**Introduction**

I was tasked with developing a digital image enhancement program that performs contrast adjustment using the histogram equalisation algorithm in C++ using OpenCL. The program should use a cumulative intensity histogram to back-project original image intensities resulting in an image of fully equalised intensities. The algorithm will work as follows:



**Implementation**

**Calculate intensity histogram from input image.**

The basic implementation of this requires an understanding of how to extract intensity data from a given image. In a greyscale image the only value provided is intensity, how black or white a given pixel is, so I can just read that data from an image and use that as shown. 

However, for more complicated image formats there are different methods required to extract intensity. For a standard RGB image, there is no intensity channel, so I need to convert them into a format that does contain an intensity channel, for this algorithm I decided that YCbCr would be appropriate, as it does contain intensity and doesn’t lose out on any colour data. I first detect if an image is in fact RGB by checking the amount of colour spectrums it contains:



I now need to convert the image to YCbCr, use the intensity channel as our image\_input value and store the Cb and Cr channels for reconstruction later, there is a built-in CImg function for this conversion, and then I can read in the channels one by one and store them as required.

Text

Description automatically generated

Now that the intensity channel has been extracted, I can construct a histogram in parallel storing this data, histograms operate by storing data in certain amounts of bins, as a basic 8 bit colour image can contain 256 colours it makes sense to use 256 bins by default, but in case the user wants to change this number I implemented a variable bin count for this:

A screenshot of a computer

Description automatically generated with medium confidence

This bin number is then used to construct the histogram vector, and cache histogram size for later:

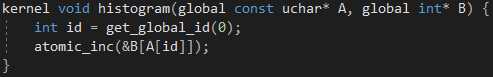
Text

Description automatically generated with medium confidence

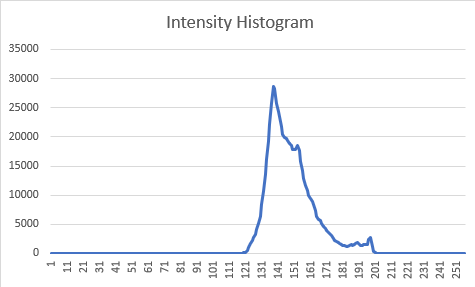
Now that everything has been calculated, I create all of the device buffers that will be used in kernel operations: Text

Description automatically generated

Next, I move on to the first step, calculating the histogram:



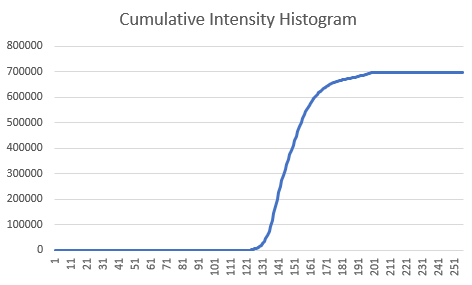
The kernel function increments the histogram at a given bin index in parallel, using atomic functions to ensure that the array is accessed in a safe way. The histogram kernel ran on the test image produces this histogram:



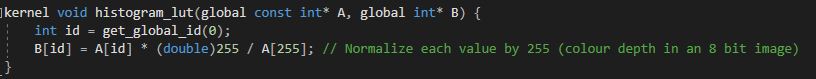
Now that the histogram is calculated I need to convert it into a cumulative histogram in parallel, to do this I’m going to again use atomic operations to ensure sequential operations, then use a kernel to add together all the previous values. Some sort of locking is important here, as this is an operation that needs to be performed in a guaranteed order, and introducing a race condition by neglecting to implement locking/barriers would be detrimental to the output of the program as it could break (Wingate, 2022).Text

Description automatically generated

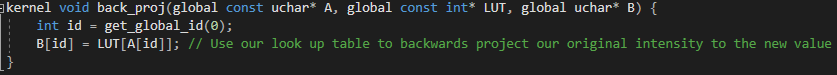
The cumulative histogram kernel ran on the test image produces this histogram:



Now the cumulative histogram is calculated, it needs to be normalised and scaled, this operation doesn’t need to be done in a guaranteed order so I can go back to performing calculations properly in parallel. As a look-up table (LUT) also needs to be constructed, these normalised and scaled values are going to be directly stored as the LUT.



The values are now normalised properly, provided the image is 8-bit. Now that the LUT is constructed I can backwards project the original values to calculate what the new intensity should be:



Finally, now that all of the values have been properly intensity equalised and stored, I can “reconstruct” the image: Text

Description automatically generated

For an RGB image I need to reconstruct it the same way I deconstructed it, as a YCbCr image, so I use the output\_image buffer as the Y (intensity) channel, and use the cb and cr channel caches I created earlier, I then convert this into an RGB image so it can be properly displayed. Providing these result with the test images provided:  

The average execution time and memory transfer was also calculated and provides the following results over 50 executions: Chart, bar chart

Description automatically generated Chart, bar chart

Description automatically generated

As evident the histogram takes the longest time to calculate and requires the most memory transfers despite being only a simple atomic incrementation. This is due to the large amount of data it is required to process as it takes in an array that is the size of the images width x height, whereas the cumulative histogram, although it uses serial atomic operations, has a far smaller set of data to operate on and therefore takes up less total execution time.

The full profiling statistics:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Execution Time [ns] | Memory Transfer Queued [us] | Memory Transfer Submitted [us] | Memory Transfer Executed [us] | Memory Transfer Total [us] |
| Histogram | 985534 | 47 | 393 | 1693 | 2134 |
| Cumulative Histogram | 136608 | 17 | 28 | 136 | 182 |
| LUT | 4256 | 10 | 20 | 4 | 35 |
| Back-Projection | 584712 | 39 | 883 | 546 | 1469 |

**References**

Wingate, J (2022) ‘CMP3752M Parallel Programming: Lecture 6: Communication & synchronisation’ Available at: <https://blackboard.lincoln.ac.uk/ultra/courses/_181985_1/cl/outline>. Accessed: 24/03/2022.