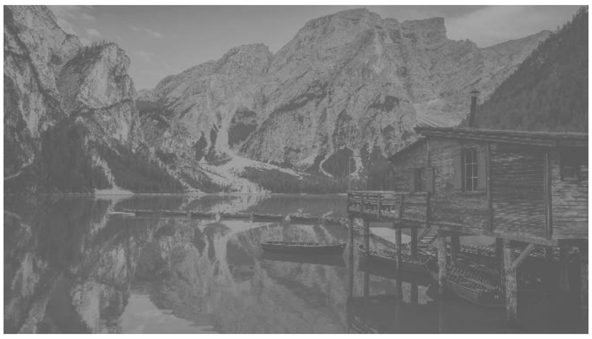
Image 1: Alps before eq

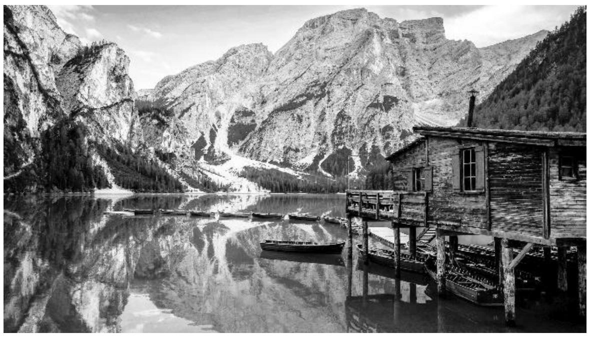
Image 2: Alps after eq

2. Baby-neq.jpg (1080x1080)
    - Average time CUDA: 0.76296 ms
    - Average time sequential: 25.2 ms

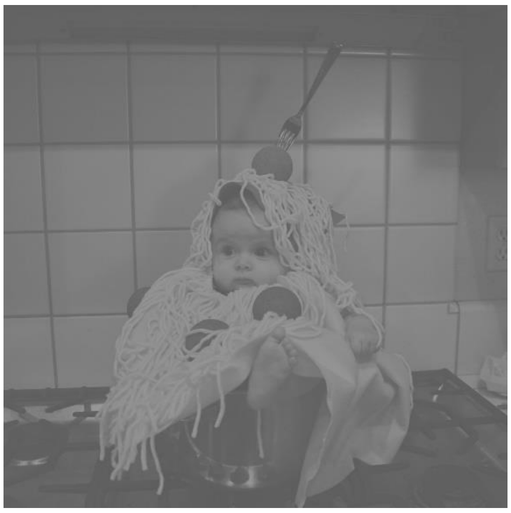
Image 3: Baby before eq

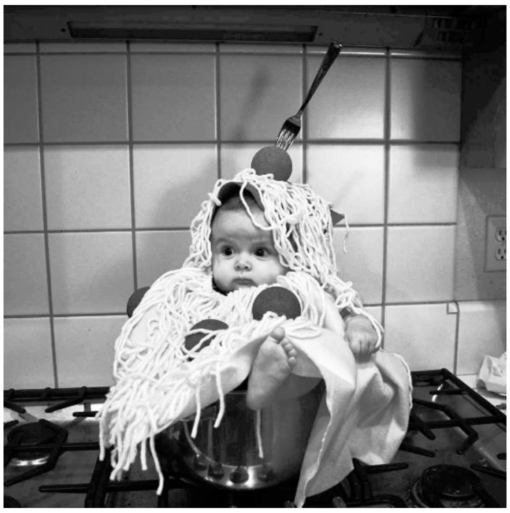
Image 4: Baby after eq

3. Car-neq.jpg (6000x4000)
   - Average time CUDA: 16.89280 ms
   - Average time sequential: 532.0 ms

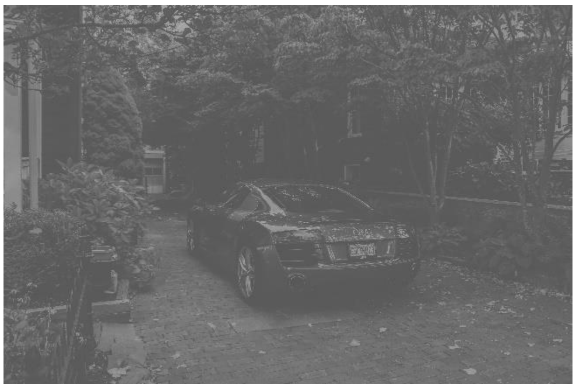
Image 5: Car before eq

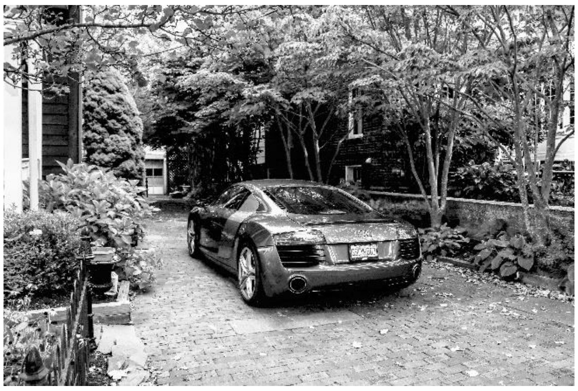
Image 6: Car after eq

4. Cats-neq.jpg (564x564)
   - Average time CUDA: 0.29303 ms
   - Average time sequential: 7.0 ms

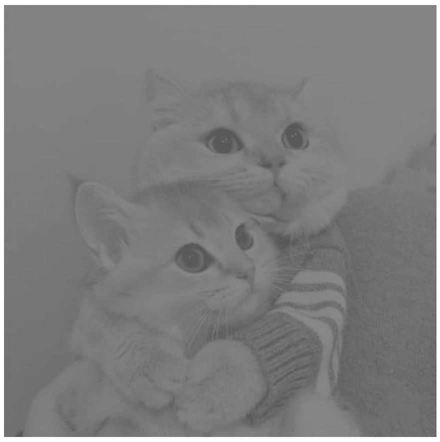
Image 7: Cats before eq

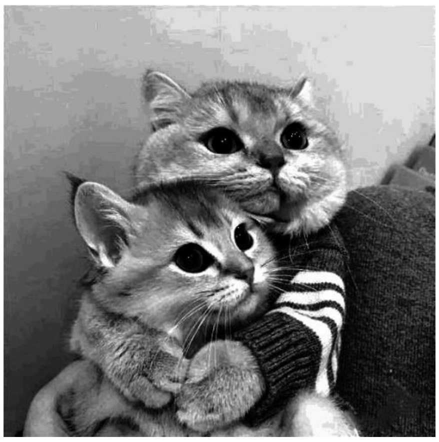
Image 8: Cats after eq

5. Kolesar-neq.jpg (1024x640)
   - Average time CUDA: 0.42180 ms
   - Average time sequential: 14.0 ms

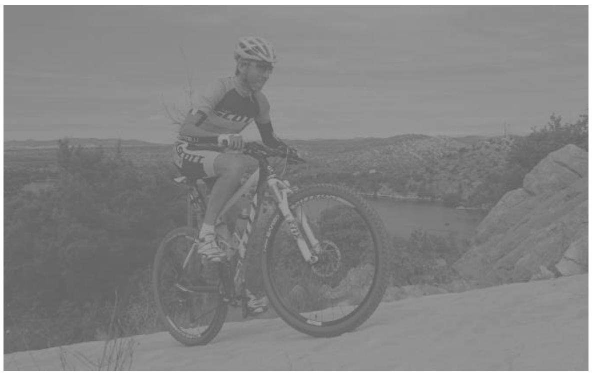
Image 9: Kolesar before eq

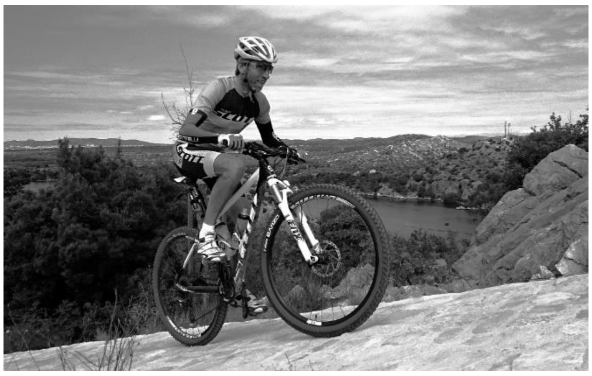
Image 10: Kolesar after eq

6. Nature-neq.jpg (2560x1440)
   - Average time CUDA: 1.79200 ms
   - Average time sequential: 80.30000 ms

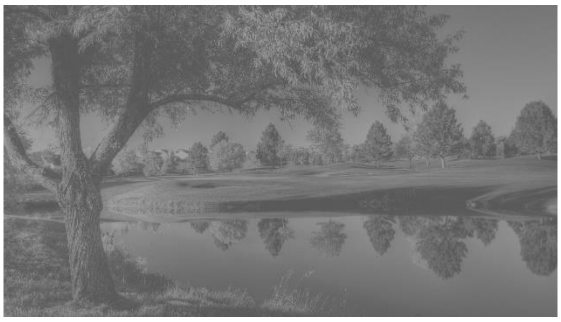
Image 11: Nature before eq

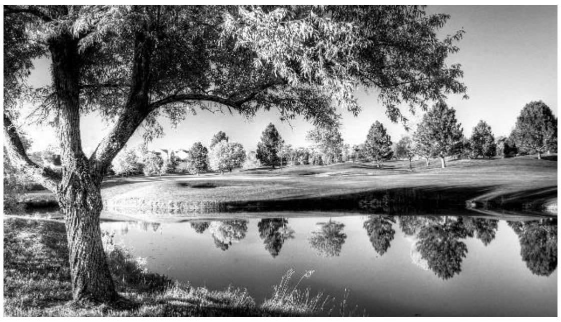
Image 12: Nature after eq

7. Panda-neq.jpg (128x128)
   - Average time CUDA: 0.09182 ms
   - Average time sequential: 0.00000 ms

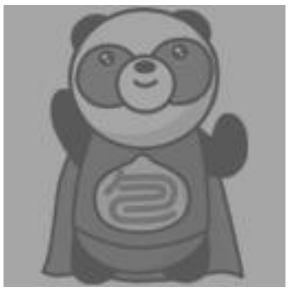
Image 13: Panda before eq

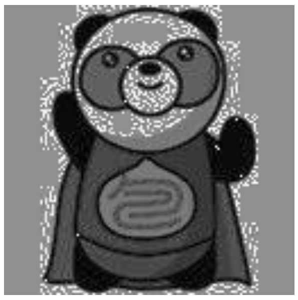
Image 14: Panda after eq

8. Shoes-neq.jpg (828x1001)
   - Average time CUDA: 0.48390 ms
   - Average time sequential: 18.00 ms

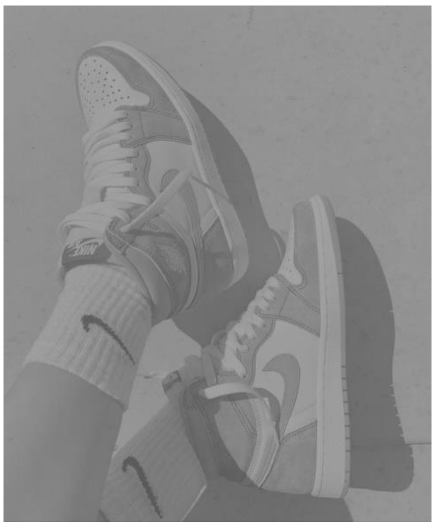
Image 15: Shoes before eq

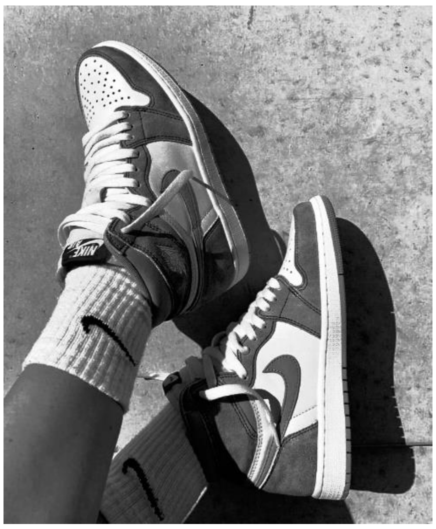
Image 16: Shoes after eq

9. Space-neq.jpg (45x80)
    - Average time CUDA: 0.06106 ms
    - Average time sequential: 0.00000 ms

Image 17: Space before eq

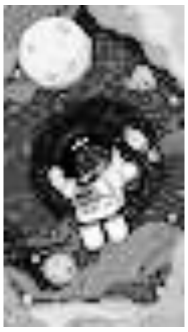
Image 18: Space after eq

10. Window-neq.jpg (300x533)
    - Average time CUDA: 0.13924 ms
    - Average time sequential: 3.00000 ms

Image 19: Window before eq

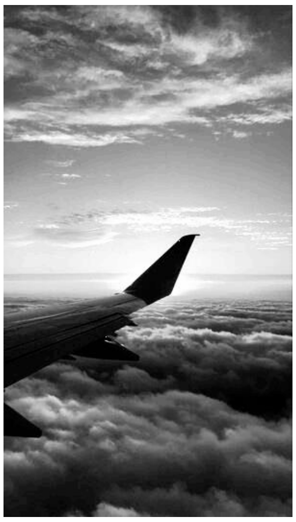
Image 20: Window after eq

From the tests and results above, we can see that CUDA given around 20-40 times the performance boost over sequential method. CUDA losses to sequential at around 175x175 pixels due to amount of pixels being so low. When looking at results we need to take into consideration that the tests were run purely to time functions performance, and not time needed for memory allocation and other necessary steps. That means that sequential code would probably be faster for a few hundred pixels more and would fall to CUDA performance at around 250x250 pixels. We can also see that performance and speed varies greatly based on graylevel arrangement.

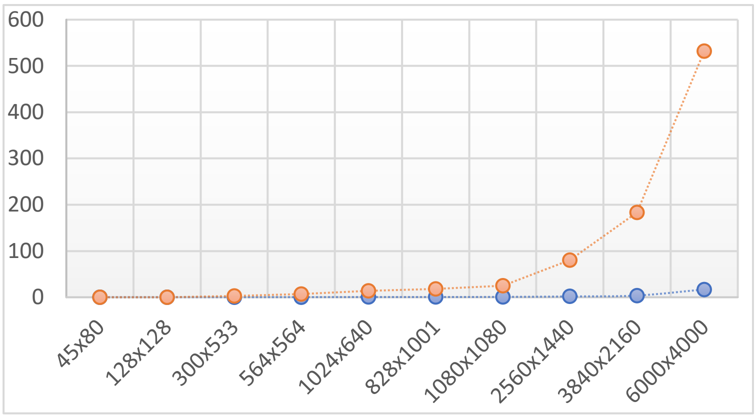

Figure 1: Chart of average ms spent running.

The following chart represents the amount of milliseconds functions needed to execute equalization on given image. The orange line represents the sequential mode, while blue line represents equalization with CUDA. We can see that time for sequential equalization greatly increases with the number of pixels the image has.

The following chart represents how many times CUDA equalization is faster than sequential equalization. Y-axis represents the speed-up ratio, and the X-axis shows the image dimensions. We can see that image size and graylevels distribution greatly impacts the number of times CUDA will be faster than sequential equalization.

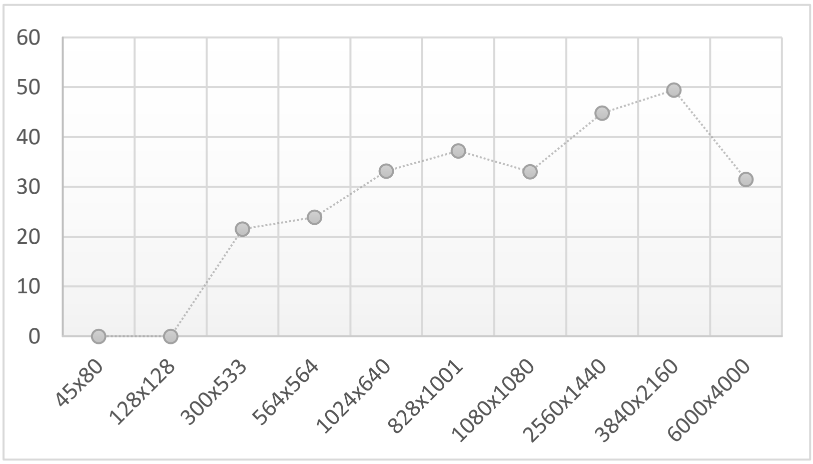

Figure 2: Chart representing time increase in CUDA.

5. Conclusion

After successfully implementing CUDA optimization and running all the tests I came to a conclusion that with larger images CUDA implementation makes a lot of sense, while sequential equalization is better for small images like icons or avatars. Overall I'm happy with my CUDA implementation, but it could be further improved by implementing technologies that help with backing algorithms, it could also be improved by fixing edge cases where images with bad graylevel distribution result into bad images. A possible improvement would also be a better grid and block layout, to make better use of the GPU abilities.

6. Sources

- VPSA online classroom
- Memory management. NVIDIA Documentation Center. (n.d.). Retrieved February 4, 2023, from https://docs.nvidia.com/cuda/cuda-runtime-api/group__CUDART__MEMORY.html
- Chapter 39. parallel prefix sum (SCAN) with Cuda. NVIDIA Developer. (n.d.). Retrieved February 4, 2023, from https://developer.nvidia.com/gpugems/gpugems3/part-vi-gpu-computing/chapter-39-parallel-prefix-sum-scan-cuda
- Pinterest. (n.d.). Retrieved February 4, 2023, from https://www.pinterest.com/
- CUDA by Example. (n.d.). Retrieved February 4, 2023, from https://ucilnica.fri.uni-lj.si/pluginfile.php/199483/mod_resource/content/0/CUDA_by_Example.pdf