Histogram Equalization

Distributed and parallel systems and algorithms

Mateusz Kowalewski

1. Introduction

Histogram equalization algorithm transforms histogram of an input image into (almost) equally distributed one. We can say that this algorithm makes image use all colours equally. The result of such action is higher contrast of an image. We represent picture as an array of picture elements called pixels. In my implementation of algorithm, I assumed that I am dealing with 8-bit monochromatic pictures.

2. Parallel implementation of histogram equalization in OpenCL

The steps for implementing histogram equalization algorithm are:

1. Create the histogram for the input grayscale image

   In this step we need to compute image histogram. In 8-bit picture we have 256 possible values for each pixel. We just count the number of occurrences of each colour through whole image. As a result, we have an array of 256 elements which values represents the number of times each colour is present on an image.

   In my implementation I decided to use local memory to speed up the algorithm. First, I assign the local and global id's and sizes. Next calculate the staring and ending point for each work group and calculate the local id in 1D representation. We need to remember about assigning 0 value for each colour in histogram before we start the main part of algorithm. After this step we need to put a barrier in order to wait for all the local work items to finish. The next step is calculating local histogram for each group using two loops and atomic operations also after that we need a barrier to wait for local work items. The last step of this part is adding local histograms to a global one using a for loop.

2. Calculate the cumulative distribution histogram.
   In this part we need to calculate the cumulative distribution histogram. It means that we need to sum up all the previous elements up to the element we are calculating histogram value. According to this the value of the last element must be a sum of all elements in histogram. In my implementation I used the simplest but not the quickest solution. Every work item calculates one value of a cumulative histogram using atomic operation and a for loop. As a result of not using local memory my algorithm doesn't share data across work group and that makes it slower than it could be.

3. Calculate and assign the new grey-level values of histogram
   That part of an algorithm is divided into 3 functions. One responsible for finding a minimum value in cumulative distribution histogram above 0, another one that calculates the new values of histogram and last main one that assign the values to the histogram.

   The function that finds minimum uses parallel reduction algorithm. Every work item compares the two adjacent elements and writes the one with a lower value into the array.

After every step we need to wait for the work items to finish so there is a barrier. We also need to divide the number of work items by 2. That simple approach is an efficient way to calculate the minimal value of cumulative histogram above 0. The weak point of my implementation is not using local memory so it's not as fast as it could be.

The second function that calculates the new values for histogram is sequential. There is nothing to parallelize there.

The main function invokes to other function and collects results. It is also responsible for assigning new values for each pixel of an image. My implementation uses global memory only so it's slower than it could be.

3. Experiment

1. Hardware:

- CPU - Intel Core i5-7300HQ @ 2.50GHz
- GPU - NVIDIA GeForce GTX 1050 (Laptop)
- RAM – 8GB

2. Measuring methodology

In case of CPU I decided to measure time from the invocation of function that calculates cumulative histogram to the end of the function that calculates new values of histogram. I skipped the freeimage library operations. On a GPU I measured time from enqueue of the first kernel to the point where the last kernel finished its work. In both cases I used ten images and measured time of the program execution ten times. Then I made an average of those measurements.

4. Results and discussion

Table 1 Execution time on GPU vs CPU

Image size [pixels]	CPU average execution time[s]	GPU average execution time[s]	Speed- up
512 x 512	0.025	0.003	8,33
300 x 200	0.006	0.001	6
1024 x 1024	0.112	0.01	11,2
600 x 338	0.019	0.003	6,33
720 x 523	0,039	0.004	9,75
228 x 226	0,005	0.001	5
1228 x 670	0,111	0.009	12,33
320 x 240	0,008	0.002	4
1600 x 448	0,074	0.007	10,57
800 x 679	0,055	0.005	11

In my algorithm the parallel is faster than the sequential one every time. This is a result of the starting and ending points in measuring time on GPU and CPU. But we can see that the larger image the greater the speed-up.

Few examples of histogram equalization algorithm:

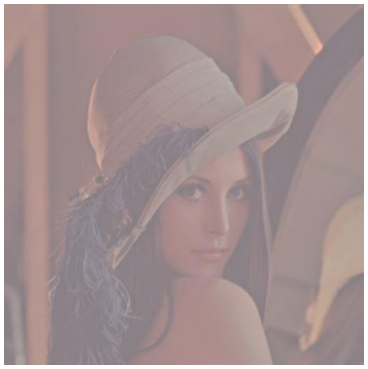

Figure 1 Before histogram equalization algorithm

Figure 1 After histogram equalization algorithm

Figure 2 Before histogram equalization algorithm

Figure 2 After histogram equalization algorithm

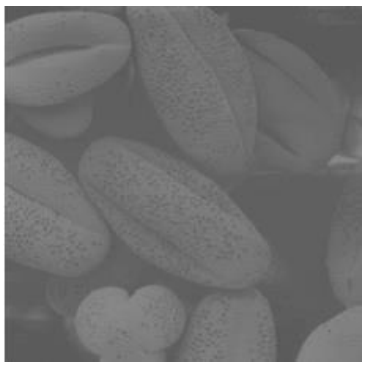

Figure 3 Before histogram equalization algorithm

Figure 1 After histogram equalization algorithm

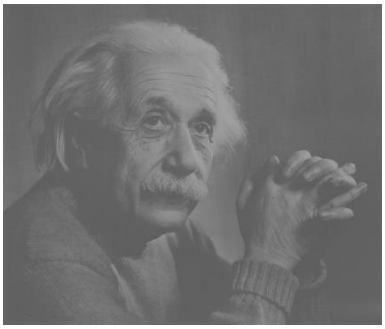

Figure 4 Before histogram equalization algorithm

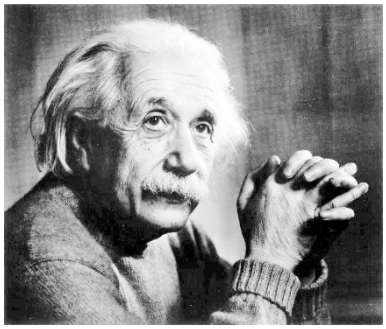

Figure 4 After histogram equalization algorithm

5. Conclusion

The parallel algorithm is faster than sequential one in most cases, but it is also harder to implement. Only on the small images sequential algorithm might be faster than parallel one. In my implementation the possible improvement is using a local memory instead of global one. I could also implement cumulative distribution histogram in a smarter way.