Assignment 7

Multiprocessor Programming 521288S

Miika Sikala 2520562, msikala18@student.oulu.fi

Task: Optimization of stereo disparity OpenCL implementation and profiling.

Expected result: A working version of the implementation, a brief report with analysis and observations (max2-3 pages), saved output images all together in the form of a compressed folder (.zip file).

The report should contain about the task solved, brief description of your implementation, comparison of profiling information and the final screenshot of your outputs asked to be displayed under the assignment.

Introduction

The whole project was done with optimization in mind so most solutions in this project were already somewhat optimal. In this assignment the main optimization was to change the codebase and kernels to support single channel images. This does not only make the calculation faster, but also consumes significantly less memory and bandwidth during transfers between CPU and compute device.

AMD OpenCL optimization guide [1] was used in looking for optimization points in the kernels. Many of the optimizations mentioned in the guide were not practical in this case, but some were:

- **`#pragma unroll` directive** in unrolling some loops to several parallel instructions
- **Using predication over control-flow wherever possible**: this allows the GPU to execute both possible paths in parallel and pick the correct one after decision has been made
- **Using largest possible work group sizes** to maximize efficiency: in my GPU, the maximum work group size is 256 so local work size of 16x16 was used (for my CPU, which has maximum work group size of 1024, I used 32x32 work groups)
- **Using built-in vector types such as float4 and int2** to allow packed SSE instructions
- **Using special instructions where possible:** for example, using `mad` (multiply-add) in grayscale conversion to make fast but not-so-accurate calculation
- **Using compiler optimizations**

Usage

To change the target device, you can change the `COMPUTE_DEVICE` flag in “Application.hpp”. The options are `TARGET_NONE` (no parallelization), `TARGET_PTHREAD` (CPU parallelization using pthread) `TARGET_GPU` (OpenCL on GPU), `TARGET_CPU` (OpenCL on CPU), and `TARGET_OMP` (OpenMP). You must re-compile the program for the change to take effect.

The application takes two to six input parameters, which are the image file names for the left and right images, and optionally some calculation parameters. See the example below, which uses image files `img/im0.png` and `img/im1.png`.

A makefile is provided for compiling with g++.

Compiling using *g++*:
```
make all
```

Running:
```
stereo.exe img/im0.png img/im1.png 15 55 8 4

# Arguments: LEFT_IMG RIGHT_IMG [WINDOW_SIZE=9] [MAX_SEARCH_DIST=32]
#            [CROSS_CHECK_THRESHOLD=8] [DOWNSCALE_FACTOR=4]
```

Testing and benchmarking the implementations

The results (intermediate and final) are stored in “img” folder. Figure 2 shows the execution of the program for three different compute targets.

The benchmarking was done using a simple occlusion fill algorithm, which seeks the nearest non-zero pixel in square-like spiral.

Used configuration:

window size:	15
maximum search distance:	55
cross-checking threshold:	8
downscaling factor:	4

Execution times:

	CPU (sequential)		**GPU (OpenCL)**	
Operation	**No optimization**	**Optimized**	**No optimization**	**Optimized**
Load + decode	2.408 s	1.011 s	2.444 s	827.147 ms
Resize	13.697 s	3.961 s	816.663 ms	497.539 ms
Convert to grayscale	61.516 ms	13.791 ms	349.622 ms	362.641 ms
Encode + save	302.215 ms	116.914 ms	301.567 ms	111.906 ms
Calculate ZNCC	181.788 s	23.179 s	510.354 ms	434.518 ms
Encode + save	288.349 ms	76.846 ms	287.210 ms	68.681 ms
Cross-check	20.732 ms	3.065 ms	179.125 ms	179.340 ms
Encode + save	108.758 ms	37.665 ms	138.841 ms	33.168 ms
Occlusion fill	37.940 ms	5.805 ms	172.679 ms	178.833 ms
Encode + save	99.736 ms	34.689 ms	108.529 ms	36.444 ms
Total	**198.812 s**	**28.440 s**	**5.309 s**	**2.730 s**

As the table shows, the improvement is quite significant. In sequential implementation there is a huge improvement on all areas, which is not a surprise since the compiler optimizations and usage of single channel images over 4-channel images improves performance, while in OpenCL implementations the kernels are executed very fast even in the non-optimized implementation. As can be seen, the load + decode and encode + write operations are so time-consuming that the improvement in the kernel calculations is not very apparent.

In sequential implementation the compiler can make in the simplest cases some advanced optimizations to allow partial parallelization and vectorized execution on the CPU (or using SIMD pipeline), which increases performance of simple loops.

In OpenCL implementation most of the performance gain in the kernelled functions comes from 4 times faster transfer times (since the images are only 1 channel). However, the kernels are also executed noticeably faster (**70.13 ms** total kernel time compared to **39.53 ms** total kernel time in the optimized version).

Figure 1 shows the dynamic memory usage during program execution (using OpenCL). The horizontal axis shows the time and vertical axis the momentary memory consumption. The consumption peaks after both images have been loaded but decreases significantly after the images are downscaled and converted to single channel. The red and yellow blocks represent the left and right images. The "base" block represents all the miscellaneous memory allocations.

CPU usage was quite stable **12,5 %** on all executions except on *OpenMP* and *pthreads* implementations. Since I have an eight-core CPU, non-parallelized execution only utilizes one core fully.

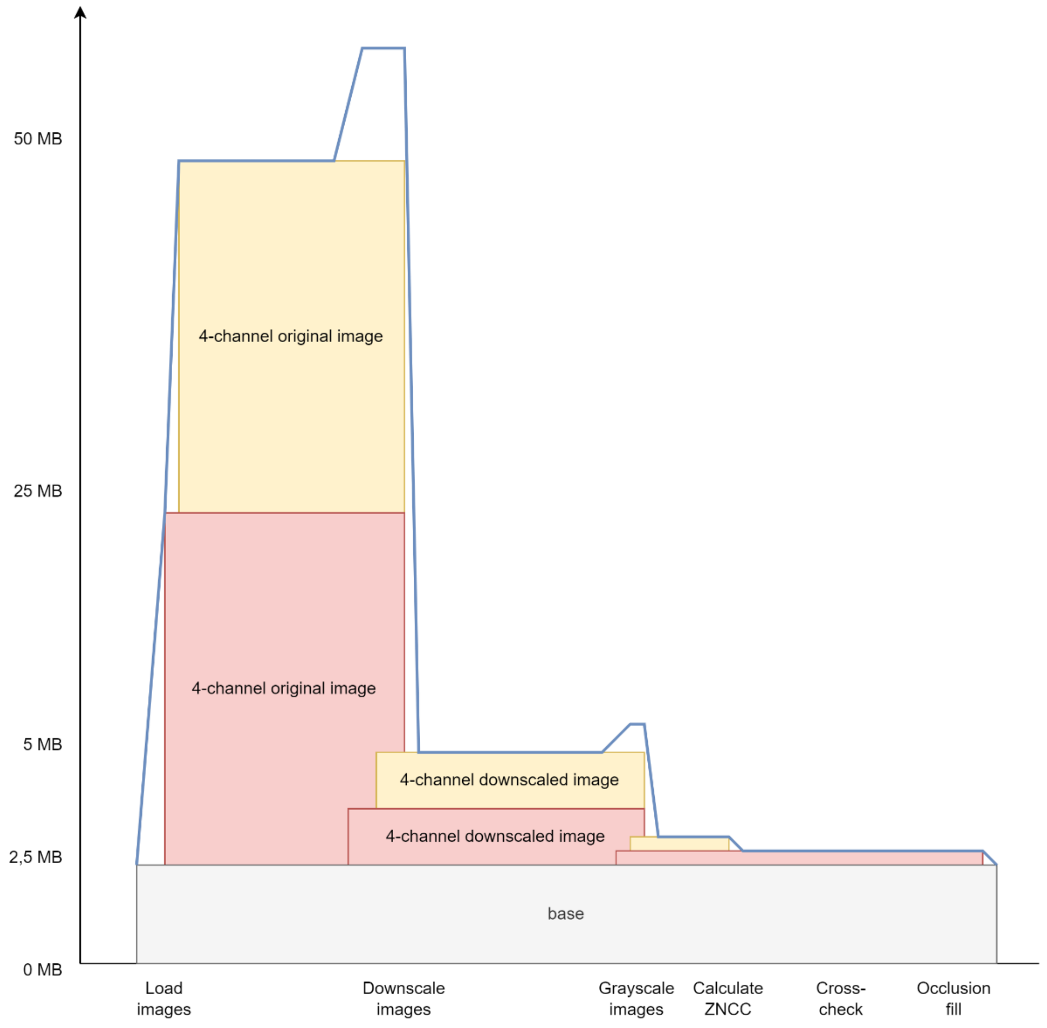
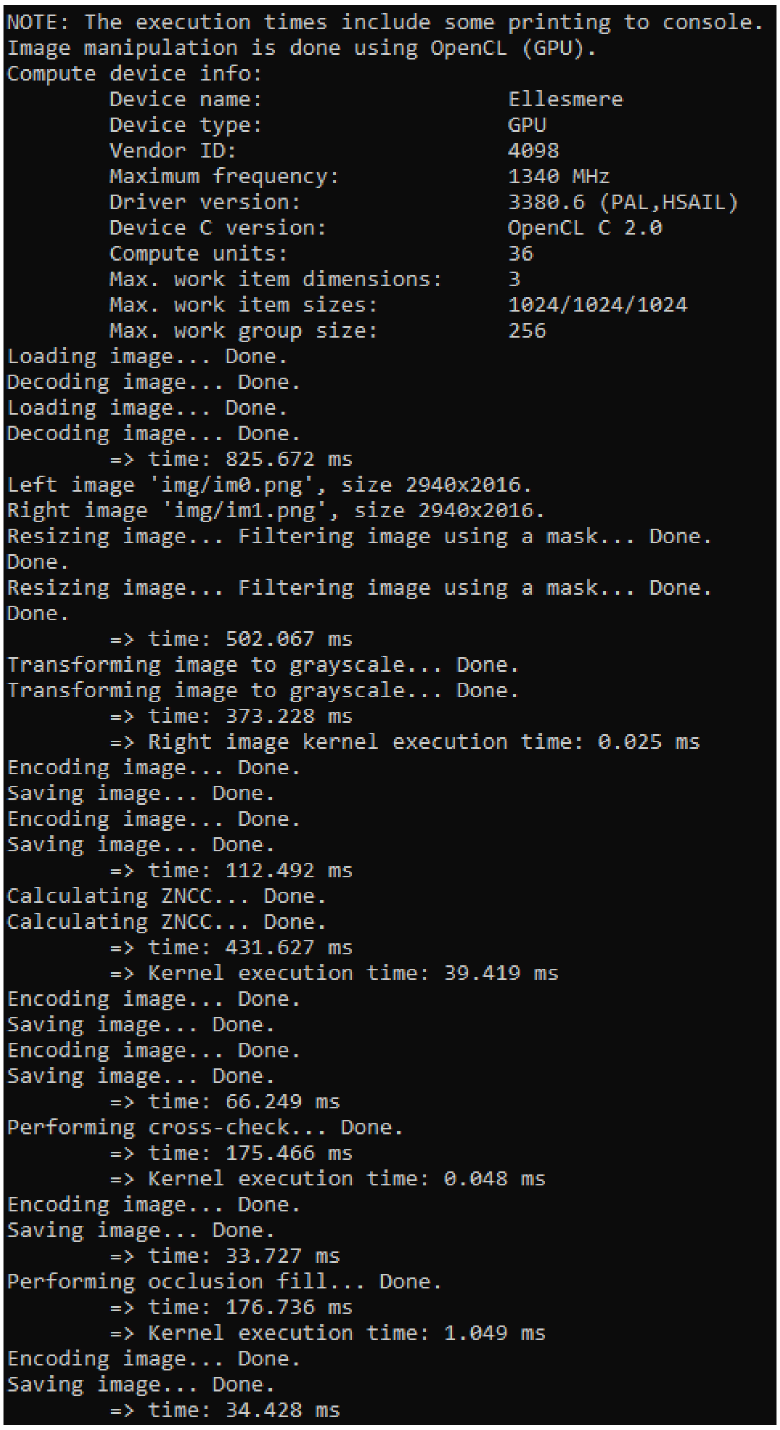


Figure 1. A mixed-format chart showing the memory consumption of the program during execution on OpenCL. The blue line shows the total consumption.



```
NOTE: The execution times include some printing to console.
Image manipulation is done using OpenCL (GPU).
Compute device info:
        Device name:                    Ellesmere
        Device type:                    GPU
        Vendor ID:                      4098
        Maximum frequency:              1340 MHz
        Driver version:                 3380.6 (PAL,HSAIL)
        Device C version:               OpenCL C 2.0
        Compute units:                  36
        Max. work item dimensions:      3
        Max. work item sizes:           1024/1024/1024
        Max. work group size:           256
Loading image... Done.
Decoding image... Done.
Loading image... Done.
Decoding image... Done.
        => time: 825.672 ms
Left image 'img/im0.png', size 2940x2016.
Right image 'img/im1.png', size 2940x2016.
Resizing image... Filtering image using a mask... Done.
Done.
Resizing image... Filtering image using a mask... Done.
Done.
        => time: 502.067 ms
Transforming image to grayscale... Done.
Transforming image to grayscale... Done.
        => time: 373.228 ms
        => Right image kernel execution time: 0.025 ms
Encoding image... Done.
Saving image... Done.
Encoding image... Done.
Saving image... Done.
        => time: 112.492 ms
Calculating ZNCC... Done.
Calculating ZNCC... Done.
        => time: 431.627 ms
        => Kernel execution time: 39.419 ms
Encoding image... Done.
Saving image... Done.
Encoding image... Done.
Saving image... Done.
        => time: 66.249 ms
Performing cross-check... Done.
        => time: 175.466 ms
        => Kernel execution time: 0.048 ms
Encoding image... Done.
Saving image... Done.
        => time: 33.727 ms
Performing occlusion fill... Done.
        => time: 176.736 ms
        => Kernel execution time: 1.049 ms
Encoding image... Done.
Saving image... Done.
        => time: 34.428 ms
```

Figure 2. Capture of the output of the program using OpenCL configuration.

Discussion and further optimizations

I saw some effort to try and make an alternative implementation where there would only be single kernel that would perform all the calculations on one go, only requiring one upload and one download to and from the compute device. The execution between compute units could be controlled using *barriers* and other control structures. However, it turned out to be impractical and hard to implement for several reasons. First, the implementation would require storing the intermediate results to device memory (e.g., both disparity maps), which would require dynamic memory allocation on the device memory. OpenCL does not offer an easy way for that.

As we saw, the peak memory usage is significantly higher than the average memory usage during computation. If we wanted to decrease the peak memory usage, we could first load the first image, downscale it, and then load the second one. This would decrease the peak almost to half.

One extreme optimization could be to balance the workload between CPU and GPU. If done correctly, this would speed up the processing even more since the CPU is currently idle while waiting for the GPU to execute.

If there was more time, I could try and optimize the algorithms themselves (kernels) to better for to the GPU world. This would include minimizing control flows and maximizing parallelization.

Also, it seems that in case of very simple kernels, there is no gain in using OpenCL. In an optimal implementation we could check the image size and compare it to some threshold to make the decision if it is worth it to transfer the image to an external compute device or just calculate it on the CPU.

References

[1]http://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide2.pdf

Reporting

Task	**Hours**
Converting to single channel	6 h
Other optimizations	3 h
Memory analysis	2 h
Benchmarking & writing the report	2 h
Total	**13 h**