CMP3752 Parallel Programming Assignment 1

By Bartosz Krawczyk – 25798652 – 25798652@students.lincoln.ac.uk

An overview of the program

The input and output of images is handled by code from the workshops. To make it clear where my code begins and ends there are long comments on the top and bottom of it. Here is an example of a functional block (See Figure 1), as requested in the assessment brief.

Each block has 3 vectors that profile the command, and the command itself labelled and

```
cl::Event e0;
// calculate image size in bytes for profiling
int image_bytes = image_input.size() * image_input.spectrum() * sizeof(unsigned char);

// input image data
queue.enqueueWriteBuffer(dev_image_input, CL_TRUE, 0, image_input.size(), &image_input.data()[0], NULL, &e0);

event_names.push_back("input image_input");
event_memory_transfer.push_back(image_bytes);
```

FIGURE 1: AN EXAMPLE OF THE LAYOUT.

in the centre. Globally, the top of the code defines some necessary variables, and then going down it follows the instructions detailed in the assessment brief for equalising the image.

```
Running on NVIDIA CUDA, NVIDIA GeForce RTX 2060

input image_input
699392
execution time [ms (ns)]: 0.055872ms (55872ns)

execute event_histogram
0
execution time [ms (ns)]: 0.300704ms (300704ns)

execute cumulative_histogram
0
execution time [ms (ns)]: 0.006144ms (6144ns)

execute normalise_histogram
0
execution time [ms (ns)]: 0.006368ms (6368ns)

execute map_image
0
execution time [ms (ns)]: 0.016384ms (16384ns)

output image_output
699392
execution time [ms (ns)]: 0.054688ms (54688ns)

total events: 6
total memory transfer between host and device: 1398784
total execution time [ms (ns)]: 0.44016ms (440160ns)
```

FIGURE 2: OUTPUT OF PROFILING DATA

The images that are to be used must be in the same directory as the executable. This program accepts only 8-bit images, although there is no error check in case someone tries to input a 16-bit image. The 16-bit image won't crash the program but there is insufficient allocated memory so the result will look like static. However, coloured images work and so do large images. OpenCL kernels are stored in "kernels.cl"

After the program runs all the steps it starts the section called "profiling data". This section iterates through all the vectors that have been storing data about the commands and displays them in a readable format. See Figure 2.

Finally, the equalised image is outputted.

ADDITIONAL FEATURES

SUPPORT FOR COLOUR IMAGES

In the brief, "support for colour images" is listed as an additional feature. This program fulfils this requirement. This works because this program works based on intensity, so the presence of colour values doesn't change much. See Figure 3.

FIGURE 3: BOTTOM IMAGE (OUTPUT) HAS BEEN EQUALISED.

VARIABLE HISTOGRAM BIN NUMBER/SIZE

At the top of my code there is a variable *"int bin_size = 1;"*. It is 1 by default, at this setting the histogram runs normally. I added a bypass because separating the pixel values into bins impacts execution time. Increasing this variable will increase the bin size and decrease the number of bins. E.G. *bin_size = 2* leads to 128 bins of size 2.

Each kernel includes *bin_size* as an argument. The bin size is used to scale the histogram that the algorithms they rely on. This also has the impact that when taking or plotting a value from an image, a calculation first has to run to sort it into a bin. This calculation works by first subtracting one from the value until it reaches a multiple of *bin_size* or 0. Then if it reaches a multiple of *bin_size* it will divide the obtained value by the *bin_size*. Now the value has been sorted into a bin and has an appropriate index for the histogram, which has also been scaled down. Increasing the bin size is very costly computationally as each pixel has to be sorted into the right bin twice as the program runs. Thankfully, the load is lessened by the fact that this is done as a parallel map pattern. It also decreases the quality of the image, so it may be useful for compression. See figure 4 for example.

FIGURE 4: THE CATHEDRAL IMAGE WITH *BIN_SIZE* SET TO 20.

PARALLEL SCAN PATTERN USED TO CUMULATE HISTOGRAM

This program uses a Hillis-Steele inclusive scan pattern, also known as a prefix sum (Harris, 2007). This pattern sums up the values of the histogram in parallel, while skipping values that have already been summed by using an offset. This pattern is more effective than summing up a histogram with series processing however, while it is theoretically efficient, it should make a negligible difference on an array this small. I researched this outside of workshops.

KEY ALGORITHMS

Almost all the algorithms are in kernels; thus, this section will talk about kernels. All kernels also take *bin_size* as an argument but I don't mention it. It just scales histogram related values.

HISTOGRAM

Take the intensity of the current pixel. The intensity is sorted into a bin (if they are being used) and atomically increments the appropriate bin value of the histogram.

CUMULATIVE_HISTOGRAM

This kernel takes a histogram as an input and runs a prefix sum on it. This adds up the values going from left to right, while adding the partial sum too. This is more efficient than a serial equivalent. This kernel is a scan pattern.

NORMALISE_HISTOGRAM

This kernel takes two int* buffers as arguments. The first one is the input. The second is blank. Then, the kernel takes the last value of the input histogram, which due to the previously used prefix sum will be the largest value in the histogram. Next, 255 is divided by the largest value to

get the ratio which will be used to scale the histogram into the normalised histogram. The normalised histogram now has 255 as the highest value, with every other value scaled appropriately such that it can be used as a look-up-table. This kernel is a map pattern.

MAP_IMAGE

This kernel takes the original input image, a normalised histogram, and a blank output buffer as inputs. The value of each pixel is taken, then the value is used to index the normalised histogram like a look-up-table, and the returned value is placed on the blank output buffer in the same position it was at in the original image. This equalises the image, by back projection. This kernel is map pattern.

OPTIMISATION AND DATA

OPTIMISATION

I do not have intermediate execution profiling results from development, however I specifically selected parallel patterns (map and scan) for the kernels so that the processing is efficient. This wasn't a hard choice as image processing is a "painfully parallel" problem.

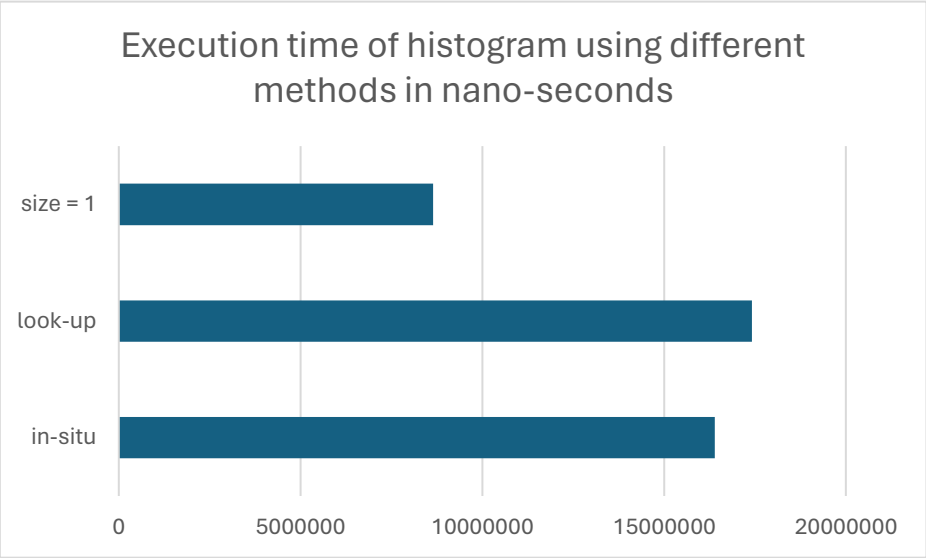

FIGURE 5: A BAR CHART COMPARING EXECUTION TIMES OF HISTOGRAM USING DIFFERENT METHODS.

The variable histogram bins add a lot of processing time. So, I added a bypass where if *bin_size = 1* the bin calculations are ignored. The problem with bins is that if *bin_size = 50* for example, for a lot of values a serial loop will have to decrement one by one until it reaches a multiple of 50. I attempted to optimise this by using a look-up-table to decide which bins values should go in, but it turned out that there was no improvement. I suspect it's because looking up the value takes the same amount of time as calculating it in-situ. Figure 5 shows this point. This time come from an average of 10 tests using each method. This doesn't show but the in-situ calculations have a greater variance, about +/-7000 from the central value. The look-up method is far more predictable with a variance of about +/-3500 but regardless their average execution time is similar. Note how much faster bypassing the bin calculation is (where size = 1).

MEMORY EFFICIENCY

The intermediate version of my program had lots of unnecessary memory transfers between the host and devices. It was inefficient and made the program confusing to read. Now, the program has 1 image input at the beginning, then 4 commands that chain into each other and one image output at the end. The small number of transfers minimises time wasted on moving data between host and device.

AVERAGE METRICS

Because images are transferred twice the total memory transfer between the host and device will always be:
*2 * input_image.sizeInBytes()*
These are the average metrics of running “test_large.pgm” for each kernel, along with the table.

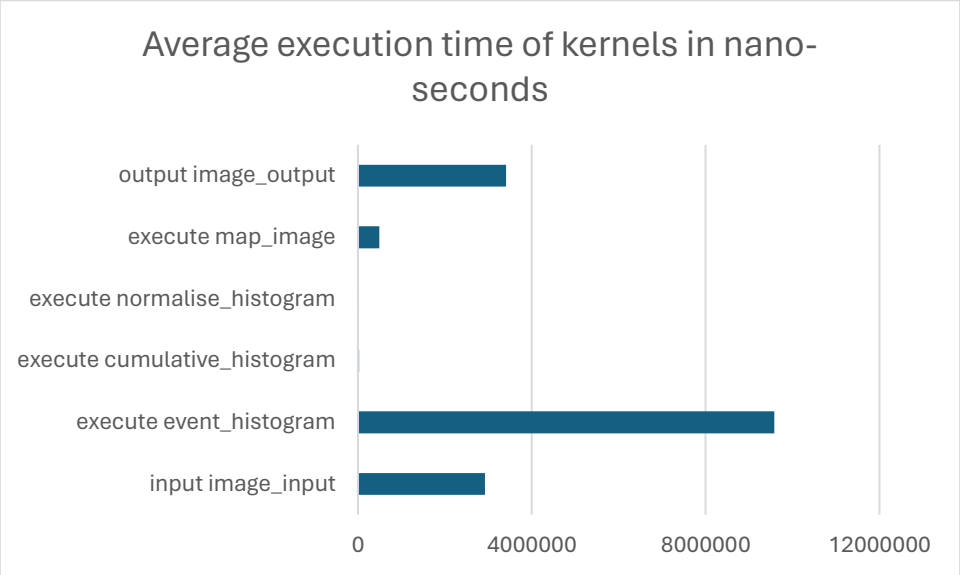

FIGURE 6: AVERAGE EXECUTION TIME OF COMPONENTS OF THE EQUALISATION PROGRAM.

trial	input image_input	execute event_histogram	execute cumulative_histogram	execute normalise_histogram	execute map_image	output image_output
1	2652128	7843520	7552	5888	389472	3961504
2	2712224	8684192	6496	5568	393472	2573312
3	2874688	9427200	7936	5056	592224	3303520
4	3188832	8165376	7968	5024	387520	3892032
5	3490016	11578048	6656	4192	386688	3960832
6	2714368	12248864	7328	6048	393664	2969440
7	3493504	10186272	7008	5824	387104	2618560
8	2577664	7379936	6176	5280	369792	2814368
9	2720256	8011776	241088	6144	630816	3248640
10	2804512	12308576	7520	4736	1010016	4751744
average	2922819	9583376	30572.8	5376	494076.8	3409395.2

It is plainly seen that the kernel with the longest execution time is *histogram*. I did not implement a very efficient parallel algorithm, so it is not surprising. Implementing a more efficient histogram generating parallel algorithm would substantially reduce the total run time of the whole program. Especially since its difficult to reduce the time spent loading and unloading the image from the device.

REFERENCES

Harris, M., 2007. *Parallel Prefix Sum*. [Online]
Available at: https://www.eecs.umich.edu/courses/eecs570/hw/parprefix.pdf
[Accessed 12 04 2024].