Parallel Programming Assignment 1 Report

Nate Y.F - 25051591

March 2023

1 Overview

In this assignment, we were asked to implement the histogram equalisation algorithm on a greyscale test image to change the colour intensities in an image. The basic program required us to be able to do this generating the histogram globally, and then using Hillis-Steele to scan the histogram before normalising it and adjusting the colour intensity values of the image. A lot of the code for this was provided in workshops and lectures, and required application as described in this document. Algorithms such as local histogram generation and the Blelloch scanning algorithm were also provided.

The first step of the algorithm was to create the intensity histogram. There are two methods of doing this - atomically and based on local memory. Atomically, we just get the global ID, make the value of the pixel at that index the bin index, and increment it to the histogram at that same index. There is no need to factor in. To do this locally, we simply need to factor in local histograms which are atomically incrememnted, which then get combined into a global histogram.

The next step is to get a cumulative histogram. In the basic implementation one needs to use the Hillis-Steele scanning method for this. This retrieved a global ID and the global size, as well as declares an array of integers (called C, both in code and hereon out), and within a for loop multiplying the stride by two each time it runs, up until the global size, sets the value of the cumulative array at the current ID to the one of the intensity array at current ID, and provided the ID is not smaller than the stride, also adds the value of the intensity array at index ID-stride. The array C is used as a placeholder to swap arrays between steps. In a non-basic implementation, we use Blelloch to scan and create a cumulative histogram. The key differences implementation is that it only takes in one histogram, and implements an integer for the downsweep rather than an integer array to swap with. The upsweep works similarly to the Hillis-Steele algorithm's for loop, but instead it checks if the number one above the ID is a multiple of the stride multiplied by two. A barrier is performed to sync this step. To downsweep, one must set the last value of the array to 0, and then construct a for loop in a similar fashion to Hillis-Steele, but instead of the stride incrememnting by multiplying by two, you must divide. Instead,

the integer declared is assigned the value of the array at the current ID, and then the array at the current ID is incrememnted by the array's value at index id-stride. the value of A at id-stride is set to the prior integer, and the loop continues. A barrier is then put in place to sync this step.

Normalising the histogram and back propagation are the next two steps. Normalisation is a simple process, as it requires creating a scale for the image. This is equal to 256 divided by the maximum number in the histogram, usually in the final index. This is put into the normalisation kernel along with the cumulative histogram and a buffer that will hold the normalised histogram. There, the value of the normalised histogram at that specific ID will be the value of the cumulative histogram multiplied by the scale. The normalised histogram can be used to get an image by putting it through a back propagation kernel.

For additional functionality, one could add the ability to run the program on colour images, to run the program on a user defined number of bins, to use different scanning algorithms etc. While previous algorithms and code could be used to implement this, original code had to be produced to allow one to run these functions on the image with the additions. These are declared in the program as norm_hs and makeImg respectively, and are small kernels work wise.

In order to allow the user to run the program on colour images, one needs to convert from RGB to YCbCr. The Y value takes in the colour intensity of the pixel currently being worked on rather than dealing with the individual colour channels. C++ has a built in "RGBtoYCbCr()" function, as well as one for the reverse. These are used to support colour imagery at user request. Options in the console are provided to choose the scan method, histogram creation method, number of bins, and image. The images used in Tutorial 2 are renamed to testColour and test_largeColour respectively to allow for testing of the program on colour images.

As part of this assignment we needed to provide metrics for the histogram generation algorithms and the histograms generated, as well as histograms, and get them in graph form. Events can be used to generate these metrics, and the program puts them into a file titled "metrics". The performance metrics are as follows:

Scan Algorithm	Queued	Submitted	Executed	Total
H-S, Small	6	131	11	149
H-S, Large	3	100	10	114
Blelloch, Small	4	117	18	140
Blelloch, Large	3	106	17	127

Histogram Generation	Queued	Submitted	Executed	Total
Global, Small	4	265	234	503
Global, Large	3	264	5851	6119
Local, Small	7	356	19	383
Local, Large	3	259	376	640

In this case, the number of bins is set to 256 for consistency. The main observation is that for all image types, creating the intensity histogram locally is the most efficient way of creating these histograms, being nearly 10% more efficient in the case of larger images. There is, however, a divergence in which scanning algorithm to use based on image size. Blelloch is slightly more efficient for scanning histograms than Hillis-Steele is, but not by a huge amount. Blelloch, however, is a lot more efficient for use on larger images. This is because in the case of Blelloch, there are more elements than number of bins for larger images, while this is not necessaril the case for smaller images.

References

1]Cielniak, G. (no date) Gcielniak/opencl-tutorials: Opencl tutorials, GitHub. University of Lincoln. Available at: https://github.com/gcielniak/OpenCL-Tutorials (Accessed: March 23, 2023).

[2]Ghafoor, M. (2023) 'Histogram - Parallelisation' [Lecture], CMP3752-2223: Parallel Programming. University of Lincoln.