# An Executive Summary of a Histogram Equalisation Solution for Digital Image Enhancement

Stewart Charles Fisher II
ID: 25020928
25020928@students.lincoln.ac.uk

March 2023


UNIVERSITY OF
LINCOLN

School of Computer Science
University of Lincoln
United Kingdom

Submitted in partial fulfilment of the requirements for the CMP3752M Parallel Programming Module

Word Count: 1444

# Contents



# List of Figures



# List of Tables

# 1 OpenCL Kernel Functions

The OpenCL framework allows for the execution of functions, known as kernels, across heterogeneous platforms (The, 2013). Parallel patterns are recurrent blueprints that are used to implement algorithms for parallel computation, allowing large amounts of data to be processed with higher efficiency. My solution contains eight kernel functions:

- Two intensity histogram functions.
- Four cumulative histogram functions.
- A look-up table function.
- A back-projection function.

## 1.1 Implementing the Kernel Functions

### 1.1.1 intHistogram

`intHistogram` uses data parallelism (IEvangelist, 2021) to execute the kernel code across multiple elements of data in parallel. Each work-item will operate on a different index of the input array and uses atomic incrementation on the respective output array, to ensure that the histogram bins are correctly updated.

### 1.1.2 intHistogram2

`intHistogram2` uses data parallelism, similar to `intHistogram` except each work-item will write to a local buffer before being reduced into the final intensity histogram. The reduction pattern sums together the local histograms from the work-groups into a global histogram. Barrier functions are used to ensure that all of the local histograms have finished calculating before any reduction can execute.

### 1.1.3 cumHistogram

`cumHistogram` uses a scan pattern, or prefix sum pattern. In this function, each work-item will compute the sum by looping over the remaining elements and adding it to the current element, which represents the current cumulative sum. Using atomic addition ensures that the updates to the output array are performed without triggering a data race (Corporation, 2023). Since the output sequence doesn't include the current element in the sum, this implementation is an exclusive scan.

### 1.1.4 cumHistogramB

`cumHistogramB` uses the Blelloch pattern (Mills, 2017) to execute the cumulative sum. This pattern divides the input array into blocks, computes the cumulative sum for each block, and then propagates the partial results to the following blocks until the final sum is computed. The 'up-sweep' computes the partial sum using a reduction operation, while the 'down-sweep' computes the final sum using a binary tree operation on the partial sums.

### 1.1.5 cumHistogramHS

`cumHistogramHS` uses the Hillis-Steele pattern (Mills, 2017) to execute the cumulative sum. Like `cumHistogramB`, this function also uses a stride method but it differs by copying the values from the input buffer to the output buffer and adding in the value at the previous stride if possible. After every stride, the buffers are swapped around. While this pattern is easier to implement, it requires a larger amount of memory to store the intermediate buffers.

### 1.1.6 cumHistogramHS2

`cumHistogramHS2` also uses the Hillis-Steele pattern, except it uses local memory to reduce the number of global memory accesses, by using the two local memory arrays; `cumHistogramHS` performs all of the algorithm in global memory whereas `cumHistogramHS2` executes the entire algorithm in local memory, improving the performance in places where global memory accesses could be a bottleneck.

### 1.1.7 lookupTable

`lookupTable` computes a look-up table to map the original intensity values onto the image, taking in values from the cumulative histogram and outputting the normalised values. The `maxIntensity` variable is determined by the bit width of the image; it is 255 for an 8-bit image, and 65535 for a 16-bit.

#### 1.1.8 lookupTable2

`lookupTable2` computes a look-up table similarly to `lookupTable`, however it also allows the histogram equalisation to maintain a higher level of accuracy across different bin sizes.

#### 1.1.9 backprojection

`backprojection` uses the Radon transform algorithm (Radon, 1986) to back-project the final image. The input array contains the image data in the Radon domain, the `LUT` argument is the look-up table, and the output array will contain the back-projected image. The value of an output pixel is set by indexing the look-up table with the original intensity value from the input buffer.

#### 1.1.10 backprojection2

`backprojection2` is another function that is optimised, to allow for higher accuracy with variable bin counts.

### 1.2 Drawbacks of the Kernel Functions

While `cumHistogramHS` is capable of producing a histogram, the image that it outputs appears to not be fully processed. I have not been able to figure out what causes this issue. Due to the fact that a histogram is still being calculated, I have decided to leave it in the final solution.

### 1.3 Testing the Kernel Functions

| Kernel Function | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
|---|---|---|---|---|---|---|
| **intHistogram** | 241664 | 239616 | 240640 | 241440 | 240640 | **240800** |
| **intHistogram2** | 163712 | 163840 | 164768 | 159744 | 166752 | **163763** |
| **cumHistogram** | 24448 | 23552 | 29376 | 24480 | 24576 | **25286** |
| **cumHistogramB** | 9216 | 30656 | 28832 | 9216 | 13312 | **18246** |
| **cumHistogramHS** | 34816 | 10336 | 9216 | 8192 | 10528 | **14618** |
| **cumHistogramHS2** | 8192 | 12288 | 8192 | 8064 | 29728 | **13293** |
| **lookupTable** | 8192 | 9216 | 29696 | 9088 | 7968 | **12832** |
| **lookupTable2** | 7168 | 7040 | 6144 | 6048 | 7008 | **6682** |
| **backprojection** | 9216 | 9024 | 8192 | 10240 | 8192 | **8973** |
| **backprojection2** | 360448 | 358400 | 359168 | 359424 | 359392 | **359366** |

Table 1: A table of the tested performance times for each kernel function, in nanoseconds.

I tested the performance of the kernel functions using a bin count of 256, and using the image `test.pgm`. The testing results in Table 1 show that the best combination of kernel functions is:

- `intHistogram2`
- `cumHistogramHS2`
- `lookupTable2`
- `backprojection2`

`intHistogram2` is the best option due to how it employs data reduction and local memory. Similarly, `cumHistogramHS2` is the best option because it is entirely executed in local memory, which will greatly increase the performance. In future implementations, I could look at the look-up table and the back-projection and see where they could be improved upon. An example of the optimal combination can be seen in Figures 1 and 2.

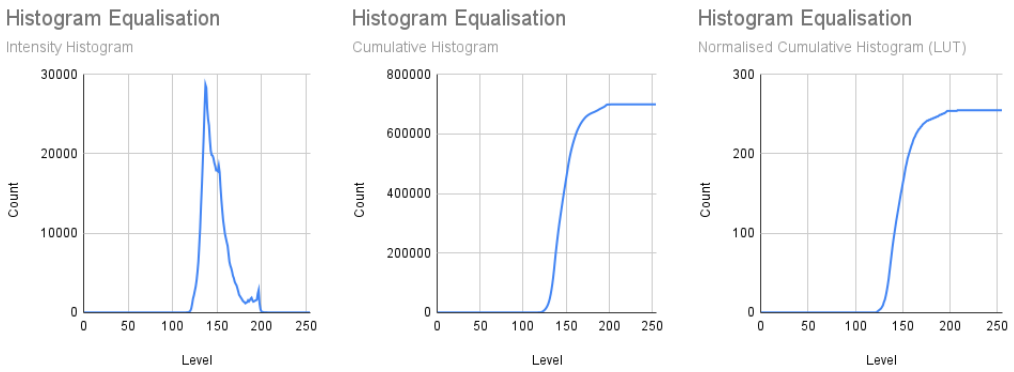


Figure 1: The individual steps of my optimal histogram equalisation procedure.

Figure 2: The input and output of my optimal histogram equalisation procedure.

## 2 Colour Functionality

In order to allow my solution to operate on coloured images, I would have to be able to properly convert them into a suitable format; the images that were being input to the solution, `test.ppm` and `test_large.ppm`, were formatted as RGB. Histogram equalisation works on the contrast of the image and none of the three channels support that; YCbCr is a format that does support contrast, in it's Y channel. To convert RGB images, I checked the spectrum count and if it was equal to 3, I would take the first channel and perform the histogram equalisation on that. At the end, I would reverse the process for the output image by adding back the chroma blue and chroma red channels and converting the YCbCr image back to RGB.

```
// Convert RGB image to YCbCr
CImg<unsigned short> ycbcrImage = tempImgInput.get_RGBtoYCbCr();

// Extract the channels setting y as the input image
imgInput = ycbcrImage.get_channel(0);
cbChannel = ycbcrImage.get_channel(1);
crChannel = ycbcrImage.get_channel(2);
```

## 3 Bit Width Functionality

To handle a 16-bit image, I had to introduce manager variables such as `is16BitUsed`, `maxIntensity`, and `consoleVariant`. In the look-up table functions, I have to be able to normalise the image according to it's bit width. If I had left the divisor as 255, the output image would have actually been output as an 8-bit image, causing a loss of information and probably a worse image. To solve this, I would set `maxIntensity` to 255 or 65535 according to the bit width of the image. Alongside this, I would set `consoleVariant` to 256 or 65536, to change the limit of the bin count and the console string text. `is16BitUsed` was used to allow for proper formatting of the output image; if the output image was read in as an `unsigned short`, it would display correctly for only the 16-bit image, while an `unsigned char` would only display correctly for the 8-bit image. The only way I could get around this was to use an if statement; this doesn't affect the performance of the kernel functions, so this wasn't an issue.

Regarding the kernel functions, the solution processes the images, 8-bit or 16-bit, as `unsigned short` rather than `unsigned char` as that is the necessary size to hold 16-bit data, and it allowed me to not have to create duplicate sets of code.

### 3.1 Drawbacks of the 16-bit Functionality

I was only able to properly implement the 16-bit functionality to work with a combination of the `intHistogram` and `cumHistogram` kernel functions. Using other kernel functions can results in corrupted image, or it can run into a lack of resources.

### 3.2 Runtime Performance

I decided to test the difference between the 8-bit imagery and 16-bit imagery, using their maximum respective bin counts and the same kernel functions, `intHistogram` and `cumHistogram`. The images used were 8-bit and 16-bit variants of the 'same' image. The results of this testing can be seen in Table 2. On average, the 16-bit image takes ~36 times longer, which is ~7 times less than the 256 times longer we might expect from a serial implementation. This shows how parallelism using GPUs or multiple cores can increase performance times over larger datasets.

| Bit Width | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
|---|---|---|---|---|---|---|
| **8-bit** | 972800 | 1043456 | 1262400 | 3221504 | 1063936 | **1512819** |
| **16-bit** | 54750016 | 55330816 | 56768512 | 54804480 | 55107584 | **55352282** |

Table 2: A table of the tested performance times for each bit width in nanoseconds.

## 4 Platforms and Devices

I built the solution on my home computer, which allowed me access to two platforms, each with one device:

**Platform 0** `NVIDIA CUDA, version: OpenCL 3.0 CUDA 12.0.89, vendor: NVIDIA Corporation`

> **Device 0** `NVIDIA GeForce RTX 3080, version: OpenCL 3.0 CUDA, vendor: NVIDIA Corporation, type: GPU, compute units: 68`

**Platform 1** `Intel(R)OpenCL, version: OpenCL 3.0 WINDOWS, vendor: Intel(R)Corporation`

> **Device 0** `AMD Ryzen 7 5800X 8-Core Processor, version: OpenCL 3.0 (Build 0), vendor: Intel(R)Corporation, type: CPU, compute units: 16`

To test the difference between the two devices, I ran the image `test.pgm` using a bin count of 256 with the aforementioned optimal kernel functions. The test results can be seen in Table 3.

| Device | Test 1 | Test 2 | Test 3 | Test 4 | Test 5 | Average |
|---|---|---|---|---|---|---|
| **GPU** | 856064 | 878592 | 939008 | 916320 | 837632 | **885523** |
| **CPU** | 8901200 | 9480700 | 10055900 | 9192200 | 9274900 | **9380980** |

Table 3: A table of the tested performance times for each device in nanoseconds.

The results of the testing show that using GPUs for parallelism is much more efficient than using CPUs; this is because of how GPUs contain more ALUs, see Figure 3, which allows for a larger volume of parallel logic operations at once.

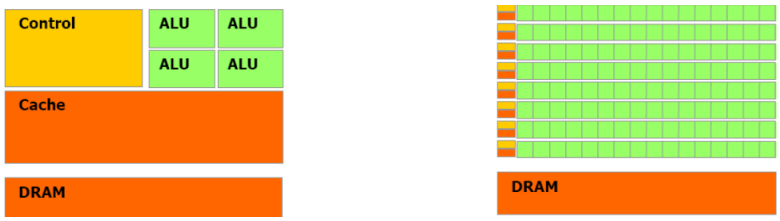


Figure 3: The architecture of a CPU and a GPU.

## 5 External Sources

- The Tutorial 2 and Tutorial 3 workshops were used to provide a basic implementation of C++ code. Some ideas for some of the kernel functions were also adapted from these workshops.
- The `intHistogram2` kernel function was an adaptation of pre-existing code.

# References


Corporation, O. (2023). 1.2 what is a data race? (sun studio 12: Thread analyzer user's guide). *Oracle.com*. Retrieved March 3, 2023, from https://docs.oracle.com/cd/E19205-01/820-0619/geojs/index.html

IEvangelist. (2021). Data parallelism (task parallel library). *Microsoft.com*. Retrieved March 3, 2023, from https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming/data-parallelism-task-parallel-library

Mills, B. (2017). Parallel scan. Retrieved March 3, 2023, from https://people.cs.pitt.edu/~bmills/docs/teaching/cs1645/lecture_scan.pdf

Radon, J. (1986). On the determination of functions from their integral values along certain manifolds. *IEEE Transactions on Medical Imaging*, *5*, 170–176. https://doi.org/10.1109/tmi.1986.4307775

The. (2013). Opencl - the open standard for parallel programming of heterogeneous systems. *The Khronos Group*. Retrieved March 2, 2023, from https://www.khronos.org/opencl/