Assignment 5

Task: Stereo disparity pthread and OpenCL implementation

There are two tasks required to be done for this assignment

Task description

Task-1:

The algorithm for "Depth estimation based on Zero-mean Normalized Cross Correlation (**ZNCC**)" should be implemented in C/C++ using **CPU-multi threads**.

Task-2:

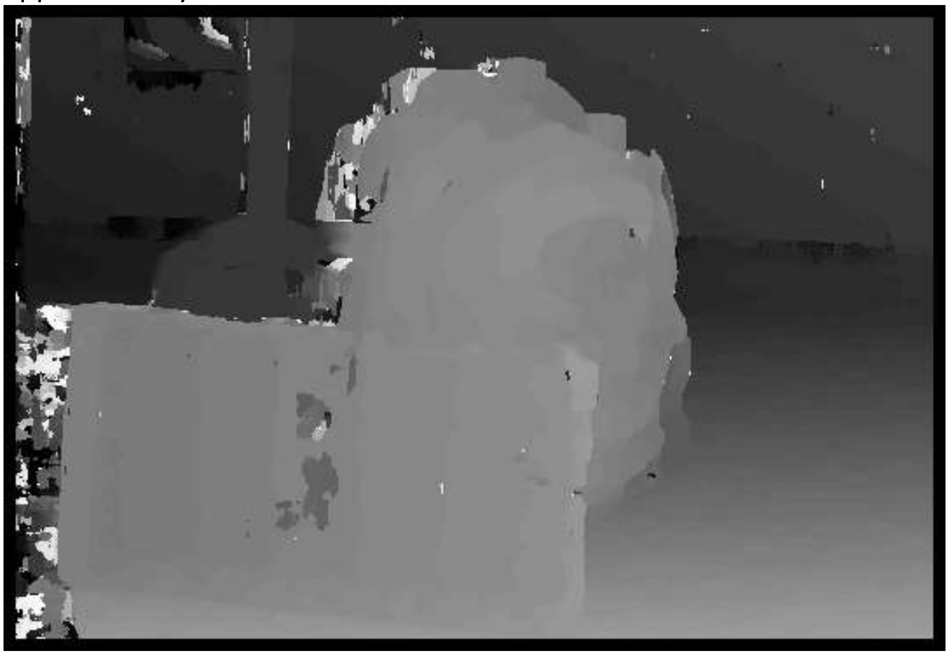
The algorithm for "Depth estimation based on Zero-mean Normalized Cross Correlation (**ZNCC**)" should be implemented using **OpenCL kernel functions**.

Stereo Disparity description and Implementation:

Regarding the stereo disparity algorithm description and custom functions required to be implemented, follow the instructions provided in assignment 4. Two input images required for the task will be provided along with the assignment documentation. Besides the algorithm description, you can follow these steps one after another for the stereo disparity implementation:

- **ReadImage** – Use lodepng.cpp/lodepng.c, lodepng.h libraries to read/decode the input images in RGBA format. Use the **lodepng_decode32_file-function** to decode the images into 32-bit RGBA raw images and **lodepng_encode_file-function** to encode the output image(s). Notice that you need to normalize the results to grayscale 0..255 as the output image should be in that format.
- **ResizeImage** – Resize the images to 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. **Write** your custom C/C++ function to perform these operations.
- **GrayScaleImage** – Transform the images to grey scale images (8-bits per pixel). E.g. Y=0.2126R+0.7152G+0.0722B. **Write** your custom C/C++ function to apply these operations. **Save** the grey scale Images as well and submit them along with the report
- **CalcZNCC -** 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 of the original images. The ndisp value is referred to as MAX_DISP in the pseudo-code. ndisp=260 for im0.png, im1.png shared under the assignment. As ZNCC is applied after resizing the images, MAX_DISP in the pseudo code refers to **ndisp/4**. Use the resized grayscale images as input. You are free to choose how you process the image borders. **Write** your custom C/C++ function that performs these operations and apply ZNCC value for pixels and compute the disparity map. Detailed description for CalcZNCC is available under assignment 4. Remember to compute the disparity images for both input images in order to get two disparity maps for post-processing. **Save** the output disparity images after applying ZNCC for each input images and submit it along

with the report. A preliminary depth/disparity map before post processing looks approximately like this:



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.

- **CrossCheck** – **Write** your custom C/C++ function to applying this post processing functionality. The description regarding this is available in assignment 4. Check again if the disparity map looks right after implementing this post processing functionality. **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. Check again if the disparity map looks right after implementing this post processing functionality. **Save** the final output image after applying OcclusionFill post processing and submit it along with the report

- **WriteImage** – Use lodepng.cpp/lodepng.c libraries to save the output images at different times of the implementation. **Remember** to comment this functionalty while measuring the timing/profiling information of other functions present under this task

- **ProfilingInfo** – Provide profiling information on each of these functionalities. You are free to choose 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. 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. You can stick to profiling tools from your previous assignment submissions as well). Check the processor load. Include such information under the report.

Task Implementation:

Task-1: Stereo disparity pthread implementation

This task 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.Try implementing the code 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 in future assignments to understand where are the opportunities for parallelization in the proposed algorithm.

Run the C/C++ code on multiple CPU cores using threads. You can use pthread library (for C) or thread library (for C++) implementation. Follow this link, for detailed explanation on POSIX/pthreads
https://computing.llnl.gov/tutorials/pthreads/#:~:text=Pthreads%20are%20defined%20as%20a,as%20libc%2C%20in%20some%20implementations.

The main idea behind this task is to parallelize the C/C++ code using multi-core CPU and threads and realize their execution times.

You must have completed C/C++ sequential implementation of stereo disparity implementation in assignment 4. The task here is to add this thread functionality to the same C/C++ implementation and realize if the execution times has got any better. Remember to save your sequential C/C++ implementation as well sepearately before adding this thread funcitonality.

Task-2:

This task is dedicated to the implementation of the code in OpenCL in a way that can run on a GPU. Try implementing the code 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 future assignments should be compared to ones obtained here

Under this task, all the custom functions in C/C++ stereo disparity implementation should be replaced with OpenCL kernel functions. Read and Write Images functions as well can be implemented using OpenCL built-in kernel functions such as read_imageui, write_imageui. You can now refer back to your previous assignments for these kernel implementations.

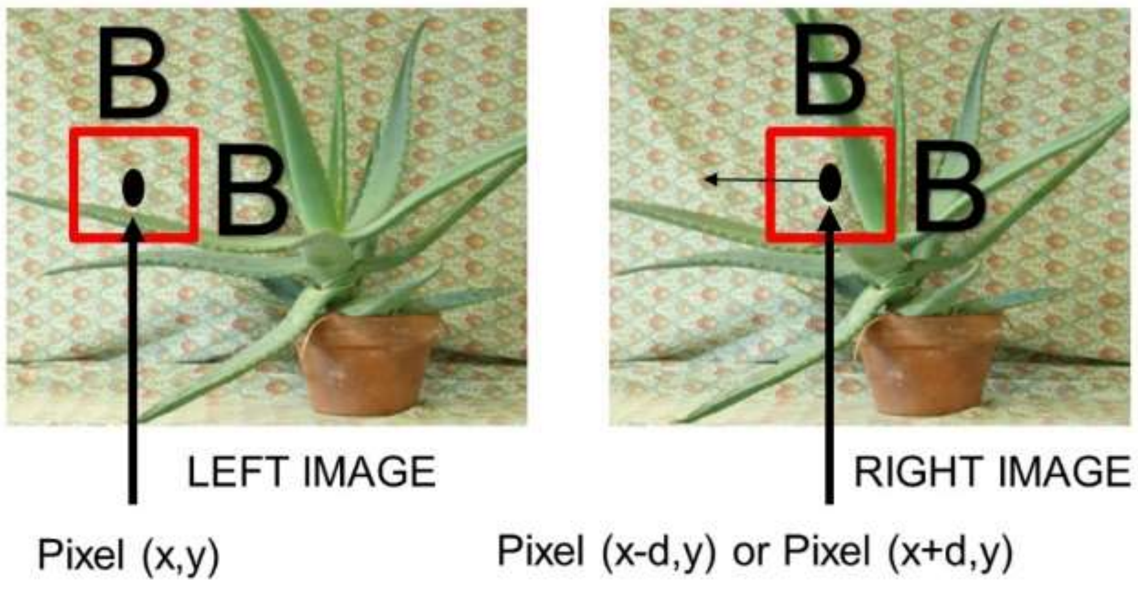
Some more explanation about the implementation steps are also included below:

- 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 from the internet freely or your previous assignments. Just cite where your implementation is from. **Note** that you still need to understand what you are doing
- Find out at least the following parameters of your GPU using the clDeviceInfo-functions and report them in your final report. All these can be the debug statements at the start of your **ProfilingInfo** as well

    - 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.
- **Save** the output images in a similar manner to C/C++ implementation and include them under one folder

Pseudo Code:

A pseudo code is also provided here for reference



ZNCC VALUE = UPPER SUM / (SQRT (LOWER SUM_0) * SQRT(LOWER SUM_1))

UPPER SUM += left_pixel_val_diff_from_avg * right_pixel_val_diff_from_avg

LOWER SUM_0 += left_pixel_ val_diff_from_avg * left_pixel_ val_diff_from_avg

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
```

Note: The previously depicted description of the task, algorithmic implementation steps 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 best) solution. You are free to experiment with other approaches.

Expected result:

A working version of the implementation, a brief report (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 for pthread and OpenCL implementations and the final screenshot of your outputs asked to be displayed under the assignment.