University of Oulu

521288S
Multiprocesor Programming

- PROJECT -

Berke Esmer

Mohamed Bekoucha

February 23rd, 2019

Part A : Self Study and Platform Installation

In the beginning of the course, we were required to find a partner and study the OpenCL platform along with the course necessities. The course was about doing a project which will use GPU and CPU together to get the most benefit out of a computer computing power.

Since we were both international students, we had hard time to find a partner. So, the teacher offered us to become a team and we accepted his polite offer. Afterwards, we immediately started working on the project.

In order to better understand the algorithm and everything that comes with it, we decided to follow an approach that would make us finish the task quickly and of course with better results. We started by each working on the algorithm alone and then merged the two codes together into one working project. In this way, we made sure that each one of us completely understood the algorithm and was ready to move on to the other tasks.

The code was implemented in C++, which we were familiar with before but had to learn further new things to finish the linear and parallel implementation of the algorithm. The structures we learned along the way are namely vectors, LodePNG library and OpenMP.

Also, both of us did not have any experience with the Visual Studio. So, we also learned this powerful IDE by studying its features and Git connection.

Part B: Algorithm Implementation in Sequential Single Thread

By following the given pseudo-code, we were able to implement the algorithm which led to some different results. Although it was more of a try and compare approach what we used in processing as we had a reference image in the project description, we had to add few more variations to the given parameters to get the required output. Moreover, in order to get a sense of what sort of output we need to expect each time, python scripts were used to generate the exact same output and compare the results. Python was chosen due to its simplicity in testing.

When analyzing the output images during the first trials, we found that it was different from the image provided in the project description. Since we had some parameters that we could manipulate to change the final output, we tested with few samples and tabulated the results as it can be seen later in the table.

Below table is a list of execution time with different used parameters and the produced output images (two disparity maps and the final images after occlusion filling) are in an attached zip.

Window size	Disparity difference	execution time (sec)
5	31	7
5	60	7
9	31	15
9	60	15
15	31	37
15	60	37
25	31	96
25	60	97
35	31	208
35	60	187
45	31	334
45	60	303
100	60	1709

Table 1: Execution times with different disparity difference and window size parameters

The output was generated in three ways, first by occlusion filling using nearest non-zero neighbor on X axis. The output as it can be seen in the output files had some shift in the X axis and this was not

fixed even after using an average filter. Then the second one was produces using Y axis, which had some similar behavior as in the first one in which it had shift in the Y axis. Finally merging the last two outputs to get a less shifted image, which gave some good results.

The output was getting smoother as we increased the value of window size, and that also had an effect on the execution time. Although smoother outputs seemed to be closer to the desired output, the smoothing effect reduced some details from the images and hence increasing the window size further would have been meaningless as it would not do the task. Furthermore changing the difference value reduced the black areas on the image and gave also smoother output. It also affected the execution time as in the case of large window size, it became less detailed since there were less operations and with the large number of loops it became significant unlike the previous cases.

The produced outcomes written on the first table can be found in the attached zip file. We wanted to document all of them considering it could be helpful in the further stages of the project. Afterwards, we decided to start in the next phase immediately.

Part C: Algorithm Implementation in Multi-Threading Environment

In this part, we decided to use the threads library in C++ as it provides functions to control their flow. The parallelization was done in a way that we could unroll the loops we had in the linear code, and for that we had a few use cases with varying number of threads.

In order to avoid shared data complications and locks which can happen while using threads strategy, we used direct operations. Such an example can be,

vector1 = (6, 9, 8) and vector2 = (4, 2, -3) => Result = (10, 11, 5)

(Result = V1 + V2)

In the beginning, we have 3 threads where every thread has read/write access only to elements of V1[i] ,V2[i] and Result[i]. By using vectors in C++, we only pass pointer to the element and hence all threads can run concurrently while not affecting each other outputs.

All the tried use cases and their statistical outputs are tabulated in the below table including run time, CPU usage and memory usage.

It is important for us to mention that parallelism was applied for the ZNCC algorithm only. It was not applied on neither on the prior processing nor post processing phases. The reason for that is because the ZNCC algorithm takes nearly over 90% of the execution time, and hence parallelizing the rest of the code could be negligible compared to parallelism in the ZNCC algorithm part.

Unrolled/partially unrolled loops	Threads	Execution time (sec)	CPU usage (I7 5500U)	RAM usage
Height and Width	735*504	133.195	90%~95%	58MB
Height	504	9.5	83%~92%	29MB
Partial: Height/2	252	9.7	83%~92%	30MB
Partial: Height/4	126	9.8	83%~92%	30MB
Partial: Height/8	63	9.7	83%~92%	30MB
Partial: Height/16	31	9.7	83%~92%	30MB
Partial: Height/31	16	9.6	83%~92%	30MB
Partial: Height/63	8	9.5	83%~92%	30MB
Partial: Height/126	4	9.6	83%~92%	30MB
Partial: Height/256	2	9.6	83%~92%	30MB
Width	735		63%~72%	29MB
None	1	17	31%~34%	28MB

Table 2: Execution times with different numbers of thread used in calculating the ZNCC part

In order to explain the loops structure, in the code, we have the following hierarchy:

```
HEIGHT: for y in range (0, HEIGHT)
        WIDTH: for x in range (0, WIDTH)
                rest of is the ZNCC algorithm
```

For the sake of comparison, we also have used OpenMP in order to parallelize the loops and tabulated the results below.

unrolled loops	Threads	Execution time (sec)	CPU usage (I7 5500U)	RAM usage
Height	4	9.07	90%~95%	34MB
Width	4	11.00	83%~89%	34MB
Height and Width	4	8.64	84%~96%	34MB

Table 3: Execution times while dividing different workload in processing ZNCC

The loop unrolling method used in OpenMP approach created only 4 threads which is the same number as the logical threads in one of our computers. We did not unroll all the loops as it was just to get familiar with OpenMP functionality.

What we could conclude from these two results is that when there is a large number of threads , the CPU usage would increase, however it will not always give quicker execution time. Sometimes having huge number of threads like in the case of the full unrolling Width*Height led to very slow execution time.