Multiprocessor Programming

Final report

Ville Kemppainen 2255141
Mikko Paasimaa 2255565

Summary

Project consisted of implementing Zero mean Normalized Cross Correlation-algorithm. First in C or C++, then in OpenCL and finally porting the OpenCL algorithm to mobile platform. Idea was to compare execution times of different implementations and to optimize them as well as possible.

Implementations

C-implementation

Structure of the C-implementation is based on the pseudocode given with the exercise instructions. Images are read using the LodePNG library. It reads the images to arrays. After that the images are scaled down from their original size (2940x2016) to one fourth by taking every 4^{th} pixel from the original image. After that the four channels for RGBA were changed to single grayscale channel by multiplying them with a suitable values.

ZNCC is done in two different functions, one for the image on left and one for the image on right. Difference between these implementations is the direction in which way you compare the window values to. It is basically multiple for loops going through every pixel of the image.

Post-processing is done in two parts. First is cross checking the images which is comparing their absolute value to determined threshold. If the value is greater than the threshold, that value is set to 0. On occlusion filling part every pixel that is zero is set to nearest non-zero pixel in x- and y-directions.

OpenCL-implementation

OpenCL implementation consists of two parts. Host code and the kernel code. Host code uses parts of the code found from the Anterus blog[1] which is listed at the additional material page on the course website. Host code uses the basic OpenCL program flow which is listed at the first seminars slides.

Our implementation has 5 kernels which are based on our 5 C-functions. Greyscaling and resizing of the image is done in one kernel. ZNCC is done in two kernels and cross-checking and occlusion filling are done in two different kernels.

We used 2Dimage objects as our memory objects. This allowed us to use the x- and y- coordinates to access image pixels. This made the kernel simpler compared to the C-implementation in which we had the images in arrays.

Mobile implementation

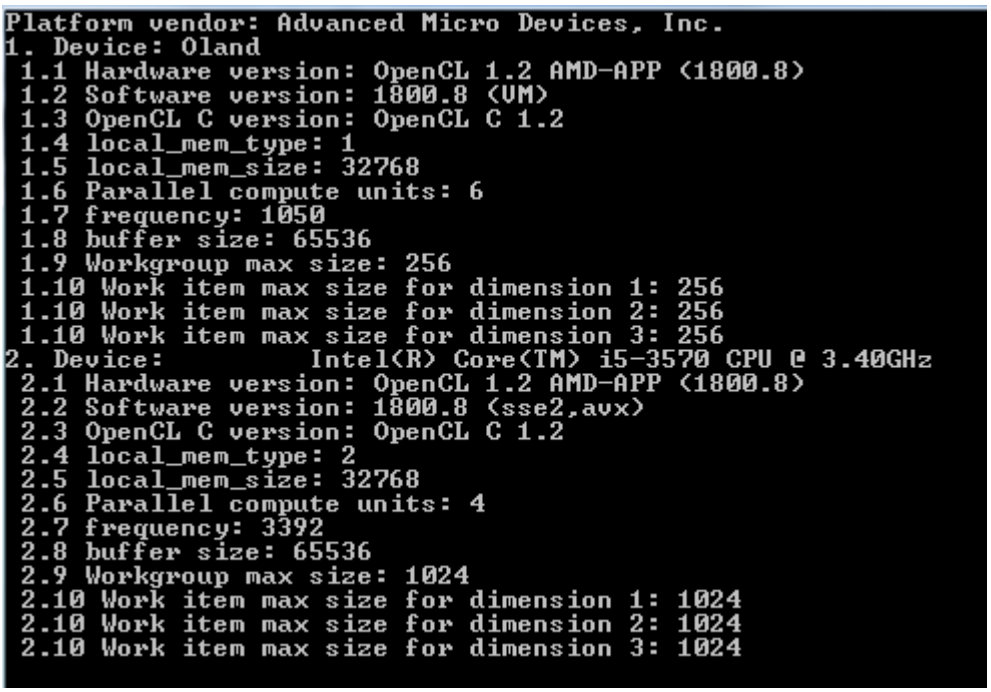
Our mobile implementation is mostly the same as our OpenCL-implementation. Differences are in the windows and Linux syntax. This basically meant changing the timing function and changing std::exit to exit. Later on in the optimization phase the mobile implementation changed more severely because we chanced all the read and write image commands to use halfs instead of unsigned integers.

Results

Device information

For the PC part we used the computers available in TS135. Mobile implementation was done on the Odroid. We made a script which displays the information about used devices.

TS135



Odroid:

Platform vendor: ARM

1. Device: Mali-T628

1.1 Hardware version: OpenCL 1.2 v1.r14p0-01rel0.c2c829708a6ea349bc7a9dc1068c92e8

1.2 Software version: 1.2

1.3 OpenCL C version: OpenCL C 1.2 v1.r14p0-01rel0.c2c829708a6ea349bc7a9dc1068c92e8

1.4 local_mem_type: 2

1.5 local_mem_size: 32768

1.6 Parallel compute units: 4

1.7 frequency: 600

1.8 buffer size: 65536

1.9 Workgroup max size: 256

1.10 Work item max size for dimension 1: 256

1.10 Work item max size for dimension 2: 256

1.10 Work item max size for dimension 3: 256

2. Device: Mali-T628

2.1 Hardware version: OpenCL 1.2 v1.r14p0-01rel0.c2c829708a6ea349bc7a9dc1068c92e8

2.2 Software version: 1.2

2.3 OpenCL C version: OpenCL C 1.2 v1.r14p0-01rel0.c2c829708a6ea349bc7a9dc1068c92e8

2.4 local_mem_type: 2

2.5 local_mem_size: 32768

2.6 Parallel compute units: 2

2.7 frequency: 600

2.8 buffer size: 65536

2.9 Workgroup max size: 256

2.10 Work item max size for dimension 1: 256

2.10 Work item max size for dimension 2: 256

2.10 Work item max size for dimension 3: 256

Unoptimized execution times

OpenCL on PC using only GPU:

Greyscale and resize kernel: 0.835 ms

Zncc_left kernel: 297.839 ms

zncc_right kernel: 297.005 ms

Postprocess kernel: 0.076 ms

occlusion kernel: 0.210 ms

total time 4.587 s

OpenCL on PC using only CPU:

Greyscale and resize kernel: 2.146 ms

Zncc_left kernel: 22920.327 ms

Zncc_right kernel: 22136.045 ms

Postprocess kernel: 3.138 ms

occlusion kernel: 3.700 ms

total time 48.767 s

OpenCL on Odroid:

Greyscale and resize kernel: 1.629 ms

Zncc_left kernel: 6976.974 ms

Zncc_right kernel: 6490.814 ms

Postprocess kernel: 1.102 ms

Occlusion kernel: 1.507 ms

Total execution time: 19.3649s

Optimization

We used several minor optimization methods. First one was changing the variable datatypes to be smaller. For example integers to chars or shorts when possible. We also planned to change floats to halfs, but the AMD does not support that so it wasn't implemented.

Second one was to use the relaxed-fast-math build flag when building our OpenCL program. This allows optimization for floating-point arithmetic that may violate the IEEE 754 standard.

Third one was that we allowed the ZNCC kernels to run simultaneously because they don't modify the same information.

These did not provide us with any noticeable difference in either devices execution times. What however gave us a small difference in Odroids execution time was to read and write the image objects using the read/write_imageh functions instead of read/write_imageui functions, which uses half values instead of integers.

We also changed structure of the given pseudocode so that the mean and standard deviation values for image 1 are not pointlessly calculated multiple times. This increased the ZNCC kernels execution times significantly.

Optimized OpenCL code on Odroid:

Greyscale and resize kernel: 1.619 ms

Zncc_left kernel: 6693.680 ms

Zncc_right kernel: 6049.471 ms

Postprocess kernel: 1.188 ms

Occlusion kernel: 1.580 ms

Total execution time:18.6392s

Future work

Our ZNCC kernels use many float variables. Changing these to half should show a noticeable difference in execution times. On our initial research it seems that AMD does not support half so implementing them on the PC implementation might not be possible. ARM platform however should have a native support for halfs so changing those should be possible in the future.

Vectorization is something that should also show a noticeable change in execution times.

Currently our implementation does not use local memory. Fully utilizing local memory would radically decrease the execution times of the different kernels. Also fiddling with the work group sizes could result in better execution times.

Reference

1. https://anteru.net/blog/2012/11/03/2009/

Appendix

zncc.cpp

zncc.c

opencl.c

kernel.cl

odroid_imageh.cl