Multiprocessor programming 521288S

Miguel Bordallo López
Praneeth Susarla
Manuel Lage Cañellas

University of Oulu
Center for Machine Vision and Signal Analysis
Faculty of Information Technology and Electrical Engineering
February 6th, 2025

GENERAL INFORMATION

Welcome to the Multiprocessor Programming course! This course consists of one exercise, and the teacher is:

Miguel Bordallo, miguel.bordallo@oulu.fi, TS349

In addition to the teacher, the course will have two assistants. They are:

Manuel Lage, manuel. lage@oulu.fi, TS313
Praneeth Susarla, praneeth.susarla@oulu.fi, TS315

This handout contains the exercise instructions for the Multiprocessor Programming course. When the exercise is completed, you will receive 5 ECTS credits. The completed answers, a training diary and all code written should be returned by email to the corresponding teacher. For inquiries, you can primarily contact the teacher, but you can also be in contact with the assistant.

Each returned exercise will be either accepted or rejected. If an exercise is rejected, the teacher or assistant will give instructions which will help you improve your answers or programs.

When the exercise is accepted, you will complete a short *exit document or email* containing some questions about the total workload, the distribution of the work and the self-assessment of the grade. The contents of this document/questionnaire will be discussed in a final **exit interview**, where the grades will be given.

Based on the overall quality of the work, you will receive a grade from 1 to 5.

There is no final exam in the course, but during the exit interview, be prepared to answer a few oral questions about the exercise and its implementation. Also, you should demonstrate (if required by the teacher) that your program is working properly. If you are unable to finish the work in time, you **SHOULD** explicitly ask for additional time. The compulsory part of the finished exercise work should be returned before **May 19th, 2025**.

In the course's beginning, there were two voluntary initial lectures, where some general topics were discussed. If you did not attend the lectures, you should download the slides to get a general idea of the organization of the course. If they raise any doubts, you can contact the teacher or assistants.

The course can be carried out either alone or *(preferably*) in groups of ***a maximum of*** two students. Each group should carry out the exercises by themselves. The assistants will give advice on request. Copying code or answers from other groups is definitely prohibited and will lead to penalties!

Before starting the work, each group must register by registering the group by sending an email with the names of the components of the team and the student numbers.

After the initial lecture, the course material and further announces will be placed in the official Moodle website:

https://moodle.oulu.fi/course/view.php?id=27283

Visit it often to see updates to the instructions or material

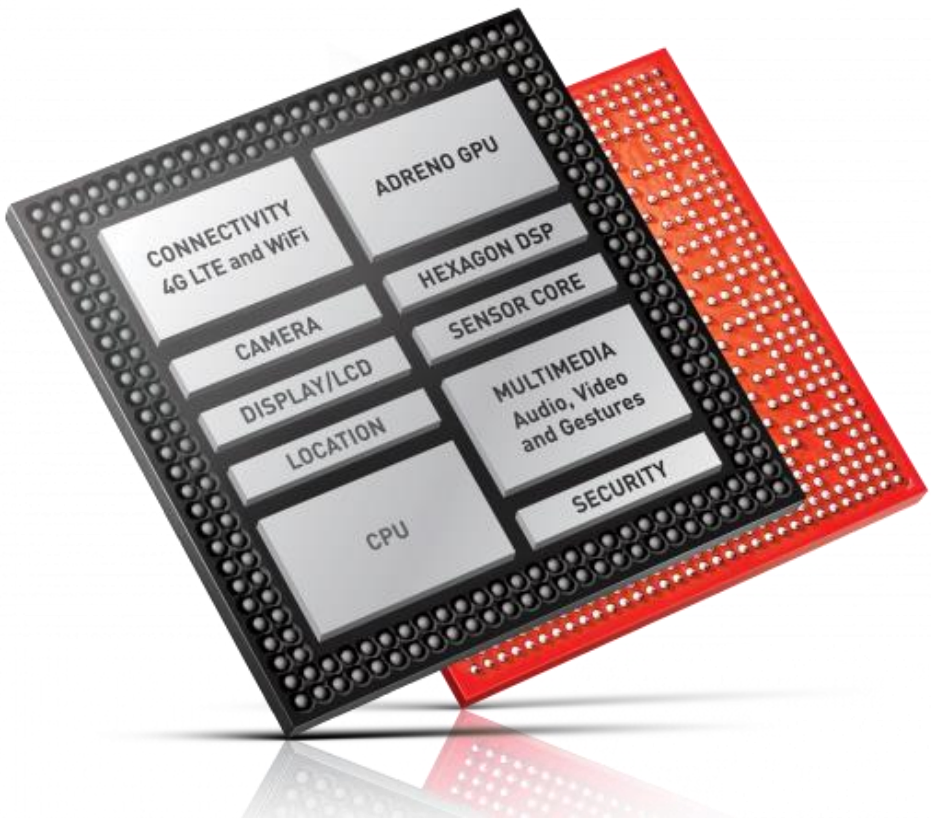
NOTE:
If you are an exchange student and/or have different grade requirements other than an ECTS grade, you must ask for it when returning to the work. If you are planning to graduate soon after completing the course, you should mention it to the teacher *(e.g., this is almost your last course).* Otherwise, the completion mark will be given within three weeks after the exit interview.

BACKGROUND

Mobile communication devices are becoming attractive platforms for multimedia applications as their display and imaging capabilities are improving together with computational resources. Many of the devices have increasingly been equipped with several sensors i.e. motion sensors, environmental sensors and position sensors that allow the users to capture several signals at different rates.

To process these signals, with power efficiency and reduced space in mind, most mobile device manufacturers integrate several chips and subsystems on a System on Chip (SoC). Figure 1 illustrates the organization of an old Snapdragon 805 system-on-chip (SoC) featured e.g. on Nexus 6 mobile phones. However, the same basic structure can be found on any high-end mobile phone or tablet.

For a long time, it was a tedious task to develop programs that could exploit all available hardware. Since Open Computing Language (OpenCL) and other similar tools were introduced, the development work was simplified significantly. However, the code still needs to be optimized using approaches that depend on the architectural differences, using strategies such as platform dependent macros embedded in the actual OpenCL code. The program development itself has changed quite dramatically. The same code can be run and tested on multiple devices. In addition, the code can even be portioned to run on multiple devices simultaneously, providing true heterogeneous computing capabilities with a single programming framework.



Qualcomm Snapdragon 805 *https://www.qualcomm.com/products/snapdragon/processors/805*

In this project, we will concentrate on a simple heterogeneous computing setup using only a multicore CPU and a GPU. The implementation will be benchmarked on both devices individually as well as with a heterogeneous setup using both devices. The CPU works as the “host”-processor in all cases according to the OpenCL platform model. You will make the final optimization choice on which parts of the algorithms to offload to the GPU.

THE ASSIGNMENT

Your task is to **implement and accelerate** a stereo disparity algorithm in **OpenCL** by offloading most of the computations to a GPU. For comparative reasons, the same OpenCL implementation should be preferably benchmarked on a CPU-only setup too. A heterogeneous CPU-GPU version where the selection of the processor is based on the kernel execution times can also be implemented.

The algorithm to be implemented is "Depth estimation based on Zero-mean Normalized Cross Correlation (**ZNCC**)". ZNCC is a function that expresses the correlation between two grayscale images (or patches). The result of the function is an integer that becomes larger the more that these two patches are correlated. Estimating depth with ZNCC can be done by evaluating the following expression.

$$\boldsymbol{ZNCC(x,y,d)} \triangleq \frac{\sum_{j=-\frac{1}{2}(B-1)}^{\frac{1}{2}(B-1)} \sum_{i=-\frac{1}{2}(B-1)}^{\frac{1}{2}(B-1)} [I_L(x+i,y+j) - \overline{I_L})] * [I_R(x+i-d,y+j) - \overline{I_R}]}{\sqrt{\sum_{j=-\frac{1}{2}(B-1)}^{\frac{1}{2}(B-1)} \sum_{i=-\frac{1}{2}(B-1)}^{\frac{1}{2}(B-1)} [(I_L(x+i,y+j) - \overline{I_L}]]^2} * \sqrt{\sum_{j=-\frac{1}{2}(B-1)}^{\frac{1}{2}(B-1)} \sum_{i=-\frac{1}{2}(B-1)}^{\frac{1}{2}(B-1)} [I_R(x+i-d,y+j) - \overline{I_R}]^2}}$$

B = window size, I_L, I_R= left and right images, $\overline{I_L}$, $\overline{I_R}$ window average, d = disparity value (0 to d)

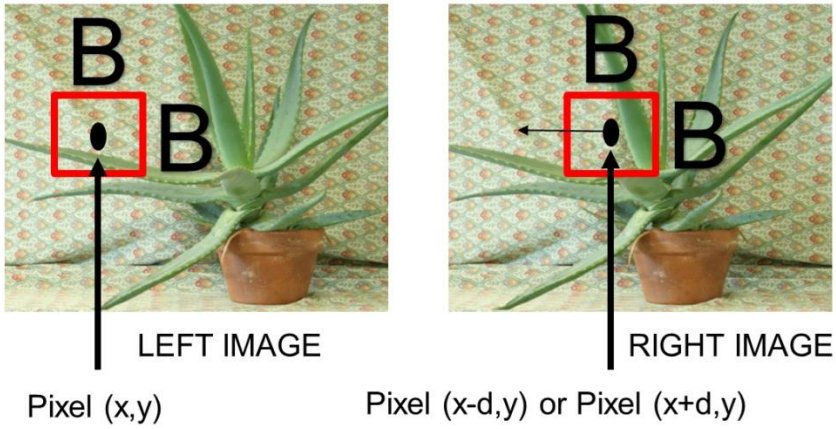
The final disparity value will be chosen based on the Winner-takes-it-all-approach, where the value of d with largest ZNCC(x,y,d) value is chosen for the pixel value (x,y) of the final disparity map.
Note that for our simple case, the disparity can be considered one-dimensional (horizontal only) and there is no disparity in y-axis. This is because the sample images that we use for our computation are already *rectified*, as it is usually the case from the input of stereo cameras.
The previous equation can be also expressed using the following Pseudo code:

```
FOR J = 0 to image height
    FOR I = 0 to image width
        FOR d = 0 to MAX_DISP
            FOR WIN_Y = 0 to WIN_SIZE
                FOR WIN_X = 0 to WIN_SIZE
                    CALCULATE THE MEAN VALUE FOR EACH WINDOW
                END FOR
            END FOR

            FOR WIN_Y = 0 to WIN_SIZE
                FOR WIN_X = 0 to WIN_SIZE
                    CALCULATE THE ZNCC VALUE FOR EACH WINDOW
                END FOR
            END FOR

            IF ZNCC VALUE > CURRENT MAXIMUM SUM THEN
                UPDATE CURRENT MAXIMUM SUM
                UPDATE BEST DISPARITY VALUE
            END IF
        END FOR

    DISPARITY_IMAGE_PIXEL = BEST DISPARITY VALUE
    END_FOR
END FOR
```



POST PROCESSING:

Once the ZNCC equation is evaluated and a first estimation of the depth map is obtained, it should be post processed for refinement.

The post processing stage consists of two phases: cross-checking and occlusion filling.

Cross checking: (Two input images, one output image)

Cross checking is a process where you compare two depth maps. The left disparity map is obtained by mapping the image on the left against the one on the right. The right disparity map is obtained analogously by mapping the image on the right against the one on the left.

To obtain a consolidated map, the process consists in checking that 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. For our case, it is recommended to start with a threshold value of 8, while the best threshold value for your implementation can be obtained by heuristic experimentation. If the absolute difference is larger than the threshold, then replace the pixel value with zero. Otherwise, the pixel value remains unchanged. This process helps in removing the probable lack of consistency between the depth maps due to occlusions, noise, or algorithmic limitation.

Occlusion filling: (One input image, one output image)

Occlusion filling is the process of eliminating the pixels that have been assigned to zero by the previously calculated cross-checking. In the simplest form it can be done by replacing each pixel with zero value with the nearest non-zero-pixel value. However, in this exercise it is recommended that you experiment with other more complex post-processing approaches that obtain the pixel value in different forms.

The previously depicted description of the algorithm and the enclosed pseudocode are just meant to help you get started with the algorithm implementation. However, notice that this is not the only (or necessarily the best) solution. You are free to experiment with other approaches. Once you understand the algorithm so you can design a sequential implementation, you can start evaluating the opportunities of parallelization that could be exploited in the exercise's several phases.

DATA SET UTILIZATION:

In this exercise, for benchmarking purposes we are going to utilize images from the following dataset:
https://drive.google.com/drive/folders/1xIFoXV9yR9aODxB1hFp8pIeG0i08G4SO
img_0.png, img_1.png

Other alternative datasets are also possible and recommended to be tried: (use the *perfect* versions)
https://vision.middlebury.edu/stereo/data/scenes2014/

As a starting point, download images *im0.png* and *im1.png*. The maximum disparity value for the two images included in the package can be found on the *calib.txt* file and is *ndisp=260* for the above mentioned two images. Take this disparity value into account if you do any image downsizing and reduce it accordingly.

When starting to code the algorithm, it is recommended that you use LodePNG to read the images, you will only need to include *lodepng.c*- or *lodepng.cpp*-source file and the *lodepng.h*-header file to your project and use the appropriate functions provided in these files to read and write the image files. If you feel it is more convenient, you are welcome to use any other image-reading library or function.

GETTING STARTED

The exercise is divided in six phases, and three of them have a specific checkpoint where the progress of the implementation needs to be reported to the teachers:

Phase 1: Self-study and platform installation
Expected result: A working development environment, knowledge of the open CL basics

Phase 2: OpenCL introduction. Matrix addition, filtering, profiling.
Expected result: A simple set of routines working on OpenCL, with auxiliary functions.

Phase 3: Stereo disparity implementation in sequential single thread C/C++-code
Expected result: A simple but correct serial implementation using C/C++ code

Phase 4: Stereo disparity implementation using in multi-threaded C/C++-code
Expected result: A C/C++ implementation that utilize more than one core of the CPU

Phase 5: Stereo disparity implementation using OpenCL for a GPU
Expected result: A simple but correct parallel implementation of the algorithm running on GPU

Phase 6: Algorithmic and implementation optimization for OpenCL in CPU/GPU
Expected result: An optimized implementation that considers the GPU architecture
Expected documents: Training diary, complete source code, final report

Phase 7 (Optional): Porting of the algorithms to a mobile platform:
Expected result: An OpenCL implementation that uses efficiently the memory hierachy ***OR***
an OpenCL implementation able to run on an ORDROID platform.

PHASE 1: Installation of the environment and self-study

Duration:

- *2 weeks*

Checkpoint:

- *7 of February 2025*

Important tasks:

- *Registration of the group with the teacher via e-mail*
- *Understanding basic foundations of OpenCL programming*
- *Environment installed and 'hello world' program working*

Recommended Lectures:

- *Three scientific articles on the historical development of multiprocessing*
- *Chapter 1 of OpenCL in Action: How to Accelerate Graphics and Computations, (Matthew Scarpino)*

This phase is dedicated to self-study and installation of the work environment. Please take your time for reading the recommended lecture as it will give you a strong understanding about OpenCL concepts. After this you will need to prepare your computer for running OpenCL code.

STEP 1. C/C++ Environment setup

Since you will be using your own computer, you will need to first install a C/C++ compiler. Start your work by setting up the development environment. We recommend any integrated development environment (IDE) suitable for your operative system such as:

Eclipse	https://www.eclipse.org/downloads/
Code::Blocks	https://www.codeblocks.org/downloads/
Visual Studio	https://visualstudio.microsoft.com/downloads/
Visual Studio Code	https://code.visualstudio.com/Download

You are free to use any other editor, compiler (GCC / G++) and debugger (GDB or DDD).

Note:
For Windows users that want to use Visual Studio, it can be obtained signing in with your O365 university credentials and joining the developing program.

STEP 2. OpenCL Environment setup

SDK Installation:
Once your C/C++ environment is working, you will need at least one OpenCL Software Development Kit (SDK). An OpenCL SDK is a set of tools and libraries that enable developers to create applications that leverage the parallel processing capabilities.
OpenCL, which stands for Open Computing Language, is not only a programming language but an open standard for parallel programming across different types of processors, including CPUs, GPUs. This means that there is not only one standard SDK version, but different versions depending on the hardware vendors. AMD, NVIDIA, Intel, etc. provide their own implementations of the OpenCL standard. You will need to use a specific SDK version corresponding to your particular hardware.
For this course, we provide a guide to install four different SDKs:
NVIDIA (GPU) **Intel** (CPU, GPU) **AMD** (GPU) and **APPLE** (CPU-GPU)

If you are using a different one, we kindly invite you to help us expanding our guide!

Note:
Different SDK can coexist in the same machine (Such as NVIDIA + INTEL) but for this course having one working SDK is enough. Please refer to the guides available in MOODLE to install one SDK in your machine.

01 OpenCL_SDK_Installation.pdf

SDK Integration:
The SDK installation provides a library for the linker and "include" folder for the compiler. Depending on the SDK and your operative system, they can be in different places.
Your C/C++ IDE needs to link against those libraries.
In this Moodle guide, we provide some helpful tips to find your library locations as well as how to integrate them with some different IDEs. Again, if you use a different IDE, we kindly invite you to help us expanding our guide.

02 OpenCL_IDE_Integration.pdf

Note:
You can use a virtual Linux machine running on your Windows/Mac machine, but there is no guarantee that you will be able to use the GPU!

STEP 3. 'Hello World' program running in OpenCL.

At this point you should be able to run OpenCL code (a kernel).
The following code shows a basic "Hello World" for OpenCL.
You can get the full code as hello_world.c in Moodle.

hello_world_open_cl.c

This simple OpenCL contains:
1 - Main: C code executed in the host (CPU)
2 - Hello: OpenCL kernel executed in any OpenCL device (CPU or GPU)

If your OpenCL installation and integration is ok, the code should give 0 warnings.

main.c

```c
/*
Example of where you can find your OpenCL based on your SDK
NVIDIA Linux
Linker:     /usr/local/cuda-11.6/lib64/libOpenCL.so
Directory:  /usr/local/cuda-11.6/include

NVIDIA Windows
Linker:     Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.6\lib\x64\OpenCl.lib
Directory:  Program Files\NVIDIA GPU Computing Toolkit\CUDA\v11.6\include

Intel CPU Linux
Linker:     /opt/intel/system_studio_2020/opencl/SDK/lib64/libOpenCl.so
Directory:  opt/intel/system_studio_2020/opencl/SDK/include

Intel Graphics Linux
install intel-opencl-icd

*/

#include <stdio.h>
#include <stdlib.h>

//Depending of your installation more includes should be uses, check your particular SDK installation
//
#include <CL/cl.h>


//This is a kernel, a piece of code intended to be executed in a GPU or CPU
const char *kernel_source =
"__kernel void hello(__global char *output) {"
"output[0]='h';"
"output[1]='e';"
"output[2]='l';"
"output[3]='l';"
"output[4]='o';"
"output[5]=',';"
"output[6]=' ';"
"output[7]='w';"
"output[8]='o';"
"output[9]='r';"
"output[10]='l';"
"output[11]='d';"
"output[12]='\\n';"
"}";

//The rest of the code is intended to be executed in the host
int main()
{
    cl_int          err;
    cl_uint         num_platforms;
    cl_platform_id  *platforms;
    cl_device_id    device;
    cl_context      context;
    cl_command_queue queue;
    cl_program      program;
    cl_kernel       kernel;
    cl_mem          output;

    char result[13];

    // PLATFORM
    // In this example we will only consider one platform
    //
    int num_max_platforms = 1;
    err = clGetPlatformIDs(num_max_platforms, NULL, &num_platforms);
    printf("Num platforms detected: %d\n", num_platforms);
    platforms = (cl_platform_id*) malloc(sizeof(cl_platform_id) * num_platforms);
    err = clGetPlatformIDs(num_max_platforms, platforms, &num_platforms);

    if(num_platforms < 1)
    {
        printf("No platform detected, exit\n");
        exit(1);
    }

    //DEVICE (could be CL_DEVICE_TYPE_GPU)
    //
    err = clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_ALL, 1, &device, NULL);

    //CONTEXT
    //
    context = clCreateContext(NULL, 1, &device, NULL, NULL, &err);

    //QUEUE
    //
    queue = clCreateCommandQueue(context, device, 0, &err);

    //READ KERNEL AND COMPILE IT
    //
    program = clCreateProgramWithSource(context, 1, &kernel_source, NULL, &err);
    err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);


    //CREATE KERNEL AND KERNEL PARAMETERS
    //
    kernel = clCreateKernel(program, "hello", &err);
    output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, 13 * sizeof(char), NULL, &err);
    err = clSetKernelArg(kernel, 0, sizeof(cl_mem), &output);

    //EXECUTE KERNEL!
    //
    err = clEnqueueTask(queue, kernel, 0, NULL, NULL);

    //READ KERNEL OUTPUT
    //
    err = clEnqueueReadBuffer(queue, output, CL_TRUE, 0, 13 * sizeof(char), result, 0, NULL, NULL);
    printf("%s", result);


    //Free your memory please...
    clReleaseMemObject(output);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);
    free(platforms);


    return 0;
}
```

The output of this code should be like

```
Num platforms detected: 1
hello, world

Process returned 0 (0x0)   execution time : 0.194 s
Press ENTER to continue.
```

If you see a similar code, congratulations, you have finished Phase 1!

PHASE 2: OpenCL introduction. Matrix addition, filtering, profiling.

Duration:

- *2 weeks*

Checkpoint:

- *7th of March 2025 (same as phase 3)*

Important tasks:

- *Set up a framework for time measurement and profiling*
- Matrix addition program in both host and OpenCL
- *Set up auxiliary functions to read/write images*
- *Implement a simple set of routines (matrix addition, grayscale conversion, filtering) in C and OpenCL*

Recommended Lecture:

- *Chapter 2,3 of OpenCL in Action: How to Accelerate Graphics and Computations, (Matthew Scarpino)*

This phase is dedicated to the setup of the framework and boilerplate code to be used during the rest of the assignment.

The first exercise goal is to compare code with same functionality working in host part and in OpenCL
The second exercise goal is to create a working program that can load and save images, allows for time measurement and profiling, and can execute simple OpenCL kernels in the desired data.
The third exercise consists in replicating the code of the second exercise but in OpenCL and compare the execution time.

Exercise 1: Matrix multiplication in C and OpenCL

In this warm-up exercise, the objective is to perform a matrix multiplication using the C programming language and then replicate a similar function using OpenCL. Throughout the exercise, participants will gain insights into the process of profiling their code to assess the time execution disparities between the Host and the parallel computing device.
Profiling requires measure at least the execution times of each of the different operations such as reading/writing images, converting to grey scale, etc. Feel free to measure the usage of memory, bus data transfer between host and device and the time as well if you are interested. and the profiling information on the terminal and save it for the report.

To profile your code, we recommend the following functions:

C/C++: gettimeofday(..) or QueryPerformanceCounter(...)
OpenCL: clGetEventProfilingInfo(..)

Tasks:

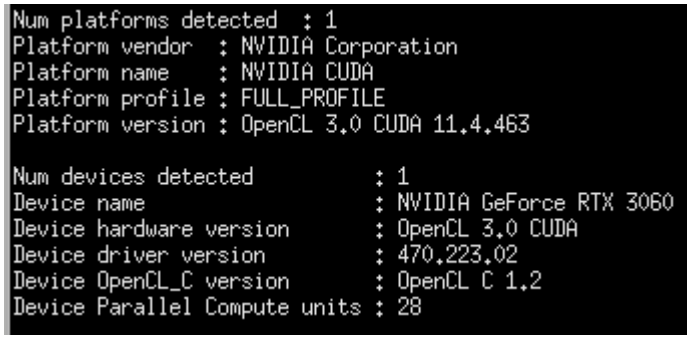
- Display your OpenCL platform information using clGetPlatformInfo function

- Implement a C/C++ program, that performs element-wise addition of two matrices. The implementation should define a "add_Matrix" function that considers two 100x100 matrices as input. The matrices must be created using dynamic memory (malloc or calloc).
  You can define them (e.g.) as float matrices: *float *matrix_1 = (float *) malloc(...)*

- Profile your code: The final host execution time of the implementation should be displayed as output.

- Similarly, implement a simple OpenCL kernel ‘add_matrix’ performing element-wise addition of two matrices. The ‘add_matrix’ kernel should consider two matrices as the input similar to C function.

- Profile your code: The final device execution time of the implementation should be displayed as output.



Exercise 2: Read/save image, convert to grayscale, reduce size, apply filter in C/C++

This exercise is designed to guide participants through the process of reading an image on the host, performing image processing tasks such as converting it to grayscale reducing its size, apply a 5x5 moving filter, and then storing the resulting image. A recommended approach for image input and output operations involves the use of LodePNG code. You can use any 5x5 moving filter such as gaussian blur.

Tasks:

- Read/decode the given image (image_0.png)
  Note1: We recommend using LodePNG, a standalone program to decode/encode images. To use it you just need to download and add the appropriate .c or .cpp files and the header file to your project. lodepng_decode32_file decodes the image into 32-bit RGBA raw.
  You can define the read images as unsigned char* image_0. Do not forget that every read pixel will be encoded as 32 bits - 4 chars.

image_0.png
image_1.png
lodepng.c / lodepng.cpp / lodepng.h

- Resize the images to $\frac{1}{16}$ of the original size (From 2940x2016 to 735x504). For this work it is enough to simply take pixels from every fourth row and column. You are free to use more advanced approach if you wish.

- Transform the images to greyscale images (8-bits per pixel).
  E.g. Y=0.2126R + 0.7152G + 0.0722B.
  Save the gray Scale Images as well and submit them along with the report.

- Apply a 5x5 moving filter on the gray scaled image matrix.

- Write/encode the resulting image (image_0_bw.png)
  Note2: You can use LodePNG to write the image
  lodepng_encode_file encodes the image with a specific bitdepth,
  Ensure your resulting image is in normalized to grayscale 0...255 as the output image should be in that format.

- *Profile your code: The final execution time of the implementation should be displayed as output.*

Implementation notes:

Although the implementation structure is left for the student to the side, it is recommended that it includes at least the following custom functions.

• ReadImage: C/C++. Use lodepng.cpp/lodepng.c libraries to read the input images in RGBA format

• ResizeImage: C/C++. Function to downscale the read input RGBA images by 4.

• GrayScaleImage: C/C++. Function to convert images to gray scale.

• ApplyFilter: C/C++. Function that applies a 5X5 filter to an image.

• WriteImage: C/C++. Use lodepng.cpp/lodepng.c libraries to save the output images. Remember to not comment this functionality while measuring the timing/profiling information of other functions present under this task.

• ProfilingInfo: C/C++. Provide profiling information on each of these functionalities. You are free to choose how to implement this part. Simplest profiling information is the timing information of each of these functionalities. Do observe the time, CPU consumes for each one of them. You can include any analysis on this in your report.

Exercise 3: Read/save image, convert to grayscale, reduce size, apply filter in OpenCL

This exercise is designed to perform the same tasks of Exercise 2 in OpenCL and compare the profiling results.

Tasks:

- Read/decode the given image (image_0.png) You can reuse the function from Exercise 2
- Resize the images to of the original size (From 2940x2016 to 735x504). For this work you will need to write a new OpenCL kernel
- Transform the images to greyscale images (8-bits per pixel).
  E.g. Y=0.2126R + 0.7152G + 0.0722B. For this work you will need to write a new OpenCL kernel
- Apply a 5x5 moving filter on the gray scaled image matrix. For this work you will need to write a new OpenCL kernel
- Write/encode the resulting image (image_0_bw.png) You can reuse the function from Exercise 2
- Profile your code: The final execution time of the implementation should be displayed as output.

Implementation notes:

Although the implementation structure is left for the student to the side, it is recommended that it includes at least the following custom functions. The structure of this exercise is the same as Exercise 2

• ReadImage: C/C++. Reuse from Exercise 2

• ResizeImage: OpenCL. Function to downscale the read input RGBA images by 4.

• GrayScaleImage: OpenCL. Function to convert images to gray scale. Save the Gray Scale Images as well and submit them along with the report.

• ApplyFilter: OpenCL. Function that applies a 5X5 filter to an image.

• WriteImage: C/C++. Reuse from Exercise 2

• ProfilingInfo: OpenCL. Provide profiling information on each of these functionalities.

PHASE 3: Sequential implementation of the algorithm (in C/C++)

Duration:

- *2 weeks*

Checkpoint:

- *7th of March 2025 (same as phase 2)*

Important tasks:

- *Implement a working version of the algorithm using C/C++*
- *Send the code, brief report and training diary of all previous phases of the assignment to the teacher and assistants via email*

This phase is dedicated to the implementation of the code in C/C++ in a sequential manner, using only one thread and one processor core. The code should be implemented in a way that the most of the parameters (such as e.g. the window size) can be easily changed. This implementation will be used by you throughout the exercise as the *golden implementation,* and the results of the next phases should be comparable to the ones obtained here.

Tasks:

- Implement the ZNCC algorithm with the grey scale images, start with 9x9 window size. Since the image size has been downscaled, you also need to scale the *ndisp*-value found in the calib.txt. The *ndisp* value is referred to as MAX_DISP in the pseudo-code. Use the resized grayscale images as input. You are free to choose how you process the image borders.

- Compute the disparity images for both input images in order to get two disparity maps for post-processing. After post-processing one final disparity map is sufficient in this work.

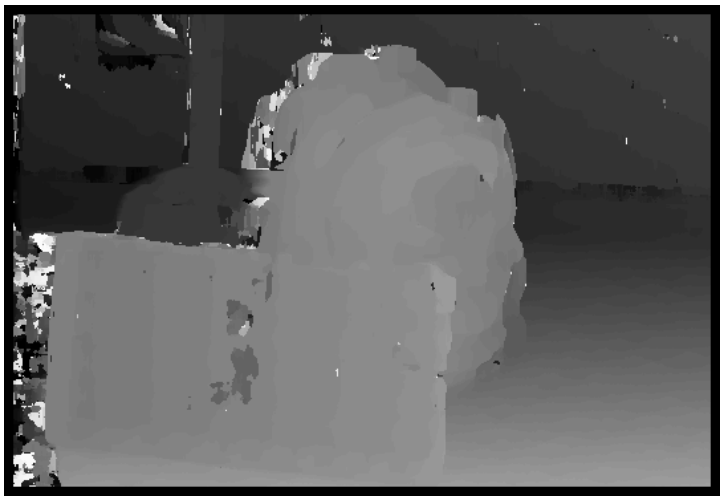
- Check that your output resembles the one presented below (Output the result image to depthmap.png Notice that you need to normalize the pixel values from 0-ndisp to 0-255. However, once you have checked that the image looks as it should, you can move the normalization process and perform it after the post-processing phase)

- Implement the post-processing including cross-check and occlusion filling (Instructions for post-processing are after the pseudo code on the last page)

- Check again that the disparity map looks right

- Measure the total execution time of your implementation including the scaling and greyscale conversion. Check the processor load. Report all these data in the email to the teacher and include it also in your final report.

- A preliminary depth map looks approximately like this:



Implementation notes:

Although the implementation structure is left for the student to the side, it is recommended that it includes at least the following custom functions.

• CalcZNCC – Write your custom C/C++ functions to compute the ZNCC value for pixels as described in the previous section. Save the output image after applying ZNCC and submit it along with the report

• CrossCheck – Write your custom C/C++ function to applying this post processing functionality. The description regarding this is mentioned under previous section. Save the output image after applying CrossCheck post processing and submit it along with the report

• OcclusionFill – Write your custom C/C++ function to apply the second post processing functionality. The description regarding this is provided under previous section as well. Save the final output image after applying OcclusionFill post processing and submit it along with the report

PHASE 4: Stereo disparity using CPU multi-threading and parallelization

Duration:

- *2 weeks*

Checkpoint:

- *11th of April 2025 (same as phase 5)*

Important tasks:

- *Implement the algorithm using CPU parallelization and multi-threading*
- *Explore alternative multithreading strategies or GPU utilization using GPUs*
- *Start the implementation of the OpenCL kernels required in the next phases*

This phase is dedicated to the implementation of the code in C/C++ in a parallel manner using more than one thread of execution and more than one processor core. Again, the code should be implemented in a way that the most of the parameters (such as e.g. the window size) can be easily changed. This implementation will be used by you throughout the exercise to understand where are the opportunities for parallelization in the proposed algorithm.

To proceed with the implementation, you can freely utilize any approach you might consider. The simplest (and still valid approach) is to utilize OpenMP pragma directives. OpenMP is an API supported by multiple vendors programming languages and operating systems that gives the programmer a very simple yet flexible interfaces for the parallelization of applications:
https://en.wikipedia.org/wiki/OpenMP

To use OpenMP, the compiler must link against the library and optionally a header file (omp.h) could be included in the code. The simplest way of using OpenMP is to include a #pragma before the sections of the code that can be parallelized (for example a for loop) and let the compiler create the threads for you. The most advanced use of OpenMP requires using some functions exposed by the library -- in other words, some rewriting -- but it is easy to get some results right away by simply *decorating* your sequential code.
You can find a small guide about how to integrate OpenMP in CodeBlocks in moodle at:

03 OpenMP_IDE_Integration.pdf

Tasks:

- For this phase, any alternative that uses more than one thread and more than one processing core is acceptable, and it is the task of the student to explore the different alternatives (pthreads, C++Threads, boost threads, even just OpenCL already). In addition to multi-threading, vectorization that uses the SIMD units of the processor (for example Intel SSE instructions) can be utilized and will speed up the computation further. For some platforms and OpenMP versions, the support of executing OpenMP code on GPU is also possible. Although this is only optional, exploring this possibility will results in extra credit.

- Measure the total execution time of your implementation including the scaling and greyscale conversion using for example QueryPerformanceCounter-function in Windows or gettimeofday-function in Linux. Check the processor load. Report all these data in the email to the teacher and include it also in your final report. A comparative analysis of the parts of

the algorithm that have been parallelized and the impact in the final computing time is also recommended.

- After you have the multi-threaded version ready, you should start right away the implementation using OpenCL, even before the checkpoint. Note that this task does not have an associated checkpoint, since it is meant to give you the chance of revisiting the first algorithm implementations while already coding the next phases of the exercise. Be mindful of utilizing all the time checkpoint for just multithreading C/C++, as it will leave you relatively short time for the OpenCL implementation.

PHASE 5: Stereo Disparity OpenCL implementation for GPU

Duration:

- *2 weeks*

Checkpoint:

- *11th of April 2025 (same as phase 4)*

Important tasks:

- *Implement the algorithm using OpenCL and a GPU*
- *Send the code and training diary of all previous phases of the assignment to the teacher and assistants via email*
- *Start the optimization of the OpenCL kernels required in the next phase*

Recommended Lecture:

- *Chapter 2,3 of OpenCL in Action: How to Accelerate Graphics and Computations, (Matthew Scarpino)*

This phase is dedicated to the implementation of the code in OpenCL in a way that can run on a GPU. If a GPU installation of OpenCL cannot be available, the use of CPU could be allowed. Consult with the teacher or assistants. The code should be implemented in a way that the most of the parameters (such as e.g. the window size) can be easily changed. This implementation will be used by you as a reference to measure the performance of your optimizations, and the results and computation times of the next phases should be compared to ones obtained here.

Tasks:

- Read the original images into host (CPU) memory (im0.png and im1.png). Use the ready C-implementation as the starting point for you host-code, but do not overwrite the C-implementation.
    - You are free to use existing host-codes from SDK-example projects or from the internet freely or ask for assistance to AI and LLM tools (chatGPT, Github co-pilot). Just cite where your implementation is from. Note that you still need to **understand** what you are doing
        - E.g. there is a vast number of examples available online

- Find out at least the following parameters of your GPU using the *clDeviceInfo*-functions and report them in your final report

    - CL_DEVICE_LOCAL_MEM_TYPE
    - CL_DEVICE_LOCAL_MEM_SIZE
    - CL_DEVICE_MAX_COMPUTE_UNITS
    - CL_DEVICE_MAX_CLOCK_FREQUENCY
    - CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE
    - CL_DEVICE_MAX_WORK_GROUP_SIZE
    - CL_DEVICE_MAX_WORK_ITEM_SIZES

- Read the images (original .png-files) into the device as **image-objects**.

- Implement a kernel or kernels for resizing and grey scaling the images

- After resizing and grey scaling, using buffer-objects can be more convenient. NOTE! You only need to read the buffers/images back to the host, if you plan to use them on the host or another device, otherwise keep them on the device in order to avoid unnecessary and costly data transfers.

- Implement the actual ZNCC algorithm. Once the algorithm works proceed to implementing the post processing. You are free to divide the work into multiple kernels. Whatever in your opinion is the best approach but justify the chosen approach in the final report.
    - Notice that transferring data between host and device is costly, so minimize data transfers

- Benchmark the execution times of each kernel using *clGetEventProfilingInfo* and the total execution time using the appropriate C-functions on the host-side and report them in your final report.

PHASE 6: OpenCL optimization

Duration:

- *2 weeks*

Checkpoint:

- *12th of May (tentative, might be subjected to change)*

Important tasks:

- *Optimize the OpenCL implementation to exploit the GPU architecture*
- *Write the final report*
- *Send the complete code of all assigments, the final report containing all tasks and the training diary to the teacher and assistants via email*
- *Agree with the teacher on a time for the exit interview and grade assessment*

Recommended Lecture:

- *Chapter 4,5 of OpenCL in Action: How to Accelerate Graphics and Computations, (Matthew Scarpino)*

Tasks:

This phase is dedicated to the optimization of the OpenCL code in a way that is able to exploit the architectural features of the GPU. This implementation is the final one that would be mainly used for the evaluation of the course and grades. In addition, this phase includes the writting of the final report and the documentation of the code and coding process.

To proceed with the optimization of the code, you can follow the next steps:

- Make a copy of the implementation before proceeding to the optimization phase! Use this implementation as the basis for optimization for both desktop (and optionally mobile) implementations. Optimize your code. All vendors have their own optimization guides with good examples. Your optimizations should take into account some of the following strategies:
  - Vectorization
  - Full memory architecture utilization (local memory)
  - Memory coalescing
  - Reducing the register usage
  - …
- Report the execution times and the CPU processor load of the optimized code in your final report
- Compare the execution times and the processor load of your final implementation to the ones obtained in the previous phases of the exercise
  - Write a reasoning about the different performance gains and their causes
  - Propose other strategies that could be made for further optimizations

PHASE 7: version 1 (*Optional*): Efficient use of memory hierarchy

Duration:

- *2 weeks*

Checkpoint:

- *26th of May 2025 (after exit interview)*

Important tasks:

- *Test and optimize the OpenCL implementation to use efficiently the multilevel memory hierarchy exposed in the framework (local, private global memory)*
- *Benchmark the memory transfers and possible overhead*

Tasks:

This is the recommended path. This optional phase is dedicated to the efficient utilization of the exposed memory hierarchy by OpenCL. This extra implementation will grant you at least one extra point in the final grade.

To proceed with the implementation, you can follow the next steps:

- Contact the teacher or assistants in order to inform them that this is the path you have chosen for the phase 6.
- Familiarize yourself with the memory hierarchy of your particular device and how it is implemented in OpenCL
- Make optimizations by efficiently use the memory architecture, including tiling in local memory
- Test your code report the execution times.
- Report the results after the optimization and compare with the non-optimized code.

PHASE 7: version 2 (*Optional*): OpenCL on mobile devices

Duration:

- *2 to 4 weeks (consult with the assistant if this option is available)*

Checkpoint:

- *26th of May 2025 (after exit interview)*

Important tasks:

- *Borrow and ORDROID platform from the teacher/assistants*
- *Port, test and optimize the OpenCL implementation for the mobile environment*

Tasks:

This is an alternative path. This optional phase is dedicated to the utilization of OpenCL in mobile environments. This extra implementation will grant you at least one extra point in the final grade.

To proceed with the mobile implementation, you can follow the next steps:

- Contact the teacher or assistants in order to borrow a mobile platform from them (ORDROID). The ORDROID board has a working environment installed. If you plan to use a different platform that you can provide (e.g. a compatible mobile phone), inform the teacher to plan accordingly.
- Familiarize yourself with the particularities of the mobile GPUs.
- Test your code in the mobile environment and report the execution times.
- Make optimizations taking into account the architecture of the mobile platform.
- Report the results after the optimization and compare with the non-optimized code.

FINAL REPORT, TRAINING DIARY, GRADING, DEADLINES, HELP

The registration of the course is done as follows:

- Register yourself in Peppi in order to get course emails and credit points.

- Find yourself a partner to create a group of two students
    - Individual groups of one person are allowed, but the workload remains the same
    - If you are unable to find a partner, contact the teacher and we will do our best to find you a suitable one.

- Register the group by sending the group members name by email to the assistant. This will in practice be the requirement of checkpoint 0 and will "officially" start your work on the exercise.

As a part of the exercise, you will write a training diary:

- After each checkpoint, send the training diary to the course assistant
- The training diary should at least include
    - Number of hours done by every student, including self-studying hours
    - Short description of tasks done by hourly basis

The grading of the course is based on self-assessment. When the exercise is completed, you will be invited for an **Exit Interview** and receive 5 ECTS:

- There are no exams:
    - Each returned exercise will be either accepted or rejected
    - When accepted, a grade 1-5 will be given depending on the overall quality of the work
    - The grade will be based on the mutual agreement and understanding of several factors, including documentation, reasoning, optimization and elegance of solutions
    - The final grades are negotiable during the Exit interview
    - The grades can be the same for both or different for each group member in case the workload has been different during the coding and documentation process
    - If you are aiming for a higher grade than the one decided, a suggestion on how to achieve the desired level will be given

The grades will roughly follow this scale:

- **GRADE 1:** Working implementation **COMPULSORY!!!**
    - *C/C++ implementation and benchmark*
    - *OpenCL-implementation and benchmarks for GPU (or CPU if not available)*
    - *Training diary and final report*
- **GRADE 2**: Complete documentation, comments, code:
    - Design, comments, structure, detailed time management
- **GRADES 3 & 4**: Optimization of your implementation(s)
    - Vectorization, utilizing the full memory architecture, …
    - Good Profiling
- **GRADE 5**: Mobile implementation or memory-optimized desktop implementation
    - Comparisons with basic desktop implementation

In addition to the grades, there are two modifiers:

- You will get one point less if you miss **more than one** deadline (Checkpoints).
- You will get one extra point if you use **any** inventive solution, or you **exhaustively explore** some of the possibilities of one of the phases, e.g.
    - Vectorization with SIMD during the multicore coding
    - GPU execution using OpenMP
    - Complete profiling of the timing from each part of the algorithms
    - Exhaustive testing of optimization techniques with analysis of the impact
    - Evaluation on multiple platforms (e.g. ARM, or GPUs from different vendors)
    - Evaluation on multiple operating systems (Windows and Linux)
    - …

The tentative deadlines for the course are as follows:

January 22th, 2025:
Initial lecture followed by a period of self-study and framework installing

February 7th, 2025:
Groups formed and registered, and exercise instructions published

March 7th, 2025:
1st checkpoint: C-implementation in single thread + basics of OpenCL

April 11th, 2025:
2nd checkpoint: Multithreaded C and OpenCL implementation with profiling

May 12th, 2025:
3rd checkpoint: Fully optimized OpenCL code, report.

May 19th, 2025:
Last possible day to return the exercises:
Further work could only be extended with explicit permission from the teacher, and recommendations on what is needed to pass the course or improve the grade.

May 26th, 2025:
Last possible day to submit the exercise and book the exit interview. After this date, help and support is not guaranteed and it might not be possible to finish the course in a timely manner.

Exactly a week before the three checkpoints, we will hold a **non-compulsory** "group" session in a classroom to discuss problems and strategies face to face. You are free to meet the teacher(s) there and share with them and other students your questions, suggestions, strategies or ideas.

In case of doubt, remember that we are here to help. Do not hesitate to contact us. We will always do our best to point you to the right direction. To contact us:

- Try to find a solution online. Someone has probably faced the same problems. Check the course webpages (Moodle) and look for the F.A.Q. (if/when available).
- If it does not help, email us with your problem
    - Always attach the code, results and possible error messages
    - Specify the problem as much as possible, so it can be reproduced
- We will answer with debugging strategies and suggestions of tests:
    - You **SHOULD** try these tests that are suggested
    - If everything else fails, we will set up a time to meet you

APRENDIX 1: PERSONAL EVALUATION FORM

Student name:

Working implementations

1. Have you implemented a working C-implementation and benchmark? (1 point)

- Yes (+1)
- No (0)

2. Have you implemented an OpenCL-implementation and benchmarks for GPU? (1 point)

- Yes (+1)
- No (0)

3. Do your output images look of good quality ”always” with no undesired data in the image?
- Yes (0)
- No (-1 or -2 points)

Reporting

4. Have you maintained a training diary and completed a final report? (1 point)

- Yes (+1)
- No (0)

5. Have you provided complete documentation, comments, and code? (1 point)

- Yes (+1)
- No (0)

6. Have you provided detailed time management records? (1 point)

- Yes (+1)
- No (0)

Code documentation

7. Have you included detailed comments in your code that explain the purpose and functionality of each segment of code? (1 point)

- Yes (+1)
- No (0)

8. Have you structured your code in a clear and organized manner? (1 point)

- Yes (+1)
- No (0)

Optimization

9. Have you implemented multi-threading techniques to optimize the performance of your implementation(s)? (1 point)

- Yes (+1)
- No (0)

10. Have you optimized your implementation(s) through vectorization, utilizing different types of memory, and use of native functions? (1 to 3 points)

- Yes (+1 to +3 points)
- No (0)

11. Have you utilized OpenCL-specific features to optimize your implementation for the GPU such as image objects instead of buffers or fast mathematics ? (1 point)

- Yes (+1)
- No (0)

12. Have you made use of the different parts of the device (GPU) memory architecture such as private and local memory, taking advantage of, e.g local groups? (1 to 3 points)

- Yes (What?, +1-3)
- No (0)

Grade modifiers

13. Have you conducted good profiling and performed exhaustive benchmarking of different alternatives? (1 point)

- Yes (+1)
- No (0)

14. Have you developed inventive or exhaustive solutions beyond the requirements of the project? (+1 point)

- Yes (+2)
- No (0)

15. Have you met all project deadlines? (-1 point for missing deadlines)

- Yes (0)
- No (-3)

14. What point value do you propose for yourself based on the criteria above? (Sum the points)

- 1-4 (Grade 1)
- 5-7 (Grade 2)
- 8-10 (Grade 3)
- 11-12 (Grade 4)
- 13+ (Grade 5)

15. Would you like to negotiate your proposed Grade?

- Yes
- No

16. If yes, what complementary experiments or improvements are you willing to undertake to raise your point value?

- We discuss this in person 😊

If you had a partner in the group:

17. What grade would you give your partner: 1-5 ?