ZNCC Algorithm Based Stereo Disparity Computation Using OpenCL

Abu Taher
Dept. of Computer Science and Engineering
University of Oulu
Oulu, Finland
abutaher.kuet.ece@gmail.com

Abstract: In this report, we describe the implementation details of ZNCC-based disparity map computation using CPU and GPU. For the CPU-based implementation, I implement both the single-thread and multithread-based setups. As expected, multithread-based implementation performed significantly better with an average execution time of 7.546 sec compared to 30.083 seconds for the single-threaded version, resulting in a 3.9× (74.92%) improvement. For the GPU-based implementation, both optimized and unoptimized versions were developed. The unoptimized ZNCC computation had an average execution time of 117.665 ms, while the optimized version reduced this to 54.758 ms—achieving a 2.14× (53.45%) speedup. I implemented the left and right disparity computation using two separate functions. I implemented the left and right disparity computations as two separate functions. To ensure a fair comparison, each function was executed five times, and the average was taken. The overall execution time for the ZNCC algorithm was calculated as the average of these two function averages.

I. INTRODUCTION

Stereo vision is the process of getting the depth information from two images taken from slightly different viewpoints, for example human eyes. Depth map is calculated by using disparity map. It is nothing but the calculation of relative shift of corresponding points between a pair of images. It has many applications such as 3D reconstruction, depth estimation and object recognition [1][2].

In binocular disparity two cameras view the scene from slightly different lateral positions, the same scene appears at different location in the left and right images [3]. The magnitude of the disparity value has an inverse relationship with the point's distance from the camera. Closer objects exhibit larger disparities, while farther objects have smaller disparities [3].

Disparity computation is also known as stereo correspondence search. There are many approaches for stereo correspondence such as local area-based methods to global optimization techniques. In this project, I focused on local area-based methods using Zero-mean Normalization Cross Correlation as a similar measure for matching [4]. ZNCC computes the similarity between two image patches. Because of normalization it is robust against noise and illumination changes [4]. ZNCC has been used as a state-of-the-art stereo matching task.

Prior work by Aleksei and Iaroslav demonstrated that an OpenCL based GPU implementation can run around two orders of magnitude faster than CPU version [5]. However, I was successful in improving this execution time to only 54.758 milliseconds.

II. ZNCC ALGORITHM

ZNCC means Zero-mean Normalized Cross Correlation. Left and right images are taken using two different cameras which are situated on the same X axis but slightly apart from each other. When these stereo cameras are used to take pictures of a scenario the images might look same, but pixels position would be slightly different. For example, a pixel at (6,4) in the left image could be situated at (8, 4) position in the right image. In this case the horizontal displacement of the left pixel would be 2 which is the disparity of the pixel at (6,2) position.

If we only compare pixels, it would be heard to recognize similarities because a slightly different pixel value could represent the same thing. Therefore, we take a patch (window) around the target pixel from the left image and compare it to the corresponding patch from the right image. To constrain the search area MAX_DISP = 65 was defined. This means a patch from the left pixel will be compared with a maximum of 65 leftward pixels from the right image. Figure 1 depicts how this computation is performed. Figure 2 shows actual implementation of final disparity map calculations.

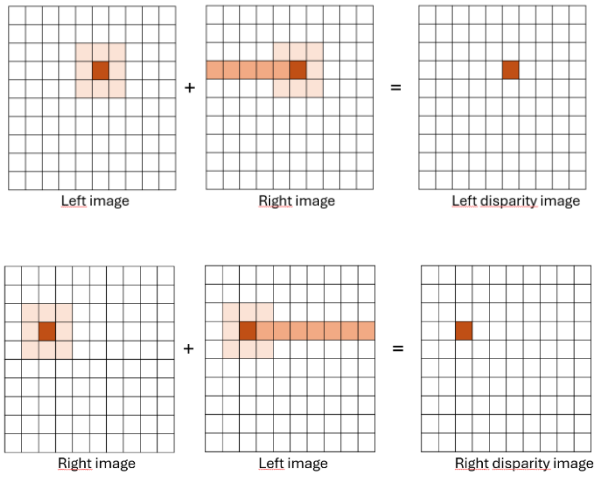

Figure 1: (a) and (b) shows how left and right disparity is computed.

To understand the algorithm ZNCC equation and the pseudocode are given below. From the below pseudo code we can see that there are multiple '*for*' loops that could be parallelized to improve the computation time. This is perfect for studying and understanding parallel processing using GPU.

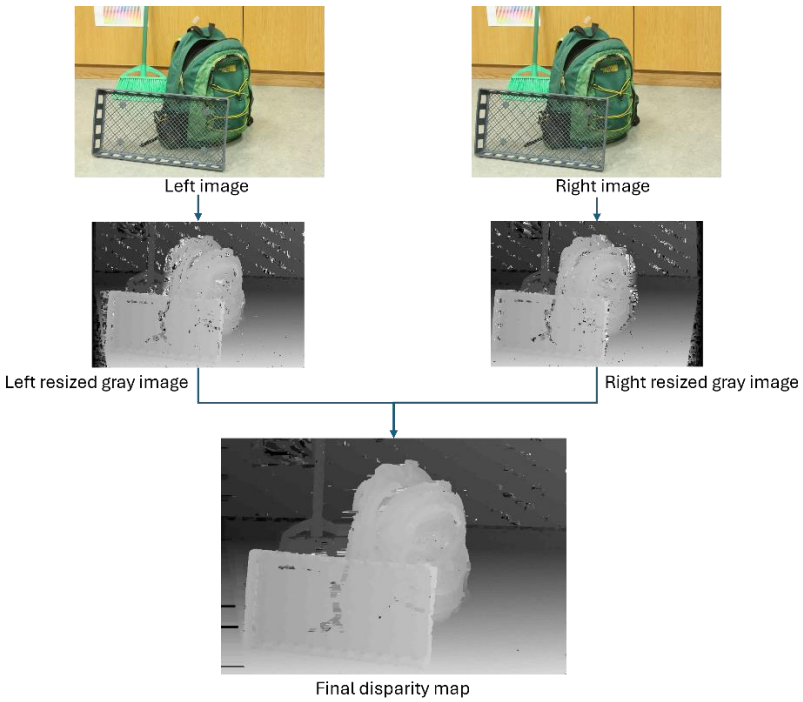

Figure 2: Intermediate images during disparity map calculations.

$$\mathrm{ZNCC}(x,y,d) = \frac{\sum_{i=-k}^{k}\sum_{j=-k}^{k}\left(I_L(x+i,y+j)-\bar{I}_L\right)\left(I_R(x+i-d,y+j)-\bar{I}_R\right)}{\sqrt{\sum_{i=-k}^{k}\sum_{j=-k}^{k}\left(I_L(x+i,y+j)-\bar{I}_L\right)^2 \cdot \sum_{i=-k}^{k}\sum_{j=-k}^{k}\left(I_R(x+i-d,y+j)-\bar{I}_R\right)^2}}$$

Symbols:

I_L = Left image
I_R = Right image
$\bar{I}_L$ = mean of a patch from left image
$\bar{I}_R$ = mean of a patch from right image
d = disparity

Pseudo Code:

For each pixel coordinate (x, y) in the image:
  Initialize max_correlation to a very small number
  Initialize best_disparity to zero

  For disparity d from 0 to MAX_DISPARITY:
    Define left_window as the patch centered at (x, y) in the left image
    Define right_window as the patch centered at (x - d, y) in the right image

```
        Calculate mean_left as the average pixel value of left_window
        Calculate mean_right as the average pixel value of right_window

        Initialize numerator to 0
        Initialize sum_left_diff_squared to 0
        Initialize sum_right_diff_squared to 0

        For each pixel position (i, j) inside the window:
            left_diff = left_window[i, j] - mean_left
            right_diff = right_window[i, j] - mean_right

            numerator += left_diff * right_diff
            sum_left_diff_squared += left_diff * left_diff
            sum_right_diff_squared += right_diff * right_diff

        denominator = sqrt(sum_left_diff_squared) * sqrt(sum_right_diff_squared)

        If denominator is not zero:
            zncc_value = numerator / denominator
        Else:
            zncc_value = 0  // or handle divide-by-zero case appropriately

        If zncc_value > max_correlation:
            max_correlation = zncc_value
            best_disparity = d

    Set disparity_map[x, y] = best_disparity
```

III. ENVIRONMENT SET UP

All the work was conducted on the windows machine with Intel® Core™ i5-8250U @ 1.60GHz processor. Firstly, I installed VS code and added C/C++ extension. Then I installed MinGW [9] and CUDA driver [7]. I downloaded 'loadpng' from [8] and added it to each subfolder. My working directory looks like figure 3. Inside the main working directory, a new directory name '.vscode' was created. Inside this folder a build task file 'task.json' was created. This file tells compiler where to look for the additional library files such as OpenCL library file. From figure 2 you can see each phase has its distinct directory. To use the 'loadpng' for image decoding and encoding I put the file in the corresponding directory. Further inspection of the directory would reveal the fact.

IV. BENCHMARK

A. Phase 1

After successfully setting up the environment, a simple OpneCL 'hello-world' program was built and executed to test if everything was working. Fortunately, everything was set up properly.

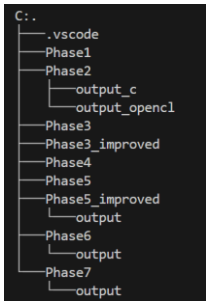

Figure 3: Project folder structure.

B. Phase 2

This phase was divided into three exercises. In exercise 1 I implemented add_matrix() function in C and OpenCL. In exercise 2 and 3 five functions: load_image(), resiz_image(), convert_rgba_to_gray(), save_image(), and moving_average_filter_float() were implemented both in C and OpenCL respectively. Figure 4 shows the platform and device information. For this project, NVIDIA CUDA platform and NVIDIA GeForce MX130 GPU were used. Execution time for each function is given in Table I. From the table we can notice that execution time in C is measured in seconds whereas execution time in OpenCL measured in milliseconds.

```
Platform 0: NVIDIA CUDA
=== Platform 0 ===
Name: NVIDIA CUDA
Vendor: NVIDIA Corporation
Version: OpenCL 3.0 CUDA 12.6.65
  Device 0: NVIDIA GeForce MX130
    CL_DEVICE_LOCAL_MEM_TYPE: CL_LOCAL
    CL_DEVICE_LOCAL_MEM_SIZE: 48 KB
    CL_DEVICE_MAX_COMPUTE_UNITS: 3
    CL_DEVICE_MAX_CLOCK_FREQUENCY: 1189 MHz
    CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE: 64 KB
    CL_DEVICE_MAX_WORK_GROUP_SIZE: 1024
    CL_DEVICE_MAX_WORK_ITEM_SIZES: 1024, 1024, 64
Platform 1: Intel(R) OpenCL HD Graphics
=== Platform 1 ===
Name: Intel(R) OpenCL HD Graphics
Vendor: Intel(R) Corporation
Version: OpenCL 3.0
  Device 0: Intel(R) UHD Graphics 620
    CL_DEVICE_LOCAL_MEM_TYPE: CL_LOCAL
    CL_DEVICE_LOCAL_MEM_SIZE: 64 KB
    CL_DEVICE_MAX_COMPUTE_UNITS: 24
    CL_DEVICE_MAX_CLOCK_FREQUENCY: 1100 MHz
    CL_DEVICE_MAX_CONSTANT_BUFFER_SIZE: 1653690 KB
    CL_DEVICE_MAX_WORK_GROUP_SIZE: 256
    CL_DEVICE_MAX_WORK_ITEM_SIZES: 256, 256, 256
```

Figure 4: Platform and device information

TABLE I
Execution time for each function in C and OpenCL

Function name	**Execution time in C**	**Execution time in OpenCL**
add_matrix()	0.037 ms	0.010 ms
load_image()	0.612 s	0.612 s
resize_image()	0.000 s	0.233 ms
convert_rgba_to_gray()	0.000 s	0.075 ms
save_image()	0.095 s	0.095 s
moving_average_filter_float()	0.078 s	0.901 ms

C. Phase 3

I implemented a working version of the ZNCC based disparity computation algorithm in C using single core single thread. When I first started working on this phase I defined a single compute_disparity_map() function to compute the left and right disparity. It correctly calculates the left disparity image, but it couldn't correctly calculate right disparity. After careful analysis I found that the pixel comparison was wrong.

When we compute left disparity, a pixel from left image is compared with leftward pixel in the right image. However, when calculating the right disparity we should do the opposite. Therefore, I used two different functions, compute_zncc_left_to_right() and compute_zncc_right_to_left() to compute left and right disparity maps. I used MAX_DISP 65 and window size 9x9 throughout the project and focused more on optimizing the main ZNCC computation functions.

For cross_check() function, I used 'Threshold = 2'. For occlusion_fill() function I used horizontal and vertical filling. To get the fair execution time, the code was executed five times. Table II provides average time for each operation. Figure 5 shows execution time in each iteration. From the table it is evident that the most time-consuming operations in both single-threaded and multi-threaded CPU implementations were the *Compute left disparity* and *Compute right disparity* functions. In the single-threaded setup, these took 29.178 and 30.989 seconds respectively (average = 30.083 sec), while in the multi-threaded setup, execution times dropped significantly to 7.462 and 7.630 seconds (average = 7.546) —demonstrating a clear performance improvement with parallel processing.

TABLE II
Comparison of execution time for single and multi-thread execution.

Functions	**Execution time (sec) (Phase 3 – single thread CPU)**	**Execution time (Phase 4 – multi-thread CPU)**
Load left image	0.610	0.615
Resize left	0.003	0.003
Convert left to gray	0.003	0.000
Save left gray	0.093	0.091
Load right image	0.613	0.620

Resize right image	0.009	0.000
Convert right to gray	0.003	0.000
Save right gray	0.085	0.102
Compute left disparity	29.178	7.462
Compute right disparity	30.989	7.630
Save raw disparities	0.187	0.195
Cross check	0.087	0.094
Occlusion fill	0.075	0.083
Apply 5x5 moving avg filter	0.154	0.149
Weighted median filter	1.426	1.452
Total Execution time	63.605	18.550

D. Phase 4

This phase is a multi-thread implementation of Phase 3. I used OpenMP to take advantage of all the available threads to parallelize the 'for' loops. The average execution time for each operation is given in Table II. Figure 5 shows execution time in each iteration. From the figure and table, it is evident that parallel execution took a lot less time than sequential implementation.

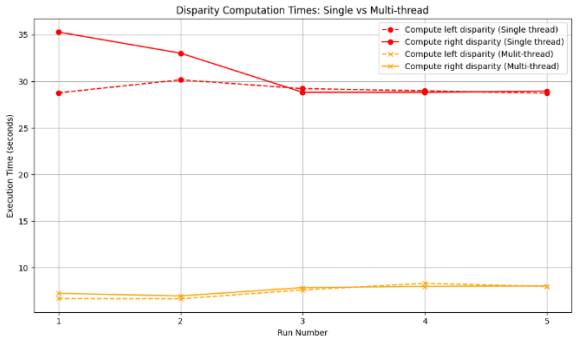

Figure 5: Execution time comparison between single and multi-thread implementation.

E. Phase 5

For this phase each C function was converted into the corresponding OpenCL kernel. Each kernel was placed in a separate file which has .cl extension. To get the correct execution time the code was run five times Table III provides the average time of each operation. In the unoptimized GPU implementation, disparity calculations (*Left* → *Right* and *Right* → *Left*) were the most time-consuming operations, taking 120.330 and 115.000 milliseconds respectively, and together accounted for the majority of the total execution time of 273.524 milliseconds.

TABLE III

Execution time (ms) for Unoptimized (Phase 5), Optimized (Phase 6) and Ultra-optimized (Phase 7)

Operation	Unoptimized (Phase 5)	Optimized (Phase 6)	Ultra-optimized (Phase 7)
Host to device transfer (left image)	16.762	16.331	16.345
Host to device transfer (right image)	16.440	16.225	16.252
Left image resize	0.239	0.240	0.239
Right images resize	0.226	0.227	0.227
Device to host (left resized)	0.857	0.855	0.859
Device to host (right resized)	0.855	0.856	0.856
Grayscale conversion (left)	0.080	0.081	0.080
Grayscale conversion (right)	0.067	0.067	0.068
Device to host (left grayscale)	0.210	0.211	0.211
Device to host (right grayscale)	0.211	0.211	0.211
Disparity (Left -> Right)	120.330	55.062	55.054
Disparity (Right -> Left)	115.000	54.455	54.435
Device to host (left disparity)	0.210	0.215	0.211
Device to host (right disparity)	0.211	0.214	0.211
Cross check kernel	0.087	0.086	0.085
Device to host (cross checked)	0.210	0.219	0.210
Occlusion kernel	0.407	0.409	0.406
Device to host (occlusion filled)	0.211	0.210	0.210
Moving average filter kernel	0.903	0.902	0.902
Device to host (filtered)	0.210	0.210	0.210
Total execution time	**273.524**	**147.284**	**147.284**

V. OPTIMIZATION OF OpenCL KERNELS

This is part of Phase 6. In this phase I optimized the previous implementation using the five strategies.

As most of the computation is done by the zncc_left() and zncc_right() kernel, I tested each method individually on zncc_left() kernel. All the methods when tested individually improved execution time. Table IV shows the average execution time for each of these methods. After testing all of these methods, zncc_left() and zncc_right() kernel was optimized using combinations of these methods except the vectorization method which introduced errors during compilation.

A. Vectorization

Instead of accessing single data OpenCL provide built-in vector datatype. Instead of processing data one by one we can use the vector to process multiple. I used vec4 for this implementation. When this method tested individually average execution time for the zncc_left() kernel was 75.970 ms – 36% faster execution.

B. Local memory utilization

Local memory reduces access time for work items. Work items could access local memory faster compared to the global memory. From Table IV the average execution time for zncc_left() kernel using this method was 98.646 ms – 18.2% faster execution.

C. Memory coalescing and

When adjacent work items access consecutive addresses, the hardware can coalesce these accesses into fewer memory transactions which reduce data fetching time. From Table IV the average execution time using these methods is 100.435 ms – 16.5% faster than unoptimized computation.

D. Reducing register usage

When we declare variables in the kernels they take some register space. We can reduce the register usage by simply reducing the number of variables in the kernels. From Table IV we can see that this methods increases computation efficiency by 21.3%.

E. Compiler flags

Compilers provide additional advantages of optimization. Using appropriate flag, we can help compiler to further optimize the code to improve the execution time. From Table IV we can see that this method can significantly improve execution time from unoptimized 120.330 ms to 94.650 ms – 21.2 % more efficient.

Table III shows the execution time after combined optimization. Figure 6 compares the optimized vs non-optimized kernel execution time.

TABLE IV
Execution time for zncc_left() using different optimization strategies.

Function	Vectorization	Local memory	Memory coalescing	Reduced register usage	Compiler flags
zncc_left()	75.970 ms	98.464 ms	100.435 ms	94.736 ms	94.650 ms

F. Using memory hierarchy

This was done during Phase 7. Phase 6 was further optimized by taking advantage of the memory hierarchy available in OpenCL and Nvidia GPU. Global, private and local memory were used. Local memory was tiled accordingly. The average execution time improved a little bit because during Phase 6 I already used tiling and memory hierarchy. Table III shows execution time for each operation in this phase.

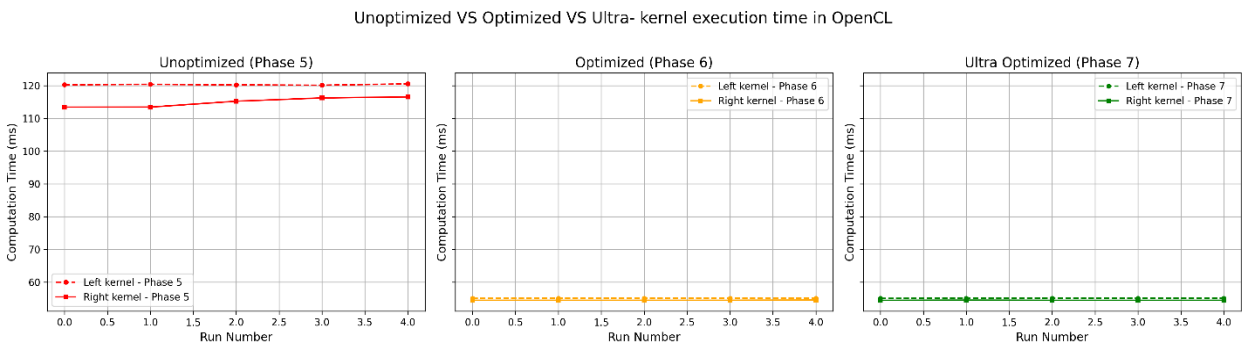

Figure 6: Execution time for zncc_left() and zncc_right() for unoptimized, optimized and ultra-optimized cases.

Figure 6 visualizes how much time was optimized. From the figure we can clearly see the huge difference between unoptimized (Phase 5) and Optimized (Phase 6). Although Phase 6 and 7 seem similar, there are some small differences. In both left and right disparity computations, Phase 7 (Ultra-optimized) achieved slightly lower execution times compared to Phase 6 (Optimized), with 55.054 ms vs. 55.062 ms for left disparity and 54.435 ms vs. 54.455 ms for right disparity. Although the differences are minimal, Phase 7 consistently shows marginally better performance.

VI. DISCUSSION & CONCLUSION

All phases have been completed step by step manner. All checkpoints are met accordingly. The GitHub link contains clean organized and well commented code. Each kernel and function have been benchmarked properly. Table I, II, III, and IV provide exhaustive data on execution time for each operation. I took all the data with great care and precision, which is evident in those tables and graphs that I provided. I was stuck while trying to solve errors when the vectorization method was combined with other methods. Vectorization method could be further tested to optimize the code. Another method I plan to test is partitioning the image and executing both zncc_left() and zncc_right() parallelly. It is interesting to see how fast GPU can compute. Before starting to implement this algorithm in C, I implemented it in Python. It took 20 minutes to complete the execution. In single thread C, it took over 1 minute to complete. In multithread C, it took over 30 seconds to complete. Surprisingly, in OpenCL it took only 54 milliseconds to complete the execution of left disparity computation.

GitHub Link: Final code: https://github.com/Abu-Taher-web/Multi-Processor-Programming.git

Note: This repository was made from (https://github.com/Abu-Taher-web/Multiprocessor-Programming-Course.git), which contains all the code that I practiced during the course.

REFERENCES

[01] A. Redert, E. Hendriks, and J. Biemond, "Correspondence estimation in image pairs," IEEE Signal Processing Magazine, vol. 16, no. 3, pp. 29–46, 1999.

[02] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," International Journal of Computer Vision, vol. 47, no. 1–3, pp. 7–42, 2002.

[03] R. I. Hartley and A. Zisserman, *Multiple View Geometry in Computer Vision*. Cambridge University Press, 2nd Ed., 2003.

[04] J. P. Lewis, "Fast normalized cross-correlation," in *Proc. Vision Interface*, 1995, pp. 120–123.

[05] A. Tiulpin and I. Melekhov, *Implementation of ZNCC algorithm for disparity map computation using OpenCL*. Technical Report, University of Oulu & Aalto University, 2015.

[06] lext, "zncc," GitHub. [Online]. Available: https://github.com/lext/zncc [Accessed: May 15, 2025].

[07] NVIDIA, "CUDA Toolkit Downloads," NVIDIA Developer. [Online]. Available: https://developer.nvidia.com/cuda-downloads?target_os=Windows&target_arch=x86_64&target_version=11&target_type=exe_local [Accessed: May 15, 2025].

[08] L. Vandevenne, "LodePNG," *lodev.org*. [Online]. Available: https://lodev.org/lodepng/ [Accessed: May 15, 2025].

[09] "MinGW: Minimalist GNU for Windows," *SourceForge*. [Online]. Available: https://sourceforge.net/projects/mingw/ [Accessed: May 15, 2025].