Histogram Equalisation

Introduction to Cuda and OpenCL

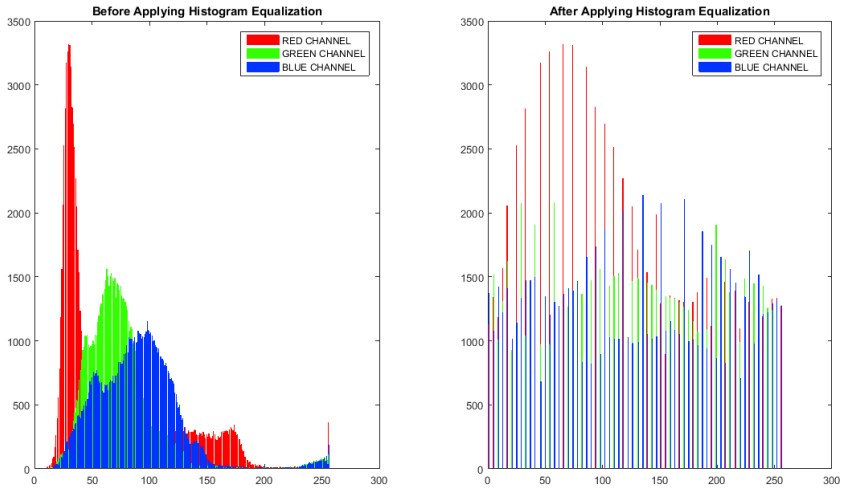
Matej Horvat & Vicente Martínez Orts

Introduction

In this report we will explain the concepts of histogram equalisation and develop a normal c and a CUDA program that can do that. We will explain the techniques used and analyse implementation.

Histogram Equalisation

Histogram equalisation is a technique used in image processing to enhance the contrast of an image. It works by redistributing the intensity values of the pixels in an image such that the resulting histogram (a graph showing the distribution of intensity values) is more evenly distributed.



Example of the histogram equalisation in action

This can be useful in situations where an image appears too dark or too light, as it can help to bring out details that were previously obscured. The process of histogram equalisation can be thought of as a method for "stretching" the intensity values of an image, making the darkest pixels darker and the lightest pixels lighter. This results in a more visually pleasing image that is easier to interpret. Example of this can be seen in the comparison on the two images below.

Image before and after histogram equalisation

Implementation

The algorithm works in 4 stages:

1. Create the histogram of the input image for each colour channel
2. Calculate the cumulative distribution for each colour
3. Calculate new colour level values using the below histogram equalisation formula
4. Assign the new transformed values to each colour channel of a pixel

$$l_{new} = round\left[\frac{cdf(l_{old}) - cdf_{min}}{N \times M - cdf_{min}} \times (L-1)\right]$$

$$cdf(l_{old}) = \sum_{i=0}^{l_{old}} n_i$$

$N \times M$ → dimensions of the image
L → number of color levels per channel
cdf → cumulative distribution
cdf_{min} → minimum value of the cdf, which is larger than 0
n_i → number of pixels with color level i

Improvements to the basic algorithm

We implemented our CUDA histogram equalisation with two improvements that increased our performance.

Histogram

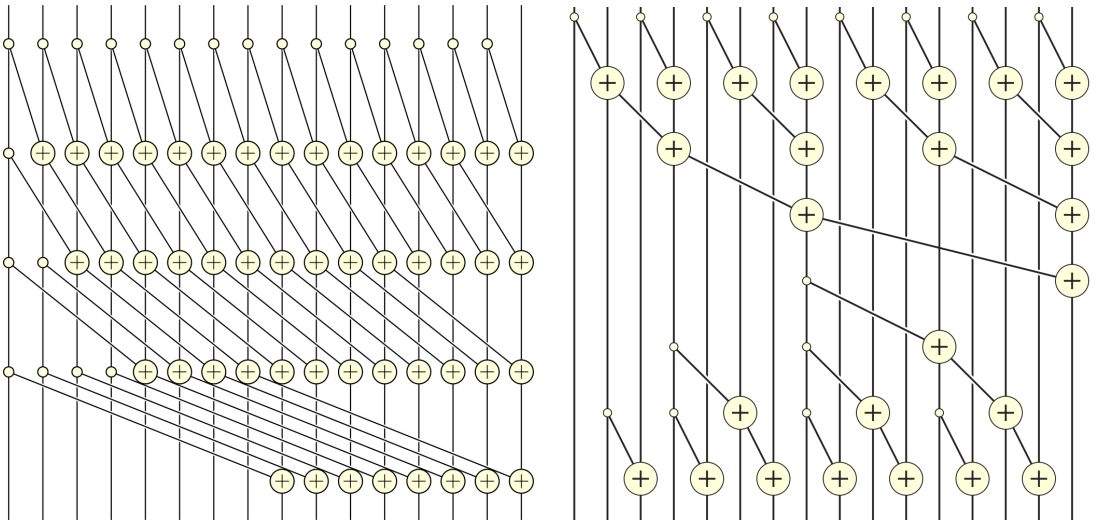
For creating the histogram we use shared memory arrays for each colour channel to reduce the amount of waiting that our atomicAdd operations need to do in each thread. After that, we reduce the local histograms to the global ones. The atomic operations are mandatory because they are needed to avoid race conditions and give us the correct result every time.

Cumulative distribution calculation

For calculating the cumulative distribution we implemented the work efficient Blelloch algorithm which follows these basic steps:

1. Divide the input data into segments of size 2^n (n is a positive integer).
2. Perform an inclusive prefix sum operation on each segment. This is done by adding each element in the segment to the element that precedes it.
3. Combine the intermediate sums from each segment to get the final prefix sum for the entire data set.
4. Repeat steps 2 and 3 recursively to calculate the prefix sum of each segment.
5. Propagate the final prefix sum back through the segments to get the final cumulative sum for each element.
6. Add the offset of the corresponding segment to each element to get the final cumulative sum.

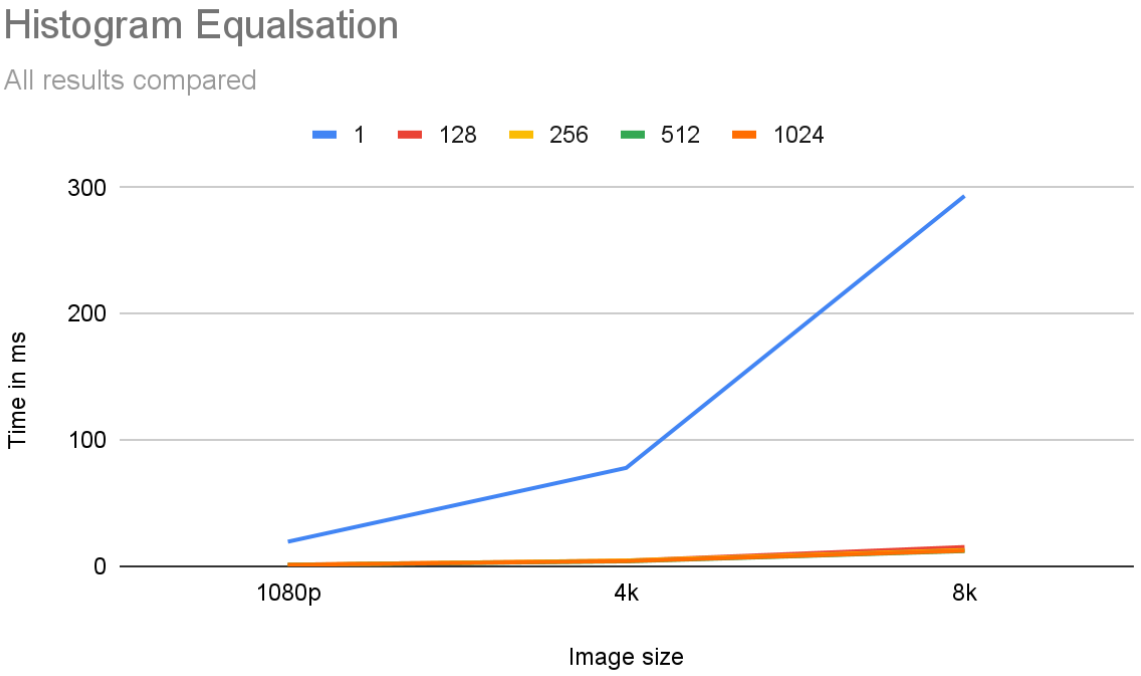
The algorithm is highly efficient due to the use of the "recursive doubling" technique, which allows it to calculate the prefix sum of each block in logarithmic time.



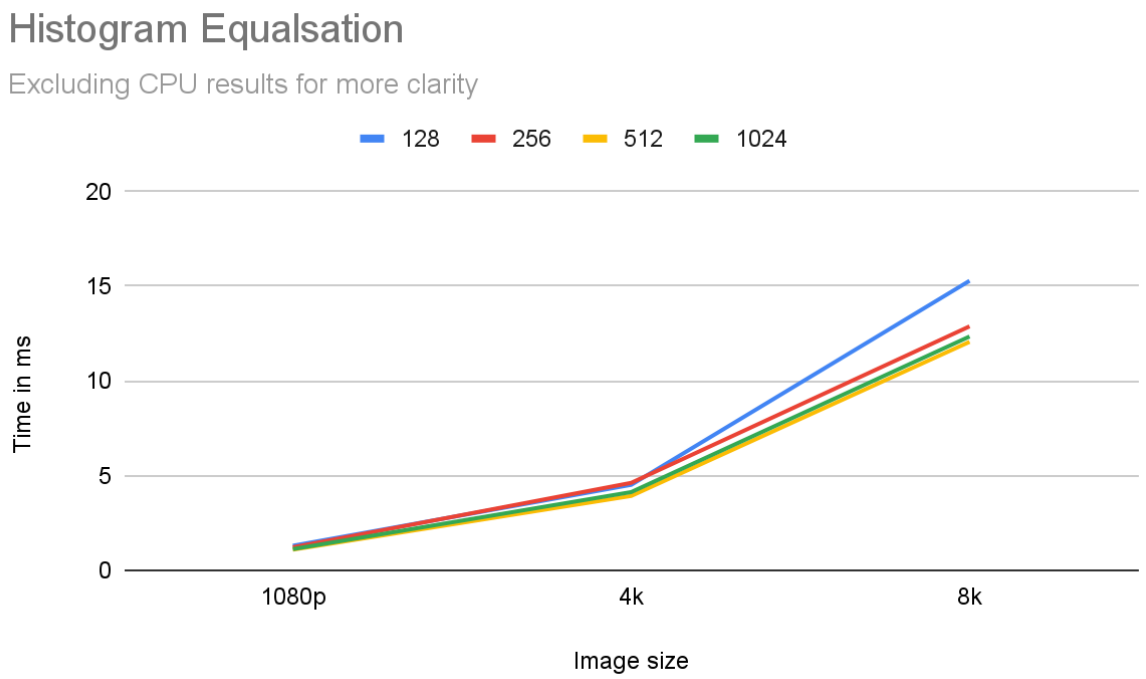
Difference in accessing of the normal and work efficient algorithm

Analysis

We analysed the performance of both the CPU and CUDA algorithms. For both algorithms we used images of sizes 1080p, 4k and 8k where each image is 4x times larger than the previous one. In cuda we also tested the performance on blocks of sizes 128, 256, 512 and 1024. The results are presented in the graphs below.



We can see that the CPU implementation (shown in blue) is almost 20x times slower than the CUDA implementation with 8k images. In the following graph we excluded the CPU implementation to better show the differences between block sizes.



In the above graph we can see that different block sizes had actually really similar performance where 512 produced the best results in all tests and other sizes had mixed results. The worst time we got was when running the implementation on an 8k image with 128 threads which was 16 percent slower than the next one, which is still only 2.4 ms slower.

It is worth noting that for the calculation of cumulative distribution, a block size of 256 was used. This was done because the histograms of the 8-bit images used in this report always have 256 elements. Additionally, to ensure efficiency, the calculation was performed using separate blocks for each colour channel.

Conclusion

In conclusion, the analysis has shown that in our case the different block sizes did not have a significant effect on the execution time of the CUDA algorithm where block size of 516 produced the best results. In comparison with the CPU algorithm the CUDA one was approximately 20x times faster.