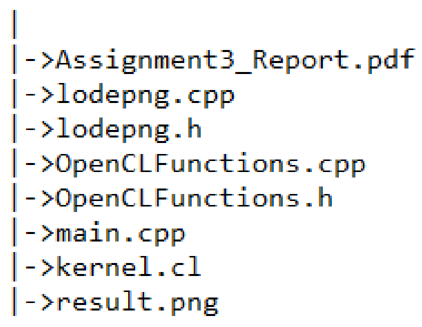
Tommi Järvenpää
2553157

Time used: ~16.5h

Assignment 3 Report

In this assignment, I developed an OpenCL implementation for image grayscaling and 5x5 moving mask bluring. The host code was developed with C++. The report's folder structure is shown in Figure 1. More detailed information on the functions discussed in this report can be seen in the extensive commenting on the code files. The codes were created on Windows 10 with Visual Studio 2019. A NVIDIA GeForce GTX 1070 GPU was used with OpenCL. Sources for different code snippets are displayed in the code files and at the end of this report. The following sections present how image loading and saving was done, and how grayscaling and using the mask were implemented.



```
|
|->Assignment3_Report.pdf
|->lodepng.cpp
|->lodepng.h
|->OpenCLFunctions.cpp
|->OpenCLFunctions.h
|->main.cpp
|->kernel.cl
|->result.png
```

Figure 1: Folder structure

Image Loading & Saving

As instructed in the assignment, the images were loaded into memory, and saved with the LodePNG[1] library. Since I was using C++, I used the the following to commands to load the image:

```
std::vector<unsigned char> img;
lodepng::decode(img, w, h, (const char*)file_name, LCT_RGB);
```

This reads the given image into memory and decodes it into a 1D array containing the red, green, and blue (RGB) values of each pixel. The image obtained after executing the two Kernel functions was endoded into PNG format and saved to disk with the following command: `lodepng::encode("result.png", result, w, h, LCT_GREY);`

As seen in the image loading code snippet, *im0.png* is read into the *img* vector. The parameters *w* and *h* represent the image's width and height. *LCT_RGB* is a value defined in *lodepng.h,* used to read the image in RGB format. When saving the result, *LCT_GREY* is used, since the image has been grayscaled. The code examples from [2] and [3] were used for creating the code for loading and saving.

Initializing OpenCL

The OpenCL parameters were initialized in the typical way. A platform and device were first selected. And a context and command queue were created for the device. *CL_QUEUE_PROFILING_ENABLE* was given to the queue as a parameter, so OpenCL kernel profiling could be used. The *OpenCLFunctions.c/h,* implemented and explained in the previous assignment, were used to assist with error checking and reading the Kernel sources into memory. During this assignment, the function *createKernel()* was added to *OpenCLFunctions* to reduce the number of lines in *main.cpp.* The function creates the specified kernel function from source. At the end of *main* the OpenCL objects are freed from memory.

Image Grayscaling

The image grayscaling was done with the Kernel function *to_grayscale*. It contains the following:

```
int id = get_global_id(0);
//          Red                 Green               Blue
result[id] = img[id*3] * 0.299 + img[id*3 + 1] * 0.587 + img[id*3 + 2] * 0.114;
```

The code creates a *w*h* image with only one color by combining the original images RGB values into one. Each color value is multiplied with a certain weight, which I found in [4]. The parameters given to the kernel are the original image and the *w*h* char array where the result will be stored. The image array is treated as a 1D array.

5x5 Moving Mask

Since what the 5x5 moving mask should do was not specified in the assignment, I decided to make it a bluring filter. The blurring is done by summing the values found in the 5x5 area, and dividing the sum with 25. When moving the mask around the image, the image's edges become a problem which must be dealt with somehow. Initially, I used zero padding, but I was not happy whit the resulting image, since it was very dark (0 = black color). Thus, the edge handling was change to mirror padding. This means, that the values outside of the image are obtained by mirroring the coordinates. Example shown in Figure 2. When executing the mask Kernel (*apply_mask*), *clEnqueueNDRangeKernel()* is called with the number of dimentions set to 2, so the current x and y coordinate can be easily checked. Due to this, the *global_work_size* is set to {width, height}. In the Kernel code, a given pixel's x and y coordinate are obtained with:

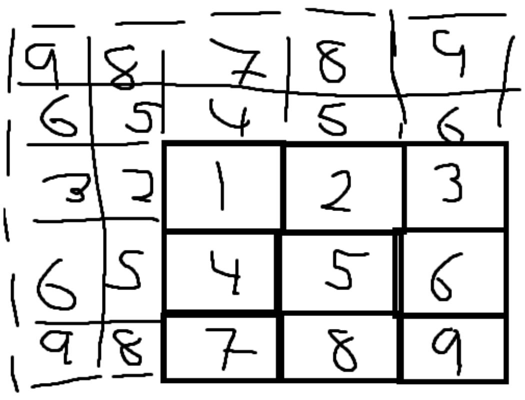


Figure 2: Mirror padding

```
const int x = get_global_id(0);
const int y = get_global_id(1);
```

Then, the moving mask is implemented by applying two for loops on the current pixel. The range for both loops is [-2, 2], i.e. five. Inside the loops, new x and y coordinates are calculated using the loop variables:

```
current_x = x+j; current_y = y+i;
```

A set of if statements is then used to determine if the new coordinates are inside or outside of the image, and if mirroring should be done:

```
if (current_y >= 0 && current_y < height && current_x >= 0 && current_x < width) {
      // Current position is inside the image
      sum += img[current_y * width + current_x];
} else if ((current_y < 0 || current_y >= height) && (current_x < 0 || current_x >= width)) {
      // x and y outside of the image, mirror both
      sum += img[(y - i) * width + (x - j)];
} else if ((current_y >= 0 && current_y < height) && (current_x < 0 || current_x >= width)) {
      // y is correct, mirror x
      sum += img[current_y * width + (x - j)];
} else if ((current_x >= 0 && current_x < width) && (current_y < 0 || current_y >= height)) {
      // x is correct, mirror y
      sum += img[(y - i) * width + current_x];
} else {
      // Just in case, should never be called
      sum += img[(y - i) * width + (x - j)];
}
```

As the name suggests, the *sum* variable stores the sum of the 25 elements inside the mask. The new pixel value is finally calculated with:

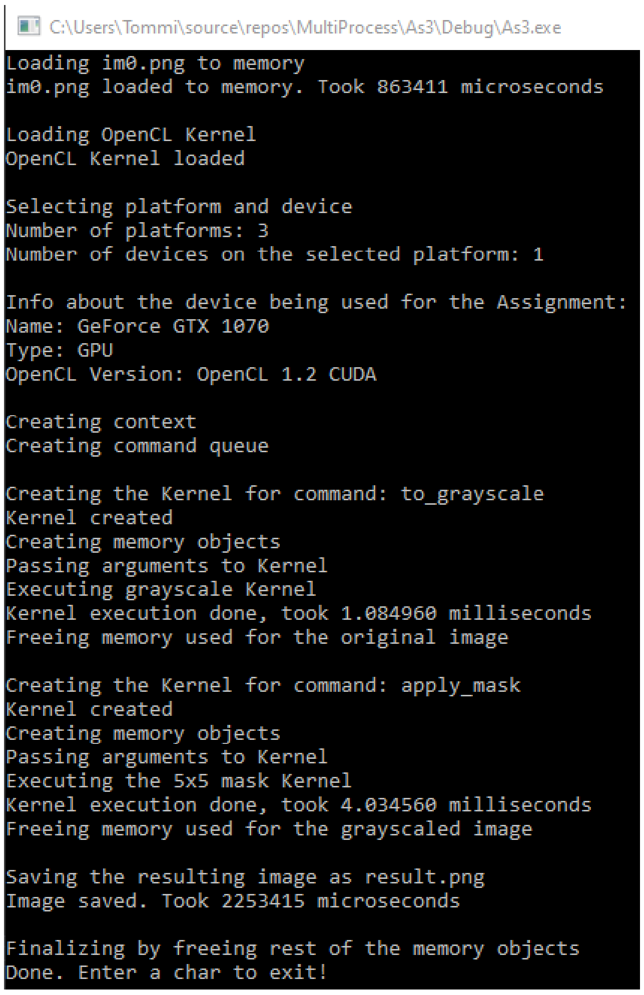
```
result[y * width + x] = (sum / 25);
```

Profiling & Output

The output of the program developed in this assignment is shown in Figure 3. From the output, we can see that loading *im0.png* into memory takes 863.4 milliseconds according to WINAPI [5], which was used to obtain its execution time. The Kernel source is then loaded, platform and device info is displayed, and a context and command queue are created. The output shows that executing the grayscale Kernel function takes 1.085 milliseconds. This time was obtained using profiling [6]. Using the moving mask takes 4.03 milliseconds. Saving the resulting image takes 2253.4 milliseconds. Finally the output shows that a finilization is done by freeing the memory used by OpenCL objects and the images.

Investigating Memory Usage

I also briefly investigated the program’s memory usage with the assistance of Visual Studio’s memory usage snapshot tool and the process memory usage graph. According to the process memory graph, the program uses
1. 57.9MB when original image is loaded
2. 143.3MB when OpenCL objects are initialized
3. 177.0MB when grayscaled image is created
4. 160.0MB when the original image is realeased from memory and mask applied
5. 154.3MB when grayscaled released
6. 163.1MB when resulting image is saved
7. 149.0MB when finalized
However, I don’t actually know how accurate these values are.
The Process memory graph is shown in Figure 4, and the memory usage snapshots in Figure 5.



```
Loading im0.png to memory
im0.png loaded to memory. Took 863411 microseconds

Loading OpenCL Kernel
OpenCL Kernel loaded

Selecting platform and device
Number of platforms: 3
Number of devices on the selected platform: 1

Info about the device being used for the Assignment:
Name: GeForce GTX 1070
Type: GPU
OpenCL Version: OpenCL 1.2 CUDA

Creating context
Creating command queue

Creating the Kernel for command: to_grayscale
Kernel created
Creating memory objects
Passing arguments to Kernel
Executing grayscale Kernel
Kernel execution done, took 1.084960 milliseconds
Freeing memory used for the original image

Creating the Kernel for command: apply_mask
Kernel created
Creating memory objects
Passing arguments to Kernel
Executing the 5x5 mask Kernel
Kernel execution done, took 4.034560 milliseconds
Freeing memory used for the grayscaled image

Saving the resulting image as result.png
Image saved. Took 2253415 microseconds

Finalizing by freeing rest of the memory objects
Done. Enter a char to exit!
```

Figure 3: Program output

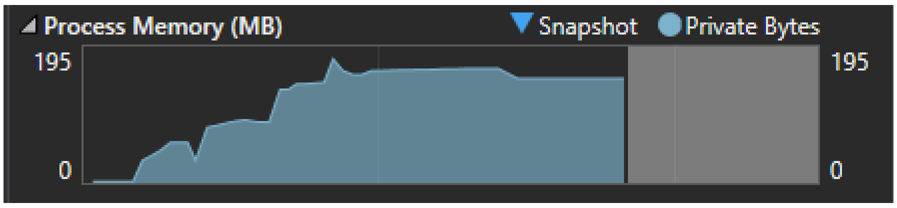
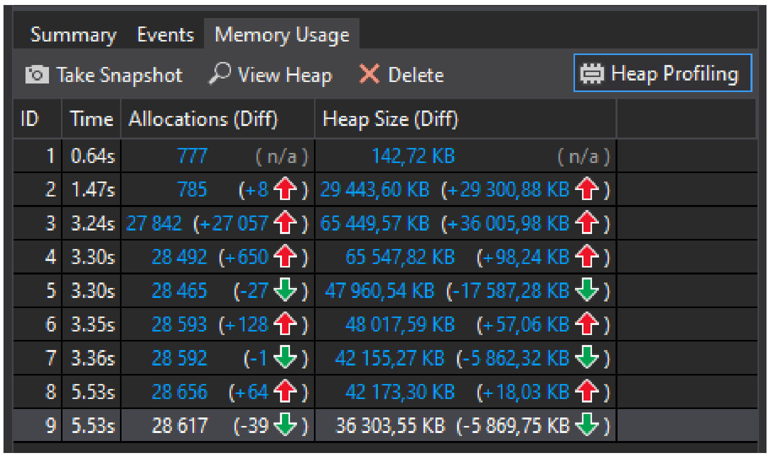


Figure 4: Process memory usage graph



ID	Time	Allocations (Diff)	Heap Size (Diff)
1	0.64s	777 (n/a)	142,72 KB (n/a)
2	1.47s	785 (+8 ↑)	29 443,60 KB (+29 300,88 KB ↑)
3	3.24s	27 842 (+27 057 ↑)	65 449,57 KB (+36 005,98 KB ↑)
4	3.30s	28 492 (+650 ↑)	65 547,82 KB (+98,24 KB ↑)
5	3.30s	28 465 (-27 ↓)	47 960,54 KB (-17 587,28 KB ↓)
6	3.35s	28 593 (+128 ↑)	48 017,59 KB (+57,06 KB ↑)
7	3.36s	28 592 (-1 ↓)	42 155,27 KB (-5 862,32 KB ↓)
8	5.53s	28 656 (+64 ↑)	42 173,30 KB (+18,03 KB ↑)
9	5.53s	28 617 (-39 ↓)	36 303,55 KB (-5 869,75 KB ↓)

Figure 5: Memory usage snapshots

References

[1] “LodePNG”. URL: https://lodev.org/lodepng/. Accessed: 29.01.2020
[2] “LodePNG Decode Example”. URL: https://raw.githubusercontent.com/lvandeve/lodepng/master/examples/example_decode.cpp. Accessed: 29.01.2020
[3] “LodePNG Encode Example”. URL: https://raw.githubusercontent.com/lvandeve/lodepng/master/examples/example_encode.cpp. Accessed: 29.01.2020
[4] “Standard RGB to Grayscale Conversion”. URL: https://stackoverflow.com/questions/17615963/standard-rgb-to-grayscale-conversion. Accessed: 29.01.2020
[5] “Acquiring high-resolution time stamps”. URL: https://docs.microsoft.com/en-us/windows/win32/sysinfo/acquiring-high-resolution-time-stamps. Accessed: 29.01.2020
[6] “Profiling Operations Using OpenCL™ Profiling Events”. URL: https://software.intel.com/content/www/us/en/develop/documentation/iocl-tec-opg/top/performance-debugging-with-intel-sdk-for-opencl-applications/profiling-operations-using-opencl-profiling-events.html. Accessed: 29.01.2020

Code References Not Mentioned in Elsewhere The Report

HelloWorld Report.pdf from Assignment 2
OpenCL Programming Guide from the course page
“List OpenCL platforms and devices” - URL: https://gist.github.com/courtneyfaulkner/7919509