Final Report

Abdurrehman Syed
Student ID: 2308546

Introduction: Stereo vision is a computer vision technique which mimics human vision to perceive depth from a pair of stereo images. This ability of calculating depth and reconstructing the 3D information from a 2D image has numerous applications in several domains such as robotics, autonomous vehicles, augmented reality, and scientific research.

A crucial step in stereo vision is disparity estimation, which determines the horizontal shift between corresponding points in the left and right images of a stereo pair. This shift represents the difference in depth between objects in the scene.

There are several algorithms which calculate disparity between stereo images. This report focuses on the Zero-mean Normalized Cross Correlation algorithm (ZNCC). ZNCC is a popular choice due to its straightforward approach and ability to normalize pixel intensities.

This report explains the concept of ZNCC, the approach taken to implement it, along with different experiments conducted which includes calculating disparity on various ranges, testing different window sizes, cross-checking disparity maps, and occlusion fitting. Additionally, the time-profiling is done throughout the implementation, parallelizing the algorithm on different CPU cores, testing it on GPU for acceleration using frameworks like OpenCL and OpenMP.

Zero-Mean Normalized Cross Correlation: ZNCC relies on the concept of normalized cross-correlation to find corresponding pixels between the left and right images of a stereo pair. The basic idea is to compare small image windows centered around a specific pixel in the left image with corresponding windows in the right image at different horizontal shifts (disparities) and vice versa. The disparity that produces the highest correlation between the windows is considered the most likely match for the center pixel.

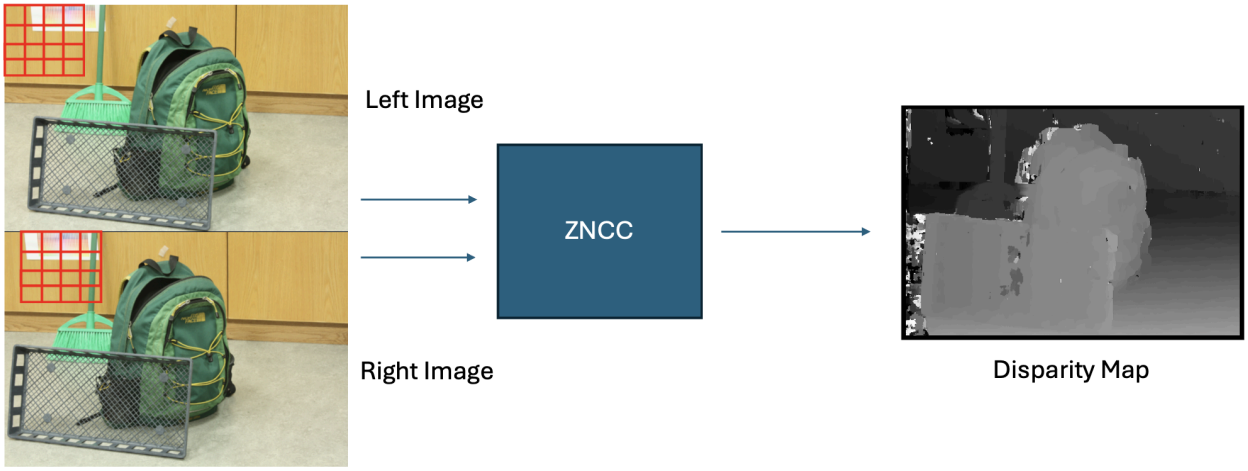


Figure – 1

Initially, a window-size is selected for comparing the corresponding image regions, the sizes can be e.g (3x3, 5x5, 15x15) in general odd sizes are preferred so that we can have a center pixel. Afterwards a disparity value is set which defines a range of horizontal shifts to search for corresponding pixels in the images.Figure – 1 highlights that there are two stereo images with red windows. The first image which is labeled as the left image is used as the target while the second image labeled as the right image is used as the template. The second window also shows a horizontal shift which highlights the disparity that the algorithm introduced. This approach ensures that one window from the target image is compared to a range of windows from the template image inorder to find the highest correlation between the two.

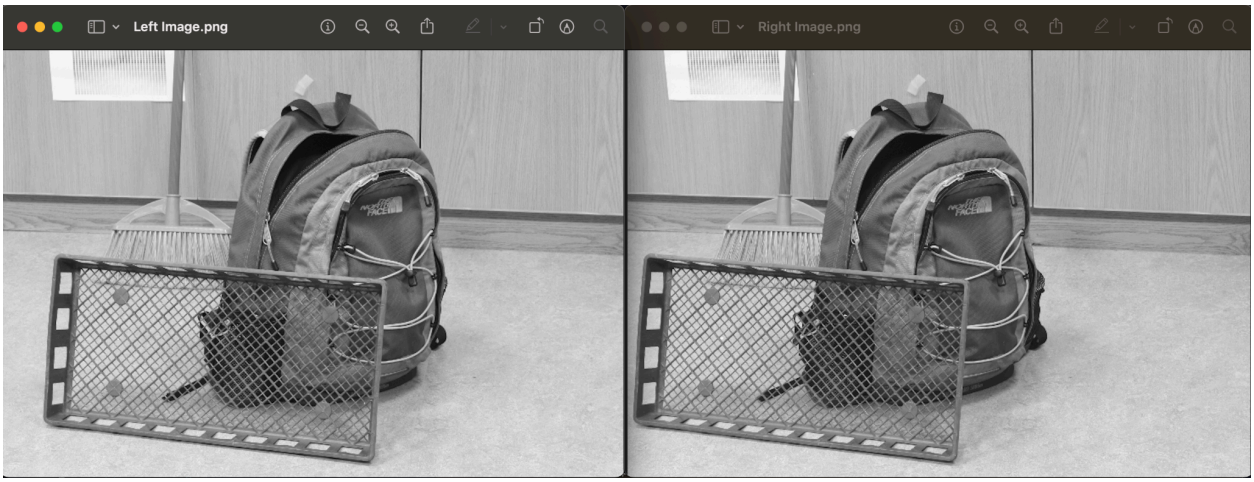
The next step involves calculating the zero-mean normalized cross correlation values which is done by taking the average for all the pixel intensities within a window and then subtracting that from each individual pixel within it (zero mean). Next, the standard deviation needs to be calculated for these window patches in order to perform the normalization step. Finally, the numerator of the algorithm consisting of zero-mean patches of both images are multiplied and then divided by the product of standard deviations of the two windows. As a result we obtain a disparity value, this process leads to a disparity map as seen in the figure above.

Methodology: The approach taken in this project is that the algorithm was first implemented using Python (due to familiarity) and then later converted to C++. It consists of several functions which are explained in detail below.

Image Resizing

To improve the processing speed, I first read two stereo RGB images (2940x2016) using OpenCV and converted them to grayscale. Created a custom resize_image function that takes the image and a downsampling factor as inputs. It checks if the image exists and then calculates the resized image dimensions by dividing the original size by the factor.

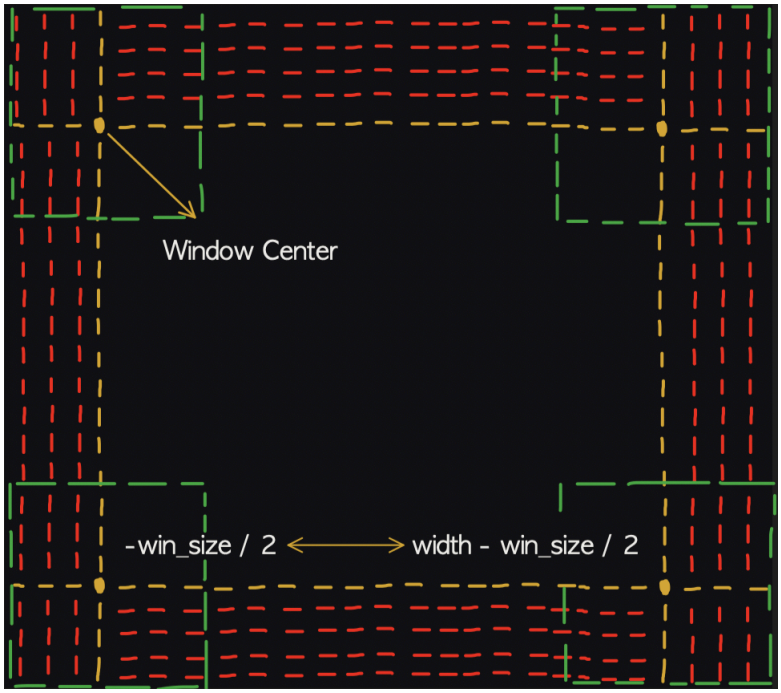
Assuming a factor of 4, the function iterates over every fourth pixel in both the horizontal and vertical directions of the original image and adds those pixels to a new, downsampled image. This approach reduces the image resolution while preserving its overall content.



Left and Right Disparity Maps

The size of resized images is then used for creating two image objects for disparity maps in which we will store the output that we get after calculating disparities between stereo images. For this function we pass the left and right images along with their width, height, window size, and disparity value. The concept used here is that a window is slid across both images, for calculating the disparity maps. For left disparity map calculation, we also subtract disparity “d” while sliding the window across the width of the right image, and vice versa we add “d” for the right disparity map calculation. That is, for the left disparity map the shift is to the left “-d” in the right image and for the right disparity map shift will be towards the right side “+d”. One constraint is to consider at which pixel we should start the window so that it does not exceed the boundaries of the image as we slide it across the image.

The approach I have taken here is that I am starting my window from the center pixel of that patch. In other words if I have a 3x3 window (9 pixels) I am starting with the (1,1) or middle pixel and to achieve this I am looping starting with “window / 2” and it goes up until “width - window / 2”. In the figure below we can see that the golden area is the size the window is covering and the center dot is where I am starting the calculation for pixels, the area on the left and on top is - window / 2 which will go towards the right. Similarly it will work from top to bottom.



The implementation works with two nested loops that iterate over the horizontal and vertical axes of the image then we have a calculate mean function which calculates the window mean of the image for which we are finding the disparity and returns that value which is later used in ZNCC calculation. Further, we have another loop that is essentially for the image that we are comparing the target image with so if we are calculating disparity for left image we will first find the mean for left one and then we will start the disparity loop which will then calculate the mean of the right image, the loop will run “d” times based on our maximum disparity value given. What happens inside is that it calculates the mean again but this time for the other image along with

disparity shifts (horizontal shifts) what this approach does is that it picks a patch from the left image and tries to locate it on the right image and it keeps on shifting the area and trying to find the best possible match within the given range “d”, it calculates that match using ZNCC function. Finally, it saves the best possible match with highest ZNCC values in the new image at the same corresponding pixel location as our target image.

Calculating Mean

This function takes x and y coordinates, the window, the image for which we are calculating this mean, and a flag which checks whether the window is for the right image or the left one. Initially, we run a nested loop to iterate over all the pixels of our window, it can be thought of as a mini image inside a big image and now we are iterating over this mini image trying to find the average of this patch. The way we are iterating is that we run a loop with - window / 2 up until window / 2 to iterate in both horizontal and vertical directions. For instance, if we have a 3x3 window we will pick the center pixel which is (1,1) now the loop will range from -1 to +1 (-1, 0, +1) and since x and y are (1,1) we can subtract or add values with them to iterate the window and take mean.

If the image has an additionally value of “d” disparity then we will add or subtract the disparity on the x-axis based on what image we are processing, for left image we would subtract the disparity value and for right we would add, its like adjusting or moving the window left if the image we are targeting is left and right if the image we are focusing is right.

Calculating ZNCC

In this function we pass in the mean values of both patches, the x and y coordinates, disparity value, window, and the images. Afterwards, re-iterating over the window but this time it is done on both patches, in order to calculate the mean and then subtract it from every pixel of the window. Following that, we calculate the numerator for ZNCC which is the product of the resulting images after subtracting mean from each pixel and the denominator is the square of the resultant numerator for both images which is then multiplied together and finally their square root is taken, basically standard deviation is calculated. The resultant value is then returned and saved in the new image at the designated pixel location.

Cross-Checking Disparity Maps

This function takes the left and right disparity maps along with a threshold value. We iterate over each pixel in both images and subtract the intensities with each other taking their absolute value and comparing it with the threshold, if the resulting value exceeds the threshold the pixel will be black otherwise the pixel will remain the same.

$$|\, left\ Disparity\ Map\ -\ Right\ Disparity\ Map\,| > threshold\ ?\ 0 : pass$$

Occlusion Filling

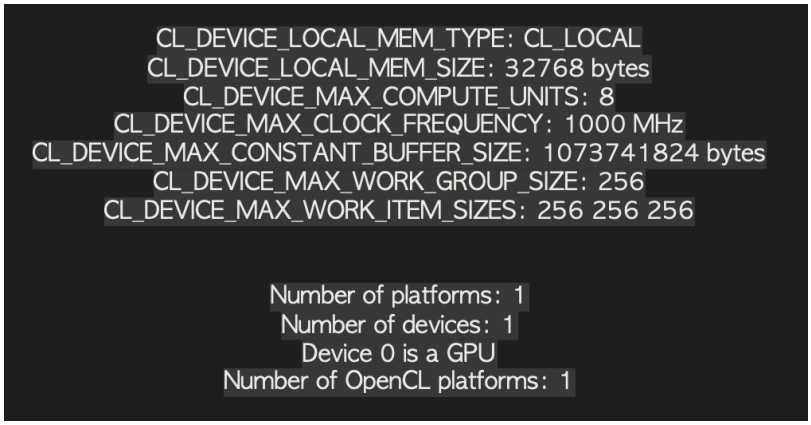
This function takes the cross-checked disparity map and performs occlusion filling on it. It does so by taking a maximum search distance parameter which acts as a search range for searching neighboring pixels and checking whether a non black pixel is present if it is then the current pixel will get that value. Basic idea of occlusion filling here is that we need to find and eliminate the pixels with zero intensity so we iterate over the image using nested loops, finding black pixels and trying to modify them based on the neighbors and maximum search distance.

Utility Functions

I have created two utility functions, one for saving the output images by creating a directory and another for creating a parameter file which stores relevant function arguments such as window_size, threshold values, etc.

Implementation: The implementation follows all the aforementioned functions in the exact hierarchy for both CPU and GPU execution. Additionally, the “chrono” header file was used for time profiling. I have also used “thread” header file for running concurrent processes such as one thread for calculating the left disparity map and second for the right one. I have also tested OpenMP implementation using the pragma directives and “omp” header file for concurrent loop execution and reduction operations.

For GPU execution I created a host file that checks the host platform, the devices present (if any) and their different configurations which can be seen in the figure below for Mac M1.



The flow of the host file was such that after checking the devices and selecting the GPU I create a context for that device and a commandQueue where all our kernels will be pushed. I have used two kernels for ZNCC implementation, the first one resizes the images and the second one performs ZNCC computation. After loading the kernel in the command queue I created the program, built it, created the kernel, the memory buffers for the image in order to have a space shared between the CPU and GPU where they can read and write image data. Passing the kernel arguments in the case of resizing it will be the factor and the images themselves. Enqueuing the kernel and then running it as an event so that the next kernel doesn't start right away and instead waits for the resizing to first occur and afterwards the kernel and program

space is released. For the ZNCC kernel I did the exact same but passed a lot more arguments and multiple read write buffers for getting different output images at each step.

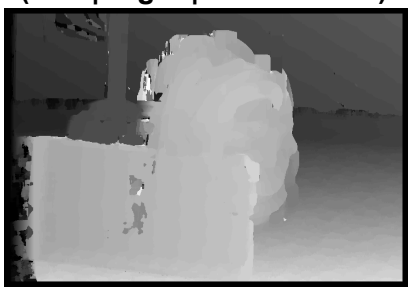
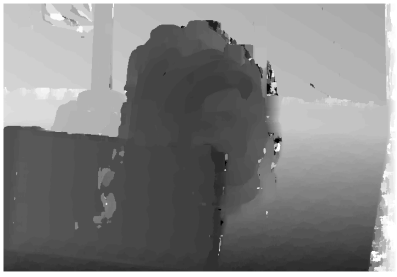
The OpenCL files contain the code which is almost identical with the CPU code. The only difference is that we have created work-items and we distribute the parallel computation among them. Additionally, I have also used barriers so that work-items know whether to wait for other workers to finish or not.

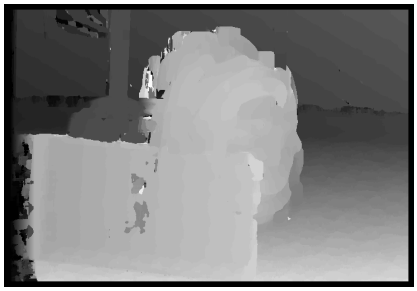
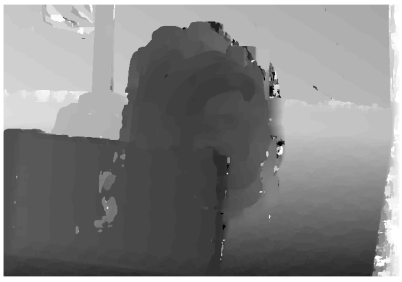
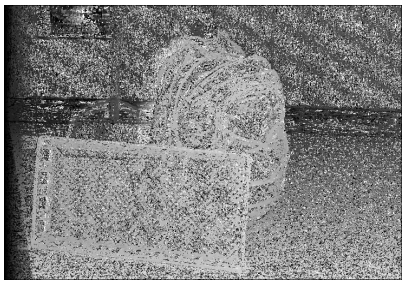
Experimentation and Results: Below are the results I have got with various experimentations and parameter tuning on CPU and GPU. First I will present some python single-threaded results (I have conducted more than 50 experiments) on linux as well as windows machines, and afterwards I will show my C++ results running on CPU, CPU-multithreaded and GPU on a macbook.

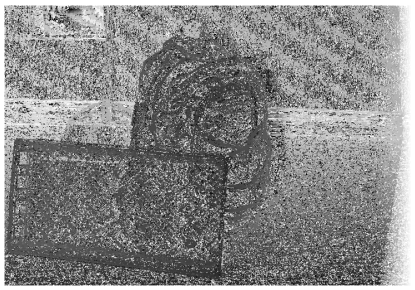
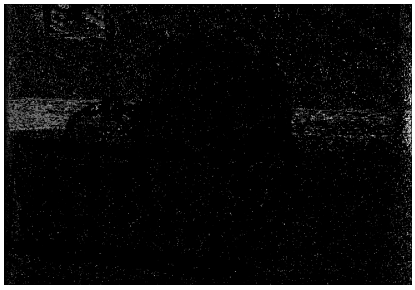
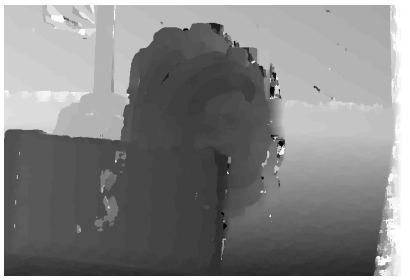
Platform and Development environment: The development platforms used are MacOS with an M1 GPU and Linux using AMD Ryzen Threadripper 3960X 24-Core Processor 2200 MHz and windows using Intel Core i5-8365U CPU @ 1.60GHz, 1896 Mhz, 4 Cores, 8 Logical Processors. The coding environment being used is Visual Studio Code.

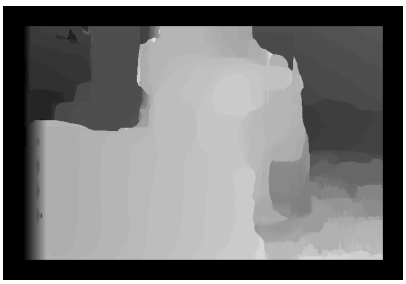
Python - Linux and Windows CPU Results

Orange represents time taken by Linux machine
Blue represents time taken by Windows machine

Patch Sizes	Downsample Factor	Max Disparity	Cross Check Threshold	Disparity Maps (Left | Right | CrossCheck)
21	4	65	8	
Disparity Map Left	**Disparity Map Right**	**Total Time**		
1276.86805 seconds	**1286.4385116 seconds**	**2565.968 seconds**		
3503 seconds	**3563 seconds**	**7846 seconds**		

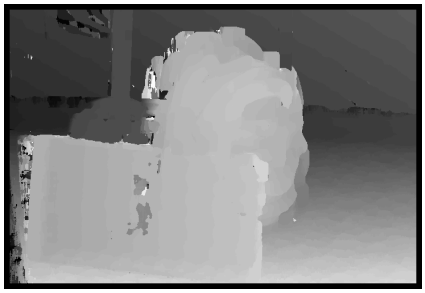
				
21 **Disparity Map Left** **1286.33588 seconds** **3644 seconds**	4 **Disparity Map Right** **1284.240600 seconds** **3650 seconds**	65 **Total Time** **2573.157 seconds** **7320 seconds**	20	  
3 **Disparity Map Left**	4 **Disparity Map Right**	65 **Total Time**	8	

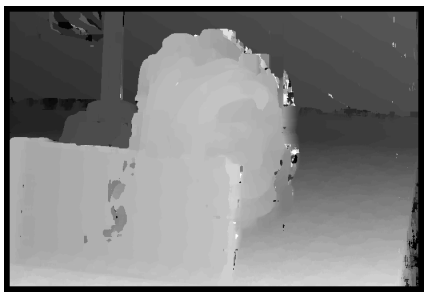
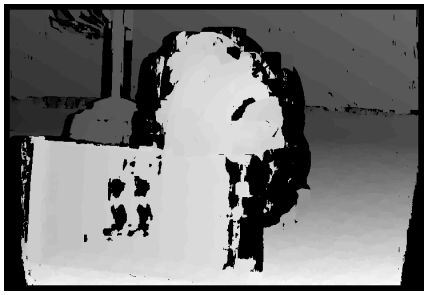
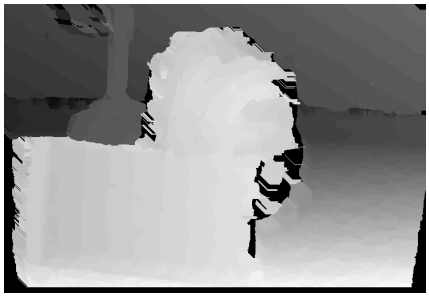
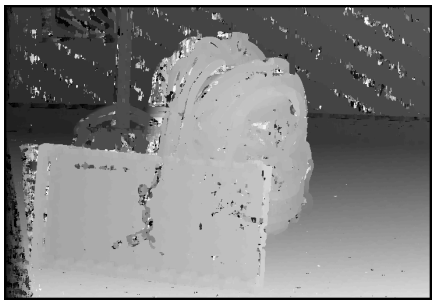
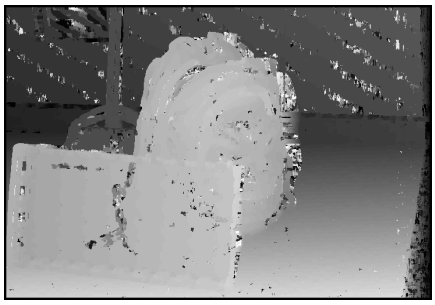
1233.51231 seconds 1860 seconds	1241.374599 seconds 1899 seconds	2475.473 seconds 3759 seconds		 
19 **Disparity Map Left** 1266.98170 seconds 3636 seconds	4 **Disparity Map Right** 1262.697680 seconds 3630 seconds	65 **Total Time** 2530.234 seconds 7266 seconds	80	  

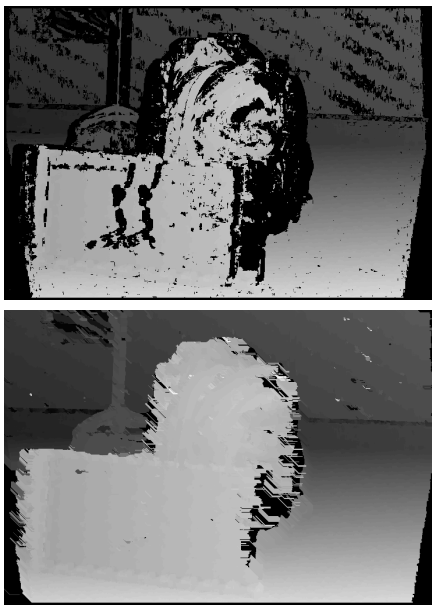
75	4	65	8	
Disparity Map Left **1588.961 seconds** **4619.9177 seconds**	**Disparity Map Right** **1587.142 seconds** **4610.54125 seconds**	**Total Time** **3178.657 seconds** **9231.121 seconds**		 

C++ Results

Red represents the time taken using threads
Purple represents openMP executions
Green represents normal CPU execution

Patch Sizes	Downsample Factor	Max Disparity	Cross Check Threshold	Max Search Distance	Disparity Maps (Left | Right | CrossCheck | Occlusion)
21	4	65	12	16	

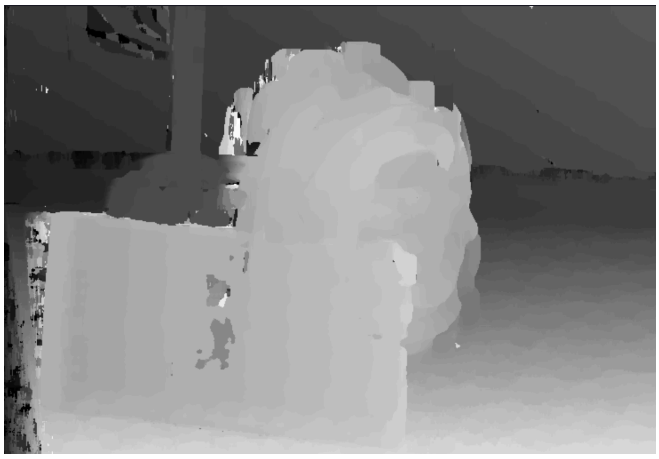
Disparity Map Left	Disparity Map Right	Total Time	Cross Checking time	Occlusion time	
321718 ms	325107 ms	330147 ms	4 ms	35 ms	
109914 ms	107603 ms	223885 ms	1 ms	14 ms	
229511 ms	216800 ms	453525 ms	4 ms	36 ms	
9	4	65	8	16	
Disparity Map Left	Disparity Map Right	Total Time	Cross Checking time	Occlusion time	
71332 ms	71672 ms	79748 ms	4 ms	38 ms	
26060	23971	59108	2 ms	11 ms	

ms 47367 ms	ms 46083 ms	ms 99869 ms	4 ms	39 ms	

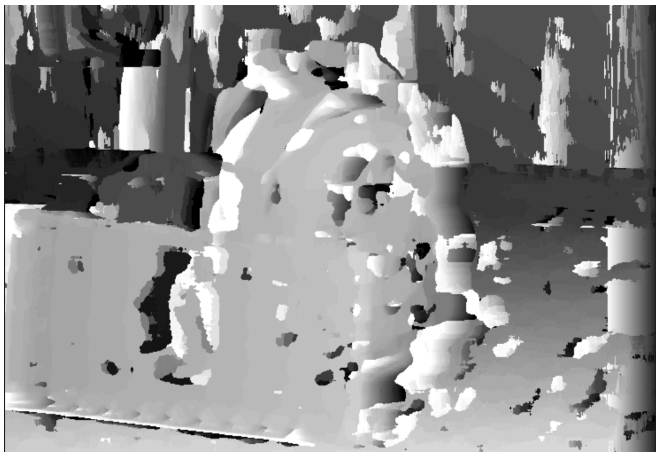
As observed OpenMP is the fastest followed by threading and then the normal CPU execution.

GPU Results – 1

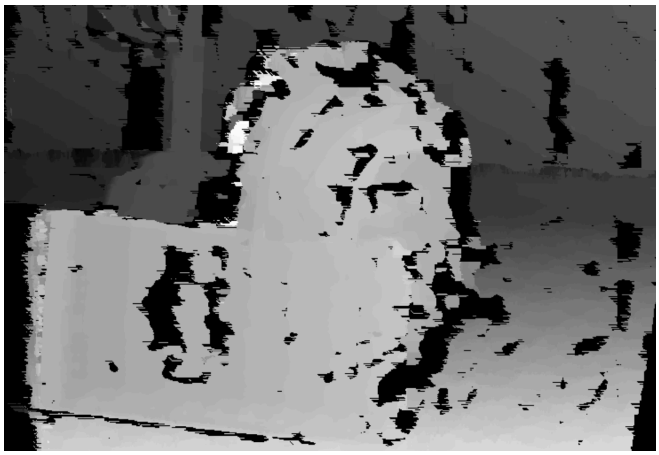
- **Resize Build log:**
  - Resize Kernel (Left Image) Time: 0.001313 ms
  - Resize Kernel (Right Image) Time: 0.001035 ms
- **ZNCC Build log:**
  - ZNCC Kernel Time: 47.7482 ms



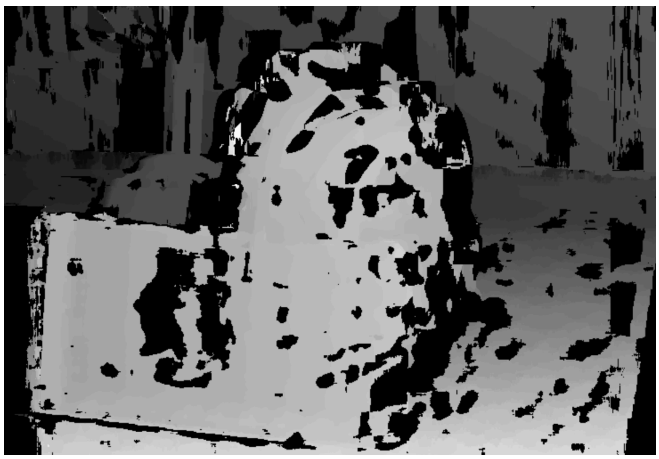
Disparity Image Left



Disparity Image Right



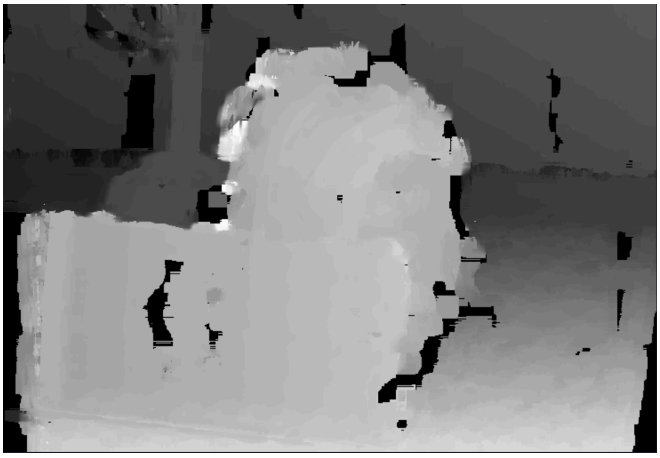
CrossChecked Disparity Map



Occlusion Filling(Neighbour Search)

Optimized GPU Results(Box Filling)

- **Resize Build log:**
  - Resize Kernel (Left Image) Time: 0.001158 ms
  - Resize Kernel (Right Image) Time: 0.00093 ms
- **ZNCC Build log:**
  - ZNCC Kernel Time: 44.6693 ms



Box Occlusion Filling

Here I used the Box Occlusion Filling approach rather than the occlusion filling with neighboring search. Box takes the window size and finds the non zero pixel instead of searching the neighboring pixels in four directions, and as we can see the results are much better than our normal occlusion. Additionally if the max distance is high and we get the worst case the computation will be much higher than we will have using this technique.

GPU Optimization Results – 2

Memory optimization by moving certain loop variables outside the loop's scope in meanCalculation and calculateZncc functions.

- **Resize Build log:**
  - Resize Kernel (Left Image) Time: 0.003183 ms
  - Resize Kernel (Right Image) Time: 0.001467 ms
- **ZNCC Build log:**
  - ZNCC Kernel Time: 42.9218 ms

Analysis: The approaches considered for this project were pretty straightforward and are explained in the methodology section in detail. After trying out different variations in patch-sizes, thresholds, search-distances, and disparities I learned that with lower patch-sizes the image becomes grainy and noisy while as we increase the patch-size it gets smoother and smoother and after a certain point the details are lost and the depth of the objects becomes foggy. Testing patch-sizes from 3 to 75, patch-size near 20 gave the optimal results.

With very low cross-check threshold values, a large number of pixels become invalidated (black) and more inconsistencies are found, and with very large values no inconsistencies get found and we just get the original image back without change. So I tried various values to find the optimal threshold between these two extremes on both CPU and GPU.

Increasing search-distance will result in lower number of pixels with zero intensity (black) but it primarily depends upon the previous cross-checking step since we search the area and if in cross checking the area is all black we will have issues fixing the intensities.

From the performance standpoint and CPU-GPU executions, I analyzed the computation speed of the GPU compared with a normal CPU.

For the same experiment, the linux machine took more than 40 minutes to run with Python, while the mac CPU took 100 seconds whereas the GPU took 0.05 seconds in C++. For GPU processing I used OpenCL and explored its various functions including waitEvents and Barriers for ensuring correct workflow. Optimized the GPU performance by trying out the box occlusion filter. Additionally, I moved out the redundant variables from the loop to avoid excess computation.

Other optimizations that can be considered for the future are memory optimization, vectorization, reducing the redundant data by calculating both disparities in the same function instead of two different ones, and trying out a more efficient occlusion filling approach.