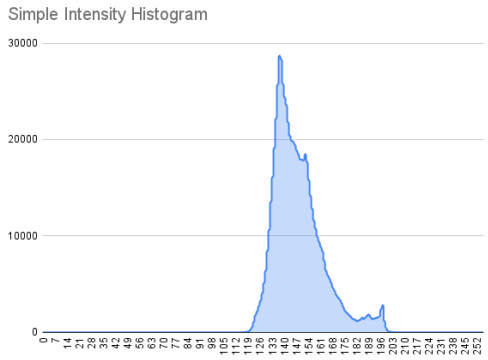
CMP3752 Parallel Programming

Jacob Morgan — 25234606
25234606@students.lincoln.ac.uk

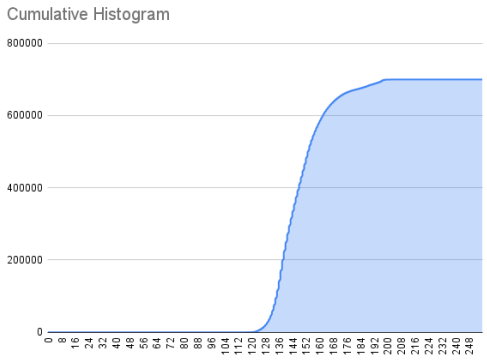
Contents

1 Kernel Implementations

The basic implementation is taken from following the tutorials provided in the workshops (Wingate 2023*c*). This includes an implementation working with 8-bit images, an intensity histogram and cumulative histogram implementation, which can be seen in Figure 1. It also includes working read and write buffers for the kernel functions and profiling for each function.



(a) Intensity histogram on 8-bit grey scale image



(b) Cumulative histogram using inclusive Hillis-Steele

Figure 1: Outputs from both histogram kernel functions

1.1 Intensity Histogram Implementation

My intensity histogram implementation is a parallel implementation of the histogram calculation which operates on global memory. This function operates using the *atomic_inc* function, which performs an increment operation on each work item based on the index of the bin into which the element falls. One issue with using atomic functions is that the atomic operation serialises access to global memory, making the process slower. To improve this function I could instead create a local barrier to ensure local memory is accessed (Ghafoor 2023).

1.2 Cumulative Histogram Implementations

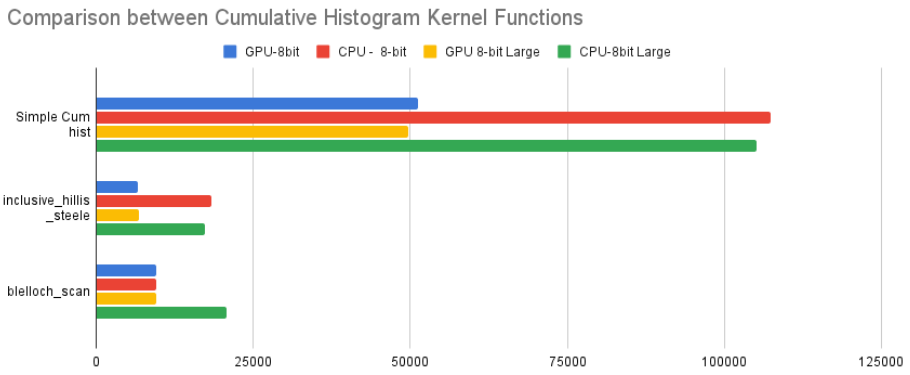


Figure 2: Cumulative Histogram Kernel Function Comparison

1.2.1 Simple Cumulative Histogram

This function is a standard cumulative sum function, which adds all elements and keeps all the results. To implement this, I used *atomic_add* which adds two values together, the two values being the current item selected in the workgroup and the item before.
This function, as seen in Figure 2, is the worst implementation as it takes the longest out of the three and its time complexity would be $O(n)$ (Wingate 2023*b*).

1.2.2 Inclusive Hillis-Steele

The inclusive Hillis-Steele function is a span-efficient kernel function that calculates the cumulative histogram in $\log_2(N)$ steps. This function forms a reduction tree for every output element and then merges the redundant elements of all the trees. As seen in Figure 2, Hillis-Steele performs best on the GPU but takes nearly twice as long on the CPU. This is because hardly any of the function is sequential (only the part in the for loop), the rest is fully parallel and runs on the GPU compute units.

1.2.3 Exclusive Blelloch Scan

This function has twice as many steps as the Hillis-Steele scan and three times more operations than the serial implementation. Blelloch scan has an up-sweep for reduction and a downward sweep to produce intermediate results. Blelloch scan has a span of $O(\log n)$ and is less efficient than Hillis-Steele (Wingate 2023*b*). This pattern can also be seen in Figure 2.

1.3 Look Up Table Implementation

This kernel normalises the cumulative distribution function using a lookup table. Input A contains the cumulative histogram and output B contains an array for mapping pixel intensities to their equalised values. The time complexity of this kernel is $O(1)$ since each work item performs a single operation to compute the value for its corresponding bin.

1.4 Back-Projection Implementation

This is a fundamental step in the reconstruction of the image from the histogram. The input A contains the image data and output B contains the back-projected image. The kernel takes in the look-up table and maps the table values to the corresponding image pixels. The time complexity for this kernel function would be $O(1)$ as each work item performs a single lookup and assignment operation.

2 Colour Functionality

In order to implement RGB functionality, I first had to convert the image into YCbCr using the *get_RGBtoYCbCr()* function. This is needed since RGB doesn't have an intensity channel, unlike YCbCr and greyscale. I check if the image has three channels and if so the image gets converted to YCbCr and the variable that the intensity histogram is run on gets set to the intensity channel of the YCbCr image. After the kernel functions are run on the intensity channel, the image is resized to the original image and converted back into RGB using *get_YCbCrtoRGB()*.

3 Variable Bins

When using a histogram, bins are used to divide the range of intensity levels into a set of ranges, according to the number of bins. I implemented this by allowing my user to decide the number of bins, this will determine how intense the image is. To figure out how big the group size will be I divided the number of bins inputted by the Intensity of the image (this depends on if the image is 8-bit or 16-bit).

```
double bin_size = (MAX_INTENSITY + 1) / bin_count;
```

To work out what bin to put the pixel value into

```
int bin_index = A[id] / binsize;
```

4 Bit Depth Functionality

At the start of the program, I use the *max()* function to check the intensity of the image. If the intensity is, 256 the image is 8-bit; if the intensity is, 65535 the image is 16-bit. This

corresponds to 2^n, n being the bit depth of the image. I then store the value of the intensity in an integer. I use this value to calculate the *bin_size* so the intensity functionality can scale with the bit depth of the image. I also set the type of my image to an `unsigned short`, since `unsigned char` can only deal with 8-bit values.

5 Platforms and Devices

I ran my kernel functions on both my GPU and CPU, accumulating the times of each function into a **total run time**. The GPU is an Nvidia GeForce GTX 1070 and the CPU is an AMD Ryzen 5 3600. The GPU overall runs better than the CPU since the GPU is optimised for data parallel and high throughput computations. The CPU also has a smaller amount of ALUs meaning a smaller computation power (Wingate 2023*a*).

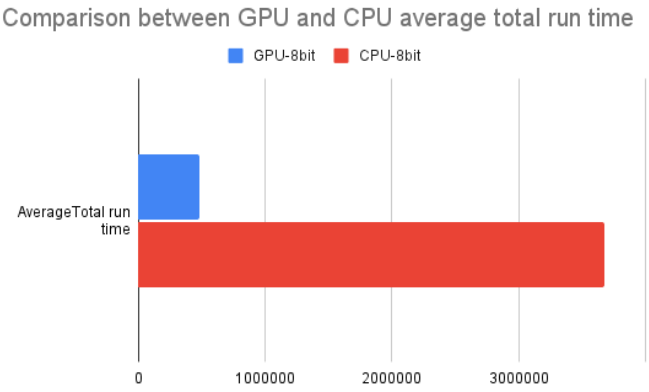


Figure 3: GPU vs CPU on 8-bit images

References

Ghafoor, D. M. (2023), 'Cmp3752m parallel programming lecture 7: Histogram - parallelisation'.

Wingate, J. (2023*a*), 'Cmp3752m parallel programming lecture 1: Introduction'.

Wingate, J. (2023*b*), 'Cmp3752m parallel programming lecture 5: Parallel patterns 3'.

Wingate, J. (2023*c*), 'OpenCL Tutorials'. original-date: 2021-09-14T00:35:20Z.
URL: *https://github.com/wing8/OpenCL-Tutorials*