FINAL REPORT

Multiprocessor programming 521288S

Students: Daniel Tisza, David Fekete

Obsah

1. Assignment

The main goal of the course task was to o implement and accelerate a stereo disparity algorithm in OpenCL. The implementation of this task was divided into five compulsory and one optional phase. The expected results after each compulsory phase were the following:

- A working development environment, knowledge of the open CL basics
- A simple set of routines working on OpenCL, with auxiliary functions.
- A simple but correct serial implementation using C/C++ code
- A C/C++ implementation that utilizes more than one core of the CPU
- A simple but correct parallel implementation of the algorithm running on GPU
- An optimized implementation, training diary, complete source code, and final report

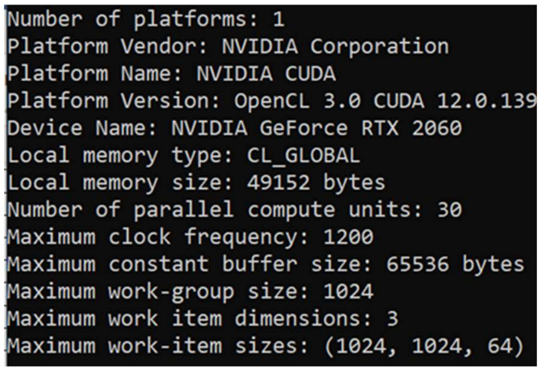
2. Platform and development environment

The goal of the Phase 0 was to set up the development environment. Based on previous experience we decided that we are going to use Visual Studio. Besides the IDE we had to install the C/C++ compiler and the OpenCL software development kit.

For the development, we were using two computers with different hardware. Table 1 shows the CPU parameter comparison of the devices and in Figure 1 and Figure 2, you can see the GPU parameters.

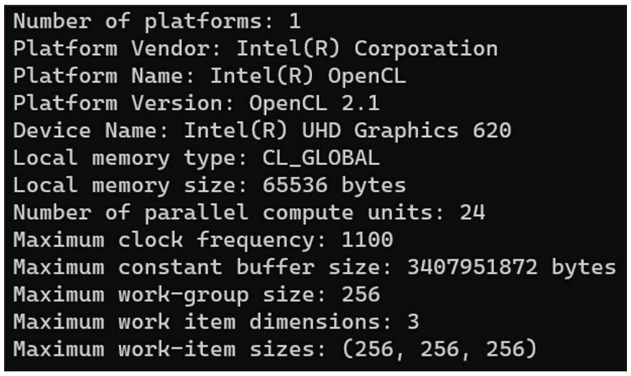
CPU information	Device 1	Device 2
Name	AMD Ryzen 7 4800H	Intel Core i5-8250U
Total cores	8	4
Total threads	16	8
Max frequency	4.20 GHz	3.40 GHz
Base frequency	2.9 GHz	1.60 GHz
Cache	12 MB	6 MB

Table 1 - CPU parameter comparison of the devices



```
Number of platforms: 1
Platform Vendor: NVIDIA Corporation
Platform Name: NVIDIA CUDA
Platform Version: OpenCL 3.0 CUDA 12.0.139
Device Name: NVIDIA GeForce RTX 2060
Local memory type: CL_GLOBAL
Local memory size: 49152 bytes
Number of parallel compute units: 30
Maximum clock frequency: 1200
Maximum constant buffer size: 65536 bytes
Maximum work-group size: 1024
Maximum work item dimensions: 3
Maximum work-item sizes: (1024, 1024, 64)
```

Figure 1 - Device 1 GPU parameters



```
Number of platforms: 1
Platform Vendor: Intel(R) Corporation
Platform Name: Intel(R) OpenCL
Platform Version: OpenCL 2.1
Device Name: Intel(R) UHD Graphics 620
Local memory type: CL_GLOBAL
Local memory size: 65536 bytes
Number of parallel compute units: 24
Maximum clock frequency: 1100
Maximum constant buffer size: 3407951872 bytes
Maximum work-group size: 256
Maximum work item dimensions: 3
Maximum work-item sizes: (256, 256, 256)
```

Figure 2 - Device 2 GPU parameters

Usually, we implemented our solution on one of the devices and evaluated the results on the other. We noticed that in some cases during the optimization the changes had different effects on the execution times. Therefore, if we developed only on one device, we could have got a more optimal solution for that device.

3. Implementation

During the phases, our task was to implement and accelerate a stereo disparity algorithm. In the end, we have four different implementations:

- Sequential implementation of the algorithm in C
- C implementation using CPU multi-threading and parallelization
- OpenCL implementation for GPU
- Optimized OpenCL implementation

During the development, the following functionalities were implemented with different approaches:

- **Load image** – For loading the images we are using the `lodepng_decode32_file()` function from the `lodepng` library. The function returns a pointer to a dynamically allocated buffer of 32-bit integers representing the RGBA pixel values of the image.
- **Resize image** – Downscaling the images to the required size.
- **Grayscale image** – The input is an RGBA image where each channel is represented as an unsigned char (or uchar4). The output consists only of one channel which is represented as a float
- **Filter image** – Input image is represented as float, we use gaussian blur to filtering, the output is represented again as unsigned char, but only 1 channel - grey
- **Adding border** – Adding a border to the image so the ZNCC algorithm can work properly.
- **ZNCC calculation** – ZNCC is function expresses the correlation between two grayscale images. The output is the depth map of the images.
- **Cross checking** – This functionality compares the depth map of the two images. We are checking if the corresponding pixels in the left and right disparity images are consistent. This can be done by comparing their absolute difference to a threshold value. We were experimenting with different threshold values in order to get the best results. If the absolute difference is larger than the threshold, then we replace the pixel value with zero. Otherwise, the pixel value remains unchanged.
- **Occlusion filling** - Eliminates the pixels that have been assigned to zero by the previously calculated cross-checking. In our implementations we are replacing the zero values the value of the nearest non-zero pixel.
- **Normalization** – Normalizes the pixel values from 0 to 255.
- **Save image** – We are using the `lodepng_encode_file` from the `lodepng` library.

3.1. Sequential C implementation

A simple sequential C implementation of the stereo disparity algorithm. It uses only one thread and one processor core. We are improving this solution in the next implementations.

3.2. CPU multi-threading and parallelization

The main task was to implement the C/C++ code that uses more than one thread of execution and more than one processor core. We decided that we will use the OpenMP directives for the implementation. The following functionalities were used from this library:

Parallelization of the for loops:

- `#pragma omp parallel for collapse()` – Most of the functions are using nested for loops. Collapse() combines multiple loops into a single iteration space and "parallel for" executes it parallel. You can specify the number of nested for loops as a parameter.
- `#pragma omp parallel for schedule(static)` – Distributes the iterations between the threads. It can use different scheduling types: `static`, `dynamic`, and `guided`. We were experimenting with these types, but there were no significant differences in the execution times, so we decided to use static. Static divides the iterations evenly between threads. The size of the iteration chunks can be also assigned manually, but we could not find a value that improved the execution time.

Parallelization in the main:

- We noticed that the loading, resizing, gray scaling, filtering, and adding border functions can be executed parallel, because the functions are working with two images that are independent from each other in this part of the code. We divided them into two sections using `#pragma omp sections` and executed them parallel with different threads using `#pragma omp parallel`. This did not affect the functions, but significantly improved the execution time of the entire process.

Data sharing attribute clauses

- There are many possibilities of specifying how parallel threads are sharing variables. We needed to specify this only once. We are setting some variables to `private` in the ZNCC calculation, so there are no synchronization issues of these variables between the threads.

Synchronization clauses

- `#pragma omp barrier` – Creates a synchronization point where the execution continues only when all the threads reached this point. It is used before the ZNCC calculation because it requires both images at the same time but one of the sections finishes earlier.

Most of the time, we were just experimenting with different pragma methods and trying different parameters. We decided to use one or another method based on the difference in the execution times.

We were implementing the parallelization on the Device 1, so the decisions were made based on the result on that device. However, we evaluated our implementation on the Device 2 and in some cases, it had a negative impact on the execution times. It happened with the shorter functions, but it had a significant improvement on the ZNCC calculation.

3.3. OpenCL implementation for GPU

In this phase we were implementing OpenCL kernels of the required functionalities. We implemented a kernel for every function in the image processing, corresponding to the C code, therefore the code is divided to sections. After the required OpenCL setup (getting platform id, getting device id, creating context, creating a command queue, creating a program and building the program) each section is following the same structure:

- Creating buffers for the input and output images using clCreateBuffer
- Transferring the data into the device using clEnqueueWriteBuffer
- Creating the corresponding kernel using clCreateKernel
- Setting the kernel arguments using clSetKernelArg
- Executing the kernel using clEnqueueNDRangeKernel
- Reading back the images from the device using clEnqueueReadBuffer

We are measuring the execution times of each kernel, and also the total execution time of the whole process (including loading and saving the images). In this first OpenCL implementation after the execution of each kernel the images are read back from the device (excluding the post processing phase – no need to allocate memory for new images, because the modifications are being done in the same image).

Corresponding to the C implementation, all the kernels are working with images represented as unsigned char, except for 2 kernels:

- grayscale_image - input image: unsigned char (RGBA), output_image: float
- filter_image – input image: float, output_image: unsigned char (G)

OpenCL implementation was being implemented mostly on Device 2, but since Device 1 has a much more powerful GPU, it turned out to run faster on it, except the loading and saving time.

3.4. Optimized OpenCL implementation

In this final implementation, we tried to optimize the OpenCL implementation even more. This implementation is done in a way to minimize the read/write operations of the buffers. Therefore, in contrast to the previous solution, where the image was read back from the device after nearly every execution of a kernel, here only the buffers of the input images are marked with CL_MEM_READ_ONLY flag, all the other buffers are CL_MEM_READ_WRITE, to enable reading the buffer and also writing into it. The image is read back to the host only after the post processing phase, when the whole process is complete, and it is time to save the final image.

In this implementation, we represent the input image as uchar4 to enable vectorization. Therefore, the kernels resize_image and grayscale_image are modified for this solution to use the input and resized images represented as uchar4 (RGBA values). After the grayscaling, we decided that it is no longer necessary to use uchar4 values, as our image is represented by only one channel (grey), so we use the same kernels as in the previous OpenCL implementation.

Also, we implemented our own version of the functions sqrt and abs placed directly in "image_processing_optimized.cl" kernel instead of using the functions provided in libraries.

4. Results

In this section we are going to compare the different implementations from the view of execution times and CPU loads. We measured the execution times of each functionality separately and also measured the total execution times. The results of the profiling are summed up in the Table 2 an Table 3.

4.1. Comparison of the execution times

	Device 1		Device 2	
Function	Without parallelization	Parallelized	Without parallelization	Parallelized
Loading image 1	2.077 s	2.049 s	1.531 s	1.232 s
Loading image 2	2.081 s	2.072 s	0.803 s	1.241 s
Resizing image 1	0.002 s	0.003 s	0.003 s	0.006 s
Resizing image 2	0.002 s	0.002 s	0.003 s	0.008 s
Gray scaling image 1	0.002 s	0.002 s	0.004 s	0.005 s
Gray scaling image 2	0.002 s	0.001 s	0.004 s	0.005 s
Filtering image 1	0.028 s	0.025 s	0.036 s	0.104 s
Filtering image 2	0.029 s	0.024 s	0.033 s	0.096 s
Adding border 1	0.001 s	0.002 s	0.001 s	0.003 s
Adding border 2	0.002 s	0.001 s	0.001 s	0.003 s
Calculating ZNCC 1	17.001 s	2.241 s	36.184 s	11.856 s
Calculating ZNCC 2	16.915 s	2.261 s	40.947 s	11.916 s
Cross-checking	0.003 s	0.001 s	0.003 s	0.001 s
Occlusion filling	0.001 s	0.000 s	0.001 s	0.001 s
Normalization	0.002 s	0.001 s	0.001 s	0.001 s
Saving image	0.173 s	0.174 s	0.135 s	0.316 s
Full execution time	38.312 s	6.794 s	79.704 s	25.469

Table 2 - Comparison of the sequential and the parallel (optimized) C implementations

In Table 2 we can see that parallelization significantly improved our implementation. By parallelizing the for loops the functions became faster. By executing parallel the first part of the program (loading, resizing, grayscaling, filtering, adding border), we also managed to improve the full execution time of the program.

	Device 1		Device 2	
Function	Without optimization	Optimized	Without optimization	Optimized
Loading image 1	2.113000 s	2.040000 s	1.070000 s	1.362000 s
Loading image 2	2.141000 s	2.082000 s	1.641000 s	0.842000 s
Resizing image 1	0.000311 s	0.000087 s	0.001280 s	0.000342 s
Resizing image 2	0.000313 s	0.000087 s	0.001356 s	0.000333 s
Gray scaling image 1	0.000115 s	0.000091 s	0.000436 s	0.000342 s
Gray scaling image 2	0.000111 s	0.000083 s	0.000449 s	0.000333 s
Filtering image 1	0.000316 s	0.000317 s	0.003005 s	0.003007 s
Filtering image 2	0.000314 s	0.000315 s	0.003014 s	0.003005 s
Adding border 1	0.000115 s	0.000116 s	0.000110 s	0.000112 s
Adding border 2	0.000118 s	0.000117 s	0.000119 s	0.000108 s
Calculating ZNCC 1	0.274170 s	0.234016 s	0.230997 s	0.221494 s
Calculating ZNCC 2	0.157428 s	0.145862 s	0.230552 s	0.227480 s
Cross-checking	0.000061 s	0.000055 s	0.000372 s	0.000374 s
Occlusion filling	0.000055 s	0.000043 s	0.000295 s	0.000299 s
Normalization	0.000055 s	0.000066 s	0.000253 s	0.000255 s
Saving image	0.156000 s	0.149000 s	0.295000 s	0.892000 s
Total kernel execution time	0.673000 s	0.593000 s	1.50300 s	0.892000 s
Total execution time (with load, save)	5.092000 s	4.872000 s	4.54700 s	3.40300 s

Table 3 - Comparison of the not optimazed and the optimized OpenCL implementation

In the Table 3 we can see the comparison of the execution times of the OpenCL implementation before and after optimization. We can see that the result are significantly improved compared to the sequential and parallelized C implementation. We can also see that the execution times are slightly shorter after the optimization. We can see the effect of the vectorization on the resizing and grayscaling functionalities and the effect of the fast mathematics in the ZNCC calculation. We also tried the vectorization on the other functions, but it had negative effect on the execution times.

4.2. Comparison of the CPU load

Using the Debug Performance Profiling feature in the Visual Studio we were also able to compare the CPU load of the parallelized C and the optimized OpenCL implementation. In the figures below we can see that by offloading most of the computations to a GPU in the OpenCL implementation the use of the CPU significantly decreased.

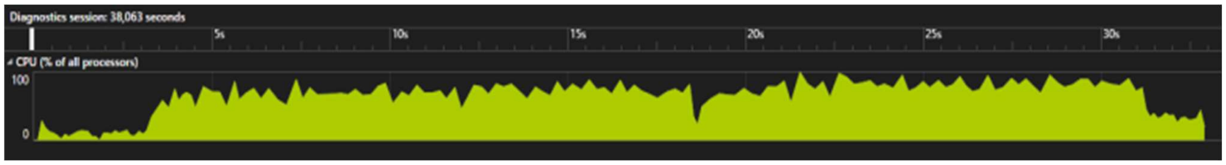


Figure 3 - CPU load during parallel C implementation

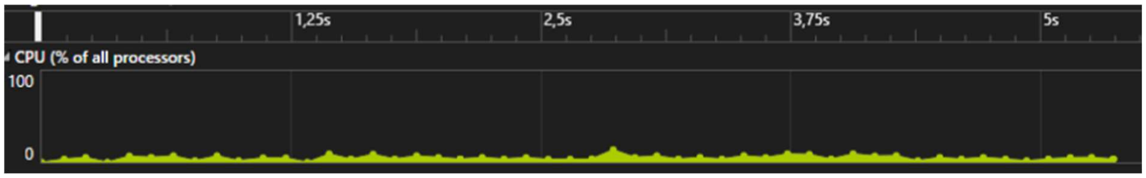


Figure 4 - CPU load during optimized OpenCL implementation

5. Summary

Looking at the execution times of all 4 implementations we can see a lot of improvement. The first implementation running sequentially being obviously the slowest one, on Device 1 at around 38 seconds, while on Device 2 at around 79 seconds.

Parallelizing this implementation using OpenMP pragmas made a lot of improvement, on Device 2 cutting the execution time just to around 1/3 of the first implementation (around 25 seconds). On Device 1 it was even more significant, with execution time being around 1/6 of the first implementation (around 6.5 seconds). Since device 1 has more CPU cores and threads, this is not a big surprise.

Using OpenCL kernels with the GPU, we cut the execution time even more. On Device 1 the execution time was around 2 seconds faster, just under 5 seconds. However, on Device 2, we managed to cut the time even more drastically and execute the process around 5-6x faster, compared to the parallelized C code implementation, being around 4,5 seconds.

We tried to optimize the OpenCL implementation even more, not reading back the images from the device, just at the end, as well as trying to use vectorization for the input and resized images. With these optimizations, we could cut the execution time by a few hundred milliseconds.

The biggest improvement is obviously the calculation of disparity images using the ZNCC algorithm, which we managed to execute at around 0,15 – 0,25 seconds in the end. Compared to the sequential C implementation (around 17 seconds on Device 1 and 35-40 seconds on Device 2) it is approximately an 85 times faster solution on Device 1 and around 175 times faster solution on Device 2.

In the end, the total execution time of the solution was around 5 seconds on Device 1 and around 3,5 seconds on Device 2, with the OpenCL execution being under 1 second on both devices. Much of the time was consumed by the operations of loading and saving the image.

6. Training diary

Functionality	Working hours	Student
Phase 0		
Setup of the environment	3	Both
Phase 1		
Load image	0.5	Both
Resize image – C	0.5	
Resize image - kernel	0.5	
Grayscale image - C	1	
Grayscale image - kernel	1	
Filter image - C	2	
Filter image - kernel	1	
Adding border - C	1	
Kernel execution environment	3	
Other(learning the basics of OpenCL, kernels etc.)	3	
Phase 2		
ZNCC calculation - C	4	Both
Save image	0.5	
Profiling info - C	2	
Profiling info - OpenCL	1	
Platform info - OpenCL	1	
Bug fixing (ZNCC)	1	
Cross-checking	2	
Occlusion filling	2	
Phase 3		
Reading about parallelization and OpenMP	2	Daniel
Parallelization C code using OpenMP	7	
Phase 4		
Adding border kernel	1	David
ZNCC calculation kernel	2.5	
Post-processing kernel	1.5	
Kernel execution environment	1	
Phase 5		
Researching the optimizing options	3	Both
Optimizing the OpenCL implementation – memory coalescing	4	Both
Optimizing the OpenCL implementation - vectorization	4	David
Other		
Writing the reports (first, second, final)	10	Both
Commenting the code	4	Both
Total hours	70	