GPU Based Real-time Trinocular Stereovision

by
Yuanbin Yao

A Thesis
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Master of Science
in
Electrical and Computer Engineering

Aug 15, 2012

APPROVED:

Prof. Taskin Padir, Thesis Advisor

Prof. Michael Gennert, Thesis Committee

Prof. Xinming Huang, Thesis Committee

GPU Based Real-Time Trinocular Stereovision

Abstract

Stereovision has been applied in many fields including UGV (Unmanned Ground Vehicle) navigation and surgical robotics. Traditionally most stereovision applications are binocular which uses information from a horizontal 2-camera array to perform stereo matching and compute the depth image. Trinocular stereovision with a 3-camera array has been proved to provide higher accuracy in stereo matching which could benefit application like distance finding, object recognition and detection. However, as a result of an extra camera, additional information to be processed would increase computational burden and hence not practical in many time critical applications like robotic navigation and surgical robot. Due to the nature of GPU's highly parallelized SIMD (Single Instruction Multiple Data) architecture, GPGPU (General Purpose GPU) computing can effectively be used to parallelize the large data processing and greatly accelerate the computation of algorithms used in trinocular stereovision. So the combination of trinocular stereovision and GPGPU would be an innovative and effective method for the development of stereovision application [18].

This work focuses on designing and implementing a real-time trinocular stereovision algorithm with GPU (Graphics Processing Unit). The goal involves the use of Open Source Computer Vision Library (OpenCV) in C++ and NVidia CUDA GPGPU Solution. Algorithms were developed with many different basic image processing methods and a winner-take-all method is applied to perform fusion of disparities in different directions. The results are compared in accuracy and speed to verify the improvement.

Acknowledgement

First of all I would like to express the most gratitude to my advisor, Prof. Taskin Padir. I really appreciate his instruction in both technical knowledge and research method during the past two years, also the opportunity to conduct my research in RIVeR Lab. My experience in RIVeR Lab is a precious property in my life as I benefited a lot from others there.

I am grateful to my thesis committee members, Prof. Michael Gennert and Prof. Xinming Huang, for their time, effort and invaluable advices.

I also need to thank my family who has supported during the past two years. Without my parents, my uncle's family and my girlfriend, I would not be able to get through my most difficult time.

Content

List of Figures......IV
List of Tables......VI
1 Introduction......1
1.1 Stereovision......2
1.1.1 Comparison between Trinocular and Binocular......4
1.2 GPU Architecture and GPGPU......4
1.3 OpenCV Library......6
2 System Overview......8
2.1 System Architecture......8
2.2 Image Processing Section......10
2.2.1 Grayscale Transformation......11
2.2.2 Sobel Filter for Different Matching Directions......12
2.3 Stereo Computation......16
2.3.1 Stereo Matching......18
2.3.2 Fusion of Disparities in two Orientations......25
3 Implementation with OpenCV and CUDA......28
3.1 Code Hierarchy of OpenCV Source Code......28
3.2 CUDA Programming Model......30
3.3 Implementation of Vertical Matching......36
3.4 Optimization and Implementation of Disparities Fusion......43
4 Result Analysis and Conclusions......49
4.1 Accuracy Results......49
4.2 Speed Results......62
4.3 Future Development......65
Bibliography......67

List of Figures

1.1 Stereovision Mathematical Principle....3

1.2 GPU Architecture 4

1.3 Comparison between CPU Architecture and GPU Architecture....5

2.1 A typical L-shape camera array setting 8

2.2 Algorithm Architecture Overview....10

2.3 Center Image of Tsukuba Dataset and Its Fourier Transform 13

2.4 Sobel Filtering Comparison with Tsukuba Center Image 15

2.5 SIFT feature key points extraction on Tsukuba dataset....17

2.6 SURF feature key points extraction on Tsukuba dataset....17

2.7 Horizontal matching result 20

2.8 Result analyses for horizontal matching result on Middlebury Stereo Page 21

2.9 Vertical matching result. 21

2.10 Result analyses for vertical matching result on Middlebury Stereo Page 22

2.11 Comparison of horizontal and vertical matching results 23

3.1 Automatic Scalability of CUDA 30

3.2 Organization of 2-D Grid 31

3.3 Organization of 3-D Grid 32

4.1 Result with sliding window size 8....49

4.2 Result with sliding window size 12 49

4.3 Result with sliding window size 16 50

4.4 Result Analysis with sliding window size 8....50

4.5 Result Analysis with sliding window size 12....51

4.6 Result Analysis with sliding window size 16....51

4.7 Result with disparity distance 9....53

4.8 Result with disparity distance 12....53

4.9 Result with disparity distance 15....54

4.10 Result Analysis with disparity distance 9....54

4.11 Result Analysis with disparity distance 12 55
4.12 Result Analysis with disparity distance 15 55
4.13 Result with fusion ratio 0.6 56
4.14 Result with fusion ratio 0.7 57
4.15 Result with fusion ratio 0.8 57
4.16 Result Analysis with fusion ratio 0.6 58
4.17 Result Analysis with fusion ratio 0.7 58
4.18 Result Analysis with fusion ratio 0.8 59
4.19 Improvement from Single match to Fusion 60
4.20 640×480 Tsukuba Result 62
4.21 800×600 Tsukuba Result 63
4.22 1280×960 Tsukuba Result 63

List of Tables

4.1 Accuracy Result for various sliding window size..53
4.2 Accuracy Result for various approximate disparity distance 57
4.3 Accuracy Result for various approximate disparity distance 60
4.4 Fusion Accuracy Improvement 61
4.5 Accuracy Comparison with Other Stereo Algorithm Implementations 62
4.6 4.6 Times per Frame for Different Size Image..63

Chapter 1

Introduction

A way to refine desired information from data acquired by visual sensor has been long pursued by both academic and industry. This kind of method could help many application and research like traffic control, ISR (Intelligence, Surveillance and Reconnaissance) and robots. For example, this method could help to construct 3-D information around for an UGV (Unmanned Ground Vehicle) in unknown environment from its visual sensor, track a certain object automatically during UAV (Unmanned Aerial Vehicle) flight or build a surgical robot to determine focus accurately during surgical operation.

Computer Vision is just the way to extract information and data from images by processing, understanding and analyzing with computer. Compared to image processing which uses image for both input and output and computer graphic which uses data as input and image as output, computer vision applies image as input and data as output [19]. Traditional image processing technologies apply computation on the image in order to obtain better image quality (e.g. use image filter to get a clearer and higher signal noise ratio (SNR) image) in 2-D space. A lot of work involving intelligence was still needed to be done by human. Limited to hardware technology, image processing used to be the best way to balance between processing quality and economic efficiency. With the development of microprocessor and computational technology, complex computation with large data which would help to solve difficult graphical problem becomes feasible. Hence the topic of advanced processing based on traditional image processing to acquire data in 3-D world raises and attracts interests of many researchers. The evolution of computer vision involves technologies from many fields including image and signal processing, computer graphics and pattern recognition. It is one of fastest and most popular field of research and development in recent years.

1.1 Stereovision

Stereovision is the view of depth for a certain scene based on information captured by multiple cameras focus on this scene [20]. Though applied very early, this topic was first raised formally in late 1970s and now it is one of most popular and efficient methods to transform image of 2-D space to 3-D world compared to many other methods (like laser scanner etc.) today. The application of stereovision includes but not limited to robot navigation, surgical robot, stereo display and 3-D modeling.

For most of computer vision systems, they could be descripted in three levels [1]. This paper will also discuss the approach of stereovision system done by this work in these three levels

- Computational theory: Describe the basic requirements (settings and environments, etc.) and principles (mathematical and physical theory) of the problem. For stereovision as an example, the method is based on the theory of vision disparity.
- Representations and algorithms: Define the input, output and the detail algorithm solution for a certain problem. For stereovision, input is a pair (or pairs) of image with disparity and output is depth image. The key for the problem is the stereovision algorithm. Different algorithms various in performance of stereo matching accuracy and processing speed.
- Hardware implementation: Provide the system which combines algorithm and actual hardware optimally to apply in the real world. Normally the hardware of a stereovision system includes two or more cameras and a computing system (CPU, GPU and FPGA etc.).Axes of cameras are parallel in most cases though some researches have started to reach unparalleled axes case. Also in this project the GPU is employed.

The basic method of stereovision is the bionics of human eyes. Human eyes could perceive real 3-D scene as the arrangement of human eyes are horizontally aligned and the position difference of same objects in left and right eyes would determine the disparity of the scene. In a similar way, stereovision means acquiring a depth grayscale image of a scene based on the images of the scene from two or more cameras. Figure 1.1 shows the principle of stereovision.

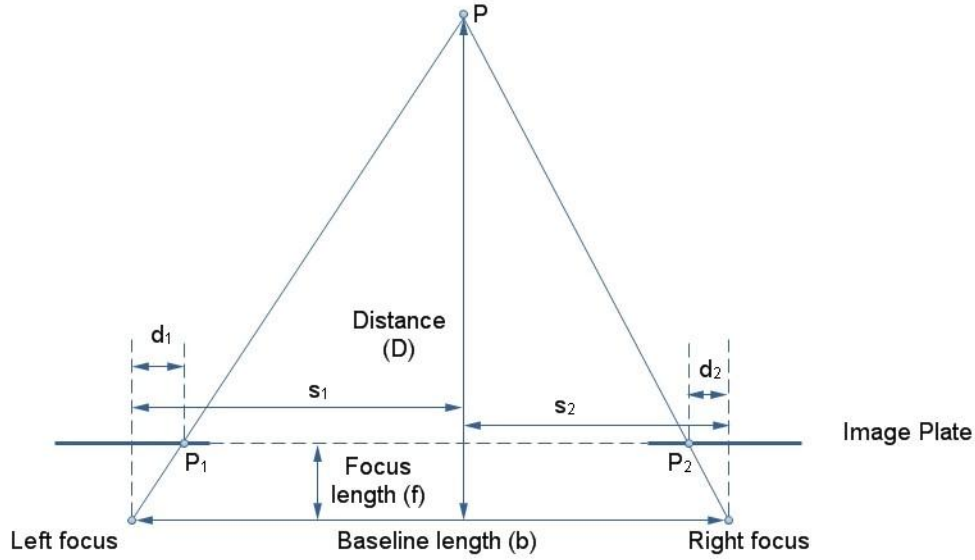


Figure 1.1: Stereovision Mathematical Principle

P is an object point in the scene. P_1 and P_2 are corresponding pixel of P in the left and right image. We assume the two cameras are parallel and in the same horizontal plane. Normally we use epipolar to describe the connection line of P_1 and P_2. As the cameras are in horizontal lines, the epipolar of P_1 and P_2 should also be in same height level. This constraint is called epipolar constraint. According to the basic similar triangle principles in geometry, we could have equation

$$\frac{d_1}{s_1} = \frac{d_2}{s_2} = \frac{f}{D}$$

We define disparity of the image as d which $d = d_1 + d_2$, baseline length of two cameras as b, then we could get the following conclusion

$$s_1 = \frac{d_1 D}{f}, s_2 = \frac{d_2 D}{f}, b = s_1 + s_2, d = d_1 + d_2$$

$$D = \frac{bf}{d}$$

As baseline length and focus length are pre-set and known, we could determine the distance of different objects in the same scene from their disparity in two images. This is the mathematical principle of stereovision and the physical meaning of the disparity image as the output of stereovision.

From the description above we could conclude that in stereovision, the most important problem is to have an algorithm which could find the corresponding pixel pairs in two images with high accuracy and efficiency. There are many approaches for this problem and most research work on stereovision focus on this topic.

1.1.1 Comparison between Trinocular and Binocular Vision

The stereovision algorithm discussed above is binocular stereovision which employed a pair of horizontal camera array. It is the most common type of stereovision application today. However, the information from two cameras is limited to solve all challenges in stereovision problems like objects occlusion and light condition differences. The algorithms become slower and slower as accuracy goes higher and higher. To break through the limits, adding more cameras in stereovision has been explored in the past [2], [3]. Previous research on trinocular stereovision has provided the following results

- Based on different settings and environments (like baseline length, camera resolution and light condition), trinocular stereo match accuracy would improve 7% - 54%
- Computational cost for the processing of the third camera would increase the overall computational cost around 25%

From this conclusion we would know that trinocular is superior to the binocular on stereovision accuracy. But twenty years ago, computational ability was not as powerful as it today. 25% more computational cost would influence processing speed performance a lot hence stop applying trinocular stereovision in many time critical applications.

There is already some previous work try to approach real-time trinocular stereovision [4], [5], [6] which achieve inspired results. The algorithm could run in almost real-time with an acceptable accuracy. However these works were implemented with CPU or FPGA. The resolution of camera was also restricted to a basic level in order to achieve real-time.

1.2 GPU Architecture and GPGPU

Graphics Processing Unit (GPU) is the device specialized to process image data and graphical computing. The task of GPU determines its architecture significantly differs from

Central Processing Unit (CPU).

The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for compute-intensive, highly parallel computation – exactly what graphics rendering is about – and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control [14]. CPU needs to execute all kinds of instructions to maintain the operation of the entire system. As a result, though multi-core CPU is well developed to meet the requirement of high parallelized computation, its architecture is Multi-Instruction-Multi-Data (MIMD) which means different cores would execute different instructions and different data at the same time. So each core needs an independent set of complex logic component to support it. This restricted the number of cores in a CPU. GPU is solely for processing large scale data with same operation; therefore, GPU architecture is Single-Instruction-Multi-Data (SIMD). The cores in GPU may be in several groups and cores in each group share same set of logic component. In this way, each group execute different instructions while cores in one group would execute the same instruction but with different input data. The concepts of 'core' in CPU and GPU are different hence number of cores in a GPU normally would be dozens of times than the number of cores in a CPU. Based on different architectures, it is clearly that GPU would be better fit for image processing and computer vision computation than CPU. The Figure1.2 shows GPU architecture. A logic component controls same operation of the cores in its group. Also Figure 1.3 shows a comparison between CPU architecture and GPU architecture. It is obviously from the figure that GPU could support more cores based on its structure.

Figure 1.2: GPU Architecture

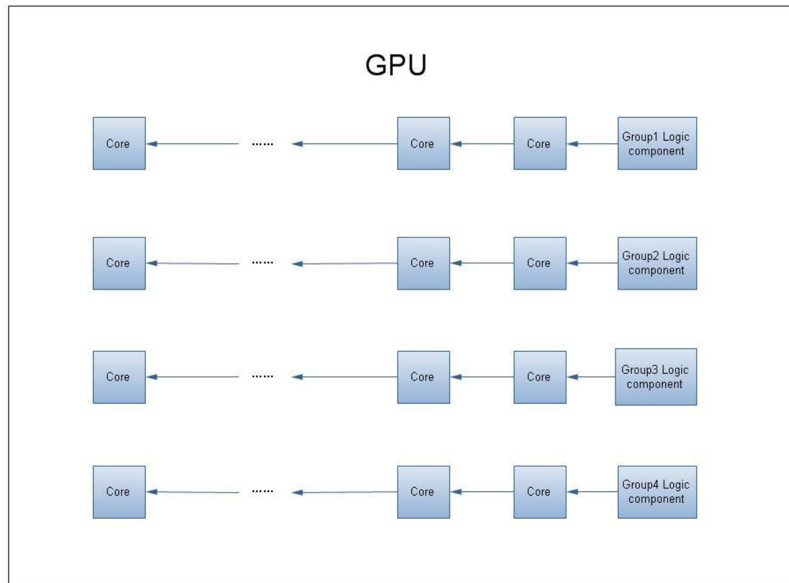


Figure 1.3: Comparison between CPU Architecture and GPU Architecture [14]

GPU was long considered as merely a toy for entertainment industry like video games and movies since it was introduced. This situation changes a lot now as General Purpose Computing on GPU (GPGPU) is widely applied in large scale computing. GPGPU takes advantage of GPU architecture to speed up large scale computing in many fields.

At the beginning, GPGPU was difficult to use for many reasons. GPU programming model is different from CPU, GPU programming environment is tightly constrained and code from CPU cannot be easily transferred to use on GPU. Also it needs to design parallelism well in order to maximize performance [7]. However, many libraries have been released to make GPGPU easy, including NVidia's Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL).

1.3 OpenCV Library

OpenCV stands for Open Source Computer Vision Library which includes functions mainly for development of real-time computer vision algorithms for research and applications. According to open accessible documentation online, "it was initially an Intel Research initiative to advance CPU-intensive applications, part of a series of projects including real-time ray tracing and 3D display walls" and first launched in 1999. After five beta

versions during five years of 2001-2005, its first formal version 1.0 was released in 2006. Since then the change log of OpenCV is available on its website [21]. In the early version of OpenCV, the only support programming language was C. A significant feature of version 2.0 released in 2009 was introduction of C++ interface together with many new functions. Later versions also add support for Python scripting language. The latest version of OpenCV is 2.3.1 which released August 17, 2011 (a newer version 2.4.0 was released on May 7, 2012 at the writing of this manuscript). OpenCV supports GPGPU (General Purpose GPU) as a CUDA based GPU module was introduced since December 2010 (version 2.2).

This work is implemented based on OpenCV 2.3.1 with the integrated CUDA module. Necessary modifications have been done to the source code of OpenCV. Results will be evaluated from speed and accuracy and expected to reach real-time speed with a

Chapter 2

System Overview

This chapter describes the whole architecture including camera setting and hardware implementation and algorithm applied in this project which corresponds to the second level (representations and algorithms) according to the description of levels in computer vision system [1].

2.1 System Architecture

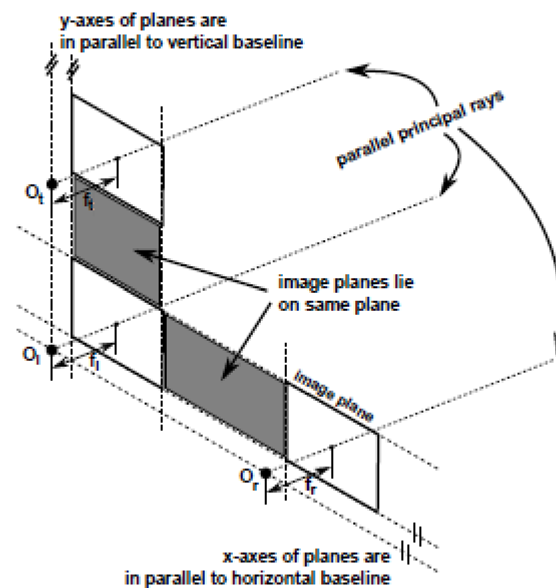


Figure 2.1 A typical L-shape camera array setting [6]

The goal for this project is to obtain real-time high accuracy stereo disparity image with trinocular system as Figure 2.1.

There are different and multiple definitions for real-time. Generally based on persistence of human vision and practical application needs, an accepted processing speed for real-time is

25Hz-30Hz frames per second (fps). This is the definition of real-time goal for this project though in some cases like high speed photography, real-time requires higher frequency. To reach the goal of real-time within this definition, for a single still image (function as one frame), the processing time is limited to 33ms-40ms. From the experience of previous research [6] [8], traditional hardware method like CPU and FPGA can hardly reach real-time or have to restrict camera resolution to reach real-time. So GPU Programming is a vital method to solve the problem.

The most novel point of this project compared to traditional binocular stereovision is the combination of GPU computation and introduction of the third camera [25][26]. Therefore, one important problem rises is how to determine the position of the third camera. Three methods of camera arrangement were considered for the project: inline arrangement that all three cameras are set in a line with parallel focus line; right triangular arrangement that all three cameras are set in three vertexes positions of a right triangular and L-shape arrangement that one camera locates in the center while one in its left (right) and one in its top. In all cases, the configurations of three cameras are same and the axes of three cameras are all parallel. This setting could make all image planes parallel hence avoid unnecessary image transformation to eliminate images distortion caused by angles between camera axes which would slow processing speed a lot [6].

Among the three arrangements, inline arrangement provides a longer epipolar line which could benefit the accuracy. However there is less common overlap area among three input images hence the fusion result will be impaired. The right triangular arrangement would provide a good vertical view and the biggest common overlap area. But as the epipolar line of the vertical pair is neither vertical nor horizontal, as a result, additional image transformation is needed and speed performance would be influenced. Based on the analysis, the arrangement of cameras in this project is in L-shape style.

The algorithm for this project takes four major steps to reach the final goal: grayscale transformation from colorful images, features enhancement filtering to fit different orientation matching with two types of Sobel Filter, horizontal and vertical block matching with Sum of Squared Difference (SSD) algorithm and finally fuse horizontal and vertical disparity image together based on probabilistic to get the final result image. All steps, except first step which

is a long existed general method in image processing, involve changes to OpenCV source code.

The basic design architecture of system is shown in the Figure 2.1. Tsukuba Stereo Dataset is used as an example to demonstrate the general outline [22].

The system is tested in Middlebury Stereovision (vision.middlebury.edu/stereo) which is the authoritative site in evaluation of stereovision algorithm. The evaluation will cover accuracy based on and speed performance.

2.2 Image Processing

From the description of system in Section 2.2, we could divide the whole work to two parts: image processing and stereo computation. Image processing section employs basic traditional image processing measure like grayscale transformation and image filtering on the input images. The purpose of this section is to do preprocess on the input images before applied to stereo computation. This preprocess would extract useful features and information such as spots and lines in certain location and direction from input images which would help to improve stereo computation performance as stronger features extracted would be more distinct to match.

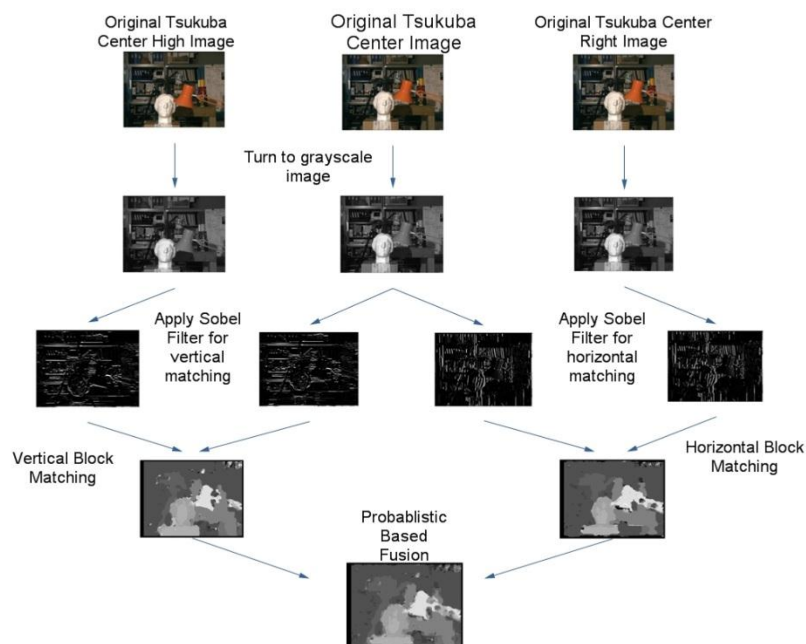


Figure 2.2: Algorithm Architecture Overview

As generally known, the digital images are stored in matrix form in computer. Also image data could be viewed as a 2-dimension signal data. Based on these features, traditional image processing methods mostly function like matrix computation or 2-D signal processing. For example, image filtering with 2-D filter is widely applied to reduce noise in the images.

2.2.1 Grayscale Transformation

Color model or color space is an important concept in image processing. Generally it is a mathematical model to present all colors. There are many different color models in digital images to interpret colors from different aspects. As normally there would be several different types of values in a color model, colorful image generally are multi-channels rather than single channel grayscale image. Different color models are based on different perspectives of color. One common color model, RGB color model, is based on additive color. For an image employs RGB color model, a pixel contains three values which stand for red value, green value and blue value. Another color model, HSV color model, is based on human vision color organization [1].

Though colorful stereo disparity image is already applied in some products like Microsoft Kinect, grayscale disparity image is still a fundamental and useful evaluating measure in research and development as it is the simplest image format. For this matching algorithm, transforming input images to grayscale is the very first step.

The test dataset (PPM format) are stored in RGB color space. The conversion from RGB color space to grayscale follows the Equation 2.1.

$$x = (R^{gamma} * Weight_R + G^{gamma} * Weight_G + B^{gamma} * Weight_B)\hat{}(1/gamma)$$

$$Weight_R + Weight_G + Weight_B = 1 \quad \text{(Equation 2.1)}$$

Where x stands for grayscale value; R, G and B stand for red, blue and green values of the pixel. $Weight_R$, $Weight_G$, $Weight_B$ are the weights values for R, G, B in the conversion and gamma is the gamma value for gamma correction. There would be minor difference for these four parameters between different RGB color spaces like sRGB, Adobe RGB and Apple RGB.

For example, for Adobe RGB, the four parameters are

$$gamma = 2.2, Weight_R = 0.2973, Weight_R = 0.6274, Weight_R = 0.0753$$

While for Apple RGB, the four parameters are

$$gamma = 1.8, Weight_R = 0.2446, Weight_R = 0.6720, Weight_R = 0.0833$$

In this project, the grayscale transformation is done with OpenCV function cvtColor.

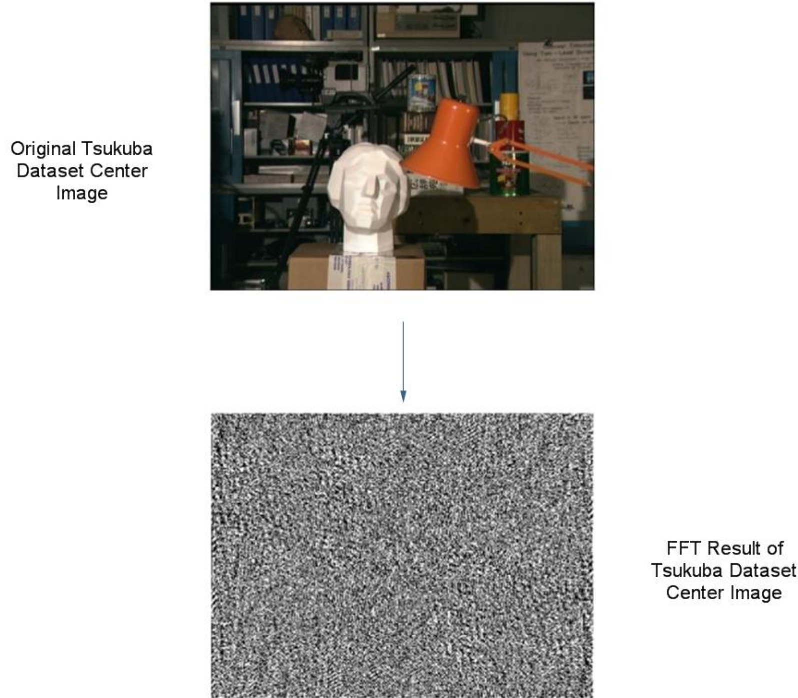
2.2.1 Sobel Filter for Different Matching Directions

As discussed in Section 2.1, 2-D filter and image filtering are widely applied in image processing. Besides reducing noise in the images, some other important applications of image filtering including extracting specific information and achieving certain effect of the images could build a good foundation for further advanced process.

Images, which can be viewed as 2-D signal, could also be transformed to frequency domain for analysis just like 1-D signal. Generally in images, high frequency area represents details and edges while low frequency area represents area with few details or consist plain color [1]. Therefore, frequency domain analysis could distinguish different area in an image. Based on this point, also reference from 1-D filter design, we could design 2-D filter for specific needs like enhance the feature in a certain direction. Figure 2.3 shows the center image of Tsukuba dataset and its 2-D Fourier Transform result.

In this project, stereo matching is in horizontal and vertical directions. The epipolar constrains are different for the two directions. For traditional horizontal matching, the epipolar line is in horizontal direction and matching block also moves along this direction. With this circumstance, features in vertical direction are easier to recognize and compute. Similarly for vertical matching, as the epipolar line is vertical, hence features in horizontal direction would be more helpful for stereo matching. We could illustrate this point by an example.

Figure 2.3: Center Image of Tsukuba Dataset and Its Fourier Transform



We want to determine a line-shape feature, F, in the depth image. We could consider the same feature in horizontal and vertical circumstances to find the difference. The grayscale values for this feature is

$$F = \begin{bmatrix} 40 & 40 & 40 & 40 & 40 \\ 80 & 80 & 80 & 80 & 80 \\ 40 & 40 & 40 & 40 & 40 \end{bmatrix} \text{ (Equation 2.2)}$$

In Equation 2.2 each value stands for a grayscale value of a pixel. Then we put F into a pair of horizontal disparity images I_1 and I_2 following the horizontal epipolar constraint.

$$I_1 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 40 & 40 & 40 & 40 & 40 & 0 \\ 80 & 80 & 80 & 80 & 80 & 0 \\ 40 & 40 & 40 & 40 & 40 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad I_2 = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 40 & 40 & 40 & 40 & 40 \\ 0 & 80 & 80 & 80 & 80 & 80 \\ 0 & 40 & 40 & 40 & 40 & 40 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \text{ (Equation 2.3)}$$

Normally matching block size is set to an odd number in order to locate a center pixel. For this case if using a 3×3 window, there would be 3 exact match for the pixel F (2, 2) as the

block moves horizontally. If the length of F is much larger than block window size (consider F is 3×100), this would lead to a bad matching result. However, for F′

$$F' = \begin{bmatrix} 40 & 80 & 40 \\ 40 & 80 & 40 \\ 40 & 80 & 40 \\ 40 & 80 & 40 \\ 40 & 80 & 40 \end{bmatrix} \quad \text{(Equation 2.4)}$$

While F′ embedded in a pair of horizontal disparity images I_3 and I_4 following the horizontal epipolar constraint, for each pixel in F′, there is only one exact match in the disparity image pair.

$$I_1 = \begin{bmatrix} 40 & 80 & 40 & 0 & 0 \\ 40 & 80 & 40 & 0 & 0 \\ 40 & 80 & 40 & 0 & 0 \\ 40 & 80 & 40 & 0 & 0 \\ 40 & 80 & 40 & 0 & 0 \end{bmatrix} \quad I_2 = \begin{bmatrix} 0 & 0 & 40 & 80 & 40 \\ 0 & 0 & 40 & 80 & 40 \\ 0 & 0 & 40 & 80 & 40 \\ 0 & 0 & 40 & 80 & 40 \\ 0 & 0 & 40 & 80 & 40 \end{bmatrix} \quad \text{(Equation 2.5)}$$

Similar result can be found in vertical matching. We could conclude that vertical features would be more helpful for horizontal match while horizontal features would benefit vertical match more. Therefore, we should design image filter to enhance features in different orientations for two match directions.

In frequency domain of images, high frequency area presents details and edges. To design a 2-D filter for horizontal matching which enhance features in vertical direction, the filter should be high-pass in vertical direction and low-pass in horizontal direction. Comparatively, the filter for vertical matching would be low-pass in vertical direction and high-pass in horizontal direction.

Sobel filter is a discrete differentiation filter computing an approximation of the opposite of the gradient of the image intensity [9]. Typically a Sobel filter for horizontal matching is a 3×3 template matrix as

$$S_h = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \text{(Equation 2.6)}$$

It is easy to conclude from the filter that the center pixel value of a block window from convolution result presents horizontal derivative and vertical smoothing as features in vertical direction are enhanced. With these characteristic, Sobel Filter is a very suitable method for edge detection and feature enhancement. Similarly we could have the filter for vertical

matching.

$$S_v = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \text{(Equation 2.7)}$$

In the project, S_h is convoluted with center image and horizontal image for horizontal matching optimization and S_v is applied with center image and vertical image. To filter the input image with specific filter, follow the Equation 2.8.

$$I_{out}(x,y) = \sum_{i=x-1}^{x+1} \sum_{j=y-1}^{y+1} I_{in}(i,j) \times S(i-x+2, \mathrm{j}-\mathrm{y}+2), x \in (1, height) and\ y \in (1, width)$$

$$I_{out}(x,y) = 0 \text{ when } x = 1\ or\ height\ or\ y = 1\ or\ width \quad \text{(Equation 2.8)}$$

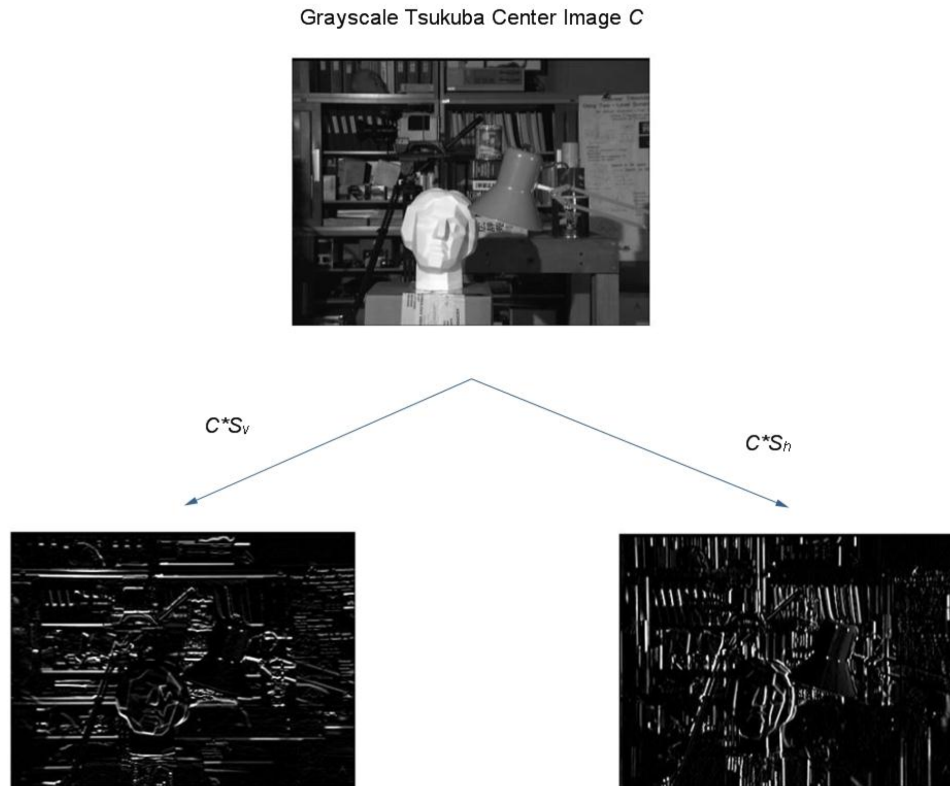


Figure 2.4 Sobel Filtering Comparison with Tsukuba Center Image

In a sub window, each pixel multiply with its corresponding factor and the sum of all elements would be the new pixel value for the center coordinate. The result samples for both filters with center image are shown in Figure 2.4. The left one is the image filtered with

Equation 2.7 while the right one is the image filtered with Equation 2.6

In OpenCV source code, horizontal Sobel filtering already exists. However, the pre-filtering function with S_v for vertical matching is needs to be implemented. The details of this implementation will be discussed in Chapter 3.

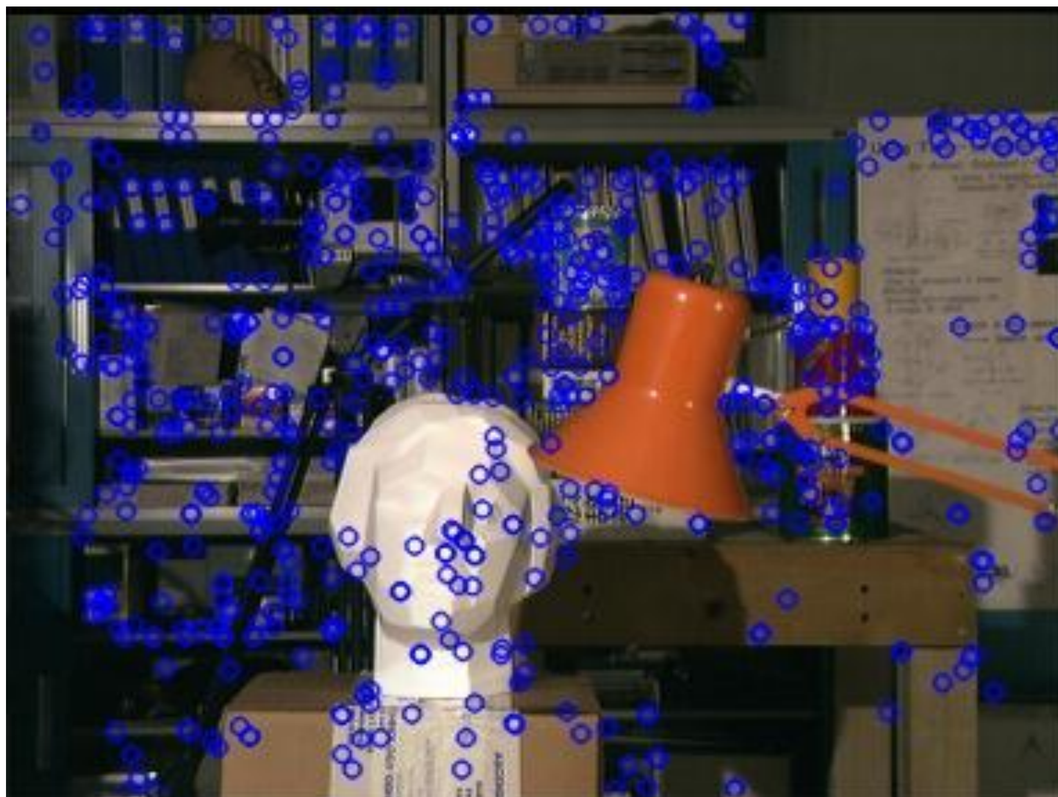
2.3 Stereo Computation

As mentioned in Chapter 1, the most difficult and important problem of stereovision is finding the matching pixel in both images. Most current stereovision research work focus on this topic [6], [8]. Briefly, stereo computation can be categorized into feature based matching and density matching.

Feature based matching is a kind of extended application from image feature descriptor. The concept of feature descriptor in image processing and computer vision is used to describe information related to the certain pixels or the structure of the image. For a feature descriptor, extracting algorithm would find the points of interest and mark it as feature key points. Though new feature descriptors are continuously being developed and researched, most popular and widely used feature descriptors today are SIFT (Scale-Invariant Feature Transform) [10], SURF (Speeded Up Robust Feature) [11], FAST [23] and ORB (Oriented FAST and Rotated BRIEF) [24]. The examples of SIFT and SURF features extraction is shown in Figure 2.5 and Figure 2.6 where blue circles represent feature key points.

Two vectors are composed as collection of feature key points extracted from two images. The basic method for feature based stereo matching is comparing two feature key point vectors to find corresponding key points. The disparity distance of corresponding key points would be found hence depth could be calculated. The depth of other pixels in the image is calculated based on constraints like continuity and epipolar.

Figure 2.5 SIFT feature key points extraction on Tsukuba dataset



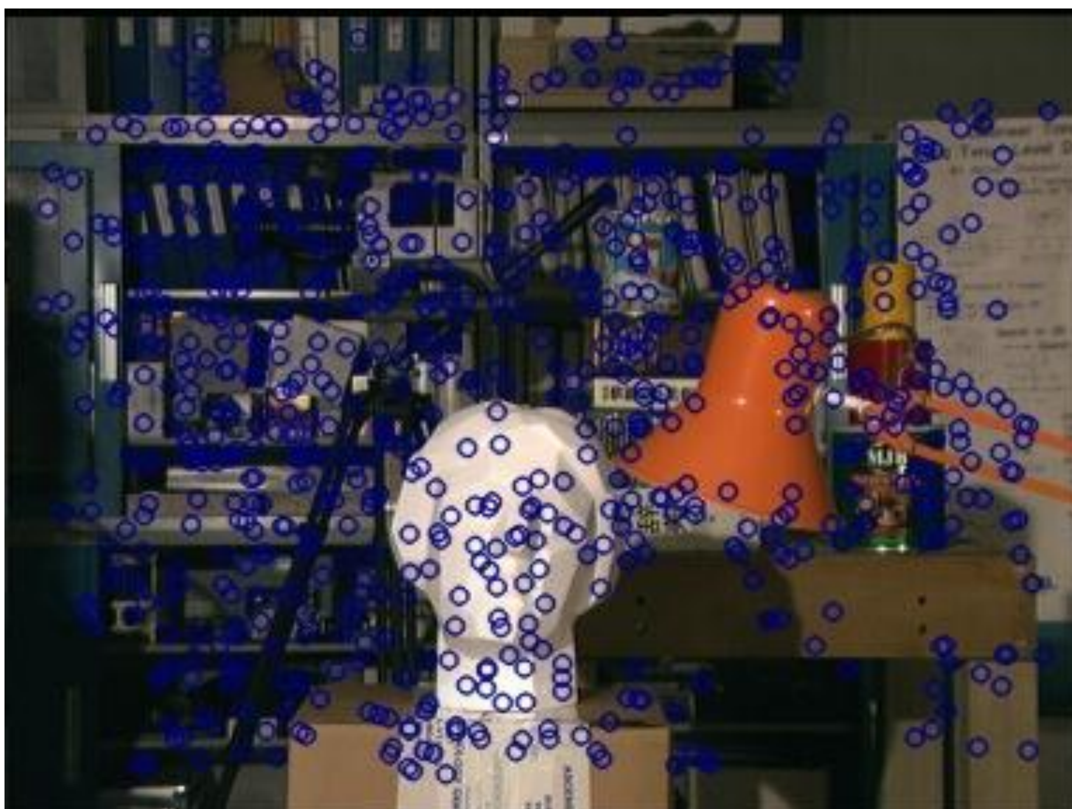


Figure 2.6 SURF feature key points extraction on Tsukuba dataset

As another approach to stereovision, density stereo matching is based on the density value of each pixel and its surrounding pixels. Density stereo matching algorithm would find the corresponding pixel for each pixel in original image from reference image. According to the depth value of each pixel, the depth image is constructed. Also, some researches try to make new methods to combine the two approaches [12].

There are advantages and disadvantages for both methods. For feature based stereo matching, as method of feature descriptor and matching between feature key points is well developed, the accuracy of depth image is higher. However, as computation for feature key points is a complex task and consumes a lot of time, the speed performance of most implementations based on this method is disappointing [8]. For density stereo matching, due to the fact that density value is influenced by many factors like brightness, color, lighting condition and texture, the accuracy of density stereo matching is not as high as feature based method. However, as computation of most density matching algorithm is simple (mostly addition ad multiplication computation) and independent by each sub window, the algorithm can be highly parallelized. With help of hardware with specific architecture, the speed performance is satisfying. Therefore, density stereo matching is a suitable choice for time critical stereovision task like robot application.

Considering real-time requirement and the help of additional camera, traditional density matching is selected for this project.

2.3.1 Stereo Matching

The computation of density stereo matching focuses on density value of each pixel and the numerical relation with its surrounding pixels. In [1], all stereo algorithms are divided into four major steps:

1. Matching cost computation;
2. Cost (support) aggregation;
3. Disparity computation and optimization;
4. Disparity refinement.

Within this manner, Sum of Squared Difference (SSD) algorithm could be described as below

1. The matching cost is the squared difference of intensity values at a given disparity.

2. Cost aggregation is done by summing the matching cost over square windows with constant disparity.

3. Disparities are computed by selecting the minimal (winning) aggregated value at each pixel.

There is a similar approach applying absolute value of pixel difference instead of squared value of pixel difference is called Sum of Absolute Difference (SAD) algorithm. Assume I_1 and I_2 are input images pair, the equation is

$$\mathrm{SSD} = \sum_{i=x}^{x+2r} \sum_{j=y}^{y+2r} [I_1(i,j) - I_2(i,j)]^2 \quad \text{(Equation 2.8)}$$

$$\mathrm{SAD} = \sum_{i=x}^{x+2r} \sum_{j=y}^{y+2r} |I_1(i,j) - I_2(i,j)| \quad \text{(Equation 2.9)}$$

In Equation 2.8 and Equation 2.9, (x, y) stands for the coordinate of most up-left pixel of the sliding window and r stands for radius of sliding window.

A widely applied method of SSD algorithm is implemented with sliding window. A certain size window fixes the elements within it in the reference image, and a same size window moves in the contrasting image along the disparity direction within a distance comparable to known camera baseline length. The direction depends on the epipolar restraints. For example, in the horizontal case and left image is set to reference image, the direction of sliding window move for contrasting image is rightward as based on epipolar restraints the pixel shows in reference image normally appears in the right of corresponding position in contrasting image. The elements in the windows of reference image and contrasting image will compute SSD accordingly and find the minimum one. The corresponding position of minimum SSD will composite disparity image and the minimum SSD value will also be recorded. To describe the algorithm based on horizontal matching case in pseudo code

for i∈[1, height-2r], j∈[1+d,width-2r]

for t∈[0,d]

if q-t∈[1,width]

$$\mathrm{SSD(t)} = \sum_{p=i}^{i+2r} \sum_{q=j}^{j+2r} [I_1(p,q) - I_2(p,q-t)]^2;$$

end for

minSSD(i, j) = min(SSD); disparity(i, j) = find(SSD==min(SSD));

end for

where d stands for an approximately possible disparity range.

Consider the example below where I_1 stands for left image as reference image while I_2 stands for right image as contrasting image.

$$I_1 = \begin{bmatrix} 23 & 50 & 85 & 75 & 24 & 32 \\ 54 & 16 & 162 & 46 & 54 & 82 \\ 64 & 6 & 85 & 216 & 58 & 76 \\ 137 & 52 & 45 & 34 & 91 & 84 \\ 81 & 46 & 248 & 57 & 81 & 79 \\ 31 & 18 & 97 & 31 & 82 & 97 \end{bmatrix} \qquad I_2 = \begin{bmatrix} 88 & 72 & 24 & 30 & 83 & 73 \\ 167 & 48 & 56 & 76 & 159 & 49 \\ 87 & 214 & 61 & 77 & 89 & 212 \\ 49 & 29 & 87 & 81 & 48 & 35 \\ 251 & 60 & 82 & 75 & 252 & 56 \\ 96 & 33 & 80 & 99 & 96 & 36 \end{bmatrix}$$

For the fixed window in reference image I_1, we could mark the position of each window in contrasting image I_2 as p_1, p_2, p_3 and p_4. The SSD for each position respectively is 58979, 36306, 71 and 62611. With the SSD data is could easily determine the corresponding position in contrasting image to the reference image is p_3 as the SSD minimum vale is at p_3. Normally the edge pixels would be padding with zero.

In the system, both horizontal and vertical matching pairs need to set a same reference image in order to make all objects in the scene are in same reference coordinate. The object position coordinate in the final disparity image is related to the position appear in reference image. It is essential to make objects in two disparity images in same position coordinate as it is a prerequisite to implement fusion of two disparities.

As a result of same reference image, the sliding window of vertical matching moves along a different direction. According to the epipolar restraints of vertical matching, a certain pixel in low image would appear at a lower position in high image. Base on the horizontal matching algorithm, we could describe the algorithm for vertical matching case in pseudo code

for i∈[1, height-2r-d], j∈[1,width-2r]

for t∈[0,d]

if q-t∈[1,width]

$$\mathrm{SSD(t)} = \sum_{p=i}^{i+2r} \sum_{q=j}^{j+2r} [I_1(p,q) - I_2(p+t,q)]^2;$$

end for

minSSD(i, j) = min(SSD); disparity(i, j) = find(SSD==min(SSD));

end for

where d stands for an approximately possible disparity range.

Consider another example with same I_1 as reference low image and I_3 stands for a high image as contrasting image.

$$I_1 = \begin{bmatrix} 23 & 50 & 85 & 75 & 24 & 32 \\ 54 & 16 & 162 & 46 & 54 & 82 \\ 64 & 6 & 85 & 216 & 58 & 76 \\ 137 & 52 & 45 & 34 & 91 & 84 \\ 81 & 46 & 248 & 57 & 81 & 79 \\ 31 & 18 & 97 & 31 & 82 & 97 \end{bmatrix} \qquad I_3 = \begin{bmatrix} 84 & 54 & 49 & 115 & 56 & 138 \\ 25 & 87 & 96 & 67 & 35 & 87 \\ 26 & 53 & 89 & 78 & 26 & 34 \\ 56 & 13 & 164 & 42 & 56 & 86 \\ 68 & 9 & 86 & 218 & 54 & 79 \\ 142 & 53 & 49 & 38 & 94 & 87 \end{bmatrix}$$

Similarly to the horizontal case, we set a fixed window in reference image I_1, mark the position of each window in contrasting image I_3 as p_1, p_2, p_3 and p_4. The SSD for each position respectively is 18940, 155330, 77 and 24309. It could conclude that p_3 is the fit position.

In this project, horizontal match already exists in OpenCV and vertical match is implemented by the author. Both horizontal and vertical matching functions are provided independently. The result of horizontal and vertical matching is shown in Figure 2.7-11. Black area in analyses images stands for bad matching

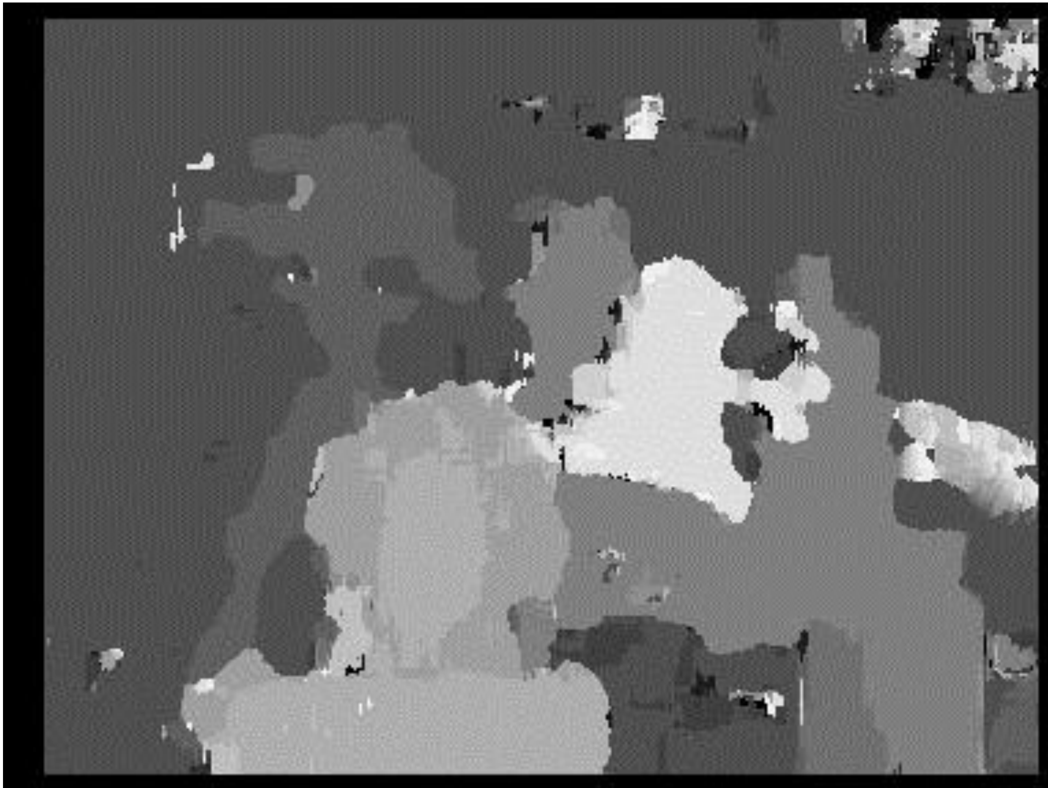


Figure 2.7 Horizontal matching result

.

Figure 2.8 Result analyses for horizontal matching result on Middlebury Stereo Page.

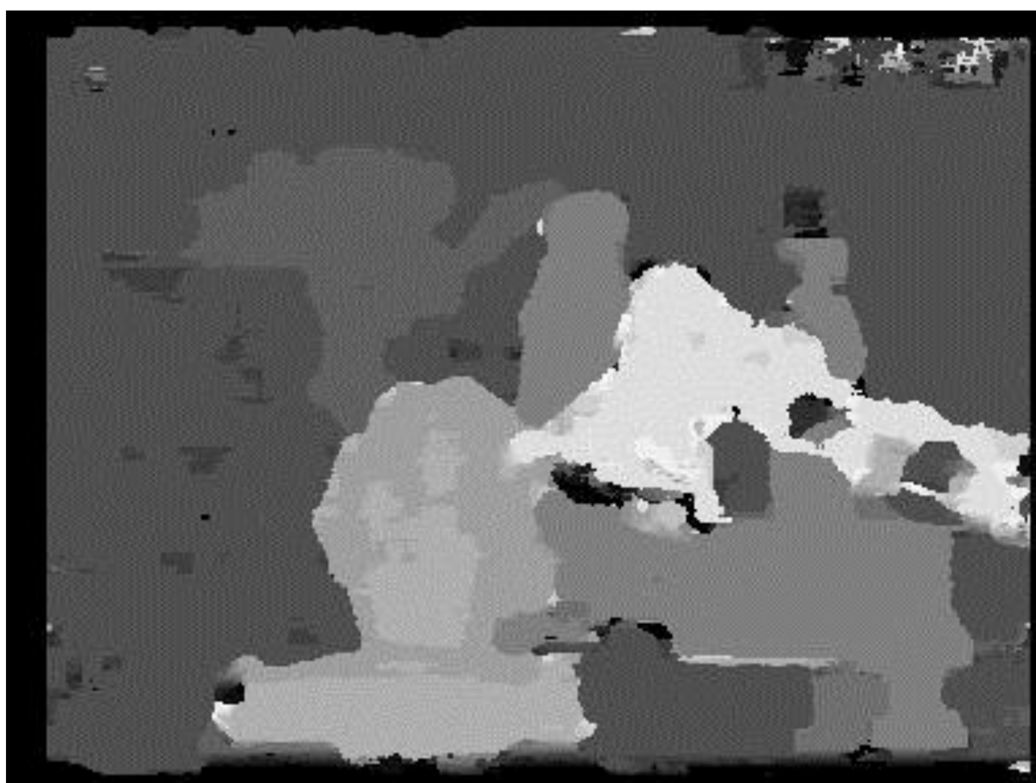


Figure 2.9 Vertical matching result.

Figure 2.10 Result analyses for vertical matching result on Middlebury Stereo Page.

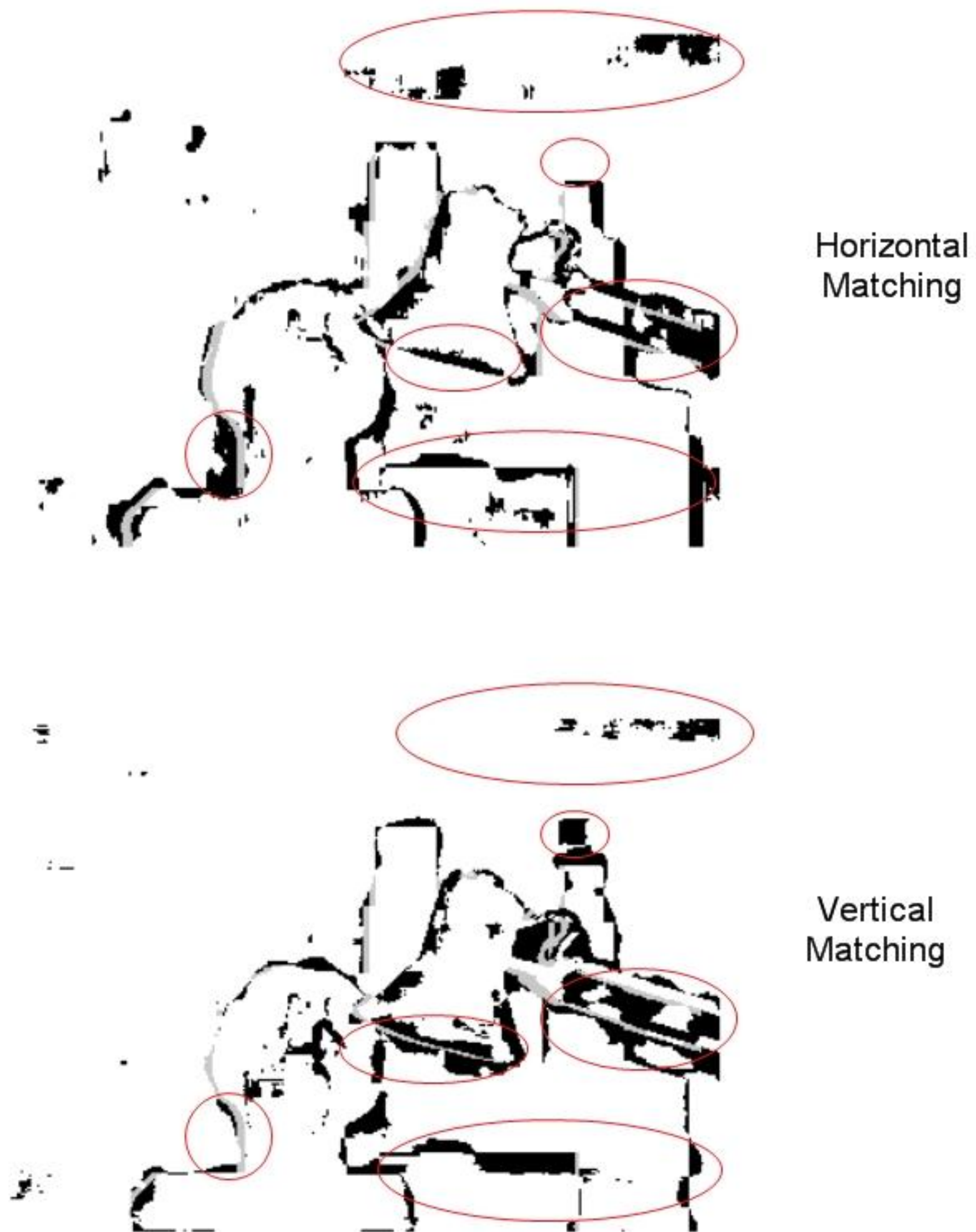


Figure 2.11 Comparison of horizontal and vertical matching results.

2.3.2 Fusion of Disparities in two Orientations

It is obvious that neither horizontal nor vertical could provide a satisfying matching accuracy with merely min SSD algorithm. It is important to combine the information from two disparity images together effectively. To combine the two disparity images, following ways were considered

1. Numerical computation with two disparity images in pixel. For instance, compute the average value of two disparity images in each pixel to composite final disparity image. However, most bad matching pixels of horizontal and disparity cannot be totally eliminated in this way. In most occasions, average value algorithm cannot improve result accuracy and even get it worse. This point is proved in practical experiments

2. Applying advanced computation model involves computation principles like Winner-Take-All. Basic method is to compare a standard value of two disparity images with a certain criteria and threshold. The one with more outstanding performance in this value is considered as the winner and the result it represents for will occupy the final result.

This project employs the second way to conduct fusion work and applies it in pixel level comparison. As mentioned before, the position of minimum SSD value composite disparity image and also minimum SSD value itself is also recorded into a matrix. To compare the minimum SSD matrix in corresponding coordinate and the one lower than k times of another one is considered as winner which k is called adjustable ratio parameter normally a float range between 0 and 1. The pixel in corresponding coordinate of winner disparity image will occupy the corresponding coordinate pixel of final disparity image. To describe the algorithm based on horizontal matching case in pseudo code

```
for i∈[1, height], j∈[1,width]
    if minSSD_h(i, j)<k×minSSD_v(i, j)
        minSSD(i, j) = minSSD_h(i, j);
        disparity(i, j) = disparity_h(i, j);
    else
        minSSD(i, j) = minSSD_v(i, j);
        disparity(i, j) = disparity_v(i, j);
end
```

where $minSSD_h$ stands for minimum SSD collection from horizontal matching, $minSSD_v$ stands for minimum SSD collection from vertical matching, minSSD stands for final minimum SSD collection, k stands for the adjustable ratio parameter, $diparity_h$ stands for disparity value from horizontal matching, $diparity_v$ stands for disparity value from vertical matching and disparity stands for final fusion disparity.

The practical experiment has proved that this method could improve matching accuracy effectively. However it cannot reach the most ideal case which eliminates all bad matching pixels appear in a signal disparity image and leave only bad matching pixels appear in both disparity images. The reason for this problem is there are many bad matching pixels station in the area where not cover by all three images. That is to say, for some bad matching pixels in one disparity, as there is no information about this area in another disparity image, their corresponding minimum SSD are even lower though their matching results are not good. A good example is in Tsukuba dataset, there is less information in higher image of vertical pair

for the area under the table. As a result the minimum SSD values in this area would be very big. However, for horizontal pair though SSD values in this area are not as big as vertical pair, there would be bad matching pixels. So in the final disparity result there would be bad matching pixels. It is possible to apply more complex and accurate fusion model to improve fusion result accuracy more.

The discussion of detail implementation of fusion will be discussed in next chapter and the result and analyses of final fusion disparity image is shown in the fourth chapter.

Chapter 3

Implementation with OpenCV and CUDA

This chapter will discuss the implementation to realize the algorithm in last chapter with OpenCV library and the CUDA module within it. According to the description of three levels in computer vision system, this chapter mainly discusses the third level: hardware implementation.

There is a lot of connection between software and hardware in this project as GPU programming is involved. Therefore, a good knowledge, skills and techniques are needed to handle it in order to maximize the performance in both speed and accuracy.

3.1 Code Hierarchy of OpenCV Source Code

As the project is based on OpenCV library, the most basic issue for implementation is to understand the architecture and source code of OpenCV [13]. The source code of OpenCV is composed by several function modules. Source code in each function module realizes a certain area of computer vision application. Major modules in current OpenCV version (The project is done with version 2.3.1. However newer versions were released while this manuscript was writing) cover almost all applied areas of computer vision applications, including image segmentation, image filtering, object detection and tracking. For example, object detection provides many functions for object detection like pyramid algorithm and stereo matching is realized in camera calibration module. Each module is independent but could call and apply functions from other modules; also in one project the source file could employ several different modules. A good example is core module which providing most

basic objects and functions for image/video loading, reading, writing and saving, it could be called in project file and also applied by any other module however no module attaches to it.

In folder of each module, there are header file and source code. The header file provides objects and interface functions and source file implements functions for applications. In a project employing OpenCV, project source code would include and call interface functions in certain module, and then the interface functions will run the code in source file. Generally to apply OpenCV in own project, developers just need to know how to handle the specific function module, its objects and interfaces. As the work on this project needs to modify OpenCV source code to realize some functions not existed in camera calibration module before, reading and understanding source code of involved modules is the very first step for implementation work of this project.

There is a GPU module for applying GPU programming on computer vision since version 2.0. This GPU module employs NVidia CUDA architecture; hence it could only support NVidia GPU to run this module. While compiling OpenCV source code, developer could switch whether to enable compiling GPU module. Another important issue is the GPU module in OpenCV is implemented with 32-bit CUDA so currently it could only work well under 32-bit operating system.

GPU module just functions like a small copy of OpenCV. In the GPU module folder it includes many different functions which cover areas of computer vision applications run in host-level. Also in the folder there is CUDA source code which called by object interface functions of the module to run in device-level. Compare to the CPU implementation, there are plenty of differences in the GPU implementation to fit features of GPU. For example, the object to store image in CPU implementation is Mat while the corresponding one in GPU version is GpuMat. To fit GPU memory, GpuMat can only support 2-dimenson and no reference return to data. Another good example to demonstrate this kind of difference is in block matching stereovision algorithm. CPU implementation employs SAD algorithm while GPU implementation applies SSD algorithm as demand for unified operation to accelerate on GPU.

To build the developing environment of OpenCV, download the source code and binaries from its website (http://opencv.willowgarage.com/wiki/). The library requires to be compiled

by CMake. Developers could choose whether compiling with several modules like CUDA, Intel Integrated Performance Primitives (Intel IPP, is a multi-threaded software library of functions for multimedia and data processing applications) and Nokia Qt (A cross-platform application framework for GUI development) by switching the corresponding options. More settings are needed to be done depends on the operating system and integrated development environment (IDE) work with. Developer should set environment variables in Windows system and add the path in Linux system. The instruction is easy and open accessible online. For this project the operating system is Linux (Ubuntu 10.10) and the IDE is Eclipse.

3.2 CUDA Programming Model

As mentioned in the beginning chapter, performance of GPGPU is superior in area like computer vision and scientific computation for the unique hardware architecture of GPU. This point attracts a lot of companies' attention. Driven by the insatiable market demand, also as a leading company in GPU industry, NVidia certainly would have own GPGPU solution. It is a general purpose parallel computing architecture named CUDA.

NVidia introduced CUDA in November 2006 with a new parallel programming model and instruction set architecture [14]. Also CUDA came with a software development kit (SDK) that developer could use a C-like CUDA language to write program. This CUDA language which the source files are in format with a suffix 'cu' is just like a supplement to C language. There are new data types and functions to help with programming on GPU in device-level. Besides these features, CUDA also supports many different API and libraries like OpenCL (Open Computing Language, an open framework for writing programs which could execute across different kinds of platform like CPU, GPU and FPGA) and DirectCompute (A Microsoft API to support GPGPU on Windows operating system, specifically Windows Vista and Windows 7). As CUDA is owned by NVidia rather than an open architecture, the discussion below will only involve how to handle CUDA rather than its detailed implementation inside.

As architecture of CUDA is so novel and the parallel level is so high, traditional parallel method no longer works under this circumstance. Therefore NVidia designs a new system for

CUDA. In CUDA architecture, the most basic computing process unit is thread. Just like the general understanding of thread in operating system, normally it is simple and straight forward operation like addition or multiplication. As in GPGPU there would be too many threads in a program, a new unit in CUDA called warp is used to describe collection of large amount threads which 1 warp equals 32 parallel threads. Also for managing so many threads in a single program, a hierarchy system is introduced. Thread is also the lowest unit in this system. In a program, equal amount of parallel threads compose blocks. Block is the smallest unit to be processed by a computation core. The CUDA programming model is automatic scalable. That means the more cores on a GPU, the less computation task distributed to a single core. Hence the more advanced GPU would have a better performance. Figure 3.1 demonstrates this feature.

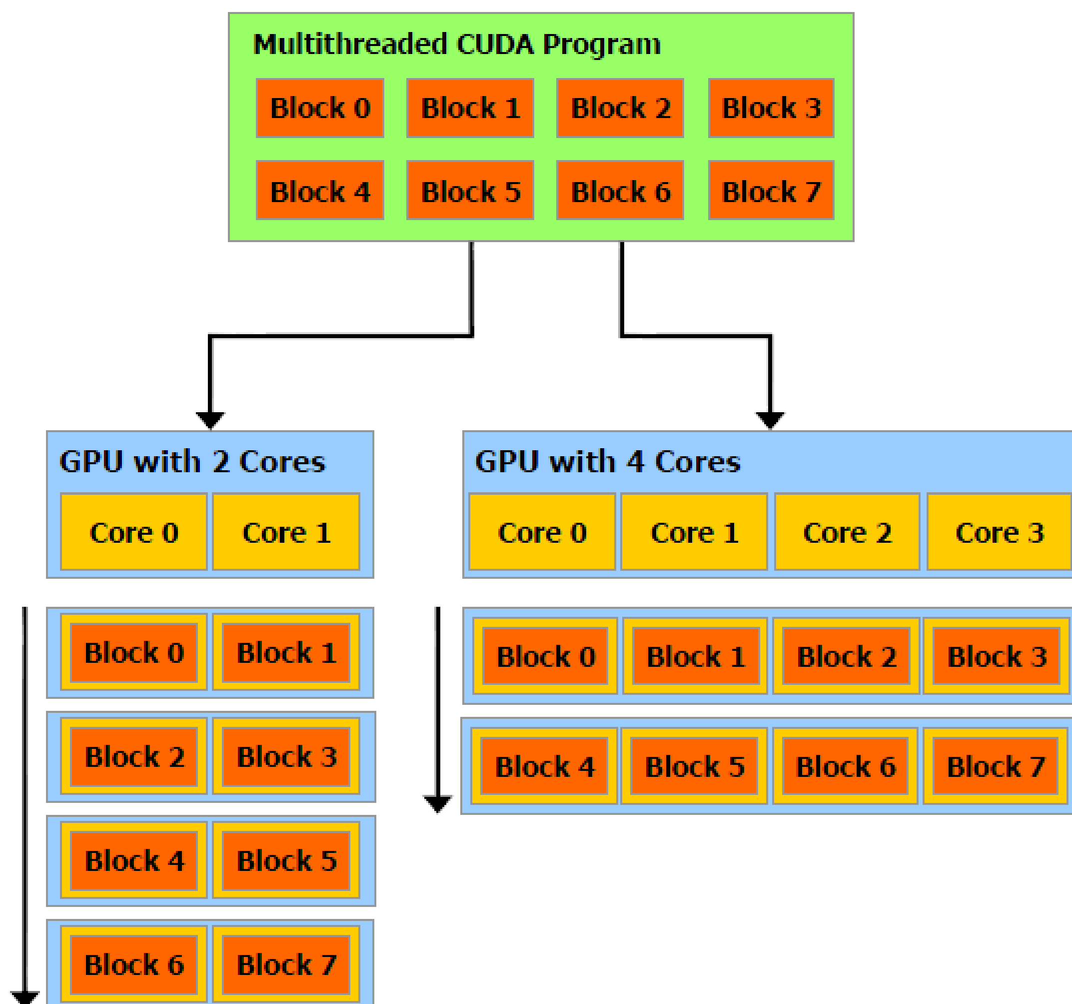


Figure 3.1 Automatic Scalability of CUDA [14]

Another new concept in CUDA is grid. A grid consists of multiple blocks. The whole computation resource for a certain piece of code or a function would be considered as a grid. Before running into the code or call the function, it is necessary to schedule the computation resource. With normal CUDA practice, a grid is divided into many same size blocks, each block includes same amount threads. The size of grid could be 1-to 3 dimensional. Figure 3.2 and Figure 3.3 demonstrate the 2 and 3 dimensional resource schedule, respectively.

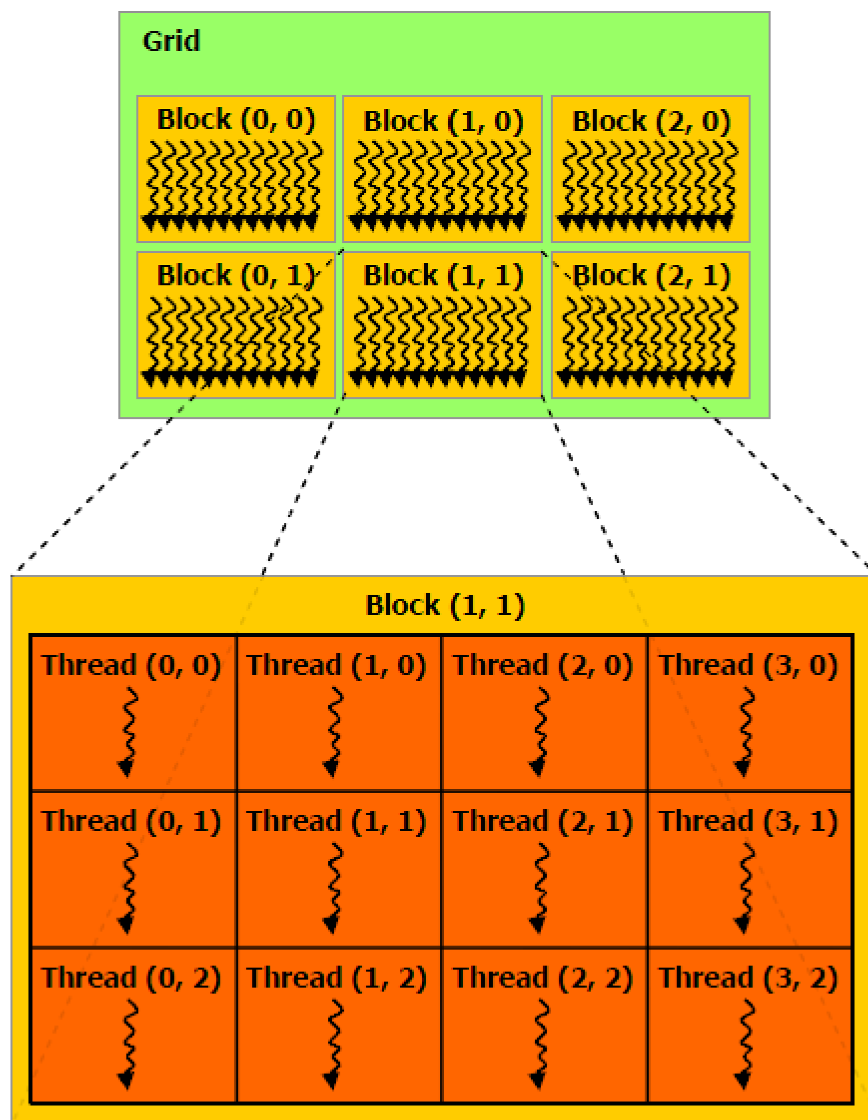


Figure 3.2

Organization of 2-D Grid [14]

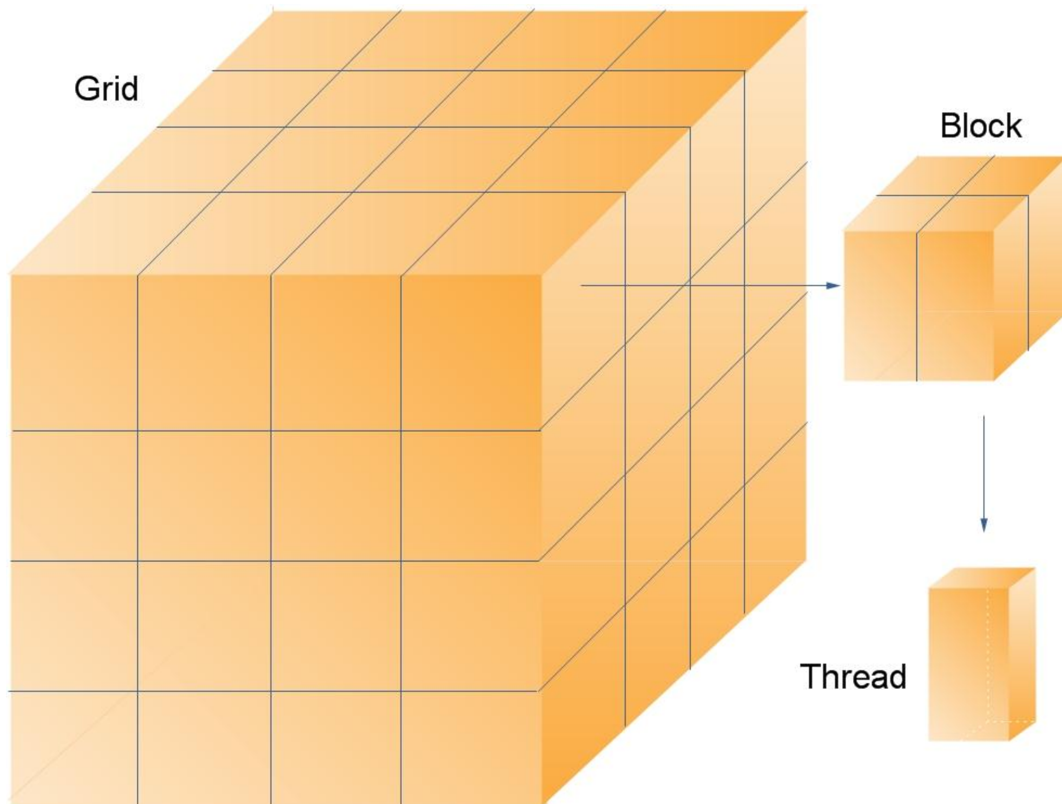


Figure 3.3 Organization of 3-D Grid

To schedule the computation resource, NVidia designs a new data type for CUDA named ‘dim3’. It is an n×1 integer vector and used to describe the size division of grid and block which n<=3. The integers indicate the amount of blocks or threads in each dimension respectively. When calling a kernel function running involves thread-level parallel operation on device, it is required to point out the resource schedule with dim3 type variables. A piece of sample code for matrix addition would demonstrate how it works.[14]

```
 // Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
float C[N][N])
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i < N && j < N)
  C[i][j] = A[i][j] + B[i][j];
```

```
}
int main() // Host code
{
  ... // Assign value for A, B and C
      // Kernel invocation
  dim3 threadsPerBlock(16, 16);
  // Each block contains 16×16=256 threads
  dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
  // Divide number of elements in each dimension by number of
  // threads in each dimension to know how many blocks needed
  // in each dimension
  MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
  // Call the kernel function with grid and block parameters.
}
```

From this piece of code we could basically understand how this hierarchy system works. While implementing kernel function, developer should know how many threads needed. When calling the kernel function developer must clearly define dim3 type variable to declare how many blocks in each dimension of grid and how many threads in each dimension of block by the way of <<<grid, block, …>>>. A restriction due to current GPU technology is there can be maximum 1024 threads in one block.

A general pattern for CUDA function is divided into three levels. The kernel function will define the most basic operation by threads in each block. A kernel caller function will schedule the resource and call the kernel function. Sometimes this kernel caller function can be defined as external and called by the host code. In some more complex cases like shared variables and shared device memory mentioned latter involve, there would be a wrapper like function to define more details and call the kernel caller function. Eventually there would be a function defined as external function to be called by host code.

One of major issues to influence the speed performance of GPGPU is data transportation between host and device. Device (GPU) and host communicates through interface of computer bus like PCI-E (Peripheral Component Interconnect Express). The specification of these interfaces restricts data transportation speed between device and host. Compared to the computation speed of GPU cores, transportation speed is too slow to satisfy computing needs.

As an example, the peak processing power of NVidia Tesla S1070 for GPU Computing Server had reached 4147.2G FLOPS (Floating-point Operations per Second, a standard measure of computing performance). However its memory bandwidth is 409.6 GB/s (All data from NVidia). It is obvious that the transportation between device and host is the bottleneck to restrict GPGPU performance. To solve the problem, shared device memory is introduced. For GPU computing cores, accessing GPU memory on device would be naturally much faster than accessing host memory as no interface involves. With this condition, it is possible to reduce reliance on host memory by locating the most often used variable to device memory. If the variable is known in host memory, an easy way to utilize device memory is allocating the device memory first and copying the data from host to device directly. However more commonly, developers could set a device memory space called shared device memory to store the variable which will be operated repeatedly in kernel function. Generally the characteristic of device memory is high speed and small capacity. According to these features, the variables in shared device memory should be used most frequently and the size is carefully calculated and strictly refrained. To apply shared device memory, using '__shared__' to define the variable stored in the shared device memory. Also it needs to define the shared device memory with type 'size_t' variable. When calling kernel function with shared variable inside, it needs to point out the shared device memory size by the way of <<<grid, block, mem_size …>>>. The usage of shared device memory and shared variables will be demonstrated with project code in latter section.

Another important issue for CUDA is the synchronization between threads. Though operation for each thread is same, however, due to hardware and operating system the processing time for each thread varies a lot. In the case applying shared variable which multiple threads would read and write to a same variable, the time difference between threads would cause disorder and lead to unexpected result. To avoid this situation, NVidia provides a function '__syncthreads()' to synchronize all threads at the point. In some cases it is extremely important to apply the function. For example, in a thread operation all threads access and write on a shared variable, it is necessary to call the synchronization function before the operation or function involve the shared variable.

Above is the brief introduction of CUDA programming model. The following sections will discuss how to implement the vertical stereo matching and fusion algorithm by modifying OpenCV source code in second chapter which would involve most knowledge of CUDA discussed in this section.

3.3 Implementation of Vertical Matching

This section will discuss how to implement the vertical stereo matching algorithm descripted in the second chapter based on current OpenCV camera calibration module source code with CUDA support. There will be code and explanation for it. As camera calibration module of OpenCV is the foundation of this project, it would be helpful to understand what the structure of the source code is and how it runs.

The header file of GPU module, ‘gpu.hpp’ gives definition of the stereo block matching object and its interface functions with some necessary parameters and private variables as below. To support vertical matching and disparity fusion, some private variables are added and some changes are done to the object. The parameters of previous operator function are changed in order to enable switching between horizontal and vertical matching. Also a new operator function for disparity fusion is added. For convenient, all codes below will only show content directly related to the algorithm and omit some of the original OpenCV code with some functions to maintain system stable

```
class CV_EXPORTS StereoBM_GPU
{
        …… // original OpenCV code
        void operator() ( const GpuMat& one, const GpuMat& two, GpuMat& disparity, bool direction = true, Stream& stream = Stream::Null());
        // a bool parameter to check matching direction is added
        void operator() ( const GpuMat& left, const GpuMat& right, GpuMat& high, GpuMat& disparity, Stream& stream = Stream::Null());
        // operator function for disparity fusion, will be discussed in next section
```

```
        private:
                GpuMat minSSD, minSSD_ve, minSSD_ho, leBuf, riBuf,
loBuf, hiBuf, ceBuf_ho, ceBuf_ve;
      // Some variables are added for vertical matching and disparity
      fusion
      };
```

In the source code folder of GPU module, the source code in file of camera calibration named 'stereobm.cpp' implement the object and interface functions. Normally users just need to call the object operator function to get the disparity image from input horizontal stereo images. The operator function would execute the caller function. The caller function will check whether filter the input images with Sobel Filter and call the CUDA stereo match function which declared as external defined function in the beginning of the stereo block matching source file.

After preparing necessary changes to the stereo matching object, the work begins with modifying stereo matching source code. Following the pattern of existing OpenCV source code, the operator function now would decide whether do horizontal or vertical stereo matching and pass the data to corresponding caller function. The new operator function code is shown as below

```
    void cv::gpu::StereoBM_GPU::operator() ( const GpuMat& one,
const GpuMat& two, GpuMat& disparity, bool direction, Stream&
stream)
    {
        if(direction==true) // direction is true
        ::stereo_bm_gpu_operator(minSSD_ho, leBuf, riBuf, preset,
ndisp, winSize, avergeTexThreshold, one, two, disparity,
StreamAccessor::getStream(stream));  // Go horizontal case
        Else        // direction is false
        ::stereo_bm_gpu_operator_vertical(minSSD_ve, loBuf, hiBuf,
preset, ndisp, winSize, avergeTexThreshold, one, two, disparity,
StreamAccessor::getStream(stream));    // Go vertical case
    }
```

While the horizontal case still apply original OpenCV implementation, the vertical case will follow the structure of horizontal matching code with new Sobel Filtering function and call a different CUDA function for vertical matching. New variables in stereo matching object will be applied in order to avoid confusion. The code of new vertical matching caller function is shown as below

```
    static void stereo_bm_gpu_operator_vertical ( GpuMat& minSSD,
GpuMat& loBuf, GpuMat&  hiBuf,  int preset, int ndisp, int winSize,
float avergeTexThreshold, const GpuMat& low, const GpuMat& high,
GpuMat& disparity, cudaStream_t stream)
    {
        ……// reference from original OpenCV codes

        GpuMat lo_for_bm =  low;
        GpuMat hi_for_bm = high;

        if (preset == StereoBM_GPU::PREFILTER_XSOBEL)
        {
            loBuf.create( low.size(),  low.type());
            hiBuf.create(high.size(), high.type());

            bm::prefilter_xsobel_vertical( low, loBuf, 31, stream);
            bm::prefilter_xsobel_vertical(high, hiBuf, 31,
stream);
            // Filtering with vertical Sobel Filter

            lo_for_bm = loBuf;
            hi_for_bm = hiBuf;
        }

        bm::stereoBM_GPU_Vertical(lo_for_bm, hi_for_bm, disparity,
ndisp, winSize, minSSD, stream);
       // call the CUDA function
```

```
 ……// reference from original OpenCV codes
}
```

Now we will discuss the CUDA code for vertical matching. When running the vertical matching function, the first CUDA function involved is Sobel Filtering function. Following the pattern of CUDA source code analyzed before, this function is actually a kernel caller which schedule resource and call the kernel function. The kernel function defines the convolution computation in each 3×3 window with Sobel Filter for vertical matching discussed in Chapter 2. The source code is shown as below.

```
extern "C" __global__ void prefilter_kernel_vertical(DevMem2D
output, int prefilterCap)
{
    ……// reference from OpenCV codes

    if (x < output.cols && y < output.rows)
    {
        int conv = (int)tex2D(texForSobel, x - 1, y - 1) * (-1)
    + (int)tex2D(texForSobel, x, y - 1) * (-2)
    + (int)tex2D(texForSobel, x + 1, y - 1) * (-1)
    +  (int)tex2D(texForSobel, x + 1, y + 1) * (1)
    + (int)tex2D(texForSobel, x, y + 1) * (2)
    + (int)tex2D(texForSobel, x + 1, y + 1) * (1);
// Convolution computation with [-1,-2,-1; 0, 0, 0; 1, 2, 1]

    ……// reference from OpenCV codes
    }
}
```

The caller code is in almost same style with the horizontal Sobel Filter caller function hence will not be shown.

After filtering with Sobel filter, the image now is more suitable for stereo matching. The vertical matching function is also implemented with the CUDA pattern. From top to bottom,

the first one is the defined as external function to be called in host code. The task for this function is to prefetch the frequent used parameters of input images to device memory and call the kernel caller function based on the sub window size of the matching object. The kernel caller function will set shared memory space for shared variable and schedule grid and block size. Then it will call the kernel function with the known parameters. As the style is almost same with horizontal case, this part of code is omitted.

Just like its name, though some computation work in it is done by other functions, kernel function is the core part of whole program to realize the algorithm. There is a lot of preparation work for kernel function of vertical stereo matching include setting shared variable, define X-axis and Y-axis indicator for lower level function to access image pixels and reserve the disparity image and minSSD matrix. In computation section the whole images are divided into blocks. With the epipolar restraint of stereovision, all computation just involves the pixels in the same vertical line of two images. The code loops in all possible disparity range. If a less minSSD value is found, the function will update the result. The minSSD values of starting window position in each block are calculated separately with rest lines as starting window difference in the block is fixed. To avoid over calculated the value of last line as starting window difference of next block will be deducted. The minSSD value will be recorded to the matrix and the corresponding window position is chosen for disparity. As the key point is the SSD computation; therefore only function of SSD computation for the starting window of the block and window position in the block is shown below.

```
    template<int RADIUS>
    __device__ void InitColSSD_Vertical(int x_tex, int y_tex, int
im_yaw, unsigned char* imageL, unsigned char* imageH, int d,
volatile unsigned int *col_ssd)
    {
        unsigned char lowPixel1;
        int idx;
        unsigned int diffa[] = {0, 0, 0, 0, 0, 0, 0, 0};

        for(int i = 0; i < (2 * RADIUS + 1); i++)
        {
```

```
            idx = y_tex * im_yaw + x_tex;
            lowPixel1 = imageL[idx];
            idx = idx + d * im_yaw;

            diffa[0] += SQ(lowPixel1 - imageH[idx + im_yaw * 0]);
            diffa[1] += SQ(lowPixel1 - imageH[idx + im_yaw * 1]);
            diffa[2] += SQ(lowPixel1 - imageH[idx + im_yaw * 2]);
            diffa[3] += SQ(lowPixel1 - imageH[idx + im_yaw * 3]);
            diffa[4] += SQ(lowPixel1 - imageH[idx + im_yaw * 4]);
            diffa[5] += SQ(lowPixel1 - imageH[idx + im_yaw * 5]);
            diffa[6] += SQ(lowPixel1 - imageH[idx + im_yaw * 6]);
            diffa[7] += SQ(lowPixel1 - imageH[idx + im_yaw * 7]);

            y_tex += 1;
        }

        col_ssd[0 * (BLOCK_W + 2 * RADIUS)] = diffa[0];
        col_ssd[1 * (BLOCK_W + 2 * RADIUS)] = diffa[1];
        col_ssd[2 * (BLOCK_W + 2 * RADIUS)] = diffa[2];
        col_ssd[3 * (BLOCK_W + 2 * RADIUS)] = diffa[3];
        col_ssd[4 * (BLOCK_W + 2 * RADIUS)] = diffa[4];
        col_ssd[5 * (BLOCK_W + 2 * RADIUS)] = diffa[5];
        col_ssd[6 * (BLOCK_W + 2 * RADIUS)] = diffa[6];
        col_ssd[7 * (BLOCK_W + 2 * RADIUS)] = diffa[7];
    }

    template<int RADIUS>
    __device__ void StepDown_Vertical(int idx1, int idx2, unsigned
char* imageL, unsigned char* imageH, int im_yaw, int d, volatile
unsigned int *col_ssd)
    {
        unsigned char leftPixel1;
        unsigned char leftPixel2;
        unsigned char rightPixel1[8];
        unsigned char rightPixel2[8];
        unsigned int diff1, diff2;

        leftPixel1 = imageL[idx1];
        leftPixel2 = imageL[idx2];
```

```
idx1 = idx1 + im_yaw * d;
idx2 = idx2 + im_yaw * d;

rightPixel1[7] = imageH[idx1 + im_yaw * 7];
rightPixel1[0] = imageH[idx1 + im_yaw * 0];
rightPixel1[1] = imageH[idx1 + im_yaw * 1];
rightPixel1[2] = imageH[idx1 + im_yaw * 2];
rightPixel1[3] = imageH[idx1 + im_yaw * 3];
rightPixel1[4] = imageH[idx1 + im_yaw * 4];
rightPixel1[5] = imageH[idx1 + im_yaw * 5];
rightPixel1[6] = imageH[idx1 + im_yaw * 6];

rightPixel2[7] = imageH[idx2 + im_yaw * 7];
rightPixel2[0] = imageH[idx2 + im_yaw * 0];
rightPixel2[1] = imageH[idx2 + im_yaw * 1];
rightPixel2[2] = imageH[idx2 + im_yaw * 2];
rightPixel2[3] = imageH[idx2 + im_yaw * 3];
rightPixel2[4] = imageH[idx2 + im_yaw * 4];
rightPixel2[5] = imageH[idx2 + im_yaw * 5];
rightPixel2[6] = imageH[idx2 + im_yaw * 6];

//See above:  #define COL_SSD_SIZE (BLOCK_W + 2 * RADIUS)
diff1 = leftPixel1 - rightPixel1[0];
diff2 = leftPixel2 - rightPixel2[0];
col_ssd[0 * (BLOCK_W + 2 * RADIUS)] += SQ(diff2) - SQ(diff1);

diff1 = leftPixel1 - rightPixel1[1];
diff2 = leftPixel2 - rightPixel2[1];
col_ssd[1 * (BLOCK_W + 2 * RADIUS)] += SQ(diff2) - SQ(diff1);

diff1 = leftPixel1 - rightPixel1[2];
diff2 = leftPixel2 - rightPixel2[2];
col_ssd[2 * (BLOCK_W + 2 * RADIUS)] += SQ(diff2) - SQ(diff1);

diff1 = leftPixel1 - rightPixel1[3];
diff2 = leftPixel2 - rightPixel2[3];
col_ssd[3 * (BLOCK_W + 2 * RADIUS)] += SQ(diff2) - SQ(diff1);
```

```
        diff1 = leftPixel1 - rightPixel1[4];
        diff2 = leftPixel2 - rightPixel2[4];
        col_ssd[4 * (BLOCK_W + 2 * RADIUS)] += SQ(diff2) - SQ(diff1);

        diff1 = leftPixel1 - rightPixel1[5];
        diff2 = leftPixel2 - rightPixel2[5];
        col_ssd[5 * (BLOCK_W + 2 * RADIUS)] += SQ(diff2) - SQ(diff1);

        diff1 = leftPixel1 - rightPixel1[6];
        diff2 = leftPixel2 - rightPixel2[6];
        col_ssd[6 * (BLOCK_W + 2 * RADIUS)] += SQ(diff2) - SQ(diff1);

        diff1 = leftPixel1 - rightPixel1[7];
        diff2 = leftPixel2 - rightPixel2[7];
        col_ssd[7 * (BLOCK_W + 2 * RADIUS)] += SQ(diff2) - SQ(diff1);
    }
```

3.4 Optimization and Implementation of Disparities Fusion

This section focuses on discussing about the implementation of disparity fusion. The input for fusion algorithm is totally different from single matching. Therefore it is necessary to change operator function in order to fit matching and fusion algorithm. The code below shows the overload operator function for fusion algorithm

```
    static void stereo_bm_gpu_operator_fusion (GpuMat& minSSD,
GpuMat& ceBuf_ho, GpuMat& ceBuf_ve, GpuMat& riBuf, GpuMat& hiBuf,
int preset, int ndisp, int winSize, float avergeTexThreshold, const
GpuMat& center, const GpuMat& right, const GpuMat& high, GpuMat&
disparity, cudaStream_t stream)
    {
        CV_DbgAssert(center.rows == right.rows && low.cols ==
right.cols && center.rows == high.rows && low.cols == high.cols);
        CV_DbgAssert(center.type() == CV_8UC1);
        CV_DbgAssert(right.type() == CV_8UC1);
        CV_DbgAssert(high.type() == CV_8UC1);

        disparity.create(center.size(), CV_8U);
```

```
        minSSD.create(center.size(), CV_32S);

        GpuMat ce_for_bm_ho = center;
        GpuMat ce_for_bm_ve = center;
        GpuMat ri_for_bm = right;
        GpuMat hi_for_bm = high;

        if (preset == StereoBM_GPU::PREFILTER_XSOBEL)
        {
            ceBuf_ho.create(center.size(), center.type());
            ceBuf_ve.create(center.size(), center.type());
            riBuf.create(right.size(), right.type());
            hiBuf.create(high.size(), high.type());

            bm::prefilter_xsobel(center, ceBuf_ho, 31, stream);
        bm::prefilter_xsobel_vertical(center, ceBuf_ve, 31,
stream);
            bm::prefilter_xsobel(right, riBuf, 31, stream);
            bm::prefilter_xsobel_vertical(high, hiBuf, 31,
stream);

            ce_for_bm_ho = ceBuf_ho;
            ce_for_bm_ve = ceBuf_ve;
            ri_for_bm = riBuf;
            hi_for_bm = hiBuf;
        }

        bm::stereoBM_GPU_Fusion(ce_for_bm_ho, ce_for_bm_ve,
ri_for_bm, hi_for_bm, disparity, ndisp, winSize, minSSD, stream);

        if (avergeTexThreshold)
            bm::postfilter_textureness(ce_for_bm_ho, winSize,
avergeTexThreshold, disparity, stream);
    }
```

As descripted in previous, the fusion is based on comparison of minSSD value of each pixel from horizontal and vertical disparity. The one more stereo block matching process

would naturally impact speed performance a lot. To achieve real-time goal, optimization for the algorithm is necessary.

As data transportation between device (GPU) and host (CPU and memory) contributes most of time consumption in GPGPU, one method to reduce processing time is to reduce data transportation by concentrating as much as possible data transportation into one transaction and finishing all computation work once for all. Following this method, the best way to optimize the fusion algorithm is to combine horizontal and vertical matching procedures into one session.

Directed by this method, the kernel function would follow the pattern of horizontal and vertical matching kernel function. However, what fusion kernel function does is to combine the horizontal and vertical block matching computation into itself, compare the results from two orientations to decide winner and feedback the final result.

As matching computation of either orientation requires occupying all threads and defining own shared variable, it is impossible to compute matching algorithm from both orientations simultaneously. The threads cannot switch between two kinds of computation continuously and repeatedly. If do so there would be conflict in threads which leads to a miserable result. One of the proper solutions is combining two processes into one function and comparing two minSSD results at the end of the function to get the final disparity result. Furthermore, as the two matching processed are taken in two steps, it could design that one matching process computes a result and store it to final disparity result then another one computes and update final result with its own better minSSD area. The code below shows crucial part of the fusion kernel function.

```
    template<int RADIUS>
    __global__ void stereoKernel_Fusion(unsigned char *center_h,
unsigned char *center_v, unsigned char *right, unsigned char *high,
size_t img_step, PtrStep disp, int maxdisp, float fusionRatio)
    {
        // Define common variables like X-axis indicator, Y-axis
indicator and result
        int X = (blockIdx.x * BLOCK_W + threadIdx.x + maxdisp +
RADIUS);
```

```
	int Y = blockIdx.y * ROWSperTHREAD + RADIUS;

	unsigned int* minSSDImage = cminSSDImage + X + Y *
cminSSD_step;
	unsigned char* disparImage = disp.data + X + Y * disp.step;

	int end_row = min(ROWSperTHREAD, cheight - Y - RADIUS);
	int y_tex;
	int x_tex = X - RADIUS;

	if (x_tex >= cwidth)
	    return;

	extern __shared__ unsigned int col_ssd_cache_vertical[];
	volatile unsigned int *col_ssd_vertical =
col_ssd_cache_vertical + BLOCK_W + threadIdx.x;
	volatile unsigned int *col_ssd_extra_vertical = threadIdx.x
< (2 * RADIUS) ? col_ssd_vertical + BLOCK_W : 0;
	// define shared variable for vertical matching only.

	…… // same with vertical matching process

	extern __shared__ unsigned int col_ssd_cache[];
	volatile unsigned int *col_ssd = col_ssd_cache + BLOCK_W +
threadIdx.x;
	volatile unsigned int *col_ssd_extra = threadIdx.x < (2 *
RADIUS) ? col_ssd + BLOCK_W : 0;
	// define shared variable for horizontal matching only.

	for(int d = STEREO_MIND; d < maxdisp; d += STEREO_DISP_STEP)
	{
	    y_tex = Y - RADIUS;
	    __syncthreads();
	    InitColSSD<RADIUS>(x_tex, y_tex, img_step, center_h,
right, d, col_ssd);

	    if (col_ssd_extra > 0)
	        if (x_tex + BLOCK_W < cwidth)
```

```
                InitColSSD<RADIUS>(x_tex + BLOCK_W, y_tex,
img_step, center_h, right, d, col_ssd_extra);

          __syncthreads(); //before MinSSD function

          if (X < cwidth - RADIUS && Y < cheight - RADIUS)
          {

             uint2 minSSD = MinSSD<RADIUS>(col_ssd_cache +
threadIdx.x, col_ssd);

              // update the final result with the one perform better
             minSSD within
              // threshold of fusionRatio
             if (float(minSSD.x)/float(minSSDImage[0]) <
             fusionRatio)            {
                 disparImage[0] = (unsigned char)(d + minSSD.y);
                 minSSDImage[0] = minSSD.x;
             }
          }

          for(int row = 1; row < end_row; row++)
          {
             int idx1 = y_tex * img_step + x_tex;
             int idx2 = (y_tex + (2 * RADIUS + 1)) * img_step +
x_tex;

             __syncthreads();

             StepDown<RADIUS>(idx1, idx2, center_h, right, d,
col_ssd);

             if (col_ssd_extra)
                 if (x_tex + BLOCK_W < cwidth)
                     StepDown<RADIUS>(idx1, idx2, center_h +
BLOCK_W, right + BLOCK_W, d, col_ssd_extra);

             y_tex += 1;
```

```
            __syncthreads(); //before MinSSD function


            if (X < cwidth - RADIUS && row < cheight - RADIUS -
Y)
            {
                int idx = row * cminSSD_step;
                uint2 minSSD = MinSSD<RADIUS>(col_ssd_cache +
threadIdx.x, col_ssd);

             // update the final result with the one perform
            better minSSD within
             // threshold of fusionRatio
                if (float(minSSD.x)/float(minSSDImage[idx]) <
              fusionRatio)             {
                    disparImage[disp.step * row] = (unsigned
char)(d + minSSD.y);
                    minSSDImage[idx] = minSSD.x;
                }
            }
        } // for row loop
    } // for d loop
}
```

Chapter 4

Result Analysis and Conclusion

This chapter will demonstrate the result of whole project and compare with previous implementation to show the improvement.

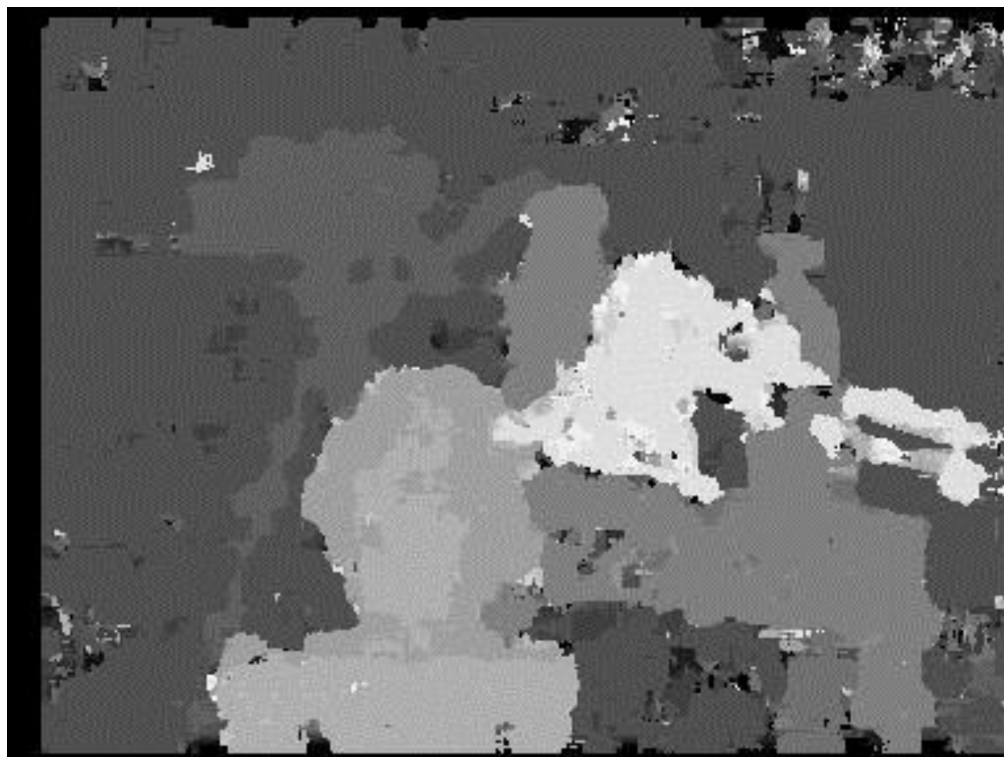
Most performance test is run with Tsukuba stereo dataset via Middlebury stereovision page [15]. The reason for choosing Tsukuba is that currently it is the only available stereo dataset fits for L-shape stereovision. To test the high resolution images, Tsukuba dataset is enlarged for test.

4.1 Accuracy Results

Besides three input images, the configuration of stereo block matching object are also parts of the parameters of the function. These configurations include approximate disparity distance, matching sliding window size for stereo matching and fusion ratio of the fusion algorithm. The setting of these configurations would also influence the final result a lot, not only for accuracy but also for speed. For example, if sliding window size is relatively small, there will be less computation in each sliding window but more computing times in each block. The best configurations should balance between speed and accuracy and it varies for different conditions and in applied stereovision system, these configurations could be easily adjusted. Therefore the way to find best setting is not the focuses of this project. To eliminate the influences of these factors, we will determine the best parameters for the functions by experiment before we run the test. As all parameters are independent to others, the experiment would be conducted by fixing two parameters whiling changing the only one parameter left to find out the best value. The Figure 4.1 to Figure 4.14 shows the experiment process and Table 4.1 to Table 4.3 compares the result. The number in tables stands for bad matching pixels

percentage in different conditions. The experiment determined the best parameters for Tsukuba approximate distance is 9, sliding window size is 12 and fusion ratio is 0.6.

Figure 4.1 Result with sliding window size 8



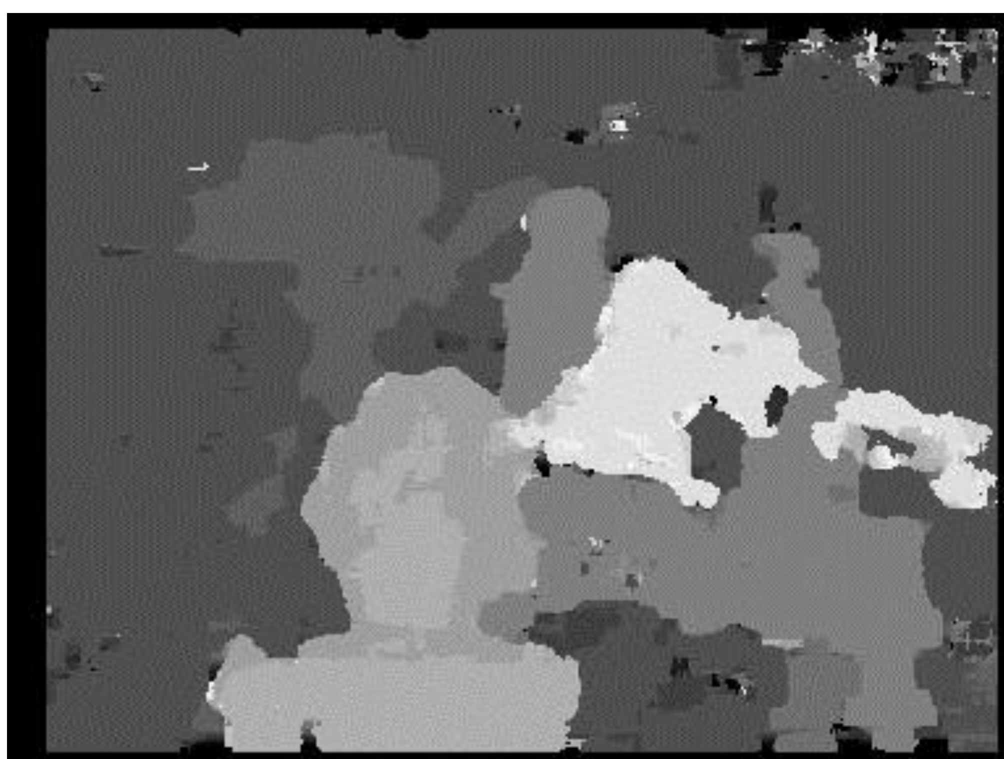


Figure 4.2 Result with sliding window size 12

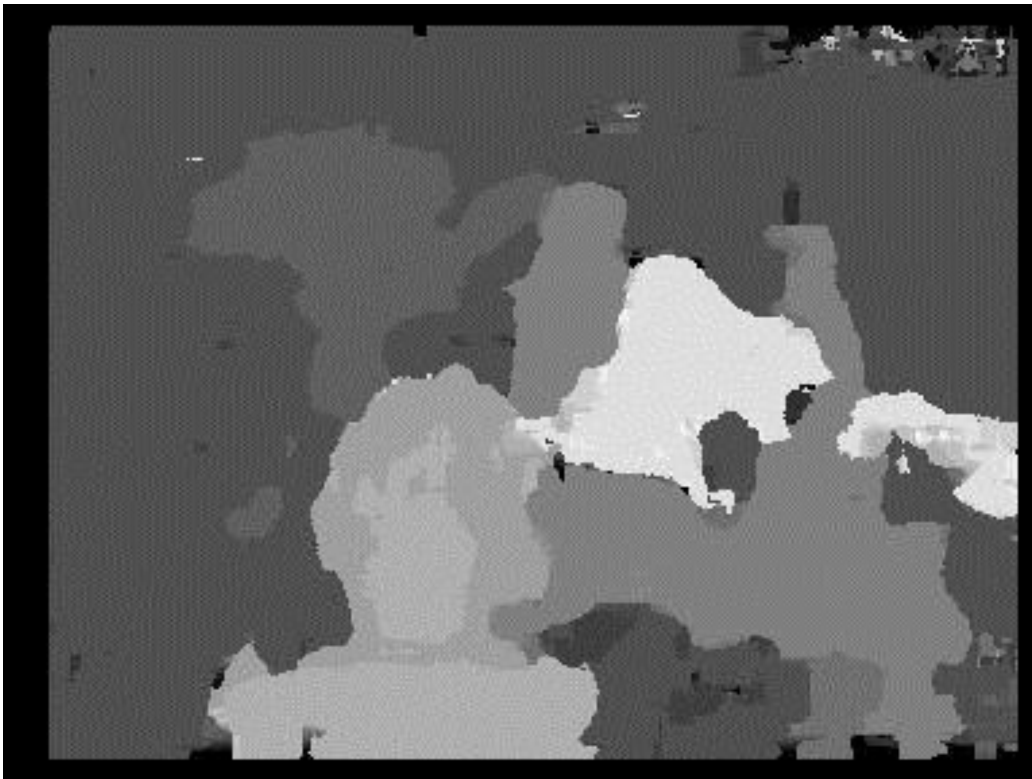


Figure 4.3 Result with sliding window 16

Figure 4.4 Result Analysis with sliding window size 8

Figure 4.5 Result Analysis with sliding window size 12

Figure 4.6 Result Analysis with sliding window size 16

	Non-occluded region (%)	All regions (%)	Discontinue region (%)
Sliding window size 8	8.51	9.78	27.3
Sliding window size 12	7.83	8.95	31.3
Sliding window size 16	7.91	8.94	33.4

Table 4.1 Accuracy Result for various sliding window size

We could observe from the test result that small sliding window size perform better in occluded area (discontinue region) which is a difficult point in stereovision. However the overall performance is not that good. Small window would be very accurate in area with strong features like boundary line as it focuses on a small region. For large open area like non-occluded area the situation is opposite. Therefore its general performance is not that good.

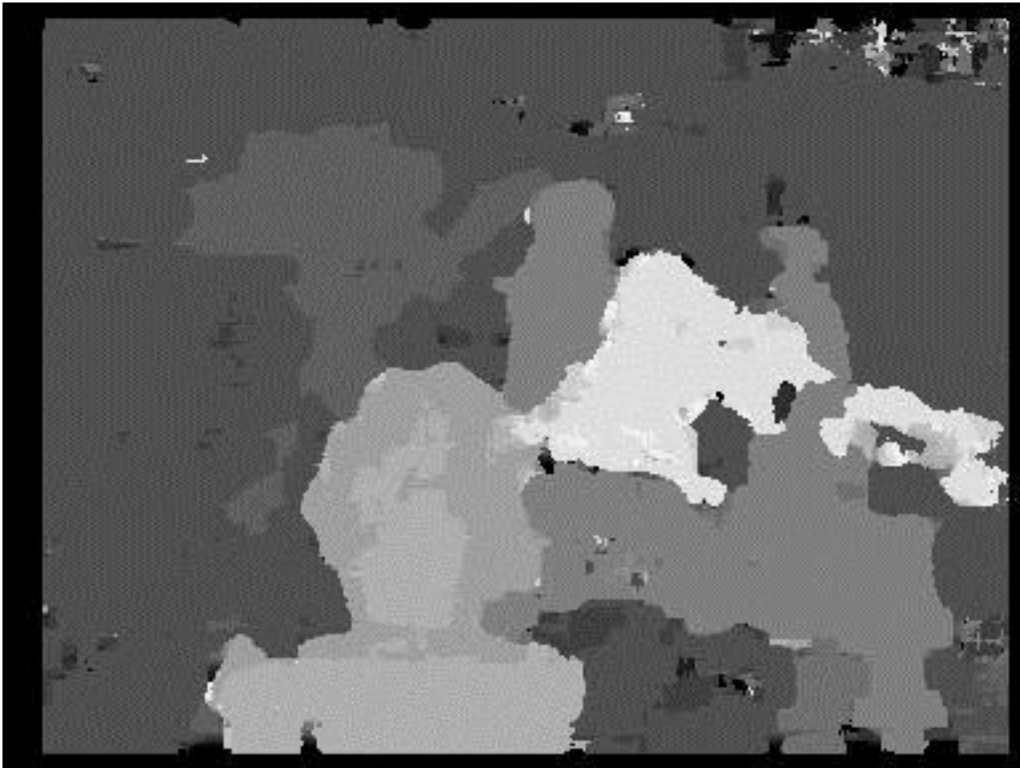


Figure 4.7 Result with disparity distance 9

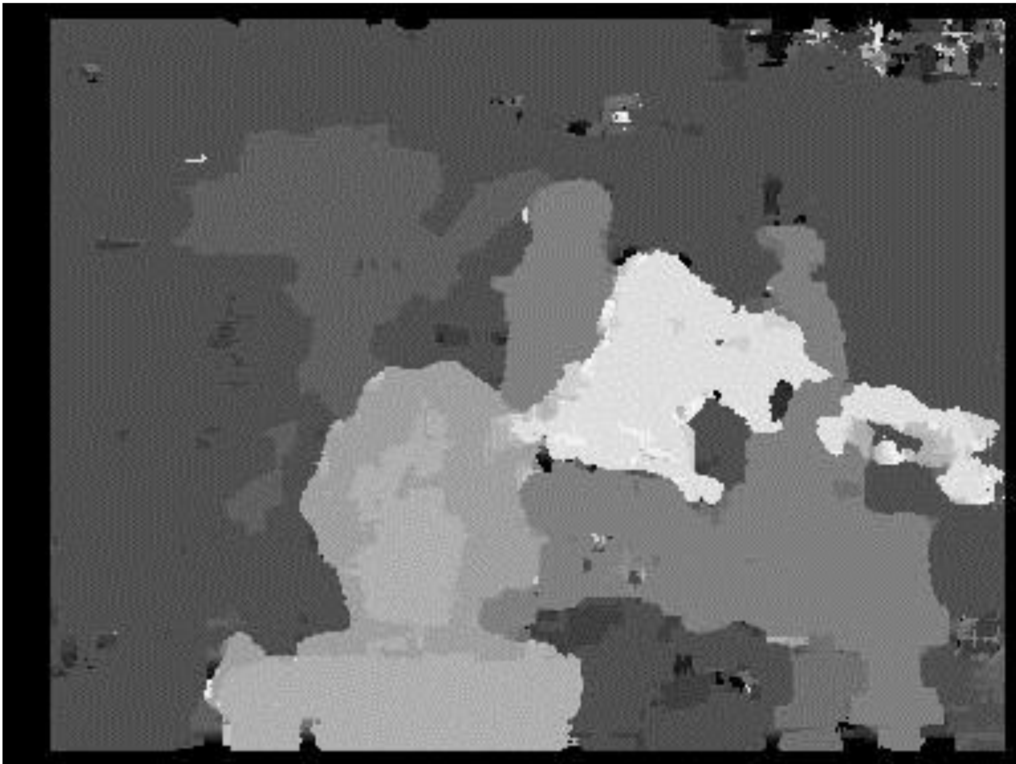


Figure 4.8 Result with disparity distance 12

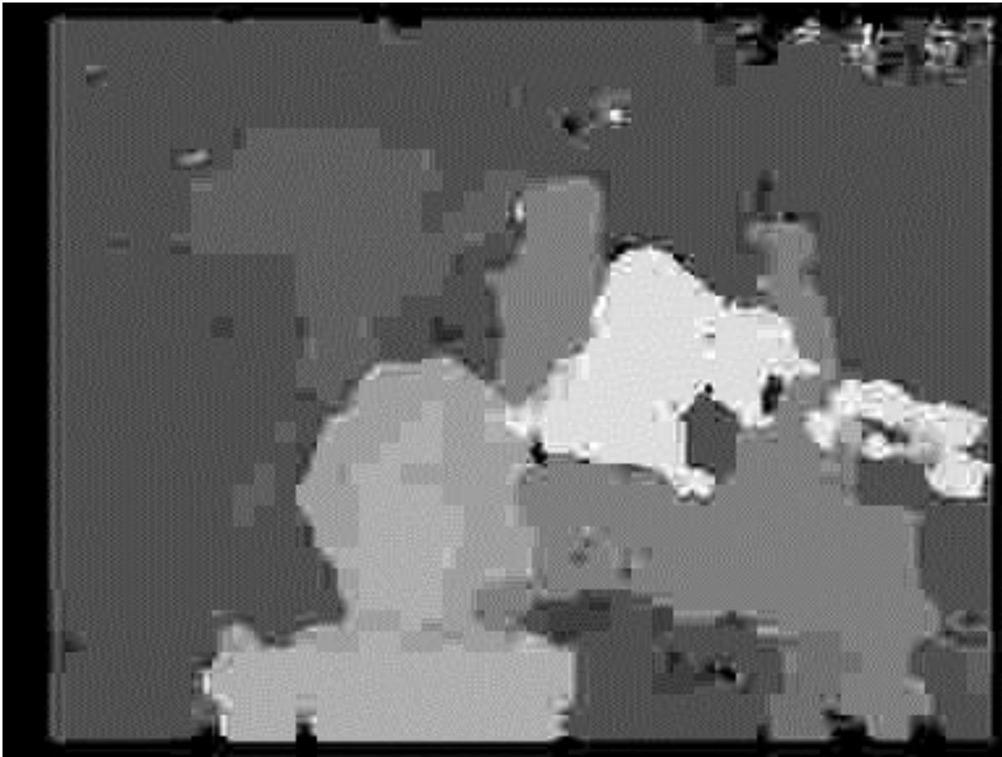


Figure 4.9 Result with disparity distance 15

Figure 4.10 Result Analysis with disparity distance 9

Figure 4.11 Result Analysis with disparity distance 12

Figure 4.12 Result Analysis with disparity distance 15

	Non-occluded region (%)	All regions (%)	Discontinue region (%)
Approximate disparity distance 9	7.83	8.95	31.3
Approximate disparity distance 12	7.83	8.95	31.3
Approximate disparity distance 15	9.29	10.5	37.1

Table 4.2 Accuracy Result for various approximate disparity distance

Approximate disparity range needs to be set carefully. A small range may corrupt the result as no valid matching would be found while too large estimated range would leave a large area blank as no matching information. In applied stereovision system, this is a known variable based on camera disparity setting.

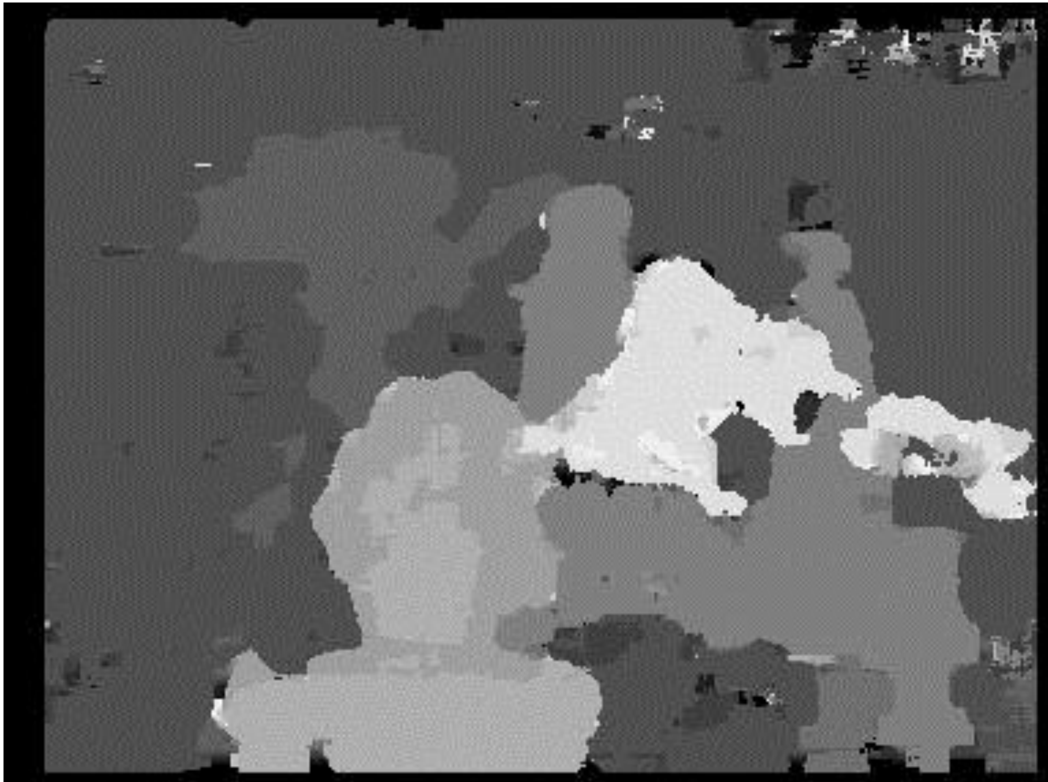


Figure 4.13 Result with fusion ratio 0.6

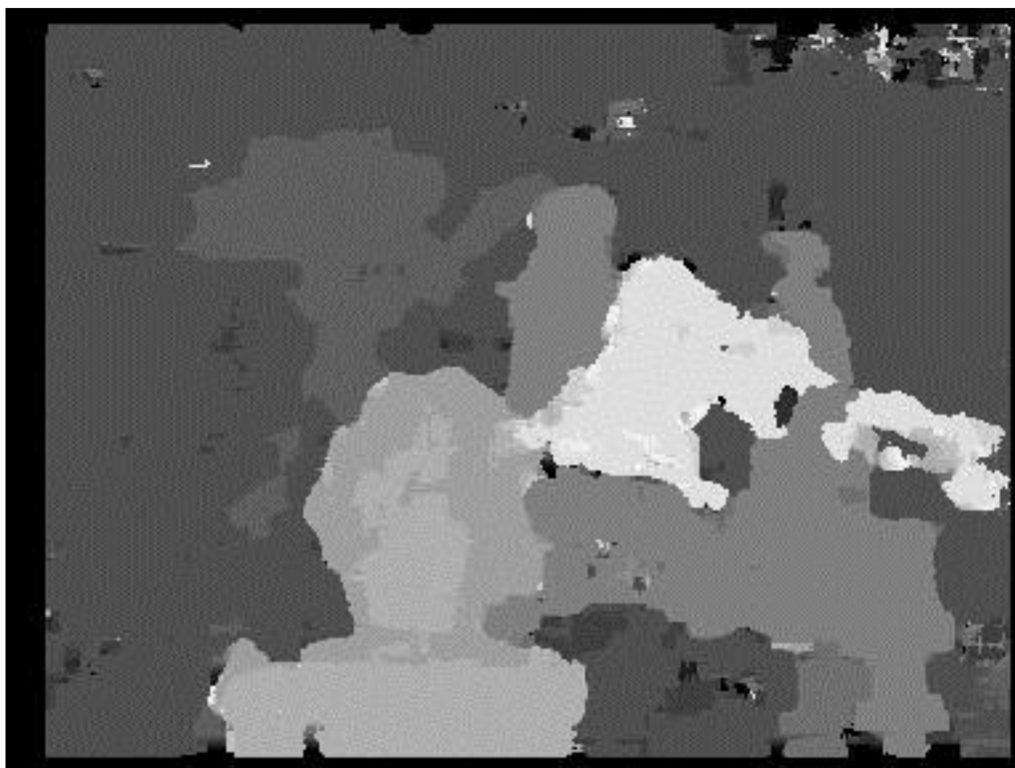


Figure 4.14 Result with fusion ratio 0.7

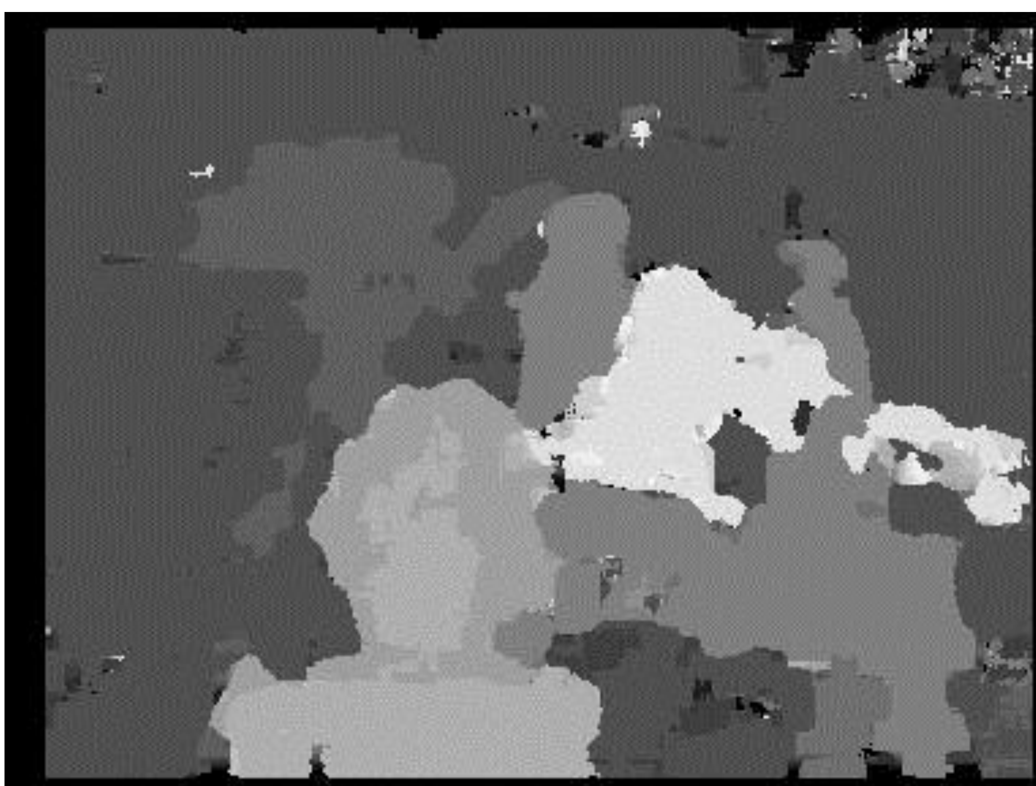


Figure 4.15 Result with fusion ratio 0.8

Figure 4.16 Result Analysis with fusion ratio 0.6

Figure 4.17 Result Analysis with fusion ratio 0.7

Figure 4.18 Result Analysis with fusion ratio 0.9

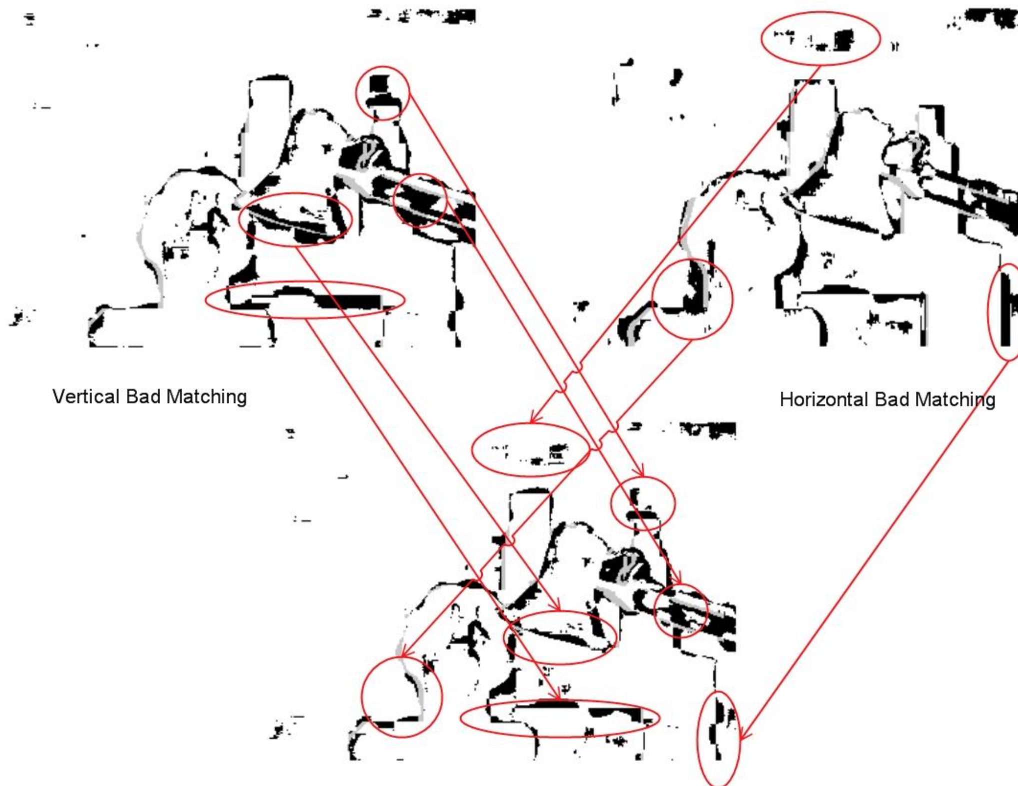
	Non-occluded region (%)	All regions (%)	Discontinue region (%)
Fusion ratio 0.6	7.8	8.94	31.5
Fusion ratio 0.7	7.83	8.95	31.3
Fusion ratio 0.8	7.86	9.01	31.1

Table 4.3 Accuracy Result for various approximate disparity distance

Fusion ratio could be adjusted between 0 and 1 in order to balance between vertical and horizontal matching. By adjusting fusion ratio we could decide to trust either vertical matching or horizontal matching more. The best parameters may vary from different scenes.

The fusion model is proven a success. Figure 4.15 show where are the improvement from original vertical and horizontal matching to the fusion result. Table 4.4 demonstrates the improvement in data.

Figure 4.19 Improvement from Single match to Fusion



	Non-occluded region (%)	All regions (%)	Discontinue region (%)
Vertical	9.03	10.2	36.2
Horizontal	8.92	10.4	34.8
Fusion	7.8	8.94	31.5

Table 4.4 Fusion Accuracy Improvement

The accuracy result of the project is acceptable. As stereo block matching is one of the simplest stereovision algorithms, there would be a promising room for L-shape trinocular to improve the accuracy. Table 4.5 shows the accuracy comparison of the project (GPU_TR_BM) with several other algorithms based on the evaluation in Tsukuba dataset from Middlebury Stereo Page.

	Non-occluded region (%)	All regions (%)	Discontinue region (%)
GPU_TR_BM	7.8	8.94	31.5
Infection [16]	7.95	9.54	28.9
BioDEM [17]	6.57	8.43	28.1

Table 4.5 Accuracy Comparison with Other Stereo Algorithm Implementations

4.2 Speed Result

One of most significant goals and achievements of this project is the real-time stereovision computation ability. With help from GPU programming, this goal is achieved perfectly. As mentioned in the beginning, a commonly accepted frequency for real-time vision is 25Hz–30Hz. So if the system could process a still image (considered as one frame) with 1/30s–1/25s, i.e. 30ms–40ms, the system is possible to achieve real-time. The tests were taken on stand Tsukuba dataset (resolution 384×288), a larger resolution which is commonly used in online video transmission (640×480), an even larger resolution which is typically used for portable device display (800×600) and an ultimate large resolution (1280×960). The important software and hardware environment configurations list as below:

CPU: Intel Core2 i7-2720QM

GPU: NVidia GTX-560M

OS: Linux (Ubuntu 10.10)

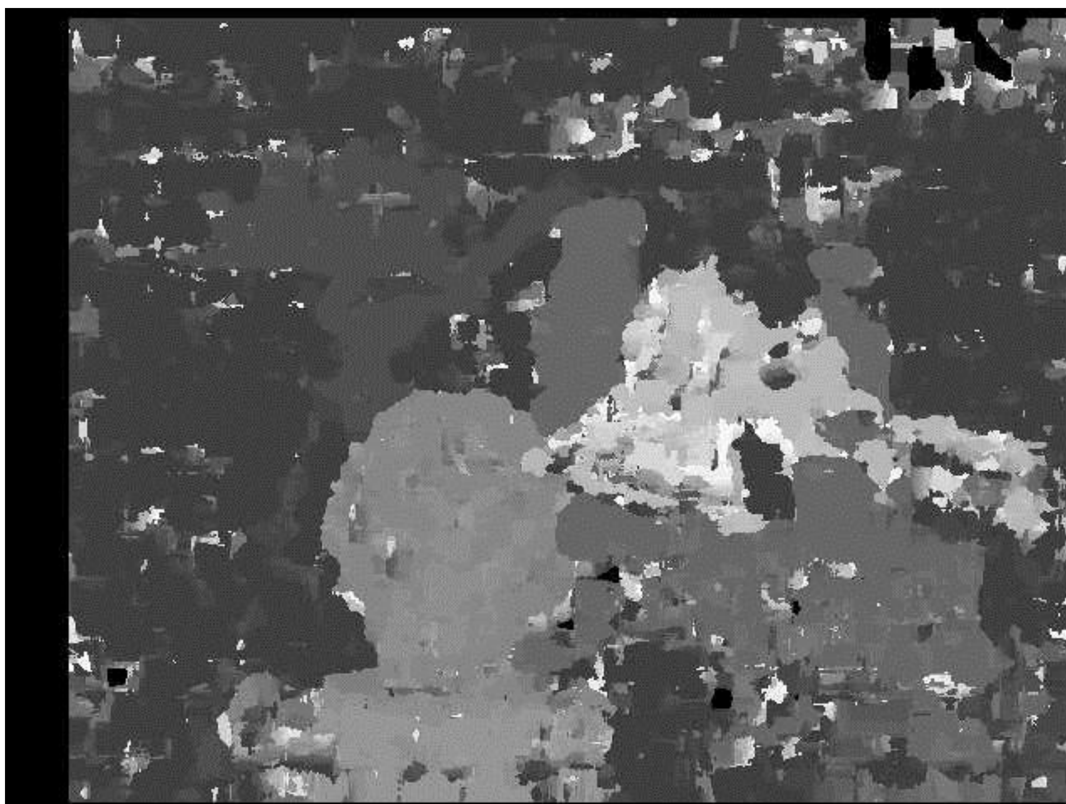
Compiler: GCC 4.4.6

Time spent for processing does not count the time consumed for data transportation between host and device. Table 4.5 shows the average time consumed for each size image. Figure 4.16-Figure 4.18 shows the result of high resolution images. The results are not as good as the original one as many pixels in enlarged images are interpolated in the image rather than original pixel values.

Size	Average Time per Frame(ms)
384×288	2.165
640×480	6.717
800×600	10.465
1280×960	18.84

Table 4.6 Times per Frame for Different Size Image

Figure 4.20 640×480 Tsukuba Result



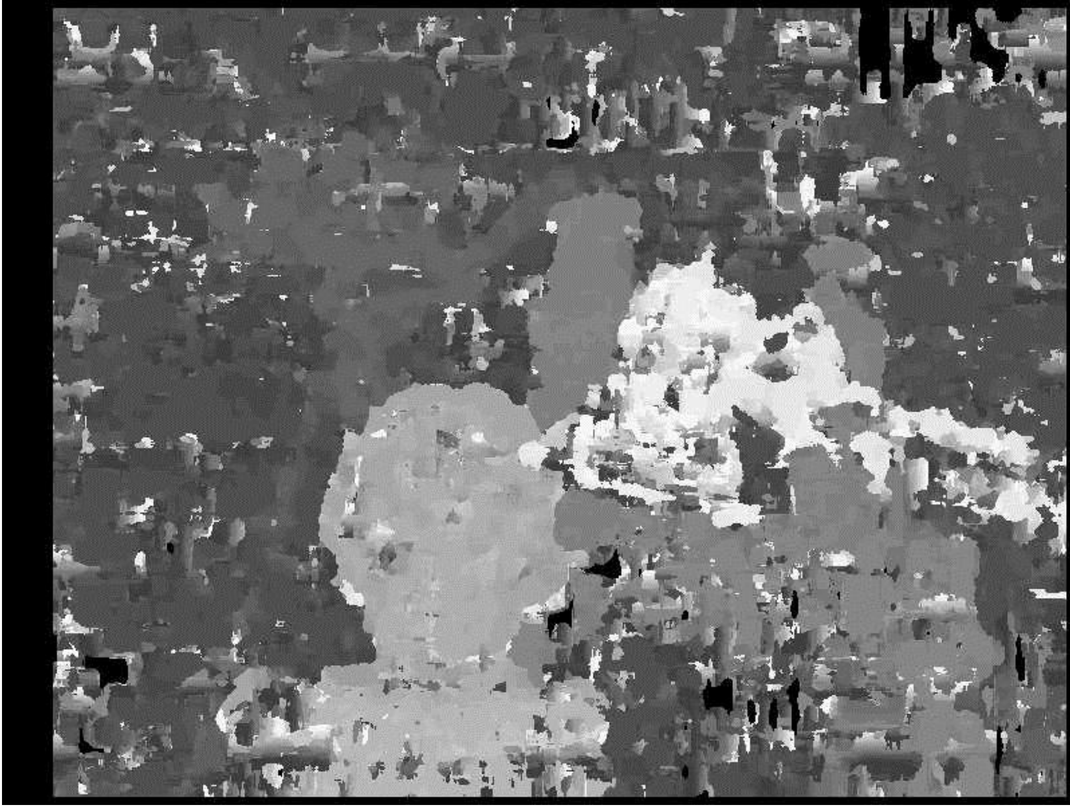


Figure 4.21 800×600 Tsukuba Result

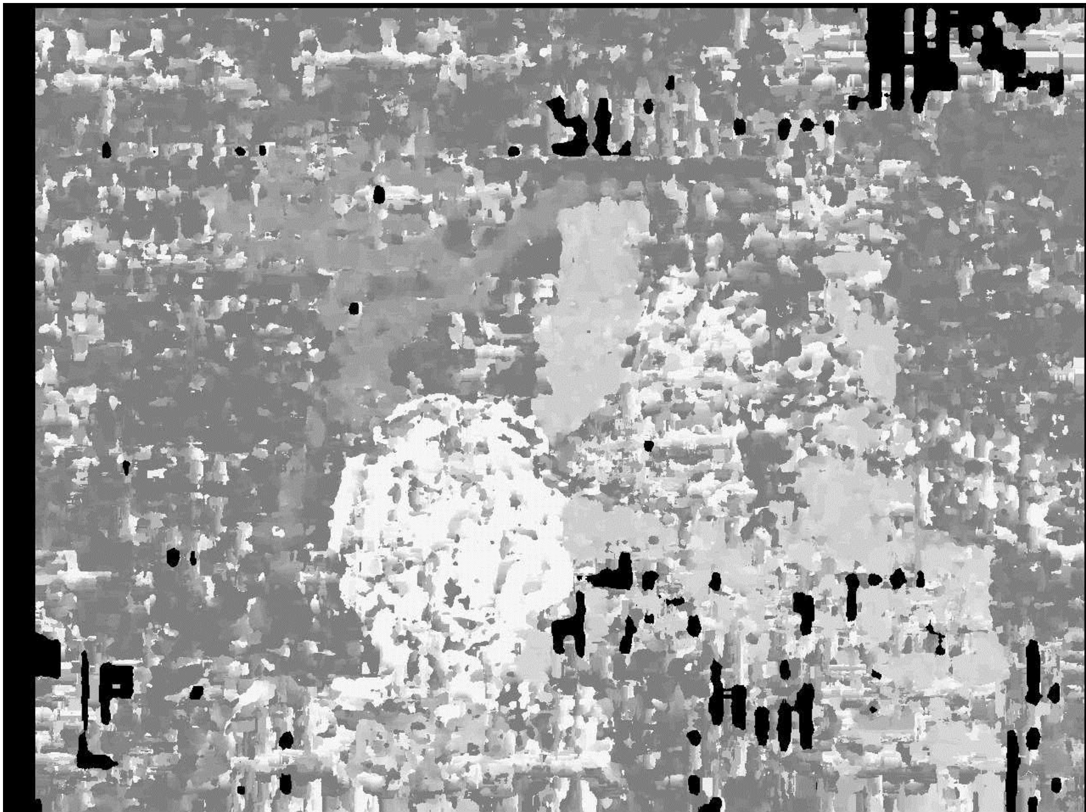


Figure 4.22 1280×960 Tsukuba Result

One problem in the test was about the approximate disparity range. As the test images were enlarged from the original Tsukuba dataset, the disparity range should also enlarge. As a special case, small disparity ranges were skipped from search. This would reduce processing time. However, the whole system is still capable of real-time even consider this factor. Experiment confirms time for 1920×960 would increase to about 25ms which is still within real-time processing speed.

The test is only for one image as a single frame. To apply the method in actual stereovision system, data transportation time must be considered. A possible solution is to set a video buffer to accumulate multiple video frames and concentrating multiple transportations into one.

4.3 Future Development

The result of test has proved the method is successful. Stereo matching accuracy of final disparity image improves a lot from any single match. The computation speed also meets real-time requirement. However, many future developments could make it better from different aspects.

The stereo algorithm applied is stereo block matching. This is one of simplest stereo algorithm applied. As research in stereo matching progresses a lot in recent years, there are a lot of effective algorithms available. Though some of them are too complex to be applied in real-time applications, there are still many algorithms balance well between speed and accuracy. If we could convert some of them into a vertical version, simplify it to accelerate, conduct the similar fusion process in the project and introduce GPU programming into the system, it could become a perfect stereovision method.

Another point could be modified is the fusion model. Currently the fusion model is based on minimum SSD value in each pixel solely. As discussed before, this method generally works fine but not so well in the area not covered by all three input images. If a more dedicated fusion model could be developed, the improvement from single match to fusion result would be greater.

Implementing a GUI (Graphical User Interface) for the project is also a good way to improve. GUI would be a helpful feature to adjust the stereo matching parameters. With a GUI user could capture disparity result with different parameters and compare them easily. It should be fairly easy to develop a GUI with Qt as OpenCV integrates API for Nokia Qt (a library specifically for GUI development).

The ultimate goal of converting the method to a real stereovision system would be a challenging but exciting work. Some configured cameras will replace still images as input data. In discussion before we have concluded that a video buffer to store data first and concentrating data transportation between host and device in order to reduce time waste on data transportation. This would be a must feature in GPU environment. Also the method of synchronization between different cameras will become a new difficulty. Asynchronous data transmission always happens due to different communication quality of camera data connection. This asynchronous data would obviously corrupt the result. Time stamp for image data may help to solve the problem.

Bibliography

[1] Richard Szeliski, *Computer Vision: Algorithms and Applications*, 2010.

[2] Umesh R. Dhond and J. K. Aggarwal, "Binocular versus Trinocular Stereo", IEEE International Conference on Robotics and Automation, 1990

[3] Nicholas Ayache and Francis Lustman, "Trinocular Stereovision for Robotics", IEEE Transactions on Pattern Analysis and Machine Intelligence 13, 1 (1991)

[4] Jane Mulligan and Kostas Daniilidis, "Real time trinocular stereo for tele-immersion", 2001 International Conference on Image Processing

[5] Jane Mulligan, Volkan Isler and Kostas Daniilidis, "Trinocular Stereo A Real-time algorithm and its Evaluation", *International Journal of Computer Vision*, Volume 47

[6] Thomas Hinterhofer and Christian Zinner, "A Trinocular Census-based Stereovision System for Real-time applications", International Conference on Signal Processing, Pattern Recognition, and Applications (SPPRA 2011)

[7] M Harris, "GPGPU: General-purpose computation on GPUs", SIGGRAPH 2005 GPGPU Course, 2005

[8] Gangqiang Zhao, Ling Chen, Gencai Chen, "A Speeded-Up Local Descriptor for Dense Stereo Matching", IEEE International Conference on Image Processing, 2009

[9] Klaus Engel, *Real-time Volume Graphics*, 2006

[10] David G Lowe, "Object recognition from local scale-invariant features", IEEE International Conference on Computer Vision, 1999

[11] Herbert Bay, Andreas Ess, Tinne Tuytelaars, Luc Van Gool, "SURF: Speeded Up Robust Features", *Computer Vision and Image Understanding*, 2008

[12] Ce Liu, J. Yuen, A. Torralba, "SIFT Flow: Dense Correspondence across Scenes and Its Applications, IEEE Transactions on Pattern Analysis and Machine Intelligence", 2011

[13] Robert Laganière, *OpenCV 2 Computer Vision Application Programming Cookbook*, 2011

[14] NVidia Corporation, *CUDA C Programming Guide version 4.0*, 2011

[15] D. Scharstein and R. Szeliski, “A taxonomy and evaluation of dense two-frame stereo correspondence algorithms”, *International Journal of Computer Vision*, 2002

[16] G. Olague, F. Fernández, C. Pérez, and E. Lutton, “The Infection Algorithm: An Artificial Epidemic Approach for Dense Stereo Correspondence”, *Artificial Life*, 2006

[17] J. Martins, J. Rodrigues, and J. du Buf, “Disparity Energy Model Using a Trained Neuronal Population”, IEEE Symposium on Signal Processing and Information Technology 2011.

[18] J. Fung, “Using graphics devices in reverse: GPU-based Image Processing and Computer Vision”, 2008 IEEE International Conference on Multimedia and Expo

[19] Milan Sonka, Vaclav Hlavac, Roger Boyle, *Image Processing, Analysis, and Machine Vision*, 1999

[20] Donald B. Gennery, “Object detection and measurement using stereovision”, 6th International Joint Conference on Artificial Intelligence, 1979

[21] OpenCV official log, http://opencv.willowgarage.com/wiki/OpenCV Change Logs

[22] D. Scharstein and R. Szeliski. “A Taxonomy and Evaluation of Dense Two-frame Stereo Correspondence Algorithms”. *International Journal of Computer Vision*, 2002.

[23] E. Rosten and T. Drummond, “Machine learning for highspeed corner detection”, European Conference on Computer Vision, 2006.

[24] E. RUBLEE, V. RABAUD, K. KONOLIGE and G BRADSKI, “ORB: An Efficient Alternative to SIFT or SURF”. International Conference on Computer Vision, 2011

[25] Minglun Gong, “A GPU-based Algorithm for Estimating 3D Geometry and Motion in Near Real-time”, 3rd Canadian Conference on Computer and Robot Vision, 2006

[26] Ohya, A, “Autonomous navigation of mobile robot based on teaching and playback using trinocular vision”, IEEE Industrial Electronics Society Conference, 2001