Parallel Programming Report

In this project the assignment tasked us with creating a digital enhancement program that will perform contrast adjustment using the histogram equalisation algorithm in which the runtimes and specific will be discussed in the document.

In this algorithm, we were tasked with using multiple parallel programming patterns in this implementation; these consisting of but not limited to, Scatter, Scan and Map. The first histogram created using a simple kernel script (labelled "hist_simple") uses the Scatter pattern to gather a large amount of data from a vector, in this case being the input image, and dropping values in separate bins to which a value corresponds.

Although a Stencil pattern was not used in this implementation it is important to note that should any other filtering or blurring/sharpening algorithms were to be implemented would adopt this; a process similar to interpolation in regular image processing terms.

Another parallel pattern that was utilised in this implementation uses the Scan pattern which can utilise the reduce and scan algorithms to best optimise array sorting in parallel when it comes to larger scale projects.

The final parallel pattern that is used is the map pattern which applies a simple operation to all elements in a data structure such as C[id] = A[id] + B[id] which unifies work groups outputs into one singular output.

Next the steps of this implementation will be discussed:

The first objective of this project is to create a basic histogram of the intensities of an input image and separate these values from each pixel into a range of 255 bins equal to the range of an RGB intensity. This is done by first using the "hist_simple" script found in Tutorial 3 of James Wingate' OpenCL tutorials code, then creating a buffer based on the size of the input image to ensure the kernel has enough resources to complete this operation. See Figures 1 and 2 for output results of this kernel operation on both greyscale (test.pgm from tutorial 2) and coloured (test.ppm for this project). The same has been done for the large variants of both

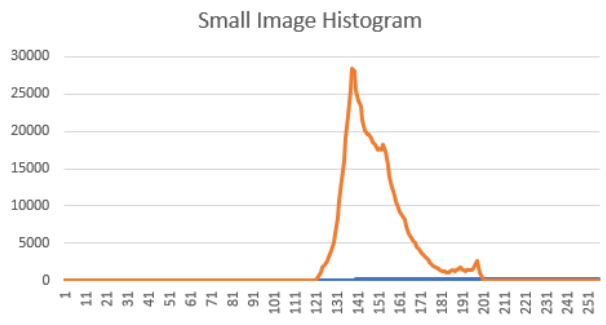


Figure 1. Small Greyscaled Image Histogram

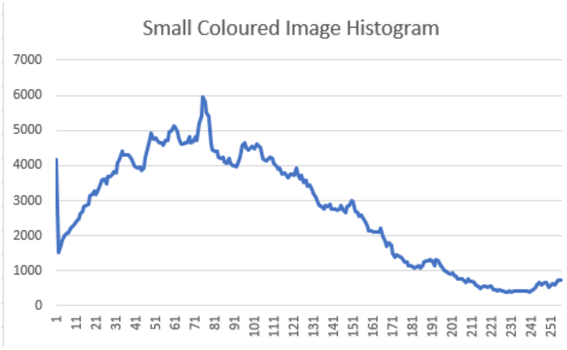


Figure 2. Small Coloured Image Histogram

The second objective requires this histogram to be turned into a cumulative histogram which is shown in Figures 5 and 6. This is useful later on for developing the lookup table for back projection. The graph values shown have been taken from greyscaled images but work for both. See the kernel code for "hist_cum" for better insight into its workings.

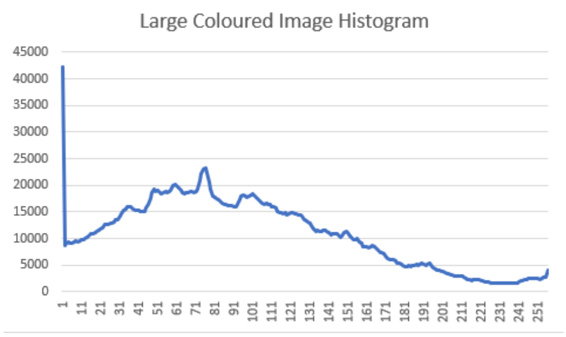


Figure 3. Large Coloured Histogram

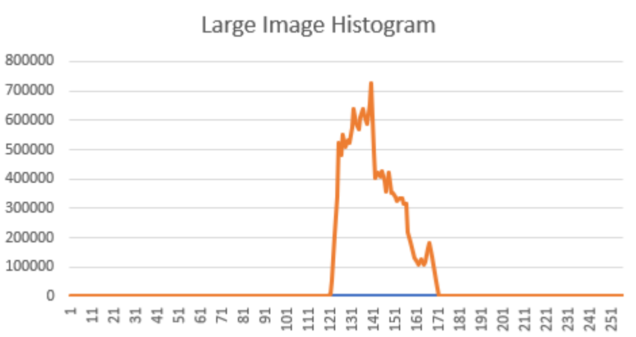


Figure 4. Large Greyscaled Histogram

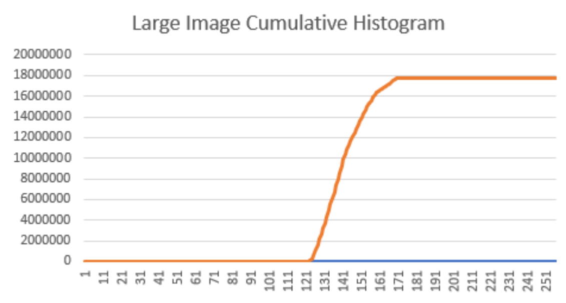


Figure 5. Large Cumulative Histogram

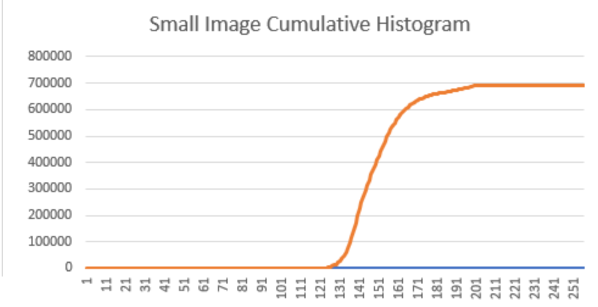


Figure 6. Small Cumulative Histogram

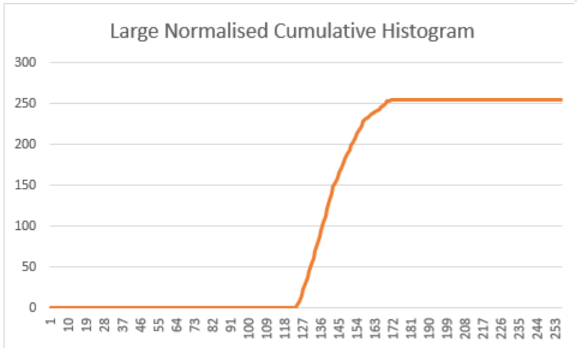


Figure 7. Large Normalised Histogram

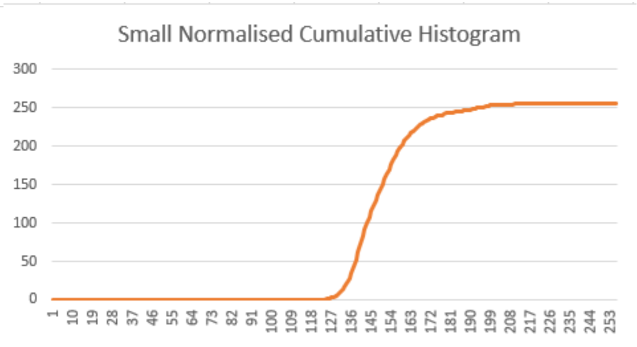


Figure 8. Small Normalised Histogram

The third step of this project is to normalise the values of the histogram by iterating through the elements shown in the graph and dividing each element by the highest value and then multiplying this by 255 to get an RGB readable value for the lookup table in the next step. See Figure 7, 8 and “norm_cumHist” and “norm_cumHistG” in the kernel code attached.

The fourth objective of this project is to use the cumulative histogram as a look-up table for mapping over the original intensities of the output image, this is shown by Figures 9 and 10. This can be used for greyscale images but will not restore or change colour in a coloured image.

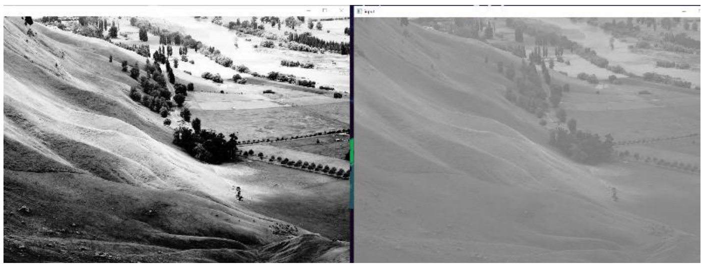


Figure 9. right to left, greyscale to altered image.

The fifth and final objective for this project is to use the normalised histogram to alter the intensities of the output image to receive an alteration as seen in Figure 10. This is obtained by first checking whether the spectrum count of the image is equal to 1 meaning greyscale or 3 meaning containing colour channels. When 3 channels have been detected, the RGB image is converted to the format YCbCr so the channel values can extracted from input image. These are then reassigned back to a blank CImg data structure in a way that uses a stencil to change the colour intensities of local pixels to get a desired output. While Figure 9 does show an intensity equalised image, this method doesn't consider colour channels.

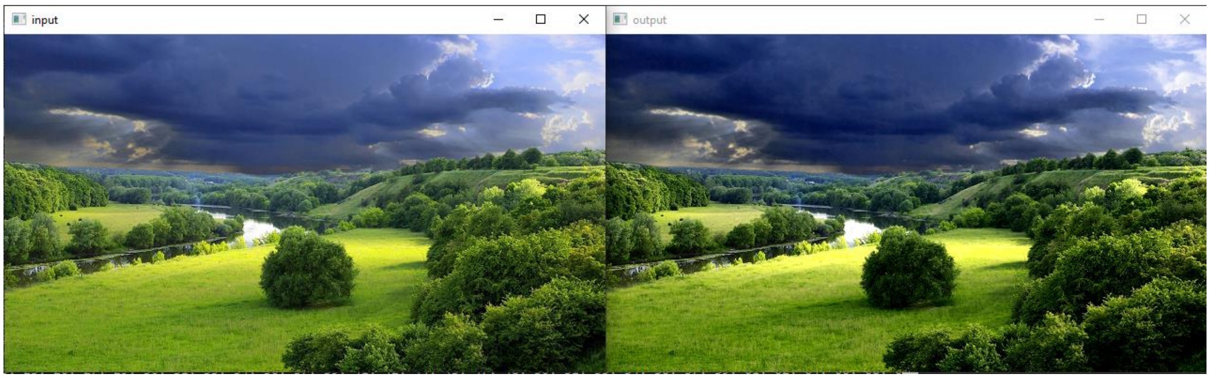
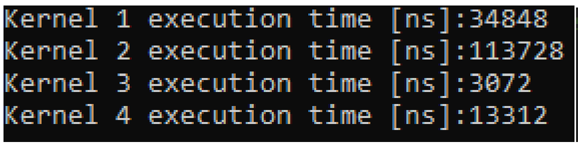


Figure 10. Colour intensity altered images left to right.

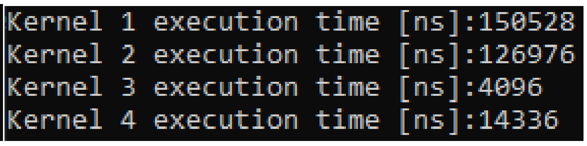
Serial computation involves the use of atomic functions, as seen in some of the kernel functions to stop raised conditions when work instances try to access the same item at once which could cause duplicate results. This process is considerably slower than working without them as shown in the profiling and runtimes section near the end of this document. Parallelised functions allow for must quicker runtimes as they pull resources solely from the global cache instead of from global memory. This ensures the resources are readily available for the kernel to work with to produce an output.

Global runtimes are slower than local runtimes because it is stored further away from the computation and has to be found before any computation can take place, this might also include latency when locating and retrieving resources which further burdens the operation. Whereas local memory would be stored beside the runtime to ensure quicker access and reduce computational power demands from each operation. This has been tested in the submitted implementation by testing runtimes on GPU & CPU for the "hist_simple" and "norm_cumHist" kernels, as seen in Figures 11 and 12. This test only worked for kernels 1 and 3 as other kernel testing was limited by coding knowledge and time constraints.



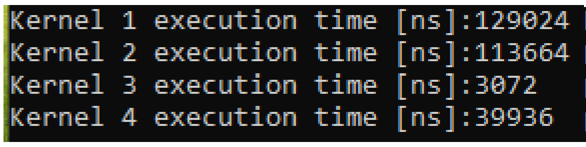
```
Kernel 1 execution time [ns]:34848
Kernel 2 execution time [ns]:113728
Kernel 3 execution time [ns]:3072
Kernel 4 execution time [ns]:13312
```

Figure 11. GPU Local Execution time (ns)



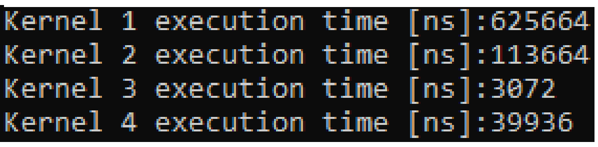
```
Kernel 1 execution time [ns]:150528
Kernel 2 execution time [ns]:126976
Kernel 3 execution time [ns]:4096
Kernel 4 execution time [ns]:14336
```

Figure 12. CPU Global Execution Time (ns)



```
Kernel 1 execution time [ns]:129024
Kernel 2 execution time [ns]:113664
Kernel 3 execution time [ns]:3072
Kernel 4 execution time [ns]:39936
```

Figure 13. CPU Local Execution Time (ns)



```
Kernel 1 execution time [ns]:625664
Kernel 2 execution time [ns]:113664
Kernel 3 execution time [ns]:3072
Kernel 4 execution time [ns]:39936
```

Figure 14. CPU Execution Time

While this implementation isn't the most optimised, the usage of local memory usage has greatly reduced the power needed for the whole program to run and making it more efficient overall. However kernels 2 and 4 require optimisation for a more confident analysis of the two.

References:

Wingate, J. (2019) OpenCL Tutorials. https://github.com/wing8/OpenCL-Tutorials

Mattson, T.G., Sanders, B. and Massingill, B., 2004. *Patterns for parallel programming*. Pearson Education.

Blelloch, G.E., 1989. Scans as primitive parallel operations. *IEEE Transactions on computers*, *38*(11), pp.1526-1538.

Hillis, W.D. and Steele Jr, G.L., 1986. Data parallel algorithms. *Communications of the ACM*, *29*(12), pp.1170-1183.