Disparity Map Estimation using Block Matching Algorithm

Chachithanandhini Bodipalayam Kalyanasundaram – 3469240, Meghana Iyer Narayana Swamy – 3440748, Swapnil Gurav – 3506611[Equal contribution - Alphabetically arranged]

st171165@stud.uni-stuttgart.de, st170072@stud.uni-stuttgart.de, st175111@stud.uni-stuttgart.de

Institute for Parallel and Distributed Systems, IPVS, University of Stuttgart

Abstract -This project estimates the Disparity mapping using the block-matching algorithm with GPU (Graphical Processing Unit) programming. Implementation on the GPU is then subjected to performance comparison with the CPU version. The disparity is nothing but a difference in x-coordinate between the corresponding pixel of two images of the same scene taken from different viewpoints. The disparity values are stored in another image which is called disparity map, which is nothing but the pixel difference or motion between the two images. It is usually done to determine the depth of the scene or to determine the distance to objects. Disparity map implementation here is achieved by the Block matching algorithm using two methods, SAD and SSD.

Keywords – Disparity Mapping; Stereo vision algorithms; Block matching algorithm; SAD; SSD; GPU programming; Opencl; Performance analysis.

1. INTRODUCTION

The stereo matching algorithm is one of the challenging problems in computer vision. Traditionally, in stereo-vision, two cameras are displaced in the x-direction to get two different views of the same scene. By comparing these two images and finding the disparity between the corresponding pixel, we can find the depth of the image, which is commonly referred to as the disparity map. The high importance of Disparity mapping is due to its advantages in modern applications like autonomous driving, robotics, augmented reality, etc.

The stereo vision disparity mapping can be basically divided into two approaches: Local method (Area-based or Window-based) and Global method (minimizing a global energy function for all disparity values). Area-based or window-based methods are more accurate because the matching process considers the entire set of pixels associated with image regions. Standard algorithms for window-based techniques include: the sum of absolute differences (SAD), the sum of squared differences (SSD), normalized cross-correlation (NCC), rank transforms (RT), and census transforms (CT) [3]. This project mainly deals with two basic algorithms, SAD and SSD, to calculate the disparity map. Further, we have discussed in detail the implementation of the

algorithms in GPU and CPU along with performance comparison using their computation time and their speedup.

2. PROPOSED SOLUTION

To calculate the disparity map, we are using the sum of difference (SAD) and the sum of squared difference (SSD) methods. The algorithm considers two pixels from two different images which are in the same scene. The absolute difference between these two pixels is nothing but SAD. To get the exact corresponding pixel, we predefine the window size, and summation of all the differences for every pixel present in this window size is stored as the SAD value. The method is iterated till we find the minimum SAD by changing the disparity value. The disparity value here is to specify how far away from the template location we want to search for the corresponding pixel. As shown in figure 1.1, there are two images: one is the reference image and the other, the matching image, and the disparity between two conjugate pixels is two here.

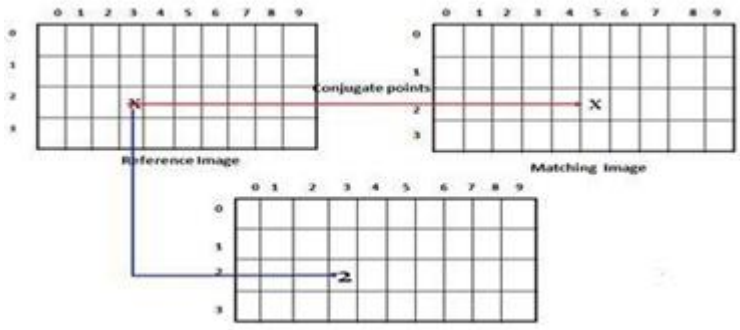


Figure 1.1 Disparity map calculation.

The Mathematical formula for calculating SAD is shown in fig 1.2. The formula has the following variable ‘d’, representing the disparity value, ‘x’ is the row value, and ‘y’ is the column value of the image pixel in the particular window size ‘w’.

$$\mathrm{SAD}(x, y, d) = \sum_{(x,y)\in w} \left| I_l(x, y) - I_r(x - d, y) \right|.$$

Figure1. 2 Mathematical formula for SAD

This method is fast, but the quality of the disparity map generated is low because of the noise at the object boundaries and in the texture-less regions.

$$\mathrm{SSD}(x, y, d) = \sum_{(x,y)\in w} \left| I_l(x, y) - I_r(x - d, y) \right|^2.$$

Figure1.3 Mathematical formula for SSD.

The sum of squared difference (SSD) algorithm is similar to and the improved version of the SAD. Here, we are squaring the difference in the pixel of the two images. The mathematical formula is shown in figure 1.3. The variables are similar to those discussed in the SAD section.

3. IMPLEMENTATION

Description:

Input: Two grey scaled images(width*height) of the same scene from two different viewpoints,

Window Size

Output: Disparity map (mapped image).

Analysis: Comparison of performance and speedup between GPU and CPU computation.

3.1 CPU Implementation

The discussed algorithms SSD and SAD are implemented in the CPU on account of performance comparison with the GPU implementation. Since CPU does not possess the parallel architecture of GPU, and the resultant computation time for each of these implementations is going to be different, and it is our goal to analyze their performance.

As part of the CPU implementation, fig.1.4 shows the main logic implementation for calculating the disparity values using either the SSD or the SAD algorithm. ‘image_width’ and ‘image_height’ are the dimensions of the input images. The window size is represented by the ‘window_range’ variable. The final value is calculated with variable ‘disparity’ and mapped to ‘output_image’. Due to the nested loops, the computation seems to take more time over a large area/window.

3.2 GPU Implementation

The GPU algorithm and logic implementation are identical to the CPU implementation. However, since GPU enables parallel processing, it should have less computation time compared to CPU.

As we have discussed in section 2 regarding the proposed solution and the two algorithms that are commonly used in calculating the disparity, now in this section, we will briefly discuss how to implement the algorithm. The implementation part remains roughly the same for both SAD and SSD algorithms.

Algorithm Algorithm to calculate disparity mapping using SAD and SSD.

INPUT: $img1$ and $img2$ - input images used for depth calculation, $image\_width$ and $image\_height$ - width and height of the image, $function\_call$ - the value decide to call SSD or SAD

OUTPUT: $output\_image$ - the disparity map image

```
1: window_size ← 13, variable declaration
2: window_range ← window_size/2, variable declaration
3: disparity_max ← 100, variable declaration
4: disparity ← 0.0, variable declaration
5: SSD ← 0.0, variable declaration
6: SSD_min ← 10000.0, variable declaration
7: SAD ← 0.0, variable declaration
8: SAD_min ← 10000.0, variable declaration
9: function DISPARITYMAPPING(img1, img2, image_width, image_height)
10:     for i ← 0 to image_width do
11:         for j ← 0 to image_height do
12:             SSD_min ← 10000.0
13:             SAD_min ← 10000.0
14:             for d ← 0 to disparity_max do
15:                 SSD ← 0.0
16:                 SAD ← 0.0
17:                 for width ← 1 − window_range to 1 − window_range do
18:                     for height ← 1 − window_range to 1 − window_range do
19:                         temp_pixel ← ABS(GETVALUEGLOBAL(img1, image_width, image_height, width +
    i, height + j) - GETVALUEGLOBAL(img2, image_width, image_height, width +
    i − d, height + j))
20:                         if function_call ≡ SAD then
21:                             SAD ← SAD + temp_pixel
22:                         end if
23:                         if function_call ≡ SSD then
24:                             SSD ← SSD + (temp_pixel ∗ temp_pixel)
25:                         end if
26:                     end for
27:                 end for
28:                 if SAD_min > SAD && function_call ≡ SAD then
29:                     SAD_min ← SAD
30:                 end if
31:                 if SSD_min > SSD && function_call ≡ SSD then
32:                     SSD_min ← SSD
33:                 end if
34:                 disparity ← d / disparity_max
35:             end for
36:             output_image[GETINDEX(image_width, i, j)] ← disparity
37:         end for
38:     end for
39: end function
```

Figure 1.4 Implementation logic for both CPU and GPU (first two for loops are not included).

3.2.1 Input to code

The images are taken in the PGM format that is in the grayscale format so that we have only one value for each pixel at it ranges from 0 to 255 for each pixel. A simple example is shown in fig 1.5. To compute SAD and SSD, the two input images: the left-image and the right-image, needs to be calibrated to achieve line alignment.

3.2.2 SAD calculation

For SAD estimation, we take the sum of absolute difference between the reference and a block. The code subtracts each pixel value from the reference to the absolute block as shown in the fig-1.5, where each pixel is subtracted with the other image pixel.

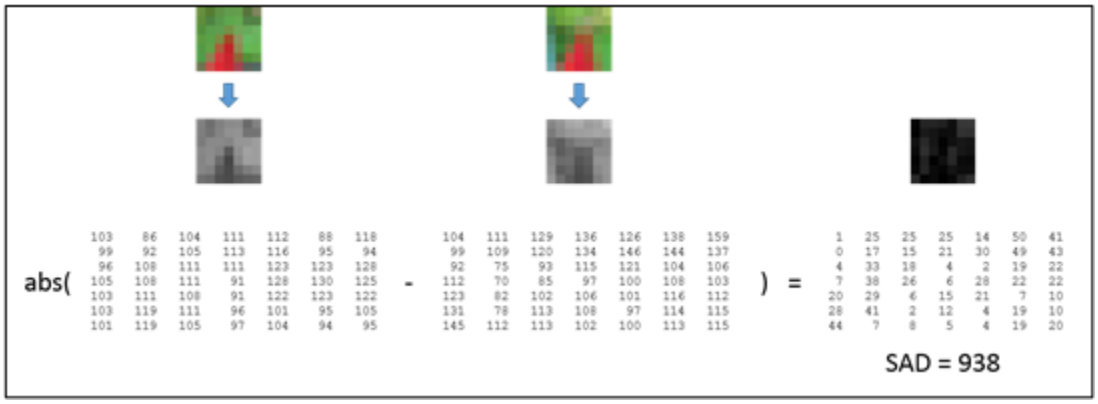


Figure 1.5 Block comparison.

3.2.3 SAD matching algorithm

- To find the corresponding pixel, we scan the left image and select the anchor point and construct the small window which is predefined.
- The window size is used to select all the pixels in the area covered by the small window.
- Similarly, we select all the pixels covered by the window in the right image.
- The pixels in the area covered by Left-Image are subtracted from the pixels in the area covered by Right-Image. The sum of the absolute values of the differences of all pixels is calculated.
- Move the small window of Right-Image and repeat the above two operations.
- Iterate the process by changing the disparity value. The minimum SAD value is stored once we find the minimum SAD value, then we store the disparity value corresponding to the corresponding pixel.

3.2.4 Pixel selection

Once the window size is selected, we choose the pixels by traversing half the window size to the left and right of the selected pixel. This is the same for y-direction as well. For example, if the window size is 9, we choose 4 pixels on the left and 4 pixels right from the anchor point. Similarly, 4 pixels up and 4 pixels down the anchor point, creating the matrix surrounding the anchor point.

For example, consider fig 1.6. If the anchor point is A33 and the window size is 5. To get the minimum point, we subtract half the window size, (X, Y) - (window size)/2, so the point is (1,1). Similarly, to get the maximum window point, we add half the window size; the point is (5,5). Also, consider the pixels up and down, which surrounds the point. We consider all the pixels in the matrix as shown in figure 1.6.

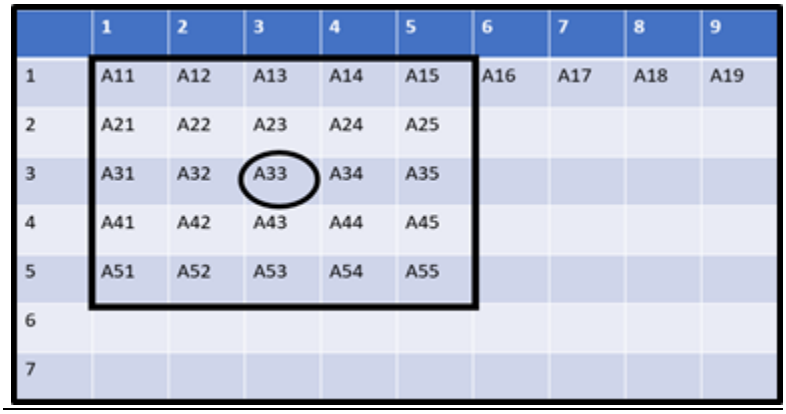


Figure 1.6 Pixel selection Algorithm

4. RESULTS

We look at the resultant images for the CPU and GPU versions of the disparity mapping and also compare the computation in each case.

4.1 CPU IMPLEMENTATION

The results for the CPU implementation for both SAD (Implementation 1) and SSD (Implementation 2) can be observed in the below images, where the disparity values were calculated for different images with different window sizes. The disparity range is set to 100.

The common observation based on the resultant images is that window size 9 gives slightly more distorted image compared to window size 13.

Output 1: Teddy

Input Images

Figure 1.7 Teddy left image.

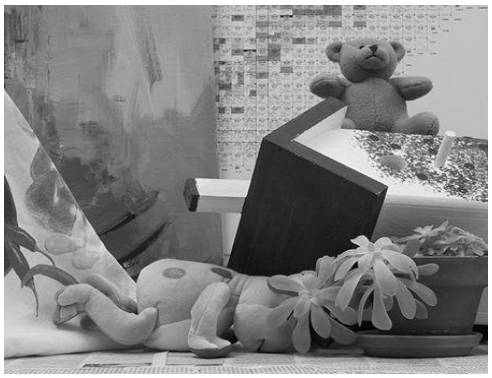


Figure 1.8 Teddy right image.

Window size – 9

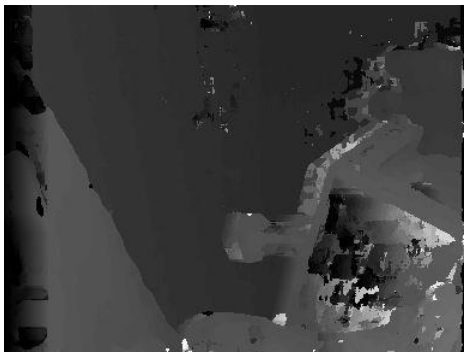


Figure 1.9 Disparity map SAD.

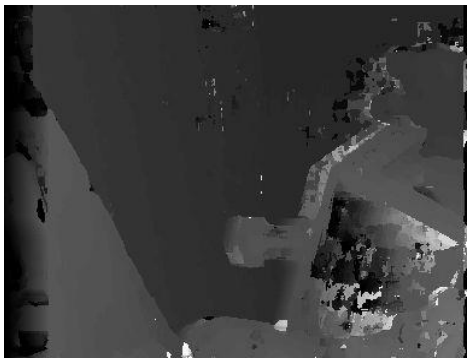


Figure 1.10 Disparity map SSD.

Window size – 13

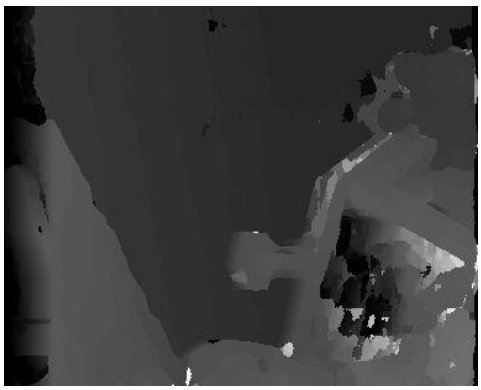


Figure 1.11 Disparity map SAD.

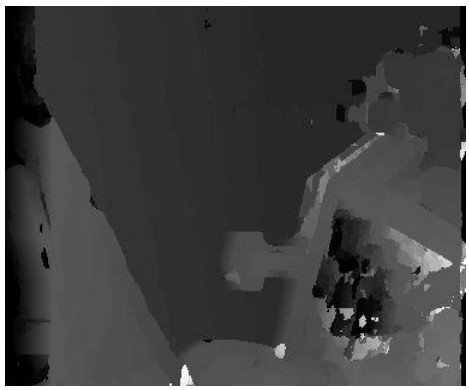


Figure 1.12 Disparity map SSD.

Output 2: Bowling

Input Images

Figure 1.13 Bowling left image.

Figure 1.14 Bowling right image.

Window size – 9

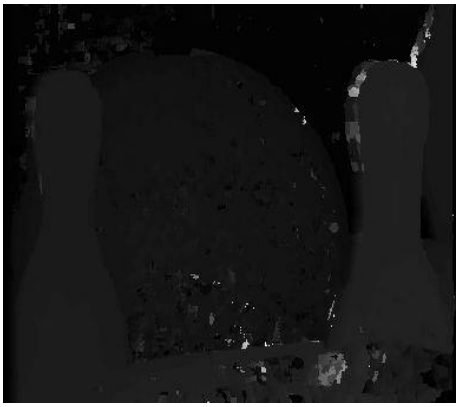


Figure 1.15 Disparity map SAD.

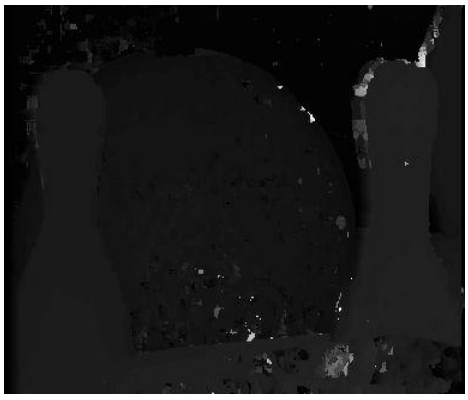


Figure 1.16 Disparity map SSD.

Window size – 13

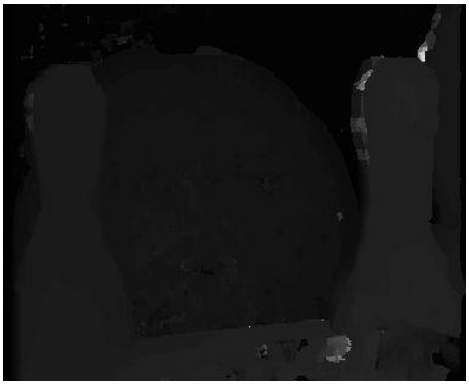


Figure 1.17 Disparity map SAD.

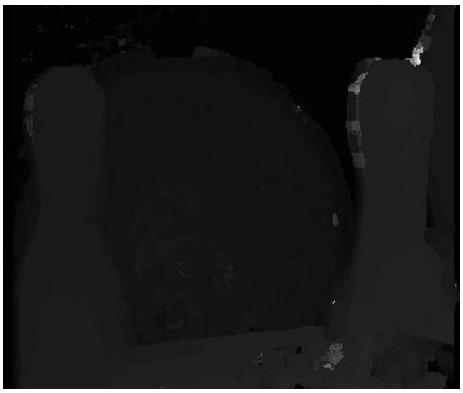


Figure 1.18 Disparity map SSD.

Output 3: Decoration

Input Images

Figure 1.19 Decoration left image.

Figure 1.20 Decoration right image.

Window size – 9

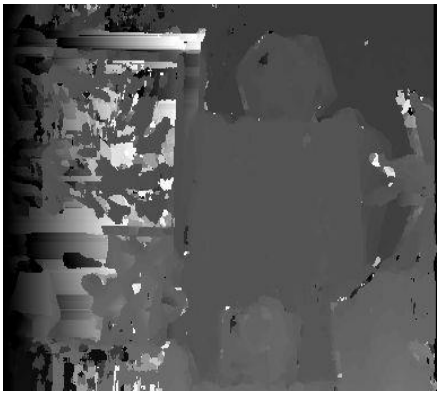


Figure 1.21 Disparity map SAD.

Figure 1.22 Disparity map SSD.

Window size – 13

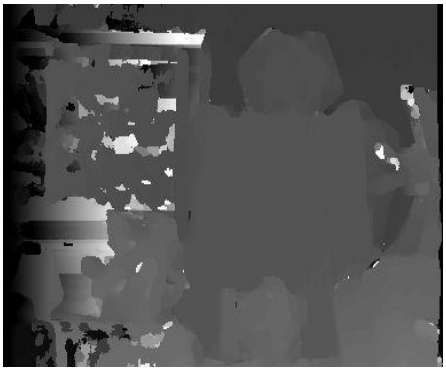


Figure 1.23 Disparity map SAD.

Figure 1.24 Disparity map SSD.

4.2 GPU IMPLEMENTATION

As already discussed in section 3.2, since GPU enables parallel processing, it should have less computation time than CPU and hence the speedup. SSD is more sensitive than SAD because it squares the difference. It also seems to be more susceptible to noise. Due to squaring, large error terms win over (many) minor errors, and hence even a single noisy pixel can screw things.

We analyze the results with two parameters: 1) Disparity Range and 2) Window size.

When the disparity range is too high, we obtain darker images due to a higher grey value corresponding to higher depth. Similarly, when the disparity range is too low, the algorithm can't differentiate well between the objects that are closer than a threshold minimum distance, and the resultant image is too bright to infer the disparity information. This is especially true of regions where there isn't too much intensity variation, and it's easy for the algorithm to get confused. So, we choose **100** (less than half of the image width, more than one-third) as the optimum value for the disparity range. The following example shows the difference in the Disparity Range with Window Size fixed to 13.

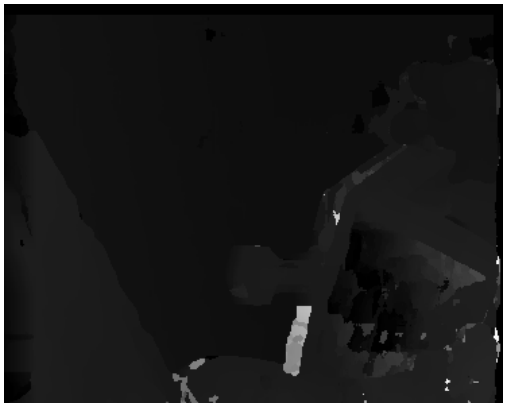


Figure 1.25 Disparity range: 300.

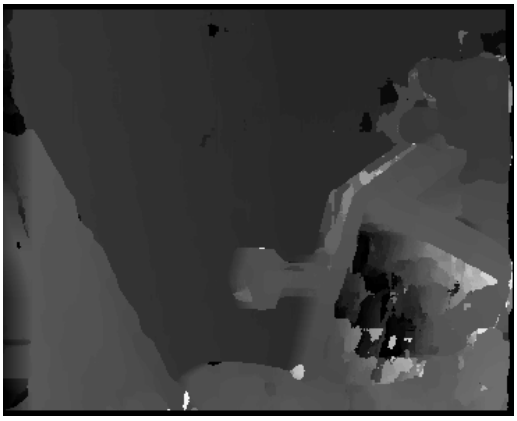


Figure 1.26 Disparity range: 100.

When the window size was fixed at 9, we obtained slightly distorted images comparing to the window size 13. The results are shown below:

Output 1: Teddy

Input Images

Figure 1.27 Teddy left image.

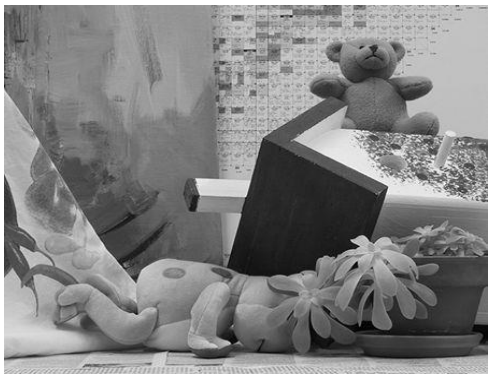


Figure 1.28 Teddy right image.

Window size – 9

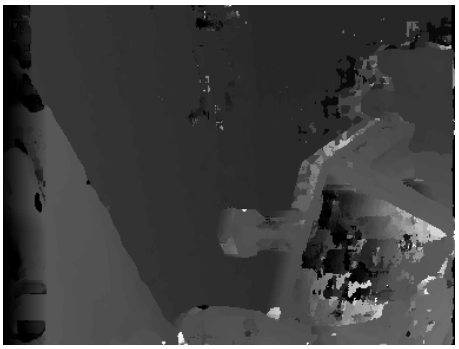


Figure 1.29 Disparity map SAD.

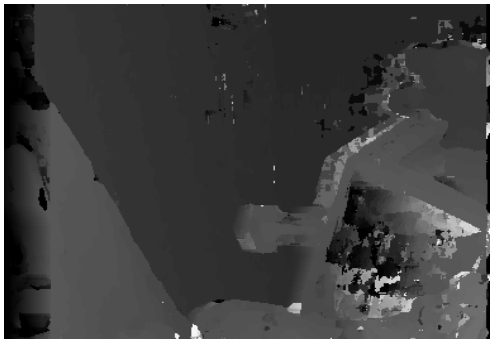


Figure 1.30 Disparity map SSD.

Window size – 13

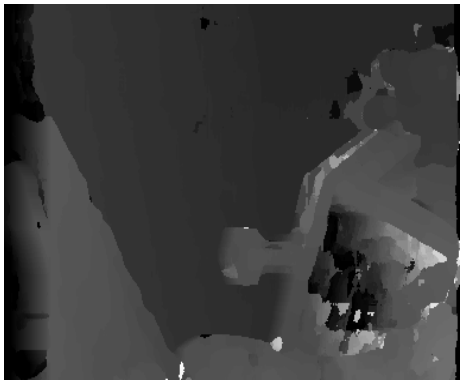


Figure 1.31 Disparity map SAD.

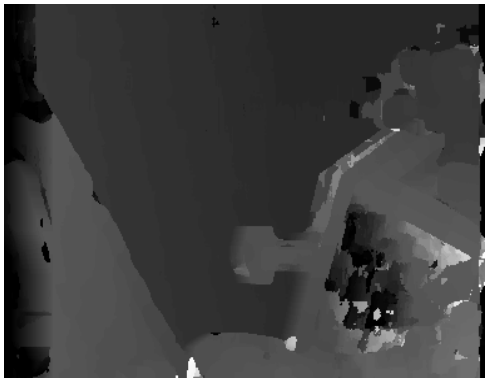


Figure 1.32 Disparity map SSD.

Output 2: Bowling

Input Images

Figure 1.33 Bowling left image.

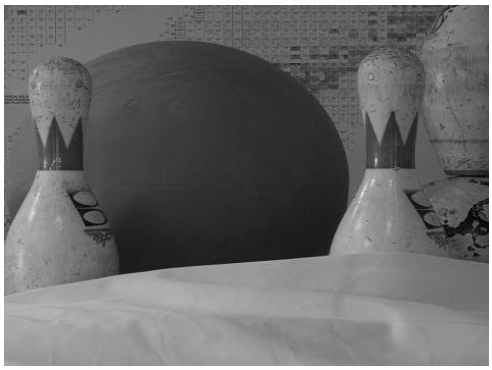


Figure 1.34 Bowling right image.

Window size – 9

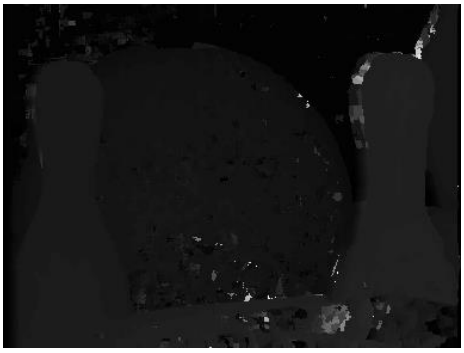


Figure 1.35 Disparity map SAD.

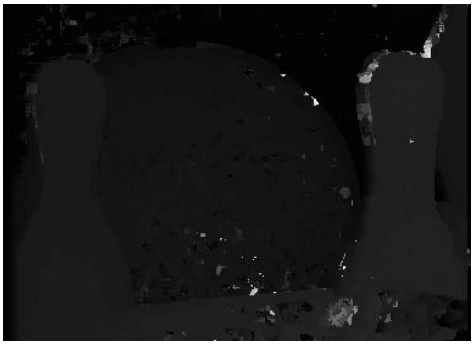


Figure 1.36 Disparity map SSD.

Window size – 13

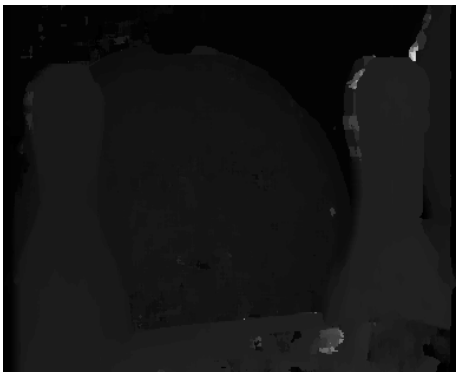


Figure 1.37 Disparity map SAD.

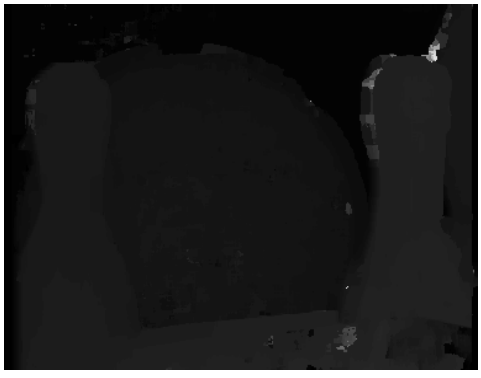


Figure 1.38 Disparity map SSD.

Output 3: Decoration

Input Images

Figure 1.39 Decoration left image.

Figure 1.40 Decoration right image.

Window size – 9

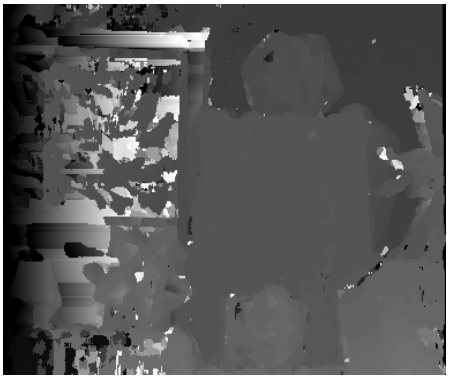


Figure 1.41 Disparity map SAD.

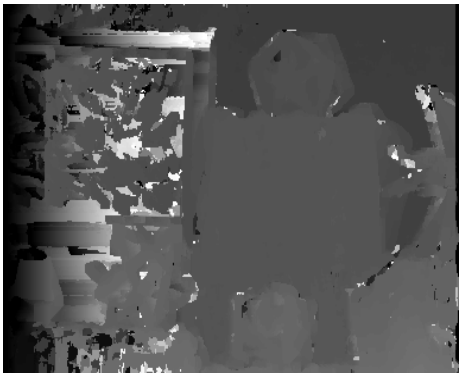


Figure 1.42 Disparity map SSD.

Window size – 13

Figure 1.43 Disparity map SAD.

Figure 1.44 Disparity map SSD.

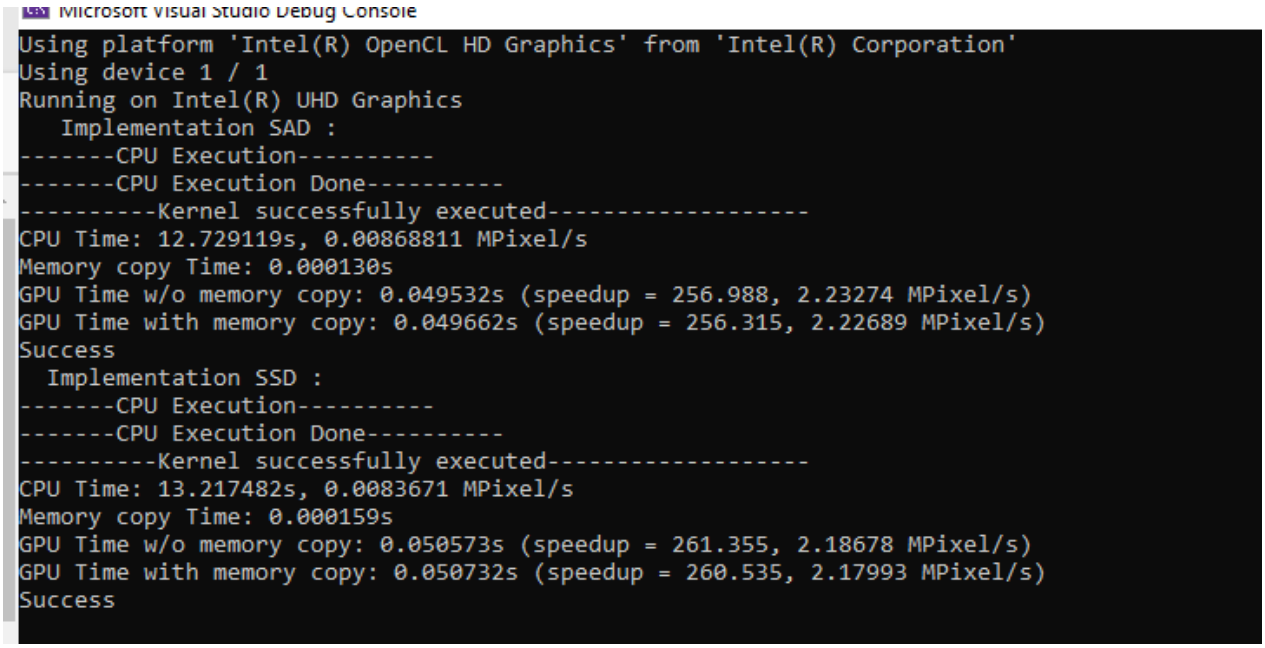
5. PERFORMANCE COMPARISON CPU vs. GPU

Comparing the performance measurements of the disparity calculation from the implementations in the above sections, we can infer that the GPU computation time and speedup are faster than the CPU. Though in some cases, single-core CPUs are faster than individual GPU cores and can manage a versatile set of instructions. When it comes to the multi-core and parallelism that GPU has to offer, GPU definitely bridges the gap to make up for the limited instruction set and the difference in clock speed. GPU is designed to render high-resolution images and video concurrency as well. The difference in the performance can be seen in the images below for each set of Implementation. Implementation 1 is the SAD algorithm, and Implementation being the SSD algorithm.

When computing the disparity using different algorithms in CPU, we can see that the SAD algorithm is slightly faster than the SSD, mainly because of the additional multiplication involved. But in case of GPU, this effect is not observed and both SSD and SAD algorithms have similar computation time, because of parallel processing. and SAD is less prone to outliers.

As an added observation on the window size per se, the computation time increases as the window size increases. Ideally, it has to increase since the larger window size contributes to more iterations, resulting in more computation time.

Teddy- Window size 9



Teddy- Window size 13

```
Microsoft Visual Studio Debug Console
Using platform 'Intel(R) OpenCL HD Graphics' from 'Intel(R) Corporation'
Using device 1 / 1
Running on Intel(R) UHD Graphics
   Implementation SAD :
-------CPU Execution----------
-------CPU Execution Done----------
----------Kernel successfully executed-------------------
CPU Time: 32.066641s, 0.00344882 MPixel/s
Memory copy Time: 0.000133s
GPU Time w/o memory copy: 0.223612s (speedup = 143.403, 0.494571 MPixel/s)
GPU Time with memory copy: 0.223745s (speedup = 143.318, 0.494277 MPixel/s)
Success
  Implementation SSD :
-------CPU Execution----------
-------CPU Execution Done----------
----------Kernel successfully executed-------------------
CPU Time: 31.964495s, 0.00345984 MPixel/s
Memory copy Time: 0.000163s
GPU Time w/o memory copy: 0.224655s (speedup = 142.283, 0.492275 MPixel/s)
GPU Time with memory copy: 0.224818s (speedup = 142.179, 0.491918 MPixel/s)
Success
```

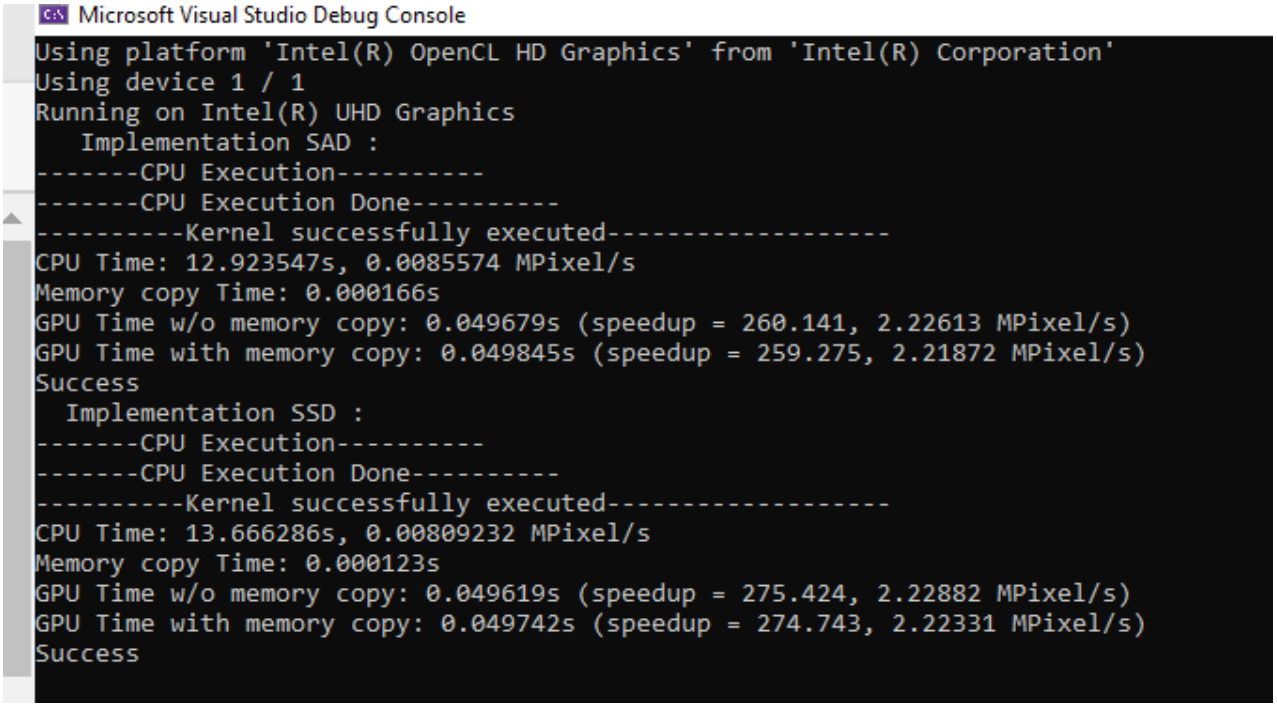
Bowling - Window size 9

```
Using platform 'Intel(R) OpenCL HD Graphics' from 'Intel(R) Corporation'
Using device 1 / 1
Running on Intel(R) UHD Graphics
   Implementation SAD :
-------CPU Execution----------
-------CPU Execution Done----------
----------Kernel successfully executed-------------------
CPU Time: 13.455101s, 0.00821934 MPixel/s
Memory copy Time: 0.000156s
GPU Time w/o memory copy: 0.049627s (speedup = 271.125, 2.22846 MPixel/s)
GPU Time with memory copy: 0.049783s (speedup = 270.275, 2.22148 MPixel/s)
Success
  Implementation SSD :
-------CPU Execution----------
-------CPU Execution Done----------
----------Kernel successfully executed-------------------
CPU Time: 13.129306s, 0.00842329 MPixel/s
Memory copy Time: 0.000149s
GPU Time w/o memory copy: 0.050028s (speedup = 262.439, 2.2106 MPixel/s)
GPU Time with memory copy: 0.050177s (speedup = 261.66, 2.20404 MPixel/s)
Success
```

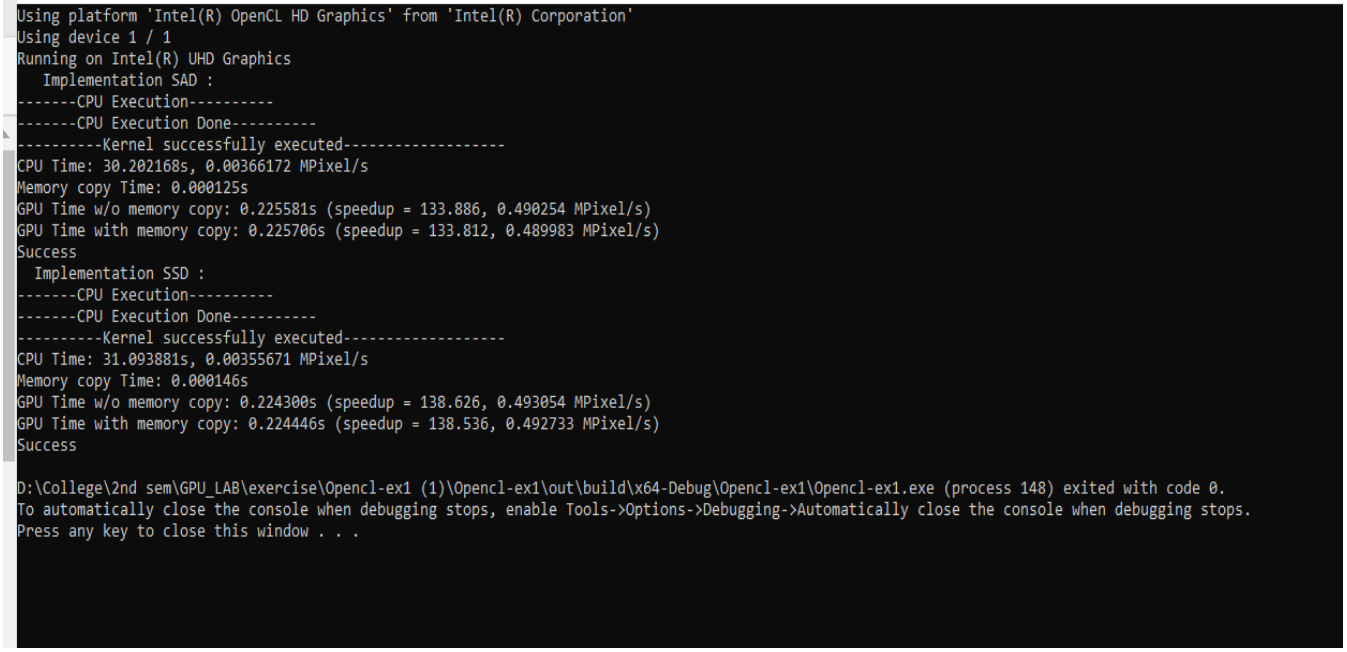
Bowling – Window size 13

```
Using platform 'Intel(R) OpenCL HD Graphics' from 'Intel(R) Corporation'
Using device 1 / 1
Running on Intel(R) UHD Graphics
   Implementation SAD :
-------CPU Execution----------
-------CPU Execution Done----------
----------Kernel successfully executed------------------
CPU Time: 30.541637s, 0.00362102 MPixel/s
Memory copy Time: 0.000131s
GPU Time w/o memory copy: 0.224545s (speedup = 136.016, 0.492516 MPixel/s)
GPU Time with memory copy: 0.224676s (speedup = 135.936, 0.492229 MPixel/s)
Success
  Implementation SSD :
-------CPU Execution----------
-------CPU Execution Done----------
----------Kernel successfully executed------------------
CPU Time: 31.498210s, 0.00351106 MPixel/s
Memory copy Time: 0.000126s
GPU Time w/o memory copy: 0.224195s (speedup = 140.495, 0.493285 MPixel/s)
GPU Time with memory copy: 0.224321s (speedup = 140.416, 0.493008 MPixel/s)
Success
```

Decoration – Window size 9



Decoration – Window size 13



6. CONCLUSION

This project has observations and conclusions that are two-fold, one being the performance analysis between the CPU and GPU, in terms of computation time and speedup. It is noted that the computation time of the GPU is more efficient and faster compared to the CPU, which is ideal due to the parallel function of GPU.

The second one being the window size, it has to be set to the proper value such that it is large enough to enable intensity variation, also small enough to avoid effects of distortion. It is observed that if the window size is too small, then the noise in the images increases, providing a resultant image with more white pixels. But, if the window size is increased, the resultant image is smoother and reduces the greyness. The grey areas of the images depict the depth of the objects, which in turn means that the larger the grey area, the lesser the distance between the object and the camera.

Thus, the two-fold observation has helped us understand the factors that impact the implementation of disparity maps specific to SAD and SSD algorithms.

7. Project Member contribution

Tasks accomplished:

Task 1 – Literature research

Task 2 – Implementation of logic for GPU SSD logic.

Task 3 – Implementation of logic for GPU SAD logic.

Task 4 - Implementation of logic for CPU SSD/SAD.

Task 5 – Debug errors

Task 6 – Result Analysis and Report

Our group with 3 members in the team have equal contribution towards the project, Task 1 - we first started with our own Literature research and understanding the concept of Block matching algorithm and disparity mapping individually. Task 2,3,4 - We divided the tasks on implementation with one for each member. Used git repository to track progress. Task 5- Errors encountered during individual implementation were discussed as a group over webex and resolved. After individual implementation the code was clubbed and results were analyzed collectively. Task 6 – Report on each of our implementation were written individually, proof read and collated collectively.

REFERENCES

[1] J. Zhang, J. Nezan and J. Cousin, “Implementation of Motion Estimation Based on Heterogeneous Parallel Computing System with OpenCL,” 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems, 2012, pp. 41-45, doi: 10.1109/HPCC.2012.16.

[2] Ambrosch K., Humenberger M., Kubinger W., Steininger A. Hardware Implement of an SAD Based Stereo Vision Algorithm. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition; Minneapolis, MN, USA. 17–22 June 2007; pp. 1–6.

[3] Rostam Affendi Hamzah, Haidi Ibrahim, “Literature Survey on Stereo Vision Disparity Map Algorithms”, Journal of Sensors, vol. 2016, Article ID 8742920, 23 pages, 2016. https://doi.org/10.1155/2016/8742920

[4] Aboali, Maged & Abd Manap, Nurulfajar & Darsono, Abd & Yusof, Zulkalnain. (2017). Performance Analysis between Basic Block Matching and Dynamic Programming of Stereo Matching Algorithm. Journal of Telecommunication, Electronic and Computer Engineering. 9. 2289-8131.

[5] Gasim Mammodov, sobel filter and Opencl, High Performance Programming with Graphic Cards Lab Course

[6] https://stackoverflow.com/questions/49524329/disparity-map-block-matching

[7] https://sites.google.com/site/5kk73gpu2010/assignments/stereo-vision

[8] https://www.programmersought.com/article/93105038292/

[9] https://medium.com/analytics-vidhya/significance-of-kernel-size-200d769aecb1

[10] https://johnwlambert.github.io/stereo/