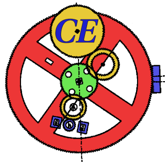


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

CE-MS-2010-32

M.Sc. Thesis

3D Information Extraction Based on GPU

Ying Zhang B.Sc.

Abstract

Our project starts from a practical specific application of stereo vision (matching) on a robot arm, which is first building up a vision system for a robot arm to make it obtain the capability of detecting the objects 3D information on a plane. The kernel of the vision system is stereo matching.

Stereo matching(correspondence) problem has been studied for a few decades; it is one of the most investigated topics in computer vision. A lot of algorithms have been developed, but only a few can be applied in practice because of the constraint from either accuracy or speed requirement.

After the vision system is built, one can get some insights from it, and determine which part of the vision system needs to be improved through experiments. The result shows that the accuracy of current block matching algorithm is enough to be applied in specific environment. Thus, the focus of the afterwards optimization for the currently built vision system is mainly from speed acceleration aspect.

After measuring each stage time cost of 3D sensing part of the vision system, the most time consuming stage is from the stereo matching which generates the disparity map or depth map. At last, the stereo matching part is executed on GPU(Graphic Processing Unit) in order to get some performance enhancement, the final result demonstrates that GPU can make the algorithm run in real time, and it is an ideal platform for the further application development of stereo matching algorithm. Because the original speedup of GPU against to CPU is round 35 times at least for desktop GPU, and the optimized speedup of GPU against to CPU can be more than 100 times at least for desktop GPU.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science

Delft University of Technology

3D Information Extraction Based on GPU

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Ying Zhang B.Sc.
born in Kunming, P.R.China

This work was performed in:

Computer Engineering Group
Department of Microelectronics & Computer Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Copyright © 2010 Computer Engineering Group
All rights reserved.

Delft University of Technology
Department of
Microelectronics & Computer Engineering

The undersigned hereby certify that they have read and recommend to the Faculty of Electrical Engineering, Mathematics and Computer Science for acceptance a thesis entitled **"3D Information Extraction Based on GPU"** by **Ying Zhang B.Sc.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: Oct. 8th, 2010

Chairman:

____
prof.dr.ir. Koen Bertels, CE, TU Delft

Advisors:

____
dr. Georgi Krasimirov Kuzmanov, CE, TU Delft

____
ir. Harry Broers, Vision Group, Philips AppTech

Committee Members:

____
assoc.prof.dr.ir. René van Leuken, CAS, TU Delft

____
ir. Alexander S. van Amesfoort, PDS, TU Delft

Abstract

Our project starts from a practical specific application of stereo vision (matching) on a robot arm, which is first building up a vision system for a robot arm to make it obtain the capability of detecting the objects 3D information on a plane. The kernel of the vision system is stereo matching.

Stereo matching(correspondence) problem has been studied for a few decades; it is one of the most investigated topics in computer vision. A lot of algorithms have been developed, but only a few can be applied in practice because of the constraint from either accuracy or speed requirement.

After the vision system is built, one can get some insights from it, and determine which part of the vision system needs to be improved through experiments. The result shows that the accuracy of current block matching algorithm is enough to be applied in specific environment. Thus, the focus of the afterwards optimization for the currently built vision system is mainly from speed acceleration aspect.

After measuring each stage time cost of 3D sensing part of the vision system, the most time consuming stage is from the stereo matching which generates the disparity map or depth map. At last, the stereo matching part is executed on GPU(Graphic Processing Unit) in order to get some performance enhancement, the final result demonstrates that GPU can make the algorithm run in real time, and it is an ideal platform for the further application development of stereo matching algorithm. Because the original speedup of GPU against to CPU is round 35 times at least for desktop GPU, and the optimized speedup of GPU against to CPU can be more than 100 times at least for desktop GPU.

Acknowledgments

Firstly, I would like to express my gratitude to my supervisors Harry Broers in Philips Apptech and Georgi Kuzmanov from CE group in TU Delft with their assistance to me in all the time of the project and thesis writing. Especially, I would like to thank Harry for offering me the opportunity to do my graduate project in Philips Applied Technology. Undoubtedly, I learnt a lot during this period. In addition, I am thankful to Harry and Georgi's patience during the project and thesis writing, and also their stimulating suggestion when I was confused.

Secondly, I am indebted to the people in Philips Apptech, Pieter-jan Kuyten and Hong Liu. Thanks for Pieter-jan's help in software tools and Hong's suggestion from algorithm aspect, and it is helpful. Although they were very busy, they provided the necessary assistance to me.

Thirdly, I am really grateful to the helps coming from two PhD students in TU Delft. Thanks for Alexander S. van Amesfoort's support in GPUs. With the GPUs provided by TU Delft and Vrije University Amsterdam, my experiment can have enough data and my project can progress forwards. Also, I would like to thank another PhD student from CE group in TU Delft, C.Gou. Every time talking with you, I can get some inspiration, thanks.

Lastly, I would like to appreciate the encouragements and understanding from my parents and other family members during the whole procedure of my study. I dedicate this thesis to you. Also, I am thankful to the discussion and supports from my friends, like C.Li, L.Zhang, and J.Xu etc, and colleagues such as two Wilco and Vikrum.

Ying Zhang B.Sc.
Delft, The Netherlands
Oct. 8th, 2010

Contents

Abstract **v**

Acknowledgments **vii**

1 Introduction **1**

1.1 Stereo Vision Survey 1
1.1.1 Stereo vision system overview 1
1.1.2 Related works 5
1.2 Context, Motivation and Objective 7
1.2.1 Context and Motivation 7
1.2.2 General Objective 7
1.3 Research Goals 8
1.4 Contributions 9
1.4.1 Strategy of Project Impetus 9
1.4.2 System Description 10
1.4.3 Results 10
1.5 Thesis Organization 12

2 3D Sensing Implementation **13**

2.1 Overview 13
2.1.1 Preparation 13
2.1.2 3D sensing part implementation requirement 14
2.1.3 3D sensing part construction 14
2.2 Disparity map generation 15
2.3 Image segmentation (object detection) 18
2.3.1 Object segmentation based on a horizontal plane 18
2.3.2 Object segmentation based on an arbitrary plane 21
2.4 Practical application of the developed 3D sensing part 25
2.5 Conclusion 27

3 3D Sensing Profiling & Stereo Matching Implementation on GPU **29**

3.1 Evaluate 3D sensing performance from the speed aspect 29
3.2 Background 32
3.3 GPU introduction 33
3.3.1 CPU vs. GPU 33
3.3.2 CUDA programming model in GPU 34
3.4 The stereo matching algorithm implementation on GPU 35
3.4.1 Introduction 35
3.4.2 WT-algorithm 35
3.4.3 Analysis of the time-complexity 37
3.4.4 Replace “parallel do”-statement above with thread index in CUDA 37
3.4.5 Implementing the design on GPU 38

4 Experimental Results & Implementation Evaluation **43**
4.1 Comparison analysis - CPU vs. GPU 43
4.1.1 Accuracy comparison 43
4.1.2 Speed comparison 43
4.1.3 Experiment results of both CPU and GPU 44
4.1.4 Discussion of results 44
4.1.5 Analysis of the speedup upper bound for one GPU comparing to one CPU 48
4.2 Evaluate the scalability among different GPUs 51
4.3 Profiling the GPU based implementation 54
4.3.1 Experiment design 55
4.3.2 Task measurement and Analysis 55
4.3.3 Discussion and conclusion 67
4.4 Conclusion 68

5 GPU Optimization **69**
5.1 Principles of the optimization in CUDA 69
5.2 Threads block configuration improvement 71
5.3 Branch or Divergent branch reduction 71
5.4 Algorithm improvement 73
5.4.1 Element based accumulation 73
5.4.2 Column based and Row based accumulation 74
5.5 Efficient management of various memories 76
5.5.1 Minimizing the use of global memory 76
5.5.2 Coalesced accessing to global memory 77
5.6 Final results and Conclusion 78

6 Conclusion **81**
6.1 Summary 81
6.2 General conclusion 82
6.3 Recommendations for future work 82

Bibliography **84**

List of Figures

1.1 Triangulation from stereo camera . . . 2
1.2 Epipolar constraint . . . 2
1.3 Stereo camera in standform form . . . 3
1.4 Depth calculation . . . 3
1.5 Disparity map example . . . 4
1.6 Overview of Stereo vision system . . . 4
1.7 Applications of the robot arm for disabled people . . . 8
1.8 Different application environments . . . 10
1.9 Semi-autonomous Robot Arm System . . . 11
1.10 3D sensing part of the system . . . 11

2.1 Stereo camera applied in the project . . . 13
2.2 The main components of 3D sensing part . . . 15
2.3 Disparity map generation with COTS components . . . 15
2.4 Different qualities of disparity map . . . 17
2.5 Combining image segmentation to the previous stereo vision part . . . 18
2.6 Camera is parallel to the plane . . . 18
2.7 Disparity map with stereo camera parallel to the plane . . . 19
2.8 Binary form generated from the disparity map . . . 19
2.9 Differentiate the blobs by labelling . . . 20
2.10 Objects detection with bounding box and center of gravity . . . 21
2.11 Stereo camera position with an arbitrary angle to the plane . . . 21
2.12 Disparity map from the stereo camera with an angle to the plane . . . 22
2.13 Illustration of two intersection lines in disparity map . . . 22
2.14 Calculated plane with different line fitting methods . . . 24
2.15 Stereo camera position with an arbitrary angle to the plane . . . 25
2.16 Disparity map generation with ROI . . . 26
2.17 Data bandwidth requirement of each stage . . . 26

3.1 User interface of the demo program (provided by Point Grey SDK) . . 30
3.2 Profiling of 3D sensing part . . . 31
3.3 Floating Point Operations per Second and Memory Bandwidth for CPU and GPU . . . 33
3.4 CPU and GPU devotion . . . 34
3.5 Various memory spaces on a CUDA device . . . 38
3.6 Relations between shared memory and thread block . . . 39
3.7 The first step of shared memory preparation . . . 40
3.8 The second step of shared memory preparation . . . 40
3.9 The third step of shared memory preparation . . . 41

4.1 Disparity maps generated by both CPU and GPU . . . 44
4.2 Illustration of Time(N, P fixed) . . . 45
4.3 Illustration of Time(P, N fixed) . . . 46

4.4 Illustration of Speedup(N, P fixed) 46
4.5 Illustration of Speedup(P, N fixed) 47
4.6 Scalability of different GPUs 53
4.7 Profiling of the GPU based stereo matching alg. (I) 55
4.8 Profiling of the GPU based stereo matching alg. (II) 56
4.9 Illustration of the granularity of the block 59
4.10 Bank conflicts of 8×8 threads block 60
4.11 Bank conflicts of 20×20 threads block 60
4.12 Granularity of fixed width block 62
4.13 Coalesced global memory accessing mode 64
4.14 Uncoalesced global memory accessing modes 64
4.15 The real situation in the implementation 65
4.16 Overview of the performance distribution 66

5.1 Illustration of element based accumulation 74
5.2 Illustration of column based and row based accumulation 74
5.3 Color based accessing to memory segment 77
5.4 GPU based 3D sensing part 80

List of Tables

3.1 Results of profiling 3D sensing part 30
3.2 Salient features of device memory 39

4.1 The GPUs in the experiment 44
4.2 Test results for the program - Time(P, N) (seconds) 45
4.3 Speedup (N, P fixed) = Time (384, P fixed) / T (N, P fixed) 45
4.4 Speedup (P, N fixed) = T (1, N fixed) / T (P, N fixed) 47
4.5 Overview of GPU speedup potential to CPU 49
4.6 CPU & GPU hardware parameters 50
4.7 '*Ec*' measurements based on local stereo matching algorithm 50
4.8 Overview of speedup potential among different GPUs 52
4.9 RealSpeedup(GPU2GPU) 53
4.10 '*Eg*' measurements based on SAD local stereo match algorithm 53
4.11 Kernel function resource usage 57
4.12 Granularity of the thread block 58
4.13 Granularity of fixed width block 61
4.14 Performance distribution of each part (seconds) 66

5.1 The original implementation performance 71
5.2 Optimization from threads block configuration 71
5.3 The results of branch reduction from CUDA profiler 72
5.4 Optimization from branch reduction 73
5.5 Result of element based accumulation from CUDA profiler 74
5.6 Results of column based and row based accumulation from '*cudaprof*' . 75
5.7 Optimization from row based accumulation 75
5.8 Effects of texture cache and CUDA array 77
5.9 The effects when using color images as the input 77
5.10 Optimization result from management of various memories on device . 78
5.11 Final results of the improved program - Time(P, N) (seconds) 78
5.12 Improved Speedup(P, N) = T(1, N fixed) / T(P, N fixed) 79

1 Introduction

Depth estimation, as the name implies, is the task that measures the distance between two objects. The immediate way to get the distance information is to use a ruler to measure it, however, in computer vision there is another method can do the same thing and it is more intelligent. Stereo vision is the intelligent one. It uses a pair of cameras to capture the images synchronously and then calculates the distance or depth of the object apart from the cameras through the differences between the two images shot by the two cameras separately. In stereo vision, the kernel function is called stereo matching which compares the differences of the two images and then generates the depth information for each pixel in one of the images. Stereo matching is the foundation and the start point of the whole project.

In the following part, it first makes a survey of stereo vision, next introduces the project background (the application context of stereo vision) and motivation, and then the research goal of the project is followed. After that, the contribution of the master project to the whole project will be provided. Last, the organization of rest of the thesis is given out.

1.1 Stereo Vision Survey

This section mainly makes a survey on stereo vision system including the investigation of relevant stereo correspondence (matching) algorithms.

1.1.1 Stereo vision system overview

Stereo vision is a technique of inferring depth with two or more cameras. It is a wide research topic in computer vision and the relevant algorithms have been developed for a few decades. The research emphasis is on the approaches which are (hopefully) feasible for real time implementation.

Accordingly, the kernel hardware component of stereo vision technique is stereo camera which consists of two or more cameras. Conveniently, with two or more cameras, if we are able to find the corresponding (homologous) points in two images, then we can infer depth by utilizing the trigonometric relation. This idea could be simply illustrated in Figure 1.1.

As seen from Figure 1.1, two points with different depth in a space have the same projection in the reference image, but their corresponding points in target image have different position, thus two points in a space and their corresponding projection in two images consist of a triangular so that it is possible to infer the real distance between P and Q based on this relation.

Now the open question is that how to find the corresponding points in the target

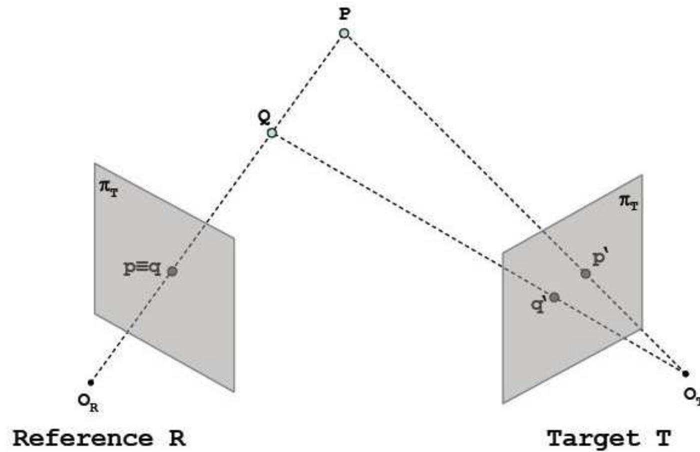


Figure 1.1: Triangulation from stereo camera

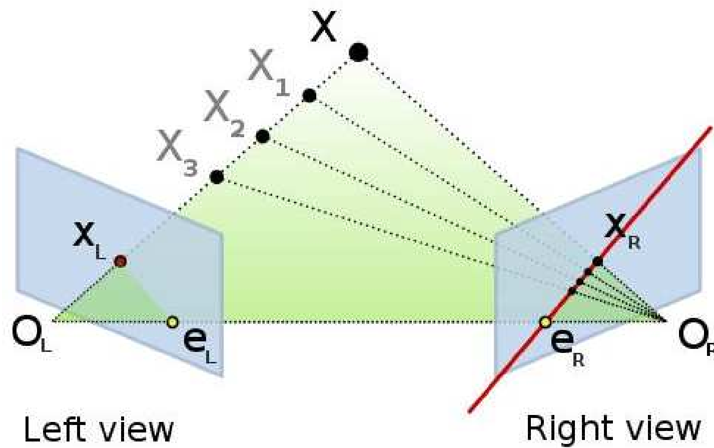


Figure 1.2: Epipolar constraint

image. The naive way is to search them pixel by pixel in the whole target image, obviously it is costly. Actually, there exists an epipolar constraint which could make the situation easier. The story of epipolar constraint could be explained in Figure 1.2.

Epipolar constraint [25] states that if the points on the same line of sight of the left view image project into the same position of the left image plane, then their corresponding points are on the (red) line on the right view image. It means that epipolar constraint reduces the search domain for the corresponding points in the whole target (right) image into a certain line on that image.

However, it is still not over, putting the stereo pair in standard form could further constrain the corresponding points on the same image scan line. In standard form, the position of the corresponding points in left and right images are only different in horizontal direction and this situation is clarified in Figure 1.3. Now finding the corresponding points in target (right) image becomes convenient.

With all the preparation before, it is easy to calculate the depth of the point apart from the camera. From Figure 1.4 and based on the similar triangles (PQ_RQ_T and

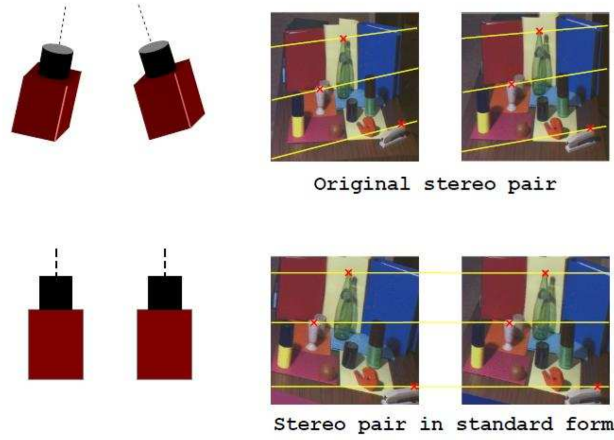


Figure 1.3: Stereo camera in standform form

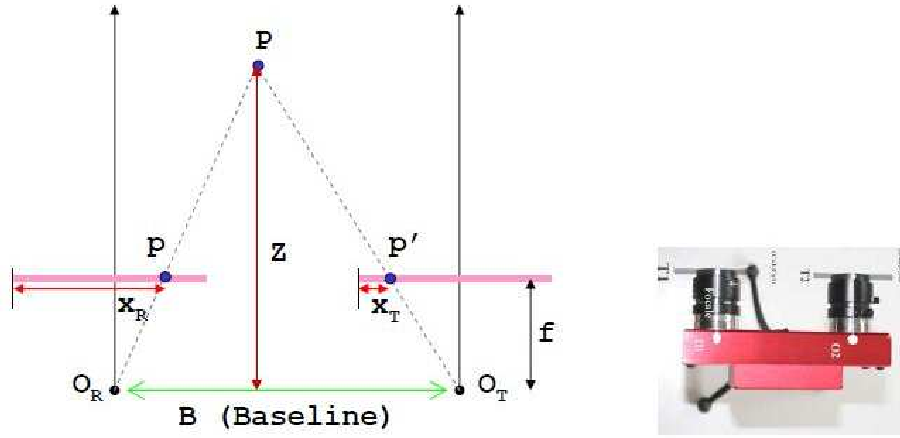


Figure 1.4: Depth calculation

Ppp'), we could get the distance value through the equation Equation 1.1:

$$\frac{b}{Z} = \frac{(b + x_T) - x_R}{Z - f} \implies Z = \frac{b \cdot f}{x_R - x_T} = \frac{b \cdot f}{d} \tag{1.1}$$

In Figure 1.4 and equation Equation 1.1 [11], Z represents the real distance of the target point P apart from the stereo camera baseline, f is the focal length of the camera and b is the baseline distance between two cameras. Finally, we replace (x_R - x_T) with d which is called ***disparity***.

Disparity is the difference of x coordinate between two corresponding points in the reference image and the target image. When we get to know the disparity value of each pixel in the reference image, we could store all these values into another image with grey scaled format which is also called disparity map. Based on equation Equation 1.1, it is known that the disparity has inverse proportion with the real distance, thus the points closer to the camera (with lower Z value) have higher disparity value and also looks brighter. Figure 1.5 illustrates this property of disparity map.

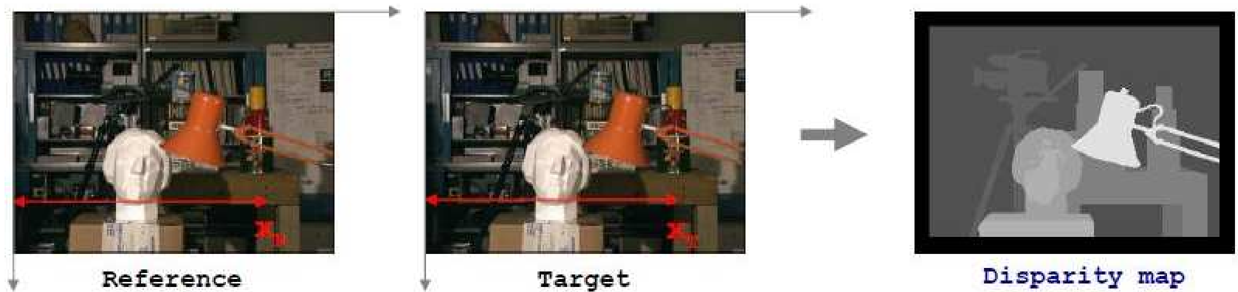


Figure 1.5: Disparity map example

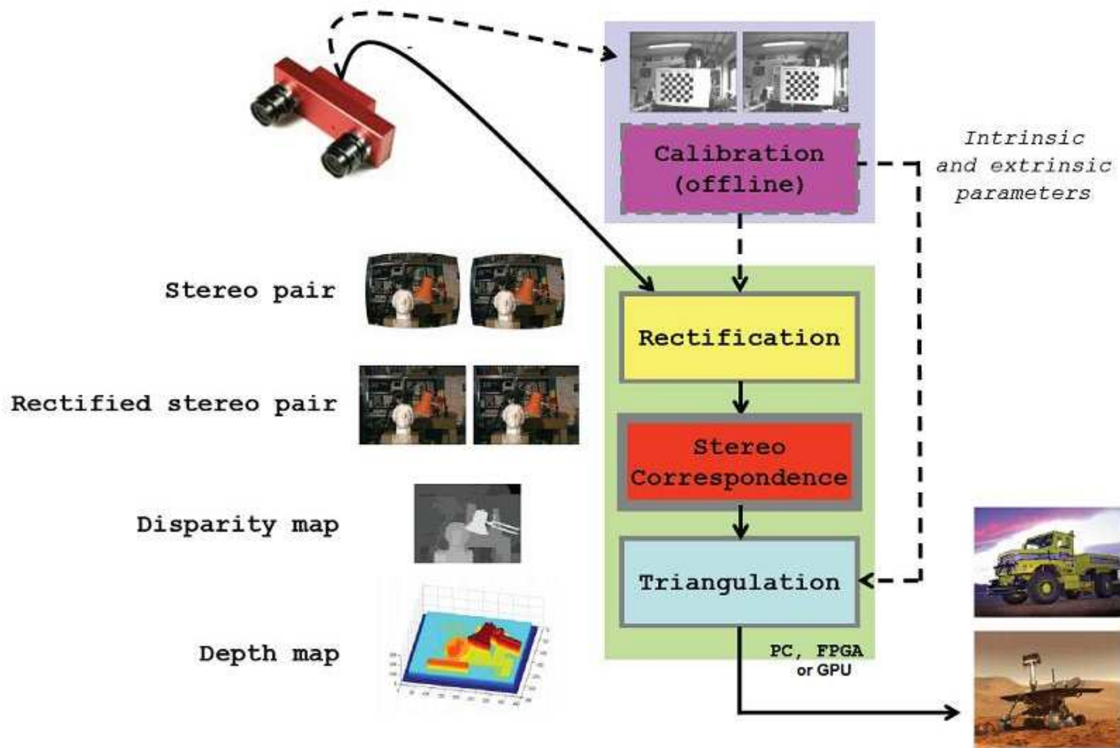


Figure 1.6: Overview of Stereo vision system

Until now, we could generate an overview of a general stereo vision system in Figure 1.6 [11]. The stereo vision system mainly consists of three components which are rectification, stereo correspondence and triangulation. Strictly speaking, calibration is exclusive of the stereo vision system, as it is a procedure to extract the cameras parameters, i.e., the focal length, image center, and lens distortion parameter etc., and then provides them for rectification before the system actually starts working.

After the stereo vision system working, as seen from Figure 1.6, the raw input images from the stereo camera have some lens distortions, so one of the functions of rectification is to remove them. In addition, although the stereo camera hardware is putted in standard form, there is still very tiny error of the position and this can only

be overcome through software way, so another function of rectification is to really turn the stereo camera in standard form.

Next step stereo correspondence is finding homologous points in the rectified stereo pair which is the output of rectification stage and generates the disparity map. This step is the kernel of the stereo vision system also called stereo matching. A lot of algorithms have been developed to deal with correspondence problem, but only very few can be applied in real time. The algorithm used in this step usually determines the quality of disparity map and the performance of the stereo vision system.

Based on disparity map, base line and focal length, the last step is to compute the position (X, Y and Z coordinates) of the correspondence in 3D space. Z could be calculated by equation Equation 1.1, and X, Y could be got through equation Equation 1.2, where x_R and y_R are the horizontal and the vertical displacement of the correspondence in the reference image respectively:

$$Z = \frac{b \cdot f}{d} \implies X = Z \cdot \frac{x_R}{f}, \quad Y = Z \cdot \frac{y_R}{f} \tag{1.2}$$

1.1.2 Related works

After getting some essential insights of stereo vision system construction, it is known that stereo correspondence (or stereo matching) is the key component of the system, accordingly, it will be the concentration of this thesis work.

In fact, until now, a lot of stereo matching algorithms have been developed and the research for stereo correspondence problem is still going on. Fortunately, Dniel's and Richard's work [18] characterized the performance of such a large number of algorithms for stereo correspondence and presented a classification of dense, two-frame stereo methods. Also, from [18], we know that, generally most stereo algorithms perform the (subset of) following steps:

1. Matching cost computation
2. Cost aggregation
3. Disparity computation or optimization
4. Disparity refinement (optional)

Matching cost computation is the method used to quantify the differences of the pixel values in two images; traditionally, SD (squared intensity differences) [15, 20] and AD (absolute intensity differences) [1, 5] are in use for this step. After that, the differences are aggregated in a support window surrounding the pixel for increasing the signal to noise ratio (SNR) and the immediate ways are SSD (Sum of SD) and SAD (Sum of AD). In Step 3, based on the previous steps computation, the disparity for the pixel is determined. There are several strategies to get the disparity and the simplest one is winner-takes-all (WTA), like [1, 5, 22] etc., which selects the disparity for specific pixel based on its minimum cost aggregation. In addition, usually the original disparities are integer values which are discrete, thus disparity refinement as

an option of further making the disparity map more smooth is applied from some specific application requirement of stereo vision.

Actually, the methods used in each step are not just restricted to the mentioned ones above and there are a lot. Usually, most stereo algorithms could be classified into local methods and global methods based on the strategy used for computing disparity in step 3. In the following part, we will further make a basic investigation for both local and global approaches.

1.1.2.1 Local methods

Local algorithms commonly perform "*local matching cost computation* $\Longrightarrow$ *cost aggregation* $\Longrightarrow$ *disparity computation (WTA)*", which first reduce ambiguity (increasing SNR) by aggregating matching cost over a support window and then adopt a simple WTA (winner takes all) strategy to select the disparity. These algorithms search the correspondence along the same scan line of the target image for the pixel in the reference image only based on the matching cost aggregation over a small region, therefore they are called local approaches.

From [10], the author made a comparison of different local stereo matching algorithms on CPU, such as census based, SSD + WTA and SAD + WTA. Overall, SSD and SAD are better than census based solution both on SNR and speed. Between SSD and SAD, for the standard test image *Tsukuba* [18], SSD and SAD have almost the same performance on the quality of disparity map, but SAD is faster than SSD. In addition, larger mask size for SSD and SAD has better SNR, but some details of disparity map will be lost.

In Tangfei's work [21], the author provided a fast block matching SAD algorithm which can get a reduction of over 55% in computational cost without accuracy loss for standard block matching method. Also, other local methods like variable windows [23] and adaptive weights [4] can further improve SNR and accuracy of the disparity map with some loss of speed.

The current state of the art local method is provided by Hosni [3] which uses geodesic distance to distribute weights during matching cost aggregation to reduce ambiguity and it has the best quality of disparity map among local stereo methods, but it takes about 1 minute on standard test images on CPU.

Surprisingly, even though there are some limitations, the widely adopted stereo matching algorithm in practice is the most basic block matching method with fixed window, because it has its own advantages, such as easy to implement, fast and limited memory requirement etc..

1.1.2.2 Global methods

Comparing to local methods, global methods perform "*local matching cost computation* ($\Longrightarrow$ *cost aggregation*) $\Longrightarrow$ *disparity computation (with global reasoning)*", simply speaking, it is a procedure of global energy minimization. The energy consists of two items, local matching cost(e.g., SD and AD in local methods) and *neighbourhood smooth* item which considers the consistency of a pixel's disparity with that of surrounding pixels, and *cost aggregation* is not necessary here. Actually, each pixel has a range of

disparity, thus there is a huge amount of combination of disparities for all pixels in the whole image and global methods select the combination which has the minimal global energy through a global energy minimizing process. Obviously, global methods have much more computation and memory requirement than local methods, thus they are usually slower than local methods. However, global methods have better quality of disparity map than that of local methods, the reason is that, even if the disparity of each pixel is not determined by the minimum local matching cost among all its disparity options, it is assigned so that the global energy of the whole image is minimum and this is the main difference between global and local methods.

For global stereo algorithms implementation, J.Sun's work [19] achieved it with belief propagation. After that, Pedro and Daniel [2] improved the running time of belief propagation strategy with some algorithmic techniques. In addition, Yang and Wang [28] implemented a hierarchical BP on GPU to make BP based solution running in real time, but it is just applicable to small size stereo pair with short disparity range. The current state of the art stereo matching algorithm is a global method which is BP based solution provided by Klaus [8] and it has the highest quality of disparity map for standard test images but not fast accordingly.

Besides belief propagation strategy, there still are other ways of implementation of global algorithms, like dynamic programming [6] and graph cuts [9].

In a word, in practical application especially in real time, although global methods obtain the best quality, they currently still have more constraints than local methods.

1.2 Context, Motivation and Objective

1.2.1 Context and Motivation

During daily life, many disabled people need some assistance for living and care from the surrounding. With the development of technology, building specific tools to help disabled people is already no longer an impossible mission. The original objective of the whole project is to exploit robotic technology for helping the disabled and the elder again obtain some basic capabilities which they lost before. One of the focus areas of this kind of technology is the application of robot arm. The blue print of the whole project is in Figure 1.7.

We can see the different usages of robot arm from Figure 1.7. For instance, Figure 1.7(a) shows the specific application of robot arm which is used for feeding people. For the general use, robot arm can be applied as the realistic extra arm of human as illustrated in Figure 1.7(b): the robot arm is fixed on the wheelchair and people can open the door, pick up objects and do something else with it.

1.2.2 General Objective

The open question is how to implement the whole task and finally to make the robot arm recognize and pick up the targets (in real time). There are several options to reach the objective. An extreme option is to implement a fully-controlled robot arm. It means that each step of a specific task is dictated by the user. Obviously it will

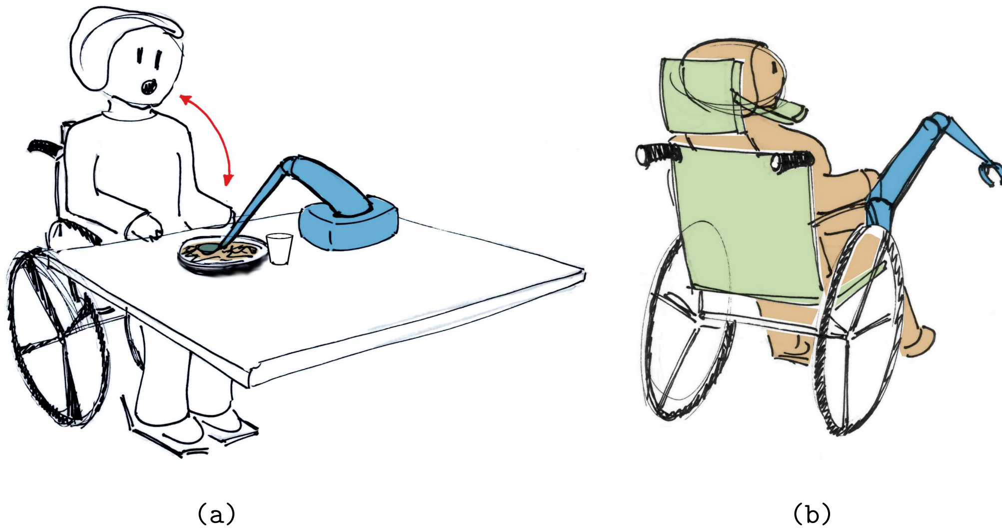


Figure 1.7: Applications of the robot arm for disabled people

be cumbersome, tedious, and even inaccurate. Another extreme is to realize a fully-autonomous robot arm. This one is the most intelligent and is not controlled by user. The system adapts to users' needs and intentions automatically. Without doubts, in a short time, the functionality is difficult to be achieved. An intermediate solution is a trade off between fully-controlled and fully-autonomous. The user just needs to do the very limited key steps and the rest of the operations will be completed by the system. For example, during the task of picking up something, the user could just select the object for the robot arm and rest of things like sensing the object, moving the robot arm and how much force needed to pick up the object are done by the system itself.

In order to realize the semi-autonomous robot arm system, the key step is to provide a specific stereo vision system with a user interface for the existing robot arm. The stereo vision system should be able to provide the distance, shape existing and position information of the object for the robot arm.

1.3 Research Goals

Based on the project objective, naturally, stereo matching and image segmentation will be applied to deal with the problems. Stereo matching is responsible to generate the 3D information of the scene and store it in an image called disparity map. After that, image segmentation technology is applied to the disparity map to extract the main 2D information of the objects, such as the object size and position while the 3rd dimensional information, the distance apart from the cameras, is still kept in the disparity map.

Until now, for the stereo vision system on the robot arm, we will have several questions to answer during the project:

- How to build up the specific stereo vision system prototype for future research

work?

- There is a speed constraint from the robot arm for the vision system which should be at least 6 frames per second for processing the images, thus it is important to know that whether the built stereo vision system could satisfy the speed requirement from the existing robot arm?
- After the stereo vision system is built, which part of it needs to be improved?
- How to improve the built stereo vision system?
- Is GPU suitable for the development of stereo vision application?
- Is it possible to make the built stereo vision system run in real time?

Furthermore, stereo correspondence (matching) problem has been studied for a few decades; it is one of the most investigated topics in computer vision. A lot of algorithms have been developed, but only a few can be applied in practice because of the constraint from either accuracy or speed requirement. Within this project, another goal is to see how far we can go with stereo matching on robotics.

1.4 Contributions

1.4.1 Strategy of Project Impetus

At the beginning of the project, we know nothing about how effectively the vision system will work and we have to build it from commercial-off-the-shelf component which is Point Grey stereo camera and its software development kit (SDK). Based on the situation, the general principle to keep the project going ahead is to start as simple as possible. Therefore, we have a basic assumption about the condition:

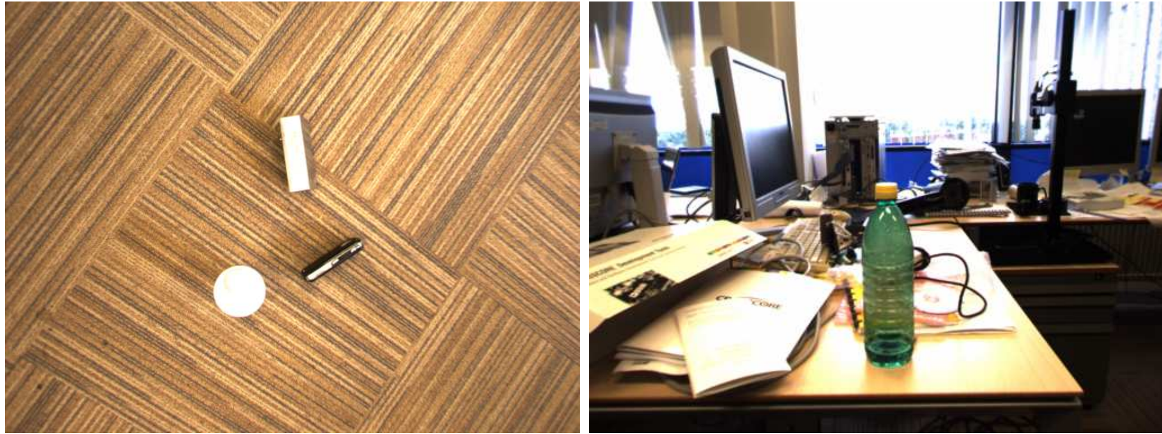
- *The vision system is just set to deal with the case, which is detecting the objects on a plane.*

Because the realistic world is very complicated and it is hard to detect the targets in a general environment, it is a reasonable way to make a progress step by step. For example, Figure 1.8 illustrates the specific environment (Figure 1.8(a)) and the general environment (Figure 1.8(b)), so that we can get an intuitive insight of different environments.

In addition, there will be several image processing algorithms or other algorithms involved in the vision system, thus it is better to make the most use of existing resources, such as the existing algorithms or COTS (commercial-off-the-shelf) components, to realize the vision system prototype. In fact, the main kernel algorithms in the vision system are stereo matching and image segmentation and the final task is how to make use of them to build a simple vision system to detect the objects on a plane.

After building up the system, we will return to evaluate the implementation and find which part still has to be improved, and then we could do further optimization for the vision system.

In summary, the principle for building the vision system is starting from simple and then keeping improving it.



(a) Specific case (b) General case

Figure 1.8: Different application environments

1.4.2 System Description

First, we will have an overview of the semi-autonomous robot arm systems and its construction could be illustrated in Figure 1.9. Actually, the system mainly consists of tree parts which are 3D sensing, user interface and the robot arm and they are controlled by the personal computer. Robot arm is already existed and it is used to verify the vision system which contains 3D sensing and its user interface. The Msc project will concentrate on 3D sensing part only.

3D sensing part is used to extract the targets information, such as the depth apart from the cameras, size, and the position, and provide them to the user interface. After that, the user could interact with the robot arm based on the object information in the user interface. In Figure 1.10, it illustrates the main functions in 3D sensing part. 3D sensing part is made up of three modules and they are rectification, stereo matching and image segmentation. Rectification is a preparation step of stereo matching, it helps stereo matching generate disparity map easily. Stereo matching is responsible to get the 3D information of the scene, especially each pixel value of disparity map stores the depth information. Finally, a simple image segmentation algorithm then could be applied to detect the target object and catch its key information from the disparity map, such as the area and the position of the target.

1.4.3 Results

The main contributions of the thesis work to the robot arm stereo vision system are referred in the following:

1. Make a survey of stereo matching algorithm, such as how it works and what the state of the art is.
2. Build the first half of the 3D sensing part, which is from the capturing images stage to stereo matching stage, with Point Grey stereo camera and its SDK so

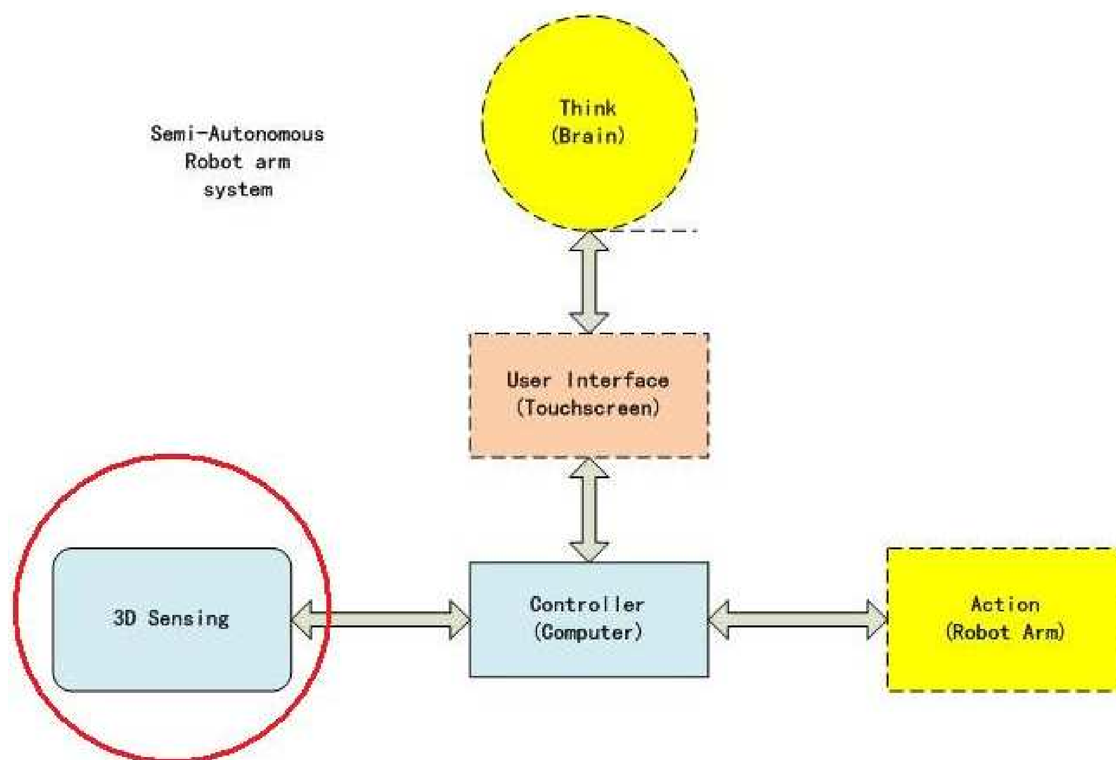


Figure 1.9: Semi-autonomous Robot Arm System

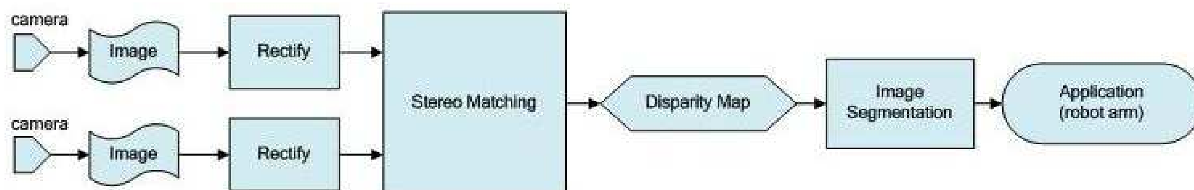


Figure 1.10: 3D sensing part of the system

that it can generate disparity map first.

3. Complete the whole 3D sensing part by combining a basic image segmentation algorithm to do some processing on the disparity map, which could extract the boundary and the position of the objects from the disparity map while the stereo camera is parallel to the plane
4. Further improve the functionality of image segmentation stage so that it can still extract the boundary and the position of the objects from disparity map while the camera has an arbitrary angle with the plane.
5. Apply ROI (region of interest) method to make sure the stereo vision system prototype speed satisfy the requirement of the robot arm, which is that the speed should always be more than 6 fps, and also more robust.
6. Evaluate the performance of 3D sensing part from its speed aspect and find that stereo matching, which applies the block matching method, is the most time

consuming stage.

7. Implement the original CPU based block matching method from scratch which is impossible to run in real time in any case.
8. Port the above block matching method on GPU with scalability to be accelerated with CUDA.
9. Evaluate the primary performance of the block matching method on GPU and find the possible optimization points based on the GPU hardware architecture.
10. Further optimize the GPU solution of block matching method and achieve a significant speedup so that the case of 640x480 image size, with block size 11X11 and 55 pixels disparity range, could run in real time on GTX 280. Especially, when the block size is down to 5X5, the block matching method could run at **128.5 frames per second** in theory with the image size 640x480 and 50 pixels disparity range on GTX 280.
11. Compare GPU based solution of the stereo matching algorithm with its CPU based solution and also compare the GPU based solution performance on different GPUs with different compute capability.
12. Demonstrate that GPU is an ideal platform for further developing the stereo vision system based on block matching method to make it run in real time or near real time.
13. (To do) Implement a GPU based stereo vision system by replacing the stereo matching algorithm of Point Grey camera with the developed GPU based algorithm in 3D sensing part.

1.5 Thesis Organization

For the rest of the thesis, it is organized as follows:

Actually, stereo matching is the kernel function in the stereo vision system, thus in Chapter 2, we will first discuss how to build the prototype of the stereo vision system for the robot arm.

In Chapter 3, based on the implementation of the specific stereo vision system, we will first evaluate its performance from speed aspect. Then we explain the implementation of the most time consuming stage of the stereo vision system, which is block matching method algorithm, on GPU to accelerate it.

in Chapter 4, after the GPU based implementation of stereo matching algorithm, we make a comparison of the block matching method on different platforms including CPU and several GPUs. Furthermore, we evaluate the GPU based solution and find the possible optimization places.

In Chapter 5, we will further optimize the GPU based solution for block matching method based on the results from Chapter 4.

Finally, in Chapter 6, we will draw the conclusion of the whole thesis work and discuss about the possible future work for the stereo vision system.

3D Sensing Implementation 2

In this chapter, the contributions made are the items from the second to the sixth referred in Section 1.4.3 and they are:

1. Generate the disparity map with a moderate quality.
2. Build 3D sensing part when the camera is parallel to the plane.
3. Make 3D sensing part still work while the camera has an angle with the plane.
4. Make 3D sensing adapt to more practical environment with ROI method.

At the same time, the following research questions are answered:

- How to build up the stereo vision system prototype?

2.1 Overview

2.1.1 Preparation

Actually, 3D sensing part is an practical application based on stereo vision technique and we will start its implementation from commercial-off-the-shelf (COTS) components. In our project, the kernel components are the Point Grey stereo camera and its SDK (refer to [17] and [16]), we could easily build up the stereo vision part easily. The stereo camera is shown in figure 3.1 and it consists of three aligned cameras, but in the application there are only two of them being used.

Figure 2.1: Stereo camera applied in the project

2.1.2 3D sensing part implementation requirement

As we explained in Section 1.4.1, our intention is to build up the 3D sensing part to satisfy the basic requirement and then further do improvement on it.

There are several basic requirements for the capability of 3D sensing part from the existing robot arm:

1. Extracting 3D information of the objects on a plane

From the robot arm side, it needs some basic 3D information of the targets so that it could do the reactions with the instructions from the user. The basic objects information includes the area, bounding box and position of the objects. The position of the objects consists of its distance apart from the camera and the XY coordinates information of the object center in the image.

2. Certain robustness

Also, 3D sensing part should be in the nature of certain robustness. The reason is that, this simple vision system is going to be used as the eyes of the robot arm, even if it is fixed in one place, it is still rotatable in a small range. Thus 3D sensing part should accordingly provide the mentioned essential information of objects when the robot eyes rotate in a small range.

3. Speed

In addition, in order to make the robot arm react in time to finish the task, such as picking up objects etc., the stereo vision system should provide the objects information for the robot arm with the speed of at least 6 Hz (or frames per second), the faster the better. Furthermore, our CPU is Xeon 5110 1.6 GHz and the Point Grey stereo camera can capture the image with the speed of 16 Hz at most. In current situation, we have to make the 3D sensing part run in the speed range from 6Hz to 16 Hz with the CPU.

2.1.3 3D sensing part construction

Considering the requirements for 3D sensing part of the stereo vision system, we could build up 3D sensing part in such a way:

For the first half of 3D sensing part, we will use stereo camera to generate the disparity (depth) map of the objects on a plane. The disparity map only stores the depth information of both objects and the plane. For the rest of 3D sensing part, we will extract or segment the objects from the plane based on the differences of the depth between objects and the plane apart from the camera in disparity map. Thus, the 3D sensing part is different from the traditional stereo vision system (Section 1.1.1) in its last stage, which is image segmentation.

The construction of 3D sensing part can be illustrated in Figure 2.2.

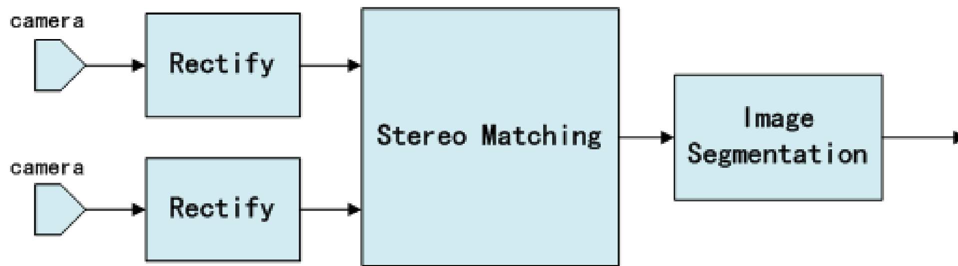


Figure 2.2: The main components of 3D sensing part

2.2 Disparity map generation

After getting some insights of 3D sensing part, the first step is to generate disparity map and this will be done with COTS (commercial-off-the-shelf) components, which are Point Grey stereo camera (Figure 2.1) and its SDK (refer to [17] and [16]). The procedure is shown in Figure 2.3. Simply speaking, the raw images are grabbed by the stereo camera and then are rectified, finally based on the rectified image pair, stereo matching is used to calculate the disparity map.

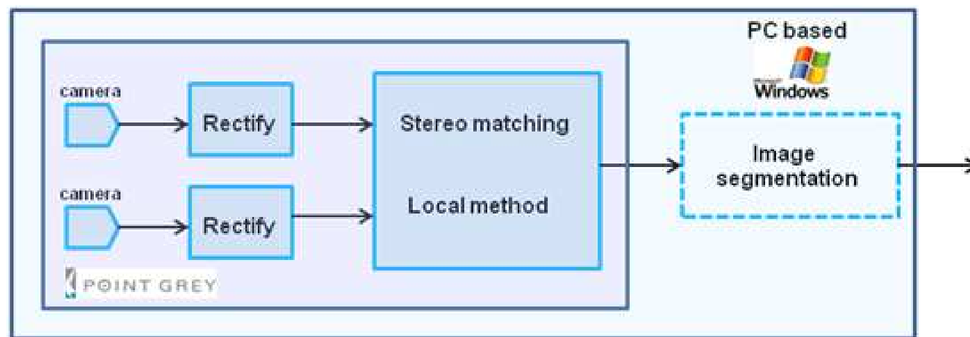


Figure 2.3: Disparity map generation with COTS components

There are several steps to complete the whole procedure of disparity map generation with the API offered by the stereo camera:

1. Open the stereo camera.
2. Initialize camera capturing context.
3. Do camera calibration - get the parameters of the stereo camera, such as focal length, base line etc.
4. Select two of the cameras for working.
5. Set the resolution for the image which will be rectified and set the stereo matching algorithm parameters, such as disparity range, mask size and so on.
6. While loop:
   (a) Grab the raw image pair
   (b) Do rectification on the image pair
   (c) Do stereo matching on the rectified image pair

(d) Display the disparity map and the reference image.

7. Close the stereo camera.

The stereo matching algorithm used by the stereo camera is based on SAD correlation in a fixed size block, which is a local method. The intuition behind the approach does the following:

- **begin**
- **for** each pixel **do**
    1. Select a neighbourhood of a given square size from the reference image;
    2. Compare this neighbourhood to a number of s in the other image along the same row;
    3. Select the best match based on the minimum difference.
- **end**

Furthermore, comparison of neighbourhoods or masks is done using the following formula:

$$\min_{d=d_{min}}^{d_{max}} \sum_{i=-\frac{m}{2}}^{\frac{m}{2}} \sum_{j=-\frac{m}{2}}^{\frac{m}{2}} \mid I_{right}[x+i][y+j] - I_{left}[x+i+d][y+j] \mid \tag{2.1}$$

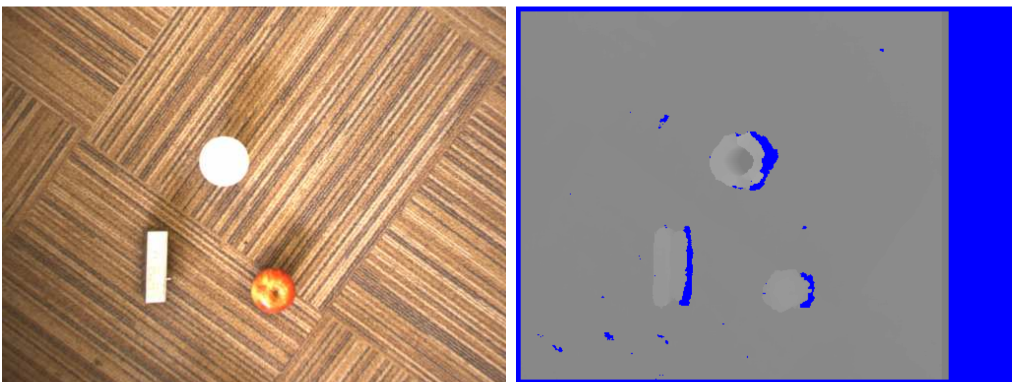
Where:
d_{min} and d_{max} are the minimum and maximum disparities.
m is the mask size.
I_{right} and I_{left} are the right and left images.

In addition, the quality of disparity map does not only determine the results of image segmentation in the next step, but also determine the reliability of the whole 3D sensing part. Therefore, it is important to get a disparity map with the quality as good as possible. With the current block matching method used in the stereo camera, there are two ways to improve the quality of disparity map. One way is to adjust the parameters of the algorithm. Disparity range, such as minimum and maximum disparity, and mask size are the two key factors. If the parameters are wrong, the disparity map will be nothing, like Figure 2.4(a). Another way is adjust the environment. Through experiments, the objects on a plane with obvious texture have cleaner disparity map than those on a textureless plane. Figure 2.4(b) is the disparity map of the objects on a textureless plane and there are a lot of noise (blue parts) in the disparity map, although we could find the objects. Undoubtedly, both Figure 2.4(a) and Figure 2.4(b) are low quality. However, Figure 2.4(c) is the disparity map of the objects on a texture plane with correct parameters, which has a clean view and this is the one it should be.

Until now, disparity map generation is completed, afterwards the image segmentation will be applied to extract the objects information on the plane.

(a) wrong parameters

(b) objects on a textureless plane



(c) texture plane with correct parameters−good quality

Figure 2.4: Different qualities of disparity map

2.3 Image segmentation (object detection)

Disparity map only stores the depth information of the whole image in every pixel with grey scaled value and it does not contain other information for describing the objects, such as the position, the area and the bounding box of the objects, thus image segmentation based on disparity map (depth information) is added to extract the basic information of the objects. The task of this section is illustrated in Figure 2.5: 3D sensing part is fully built up with combining an image segmentation algorithm.

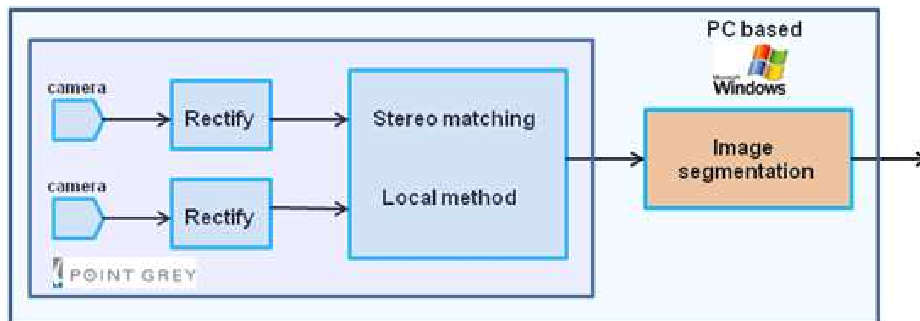
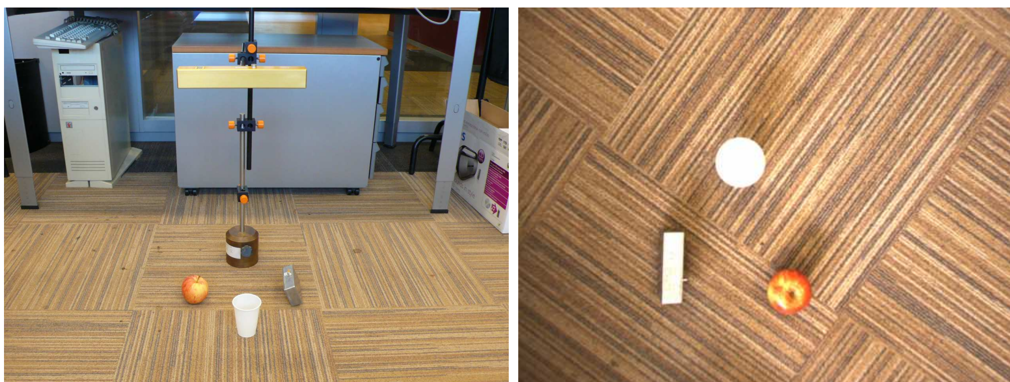


Figure 2.5: Combining image segmentation to the previous stereo vision part

2.3.1 Object segmentation based on a horizontal plane

In order to make the situation simple, we first assume that the objects are on a horizontal plane, which means that the stereo camera is parallel to the plane as shown in Figure 2.6.

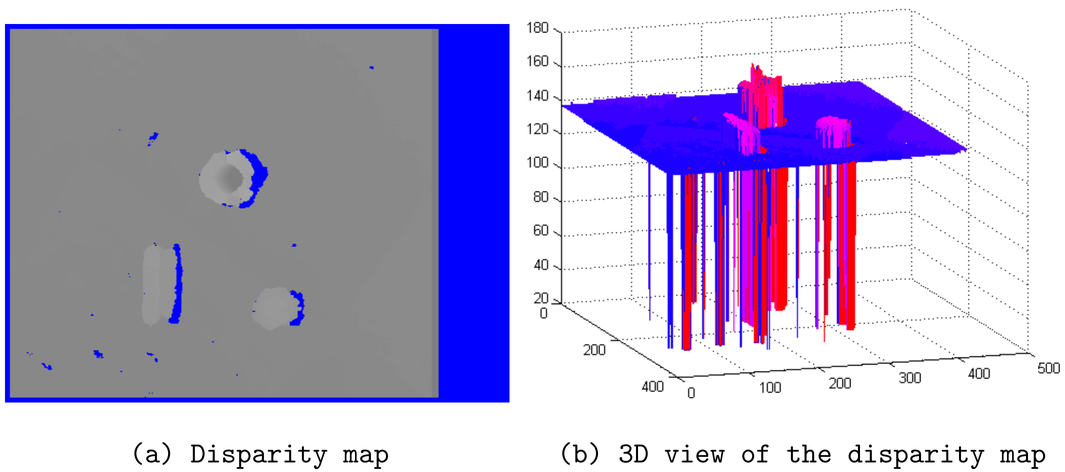


(a) Camera position

(b) Grabbed reference image

Figure 2.6: Camera is parallel to the plane

Based on the parallel position of the stereo camera to the plane, the disparity map generated is shown in Figure 2.7(a) and Figure 2.7(b) illustrates the disparity map in a 3D view based on the pixel values in Matlab:



(a) Disparity map (b) 3D view of the disparity map

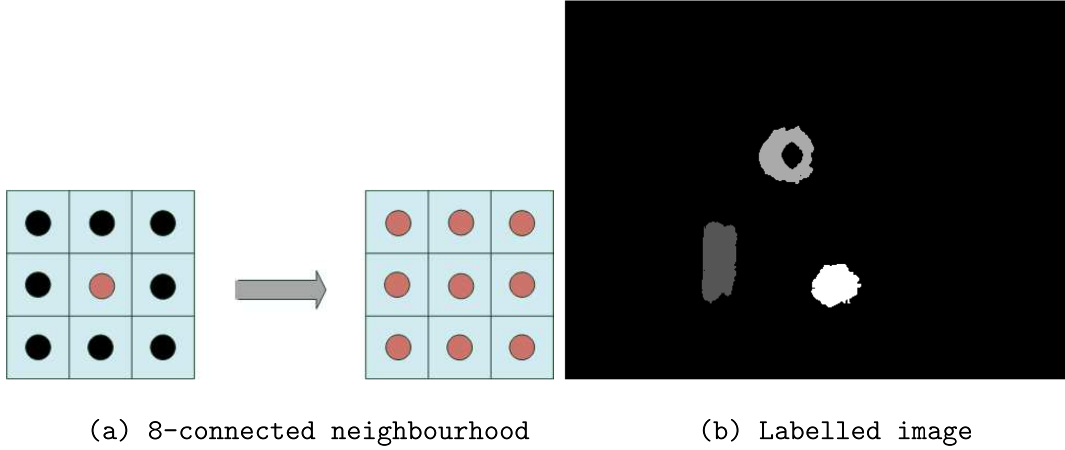
Figure 2.7: Disparity map with stereo camera parallel to the plane

From Figure 2.7, it is easily seen that, as the camera is parallel to the plane, so everywhere of the plane has almost the same distance apart from the stereo camera and accordingly they have almost the same pixel values. However, the objects are on the plane and closer to the camera, so their pixels values (disparities) are larger than those of the plane. Based on this state, the objects can be extracted easily from the background (the plane) with their pixels values. After that, we could calculate the basic information of the objects. There is a simple image segmentation algorithm to complete the task. In this simple image segmentation algorithm, there are several steps to get the area, bounding box and the center information of the objects and these steps are in the following:

1. Extract the objects from background (plane) by setting a threshold value manually.
   By setting a threshold value, the pixels below this threshold are set to zero, and others are kept in the image, which means the plane is removed while the objects regions are preserved. The result of this step is shown in Figure 2.8.

Figure 2.8: Binary form generated from the disparity map

2. Differentiate the blobs by labelling based on 8-connected neighbourhood.
   Considering the filtered disparity map (Figure 2.8) by threshold, the objects can be labelled with different values based on 8-connected neighbourhood. Generally, around a pixel, there are 8 positions close to it shown in Figure 2.9(a), if some of these 8 positions have the same value (non zero) with the center pixel, and then all these pixels will be seen as being connected with a label. Finally, the objects will be differentiated by their own labels. The labelled image is shown in Figure 2.9(b).



(a) 8-connected neighbourhood (b) Labelled image

Figure 2.9: Differentiate the blobs by labelling

3. Get the blobs properties−area, bounding box, and center of the objects
   Based on the labelled image, we could easily calculate the basic information of the objects. For the area of the object, it is gotten by accumulating the number of pixels with the same value (label). For the bounding box of the object, it is define by the left most pixel, right most pixel, top most pixel, and bottom most pixel coming from the blob in the image. Last, for the center of the object, we will use its center of gravity to represent it. Centre of gravity is the average location of the weight of the object and here we will use the area instead of the weight to compute the objects center of gravity in the image. Equation 2.2 is used to calculate the center of gravity of the object in the image. After getting the basic information of the objects, we could locate the objects and the result is show in Figure 2.10.

$$Centre(X, Y) \Longrightarrow \begin{cases} X_{coordinate} = \frac{\sum_{i=1}^{n} x_i}{blobArea} \\ Y_{coordinate} = \frac{\sum_{j=1}^{n} y_j}{blobArea} \end{cases} \tag{2.2}$$

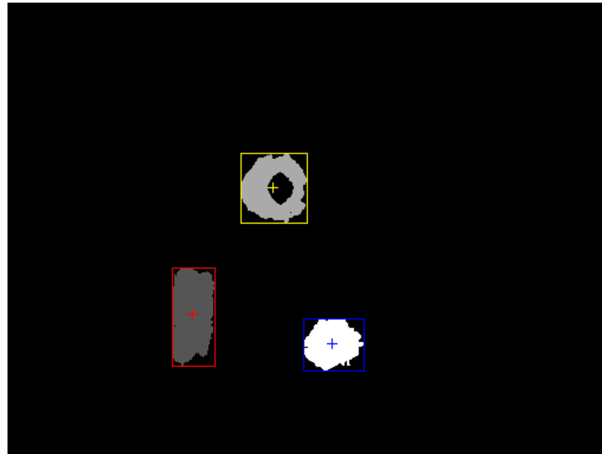
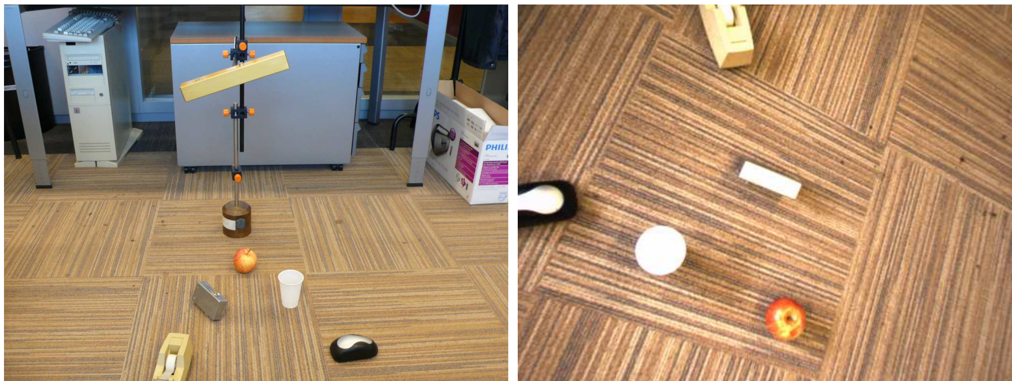


Figure 2.10: Objects detection with bounding box and center of gravity

2.3.2 Object segmentation based on an arbitrary plane

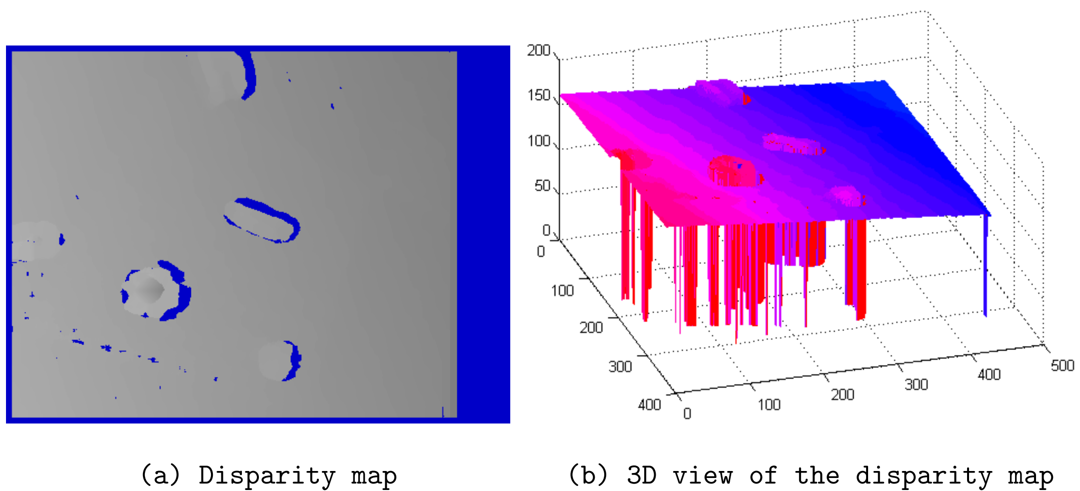
In consideration of the requirement for the certain robustness of 3D sensing part in Section 2.1.2, the fixed stereo camera is rotatable in a small range, so the camera can not always be parallel to the plane. When the stereo camera has an angle with the plane (shown in Figure 2.11(a)), the disparity map will be different from the previous one and the result is illustrated in Figure 2.12.



(a) Stereo camera with a slope (b) Reference image

Figure 2.11: Stereo camera position with an arbitrary angle to the plane

From Figure 2.12, every point of the plane does not have (almost) the same disparity value ever, or to the stereo camera, the plane is not horizontal any longer. Right now, we can not easily remove the plane (background) by setting a threshold, thus the previous image segmentation can not be used directly here to extract the basic information of the objects. Only if we could find what the plane is, then the objects could be extracted from the background by subtracting the plane from disparity map, and the rest of the previous image segmentation algorithm can be used to get the information of the

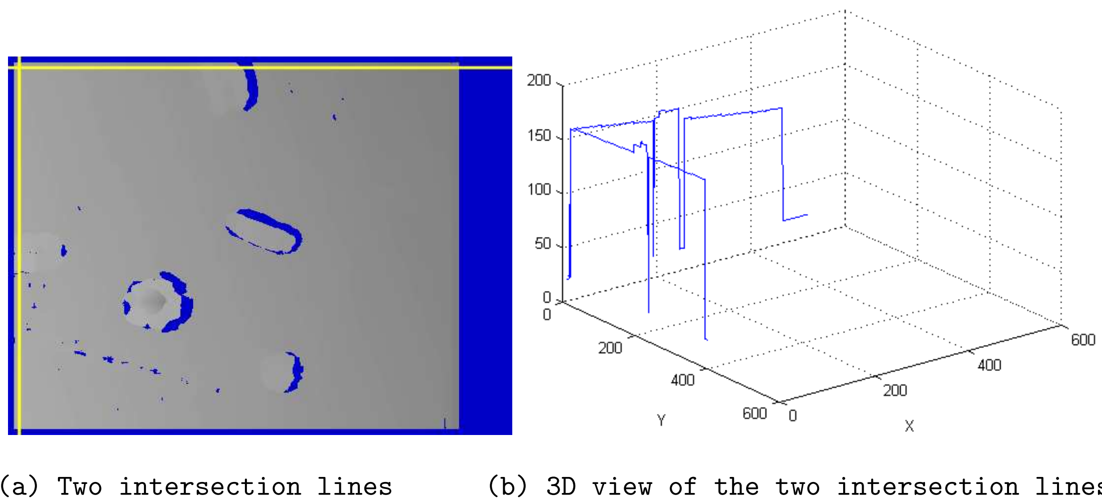


(a) Disparity map (b) 3D view of the disparity map

Figure 2.12: Disparity map from the stereo camera with an angle to the plane

objects. Thus, what we will contribute to the previous image segmentation algorithm is mainly on its first step.

There are two ways to calculate the plane: one is to do plane fitting directly, another one is doing line fitting to two intersection lines on the plane and then calculating the plane from these two intersection lines. Obviously, the first method is more reliable because it will consider all of the points ($O(n^2)$)on the plane, however it is possible to be so slow that 3D sensing part can not be used by the robot arm. Thus we will finally use the second method which is using line fitting ($O(n)$) to fit the plane and the two intersection lines are illustrated in Figure 2.13.



(a) Two intersection lines (b) 3D view of the two intersection lines

Figure 2.13: Illustration of two intersection lines in disparity map

From Figure 2.13(a), the two selected lines are close to the boundary of the disparity map, as most objects are close to the center region of the image, so the values of the two

line are not often affected by the objects. Also we assume the intersection point of the two line is (m, m) in XY coordinate, in addition, the horizontal line is in X-direction and the vertical line is in Y-direction. From Figure 2.13(b), we could see the 3D view of the two lines we select from disparity map. Naturally, if we can calculate the two lines in 3D space, then we could find the plane they locate. The way to calculate the plane with two intersection lines is in the following:

The vertical line in 3D space is:

$$\begin{cases} Z = a_1 Y + b_1 \\ X = 0 \end{cases} \tag{2.3}$$

$\implies$ Its vector is: $0\vec{i} + \vec{j} + a_1\vec{k}$

The horizontal line in 3D space is:

$$\begin{cases} Z = a_2 X + b_2 \\ Y = 0 \end{cases} \tag{2.4}$$

$\implies$ Its vector is: $\vec{i} + 0\vec{j} + a_2\vec{k}$

Thus the normal vector of the plane including these two lines is calculated by the cross product of the vectors of the two lines:

$$(0\vec{i} + \vec{j} + a_1\vec{k}) \times (\vec{i} + 0\vec{j} + a_2\vec{k}) = \begin{vmatrix} \vec{i} & \vec{j} & \vec{k} \\ 0 & 1 & a_1 \\ 1 & 0 & a_2 \end{vmatrix} = a_2\vec{i} + a_1\vec{j} - \vec{k}$$

$\implies$ The plane is:

$$a_2 X + a_1 Y - Z + c = 0 \tag{2.5}$$

Substitute Equation 2.5 with Equation 2.3 or Equation 2.4 $\implies$

$$c = b_1 = b_2 \tag{2.6}$$

In addition, we know that the intersection point of the two lines in XY coordinate is *(m, m)*.
So, substitute Equation 2.5 with intersection point, Equation 2.3 and Equation 2.6, we get:

$$c = b_1 - a_2 m \tag{2.7}$$

Therefore, substitute Equation 2.5 with Equation 2.7, the final plane is:

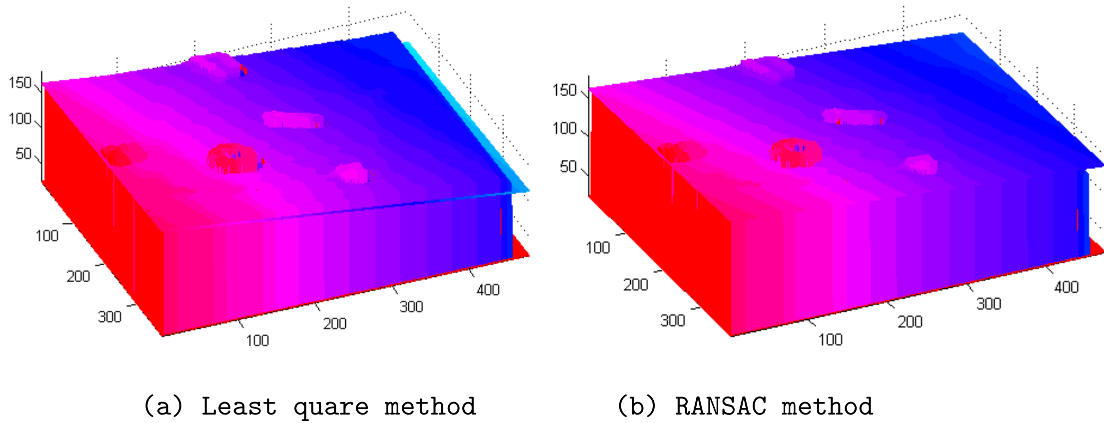
$$a_2X + a_1Y - Z + (b_1 - a_2m) = 0 \tag{2.8}$$

Based on Equation 2.8, as m is a constant, so if we know a_1, a_2 and b_1 of the two lines, then we could calculate the plane.

Now there are two methods to do the line fitting, one is least square method ([26]) and another is RANSAC (RANdom SAmple Consensus) method ([27]).

For the least square line fitting method, it will find the best fitted line based on all of the points. Based on Figure 2.13(b), the points do not only contain the ones on the plane, but also take the points of the object and the noise points into account. With the affection of the points out of the plane, the fitted line will not be the real one on the plane. We could see the final plane we calculated from the fitted lines with least square method in Figure 2.14(a), which is not fully fitted to the real plane. There are obvious errors between the calculated plane and the real plane. Afterwards when we want to extract the objects based on the calculated plane, it is hard to do it and the result is shown in Figure 2.15(a). As the calculated plane is not fitted to the real plane well, the binary image of removing the calculated plane contains both the information of the objects and that of the rest of the real plane.

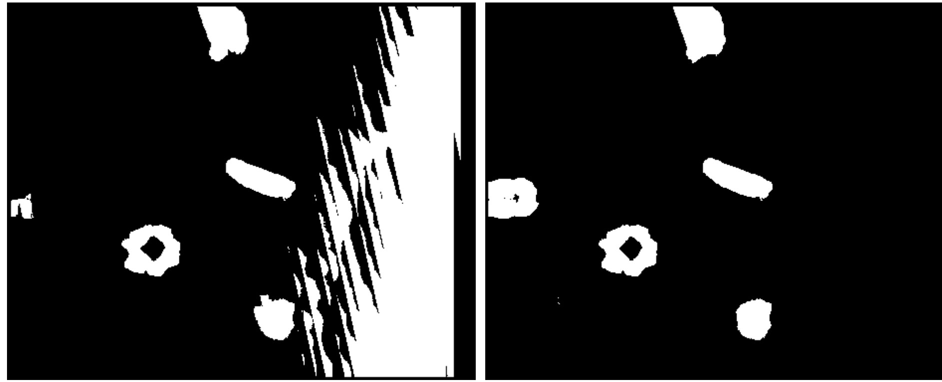
In addition, for RANSAC line fitting method, it can take the points on the real plane as many as possible into account to find the optimally fitted line by ignoring the points out of the real plane as many as possible. With this property of RANSAC line fitting method, the fitted lines will be almost the same as the real lines on the plane, so that the calculated plane from them is almost fitted to the real plane and the result is shown in Figure 2.14(b). With the calculated plane, the objects could be extracted perfectly from the real plane and the extracted image of the objects is illustrated in Figure 2.15(b) in a binary form.



(a) Least quare method (b) RANSAC method

Figure 2.14: Calculated plane with different line fitting methods

Thus, based on the results of least square method and RANSAC method, we determine to use the existing RANSAC line fitting algorithm in the first step of previous image segmentation algorithm to extract the objects from the plane and the rest steps of segmentation algorithm are the same as before.



(a) Least aqure method (b) RANSAC method

Figure 2.15: Stereo camera position with an arbitrary angle to the plane

Right now, when the camera has an angle with the plane, the image segmentation could still get the information of the objects. Another advantage of current image segmentation algorithm is that, because the plane can be calculated automatically, there is no need of manual efforts to find the plane in the procedure while the camera is parallel to the plane.

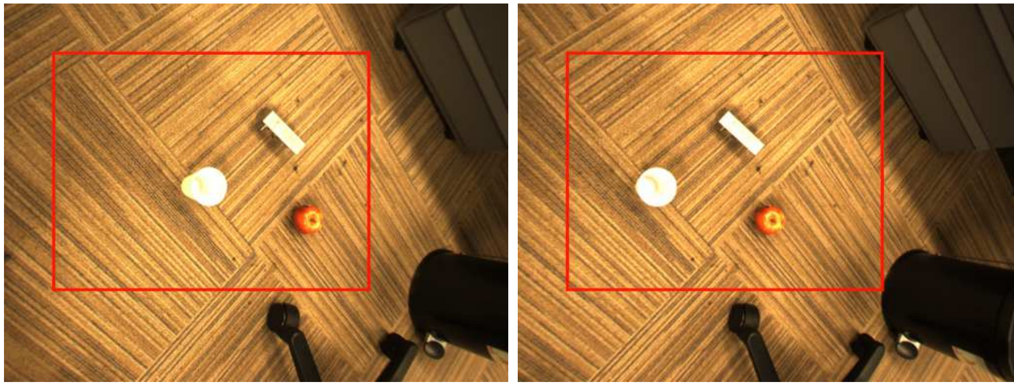
Totally speaking, the current 3D sensing part is more robust than before and it almost satisfies the implementation requirements of the functions.

2.4 Practical application of the developed 3D sensing part

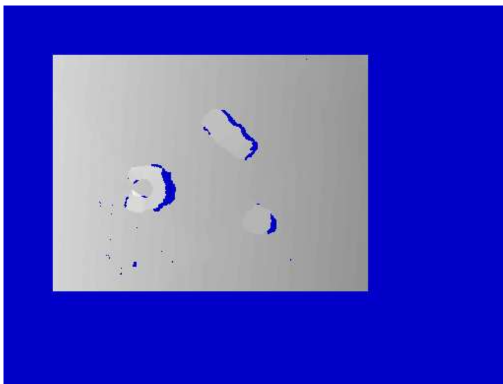
When the objects on a huge plane, 3D sensing part can satisfy the requirements from robot arm. However, in a more practical application environment, there are something else besides the objects and the plane, such as the case illustrated in Figure 2.1(a), only a region in the image is pure plane and other part of the image is unrelated stuff. Thus, in this case, it could be imagined that 3D sensing part can not work well in this environment. In addition, in some cases, large image size could be helpful on the accuracy of the result of 3D sensing part, but from Table 3.1, the image size with the width of 800 or 1024 is not available with the current stereo camera because of its low speed. Thus, the speed somewhat constraints the further application of 3D sensing part in some cases.

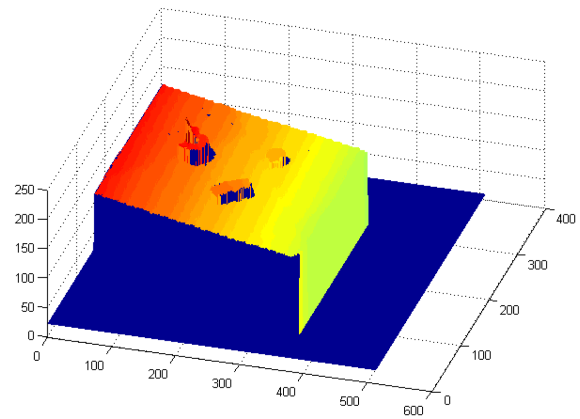
Based on the two practical application requirements above, 3D sensing part needs a certain improvement so that it could be more widely used. The solution or the trick to solve the problems above is to set a region of interest (ROI) in the images and just do all the processing in ROI, which is shown in Figure 2.16.

From Figure 2.16, within ROI, the case of the objects on a plane is separated from the real environment and it returns to the state in Section 2.3 or Section 3.1 so that 3D sensing part could be used directly on the region of interest (ROI). In addition, 3D sensing part only works on a part of the original images with ROI, so the computation is accordingly reduced and its speed is also faster than before for all cases of different



(a) left image (b) right (reference) image





(c) disparity map (d) 3D view of ROI

Figure 2.16: Disparity map generation with ROI

image sizes. We could use the data bandwidth requirement of each stage of 3D sensing part, as shown in Figure 2.17, to explain the reason.

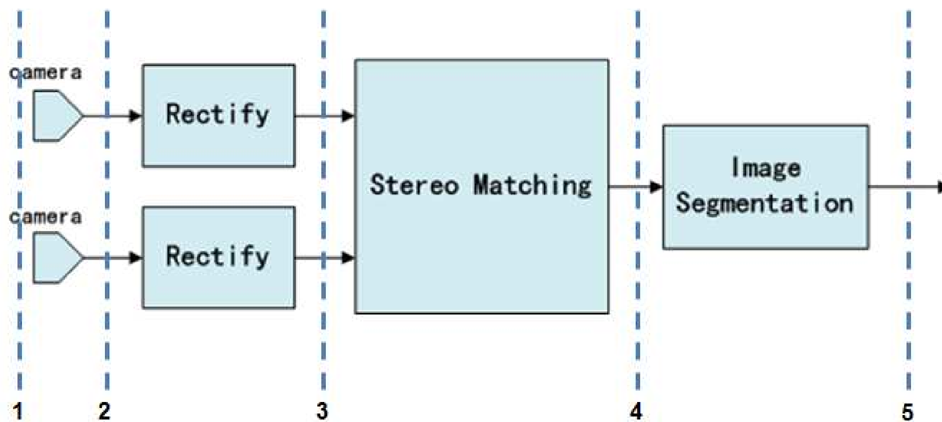


Figure 2.17: Data bandwidth requirement of each stage

From Figure 2.17, different stages have different data bandwidth requirements and they are:

1. 2 × original image resolution × 3 bytes (pixel size for color image)
2. 2 × sampled image resolution × 1 byte (pixel size for grey scaled image)
3. 2 × (ROI of) sampled image resolution × 1 byte
4. 1 × (ROI of) sampled image resolution × 1 byte
5. 1 × (ROI of) sampled image resolution × 1 byte

Original image resolution is $1280 times 960$ which is a fixed number determined by the CCD sensor size of the stereo camera, and sampled image resolution is changeable, all its cases are shown in Table 3.1. ROI usage could starts from the third stage and it accelerates 3D sensing part by reducing the data bandwidth requirement of some stages. An intermediate result is shown in Figure 2.16(c), 3D sensing part only calculates the disparity map of ROI, thus the time spent on stereo matching is less than that of computing whole disparity map.

Totally speaking, ROI is helpful to current 3D sensing part, it expands its application range without changing the hardware (camera) and without improving the function modules in 3D sensing part.

2.5 Conclusion

Above all, first we built up the 3D sensing part only for the stereo camera parallel to the plane. In this case, one has to manually set the threshold to find the plane and then the objects can be detected.

Later, we applied an existing RANSAC line fitting method to make the original 3D sensing part more robust. With this line fitting method, 3D sensing part could calculate the plane automatically. As a result, even the camera has an angle with the plane, the objects on the plane can still be detected

Lastly, in order to make the built 3D sensing part more stable in some practical application environment, we used ROI (region of interest) to retrieve the situation which 3D sensing part could deal with and at the same time reduce its computation range so that 3D sensing part there-for runs faster with the same input image resolution.

3D Sensing Profiling & Stereo Matching Implementation on GPU

3

This chapter concentrates on the implementation and the evaluation of the stereo matching stage in 3D sensing part on GPU, and the main contributions of this chapter are:

1. Evaluate 3D sensing part performance from its speed aspect.
2. Implement the original CPU based stereo matching algorithm from scratch.
3. Port the CPU based stereo matching algorithm on GPU.

From the work of this chapter, the following research questions in the first chapter are answered or partly answered:

- Whether the built stereo vision system could satisfy the speed requirement from the existing robot arm?
- After the stereo vision system is built, which part of it needs to be improved?
- How to improve the built stereo vision system?

3.1 Evaluate 3D sensing performance from the speed aspect

Until now, the 3D sensing part has been built up and the open question is whether its speed could satisfy the requirement from robot arm. For the robot arm, the 3D sensing part have to run at more than 6 Hz which means it should process the images at least 6 frames per second. In addition, if the speed of 3D sensing part is not fast enough, we have to know where the bottleneck is. In order to get the information of 3D sensing part performance (speed and time), we measure it in the following way:

Firstly, we put 3D sensing part into a demo program (Triclops Demo) provided by Point Grey SDK(e.g., [17] and [16]), in that demo program (shown in Figure 3.1), there are a user interface which could control function modules in 3D sensing part and a timer which could measure the speed of the program in the form of FPS. With this demo program, we can make a profiling of 3D sensing part.

Secondly, with the demo program, we will measure the speed of 3D sensing part in different image resolutions, while for stereo matching stage, the mask size is fixed at 11 and the disparity range is less than or equal to 50.

Thirdly, for each image resolution, we will use the user interface to turn on or off the function module of 3D sensing part, such as rectification, stereo processing and image segmentation shown in Figure 2.2, measure the variation of the speed of 3D sensing part, and finally calculate the time cost of each stage in 3D sensing part. The

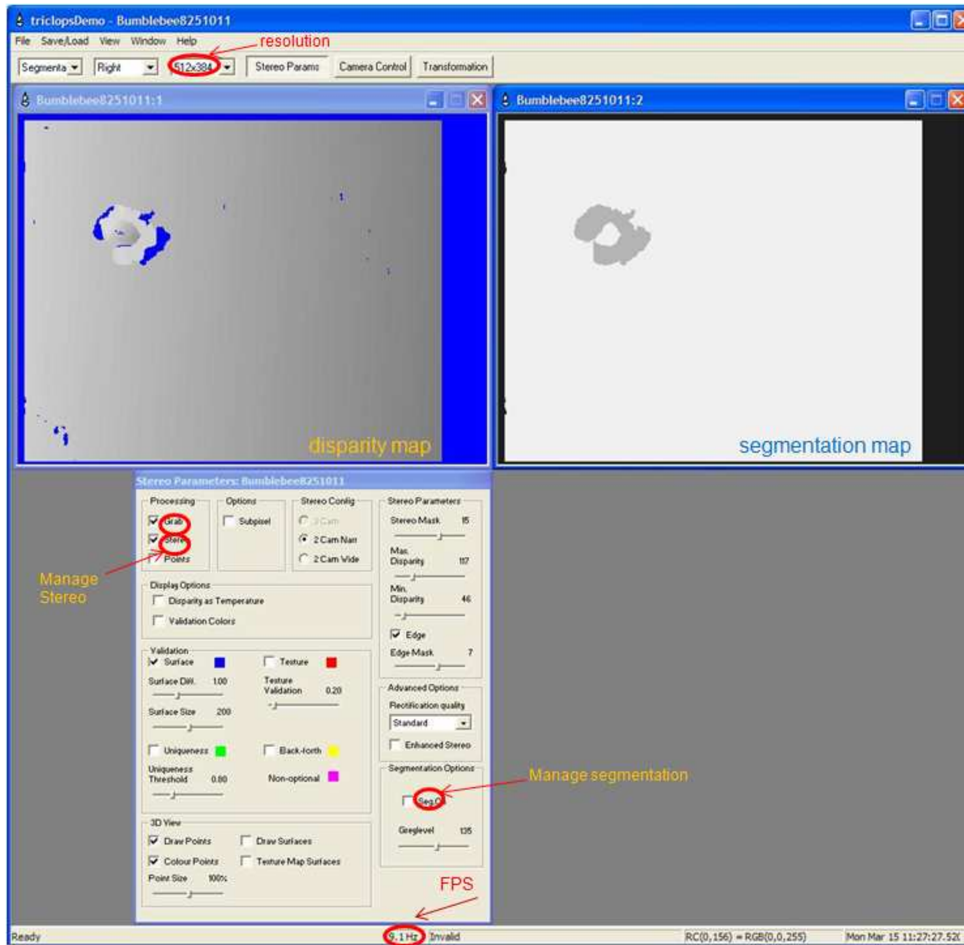


Figure 3.1: User interface of the demo program (provided by Point Grey SDK)

Table 3.1: Results of profiling 3D sensing part

Resolution	Grabbing (ms)	Rectification (ms)	Stereo match (ms)	Segmentation (ms)	Total time (ms)	Speed (Hz)
160x120	62.5	0.0	0.0	24.5	87.0	11.5
256x192	62.5	0.0	0.8	31.0	94.3	10.6
320x240	62.5	0.0	7.4	31.1	101.0	9.9
400x300	62.5	0.0	10.5	34.5	107.5	9.3
512x384	62.5	3.3	28.6	37.2	131.6	7.6
640x480	62.5	7.9	50.1	43.5	163.9	6.1
800x600	62.5	22.2	97.1	45.5	227.3	4.4
1024x768	62.5	34.6	159.3	56.1	312.5	3.2

profiling results of 3D sensing part in different resolution are recorded in Table 3.1 and illustrated in Figure 3.2.

From Table 3.1, we see that, except the cases of $800times600$ and $1024times768$ images, others could make 3D sensing part run above 6 fps (Hz). Also, through the

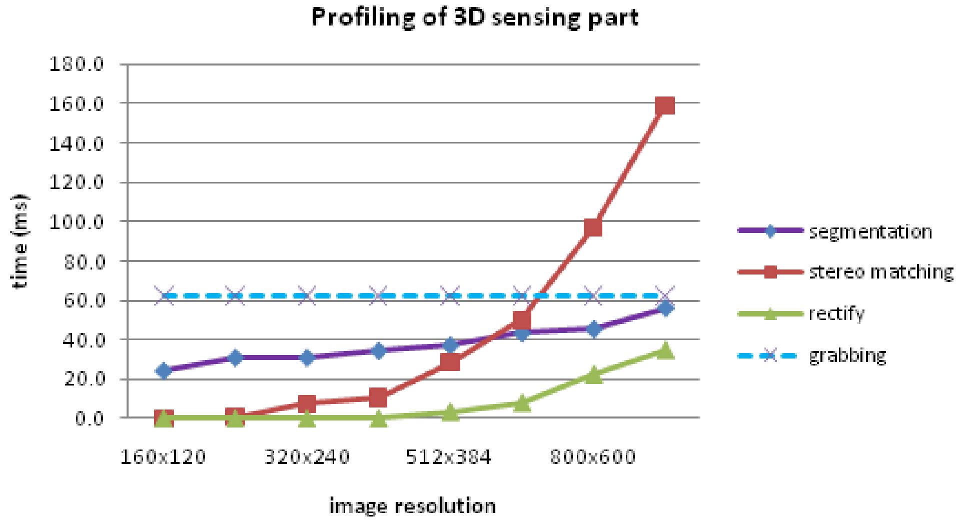


Figure 3.2: Profiling of 3D sensing part

experiment we found that, for 160$times$120 and 256$times$192 cases, as the images are so small that their results are not stable and not accurate enough, thus in real application only the image size with the width from 320 to 640 is available with current 3D sensing part.

In addition, from Figure 3.2, we could get an insight of the time contribution of each component in 3D sensing part. The grabbing time is almost the same for every image resolution. In fact, the stereo camera has a color CCD sensor with the fixed resolution 1280$times$960 and the camera is connected to PC with Fireboard-800, which is an IEEE-1394b to PCI OHCI compliant adapter capable of 800Mbit/sec performance, these two factors make the stereo camera grab the image at the speed of 16 Hz (frames per second). Accordingly, when the software section of 3D sensing part is combined with the hardware (stereo camera), the whole system can not run faster than 16 Hz. Thus, the stereo camera itself is one of the bottlenecks of 3D sensing part and the immediate improvement for it is to change a faster camera.

Another bottleneck of 3D sensing part is undoubtedly stereo matching component. When the image size increases, the time spent on stereo matching will be the most and increases faster than other components. For segmentation component, the variation of image size has a little effect on it. For rectification component, it consumes the least time in 3D sensing part.

Above all, if we can accelerate current stereo matching component, then the whole 3D sensing part with current stereo camera will get a significant performance enhancement. Based on the current stereo matching algorithm used in the camera, which is just known a local method, as its processing on each pixel of the image is independent, so it could have a large amount of data parallelism. In addition, current NVIDIA GPU is a typical SIMD platform which is suitable for massive data parallel programming with CUDA. Therefore, our further improvement on 3D sensing part in next section will consider implementing the stereo matching algorithm on GPU to see whether GPU is a suitable platform for future development of 3D sensing part.

3.2 Background

From results of Section 3.1, it is known that, besides the constraint of the speed from the hardware itself (bandwidth between stereo camera and PC), another critical bottleneck, shown in Table 3.1 and Figure 3.2, is from stereo matching component in 3D sensing part which belongs to the software aspect. Thus, in order to get further performance enhancement for 3D sensing part, the immediate way is to reduce the time spent on stereo matching stage. As the stereo matching algorithm provided by Point Grey SDK is a local method which calculates the disparity map pixel by pixel and it means the computation on each pixel is independent, so this algorithm has high potential data parallelism. If we could realize the parallel execution of the local method, then the time spent on stereo matching component will be reduced.

Currently, there are two kinds of platform supporting parallel computing, one is GPU and another one is FPGA. We will choose Nvidia GPU as the target platform for running local stereo matching algorithm and there are several reasons in the following:

Firstly, GPU is intrinsically SIMD architecture, so the stereo matching algorithm with massive data parallelism is suitable to it.

Secondly, with the help of CUDA (computing unified device architecture) from Nvidia GPU, it is very flexible to implement the algorithm on GPU without considering how to design the hardware architecture like FPGA development. For example, if someone wants to change the implementation, with GPU, he or she does not need redesigning the hardware architecture and this could save the developing time.

Thirdly, as GPU is one part of PC, so we could use it directly without appending a new hardware. Furthermore, usually GPU has lower price than FPGA with the similar computing capability.

Fourthly, although FPGA is suitable for embedded application with low power, as the 3D sensing part for the robot arm faces to desktop application at the beginning, so there are no needs to consider the power consumption for it in current time.

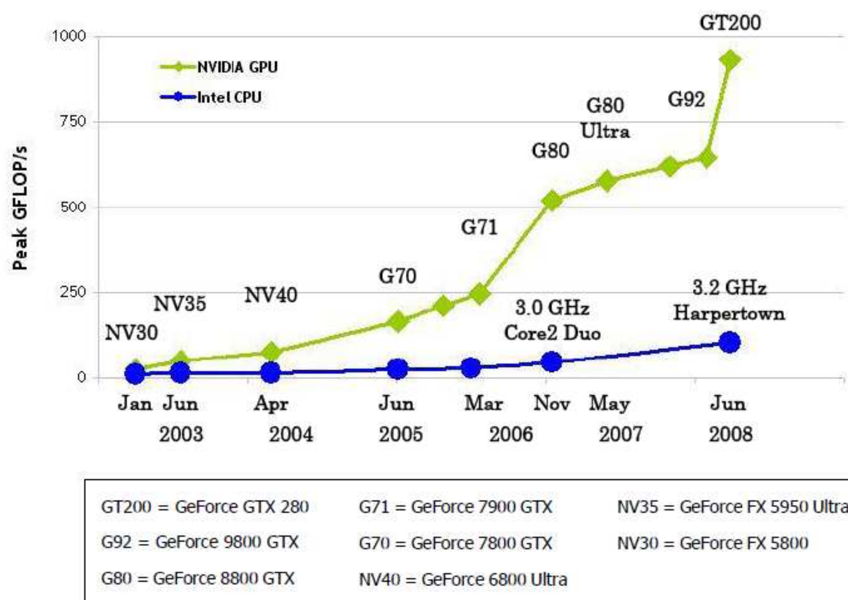
After determining GPU as the platform for running stereo matching component in 3D sensing part, the open question is how to do it. Unfortunately, the stereo matching algorithm applied by Point Grey stereo camera is not open source, so we cannot port it on GPU directly. However, it is known that the stereo matching algorithm is a local method based on SAD correlation in a fixed size block. With this limited information, the GPU work will be:

1. Implement the CPU based stereo matching algorithm from scratch with the same fundamental features as the algorithm used by Point Grey camera.
2. Port this CPU based algorithm on GPU to make it run as fast as possible.
3. Consider whether this GPU based stereo matching algorithm could replace the original one in stereo camera with higher speed.
4. Consider whether GPU is suitable for the further development of 3D sensing part on robot arm

3.3 GPU introduction

Before starting the GPU work, we will first get some intuitive insights about GPU capability (reference to [13]).

3.3.1 CPU vs. GPU



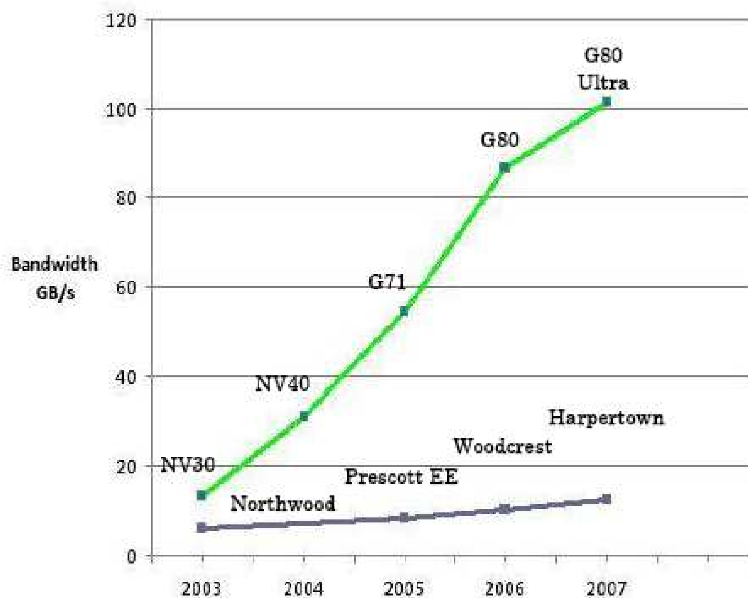


Figure 3.3: Floating Point Operations per Second and Memory Bandwidth for CPU and GPU

Figure 3.3 illustrates the technology progress of CPU and GPU in recent years, it is seen that the discrepancy in floating point capability and memory bandwidth between the CPU and the GPU is larger and larger. For floating point capability,

comparing to the sequential execution in CPU, GPU is specialized in compute-intensive, highly parallel computation and therefore designed so that more areas are devoted in data processing rather than data caching and flow control like CPU. The devotion of CPU and GPU is shown in Figure 3.4. More specifically, the GPU is especially good at dealing with data-parallel computations, as it is SIMD architecture which is that the same program (thread) is executed on many data elements in parallel. Thus applications that process large data sets independently in a same way can use a data-parallel programming model to accelerate the computation.

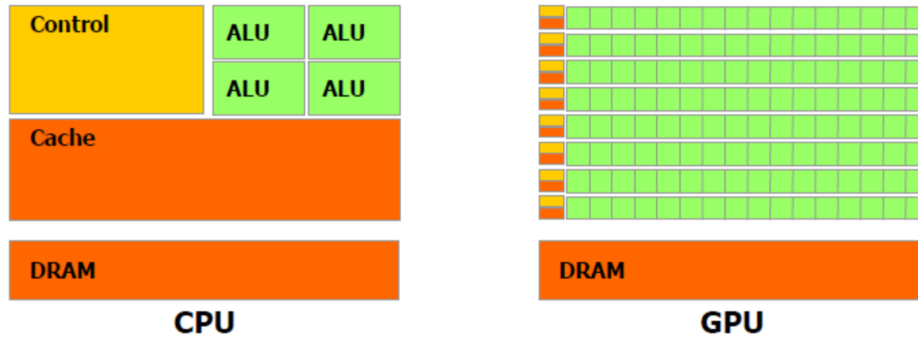


Figure 3.4: CPU and GPU devotion

For memory bandwidth, as the GPU provides the usage of various memory with parallel accessing pattern, so it makes the memory bandwidth of GPU larger and larger than that of CPU recently. For example, the shared memory on GPU is split into 16 banks (for compute capability 1.x) which can be accessed in parallel. Also in coalesced accessing mode, global memory can feed several threads together not one by one. In addition, as the memory access latency can be hidden with calculation, so the memory could have high toleration for much accessing.

3.3.2 CUDA programming model in GPU

CUDA is a general purpose parallel computing architecture, and the developer can achieve massive data parallel programming with it. Actually, CUDA program expresses data level parallelism (DLP) in terms of thread level parallelism (TLP) and then hardware converts TLP into DLP at run time. The whole procedure of CUDA program is like this:

Firstly, it creates as many as possible parallel threads to represent the parallel data elements.

Secondly, as all the threads will execute in a same way, so it just needs to write the program for one thread and then all of threads can make use of this program based on the threads indices.

Lastly, during run time, all these threads will be automatically distributed to different processor cores in GPU to run in parallel. It means, with more processor cores in GPU, the threads can get more parallelism and the program can be executed in less time. In summary, CUDA program can transparently scales its parallelism to leverage the increasing number of processor cores.

After knowing the principle traits of CUDA GPU, the rest of work is to implement a local stereo matching algorithm with high potential data parallelism on CUDA GPU.

3.4 The stereo matching algorithm implementation on GPU

3.4.1 Introduction

a) Problem requirement:

Design and implement an efficient parallel stereo matching algorithm with block matching method on CUDA GPU to generate the disparity (or depth) map in real time or near real time.

b) Algorithm description:

The inputs are two M*times*N rectified grey scaled images, which are generated from stereo camera. The output is a disparity map which is calculated based on x-coordinate difference between the correspondences in the two images. Here we assumed the left image as the reference and the right image as the target one for finding the correspondences.

First, calculate the fixed window SAD (sum of absolute difference) of one pixel in the valid range of disparity. Second, find the minimum SAD under specific disparity and then set the pixel value in output image with the disparity which has the minimum SAD. Finally, do the same task to all pixels in reference image.

c) Algorithm proof:

Because for a pixel or an object in the reference image, if we want to find its correspondence in the target image, then it is true that the pixel or the object should have the least or even no difference with its correspondence.

So we have:

if $SAD_{min} = SAD(k)$, where $disparity_{min} \leq k \leq disparity_{max}$;

then $Disparity(i_{pixel}) := k$.

3.4.2 WT-algorithm

We will use the WT-paradigm as assistance to simply analyse the time complexity of the stereo matching algorithm.

For simplifying complexity analysis (but not affect the conclusion), we assume the image (Left or Right image) resolution is $n \times n$, the mask (block or fixed window) size is $m \times m$ and the disparity range is l.

Firstly, the CPU based solution for the stereo matching algorithm is implemented in the following way; it is a very basic one:

a) - CPU based version

b) - GPU based version

Algorithm 1 Sequential stereo matching

Require: $L(n \times n), R(n \times n), m = 2k+1, k \geq 0$
Ensure: $D = stereomatch(L, R)$
  for $1 \leq row, col \leq n$ **do**
    for $d = 1$ to l **do**
      for $1 \leq x, y \leq m$ **do**
        $SAD+ = abs(L[row+y][col+x] - R[row+y][col+x-d]);$
      end for
      if $SAD_{min} > SAD$ **then**
        $SAD_{min} := SAD;$
        $Disparity_{_true} := d;$
      end if
    end for
    $D[row][col] := Disparity_{_true};$
  end for

After the CPU based stereo matching algorithm is implemented, we will port it on GPU. For parallelizing the CPU based solution, we have several choices:

1. Parallelize the first for loop only.
2. Parallelize the second for loop only.
3. Parallelize the third for loop only.
4. Parallelize both the first and the second for loop together.

At first glance, the fourth option seems the best, as it can parallelize the CPU solution mostly. In order to achieve it, we have to build up a 3D thread block to hold the necessary information for computing. It can be like this, two dimensions represent the width and the height of the image and the 3rd dimension store the image information in different disparity value. Unfortunately, the CUDA only provides the maximum of 64 in z-dimension, but in usual cases, the disparity exceeds 64 sometimes. In addition, if we build up 3D computing architecture for parallel processing, the communication overhead among different elements will be too much to improve the performance. Thus, the fourth option should be denied.

Secondly, for the second or third option, we can easily reduce their time complexity from $O(l)$ or $O(m^2)$ to $O(log(l))$ or $O(log(m))$ with balanced tree technology through the parallel architecture on GPU. If we do like this, we can parallel deal with the computation for each pixel, but all pixels should be processed sequentially. Generally, n is much greater than l or m, so parallelizing the computation on each pixel is not a wise strategy.

Based on the above discussion, the rest we can do is the first option which is that all pixels run concurrently, but within each pixel the procedure is sequential. Now we could have the following algorithm which runs on GPU:

Algorithm 2 Parallel stereo matching

Require: $L(n \times n), R(n \times n), m = 2k+1, k \geq 0$
Ensure: $D = cuda_stereomatch(L, R)$
  for all $i, j = 1$ to n **parallel do**
    for $d = 1$ to l **do**
      for $1 \leq row, col \leq m$ **do**
        $SAD+ = abs(L[row+y][col+x] - R[row+y][col+x-d]);$
      end for
      if $SAD_{min} > SAD$ **then**
        $SAD_{min} := SAD;$
        $Disparity_{true} := d;$
      end if
    end for
    $D[row][col] := Disparity_{true};$
  end for

3.4.3 Analysis of the time-complexity

Based on the above CPU- and GPU-based stereo matching algorithms, the analysis for their time and work complexities are discussed in the following:

a) CPU based version
$Time = n^2 \times l \times m^2 + 2 \times l \times n^2 + n^2 = O(n^2 \cdot l \cdot m^2);$
$Work = n^2 \times l \times m^2 + 2 \times l \times n^2 + n^2 = O(n^2 \cdot l \cdot m^2);$

b) GPU based version
$Time = l \times m^2 + 2 \times l + 1 = O(l \cdot m^2);$
$Work = n^2 \times l \times m^2 + 2 \times l \times n^2 + n^2 = O(n^2 \cdot l \cdot m^2);$

Because the CPU based algorithm is sequential and we see the GPU based parallel algorithm has the same work complexity with CPU based sequential algorithm, GPU based solution is work efficient.

3.4.4 Replace "parallel do"-statement above with thread index in CUDA

In order to make the algorithm run in parallel, an appropriate approach for processing the input images with thread block is needed.

For the special case in the algorithm which is to use one thread to compute one pixel value of the disparity map, as the maximum threads in one thread block are 512 and 1024 for compute capability 1.x and 2.0 device respectively, which is even far less than a small image with 320 × 240 pixels, so a hierarchy of multiple 2D thread blocks is needed to cover all pixels of the image so that one pixel can be controlled by one thread.

In addition, for each thread block, we will use a 16 × 16 square thread block as the unit of thread groups (reference to [7]). For 16 × 16, we have 256 threads per block. Since each streaming multiprocessor in GPU can take up to 768 threads, it can take up to 3 blocks to achieve full capacity.

With the above thread block configuration, all the threads can run parallel to generate all pixel values of the output image together.

3.4.5 Implementing the design on GPU

a) Implementation strategy

The stereo match algorithm does that, for each pixel in the reference image, it will search for its correspondence in the target image within a certain disparity range. It means that the information of reference image is static and we should shift the target image in a range to make all pixels in reference image find their correspondence. Simply speaking, we should update the information of target image by shifting it.

In addition, GPU provides various memory spaces for the application show in Figure 3.5 (reference to [12]), such as the off-chip memories, which are local, global, constant and texture memory, and the on-chip memories, which are register and shared memory. Different memories have different features which are shown in Table 3.2. One could get performance improvement for the application by making a good use of different memories. Usually, the usage of register and local memory are done by nvcc compiler of GPU. Accordingly, for other memory spaces based on Table 3.2, global memory accessing is the slowest while shared memory accessing is the fastest, because shared memory is closer to multiprocessor. Therefore it is better to access global memory as few as possible.

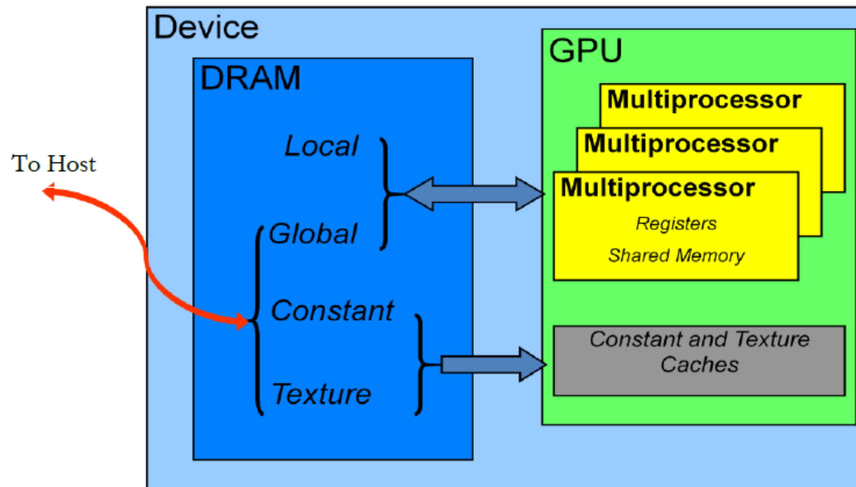


Figure 3.5: Various memory spaces on a CUDA device

Because the images are stored in global memory after being transferred from host memory, in order to avoid frequent read/write operations in device (global) memory, we should first transfer the images data to shared memory in each thread block, because the performance improves with the use of shared memory which can be accessed faster than the device memory (global memory).

Based on the above memory usage, the actual procedure on GPU is:

1. Download the two input images to GPU global memory.

Table 3.2: Salient features of device memory

Memory	Speed Rank	Location on/off chip	Cached	Access	Scope	Lifetime
Shared	1	On	No	R/W	1 block	Block
Register	2	On	No	R/W	1 thread	Thread
Constant	3	Off	Yes	R	all threads + host	Host allocation
Texture	4	Off	Yes	R	all threads + host	Host allocation
Local	5	Off	CC 2.0	R/W	all threads + host	Host allocation
Global	6	Off	CC 2.0	R/W	all threads + host	Host allocation

2. Tile the reference image to each thread block shared memory.
3. Tile the shifted target image to each thread block shared memory under certain disparity.
4. Each thread computes SAD, finds the minimum SAD and then stores the disparity which has the minimum SAD into register.
5. Go to 3 until the target image is shifted out of the valid disparity range. (step 3 step 5 runs parallel for all threads).
6. Write the final disparity of each pixel to global memory parallel.
7. Upload the output image to CPU.

b) Shared memory management

Until now, the only open question is that how to manage shared memory in each thread block. As we know, for every pixel, we have a fixed-size window to cover it including its surrounding ones for computing SAD. If the number of image data hold in the shared memory of each thread block equals the number of threads in each block, then when computing SAD at the boundary pixels, there will not be enough information for them. For short, at the boundary of each thread block, it will cause errors. The situation can be illustrated in the following Figure 3.6:

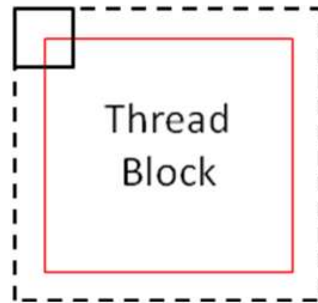


Figure 3.6: Relations between shared memory and thread block

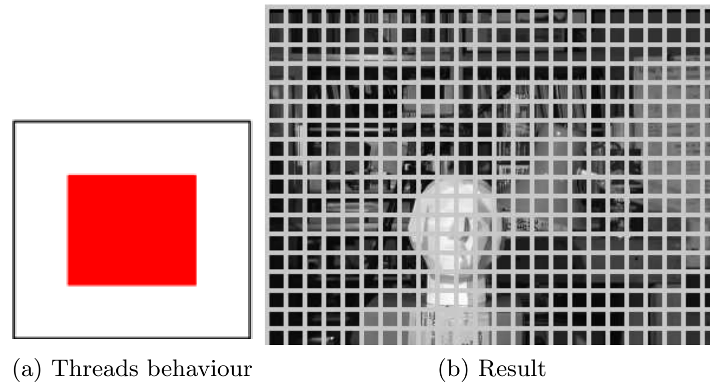
As we see, it is obvious that the number of elements in shared memory should be more than that of its thread block. Their relations are:

- Thread block size: $tile_width \times tile_width$

- Shared memory size: $(tile_width + mask - 1) \times (tile_width + mask - 1)$

After that, we will evaluate the shared memory through several steps:

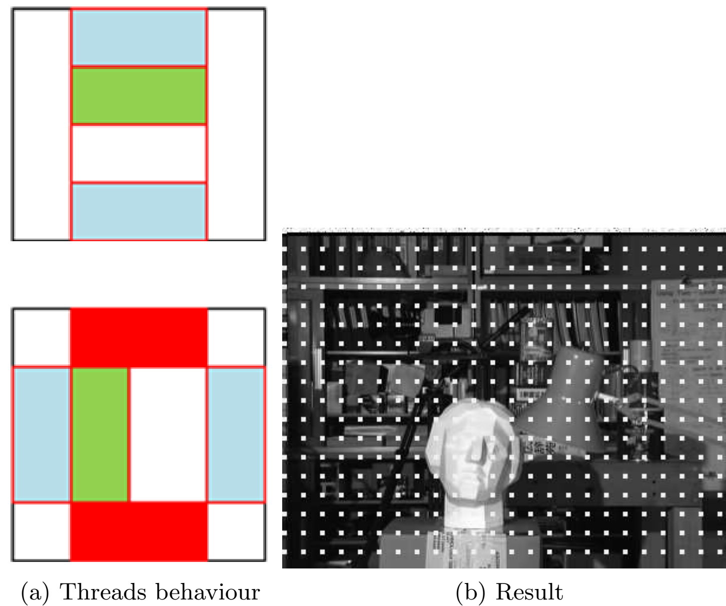
Step 1: Evaluate the central part of shared memory with all threads in all thread blocks. The procedure and the intermediate result are illustrated in Figure 3.7.



(a) Threads behaviour (b) Result

Figure 3.7: The first step of shared memory preparation

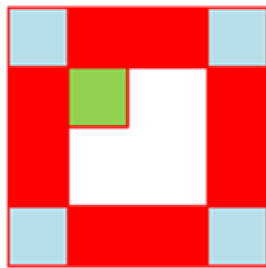
Step 2: Evaluate the upper (left) and the lower (right) part of shared memory with upper (left) threads in all thread blocks. The procedure and the intermediate result are illustrated in Figure 3.8.



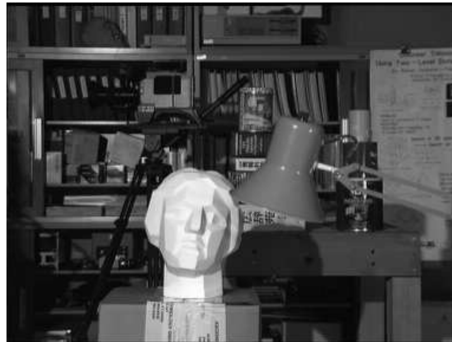
(a) Threads behaviour (b) Result

Figure 3.8: The second step of shared memory preparation

Step 3: Evaluate the rest of four corners of shared memory with the upper left threads in all thread blocks. The procedure and the intermediate result are illustrated in Figure 3.9.



(a) Threads behaviour



(b) Result

Figure 3.9: The third step of shared memory preparation

4 Experimental Results & Implementation Evaluation

After the stereo matching algorithm is implemented on GPU, we will make a evaluation about the quality of the GPU based solution through several aspects, for example, firstly, we will make a comparison of performance between CPU and GPU; secondly, we will make a comparison of performance among different GPUs; after that, we will profile the GPU based solution to find whether there exists some possible further optimization places in GPU.

Thus, this chapter concentrates on the evaluation of the stereo matching stage in 3D sensing on GPU, and the main contributions of this chapter are:

1. Comparing GPU based solution of the stereo matching algorithm to the CPU based solution and to other GPUs with different compute capability.
2. Profile the GPU based implementation of stereo matching algorithm and find the possible optimization places based on the GPU hardware architecture.

From the work of this chapter, the following research questions in the first chapter are answered or partly answered:

- Is GPU suitable for the development of stereo vision application?
- Is it possible to make the built stereo vision system run in real time?

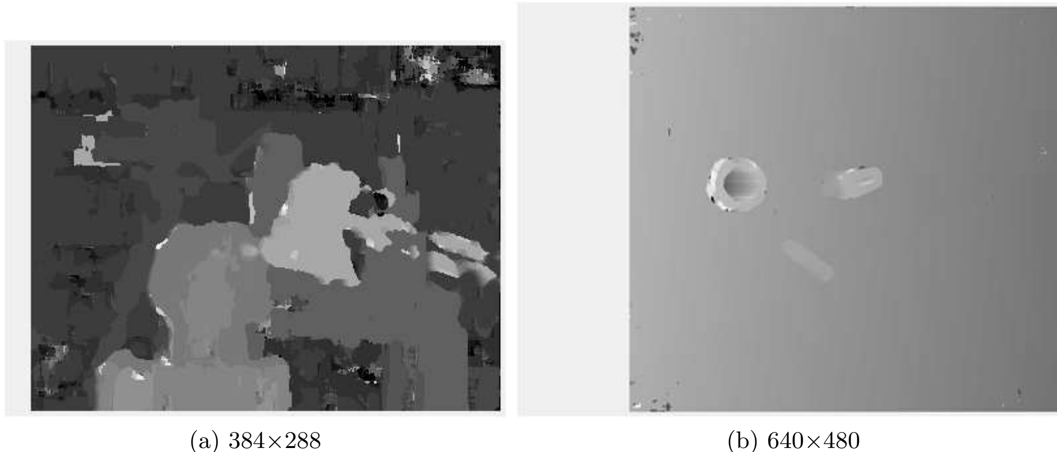
4.1 Comparison analysis - CPU vs. GPU

4.1.1 Accuracy comparison

In fact, the CPU output and the GPU output have the same accuracy, as they use the same algorithm and the same data type which is just integer data. In addition, we compared the disparity maps generated by CPU and GPU respectively pixel by pixel and found that they had the same pixel values. The disparity maps of the standard test image pair "tsukuba" and the experiment image pair from both CPU and GPU are shown in Figure 4.1.

4.1.2 Speed comparison

Here for simplification, we will just use Intel Xeon 5110 1.6 GHz dual core CPU as the PC based reference, and then we are going to use different NVidia GPUs with different cores and computability to do speed comparison with the CPU. Note that, for the users, they are not allowed to just use some of the CUDA cores in one GPU, as it is not provided to do so and all of the CUDA cores in one graphic card always run together. Thus, we can just increase or decrease the number of CUDA cores we use by changing a new GPU. Additionally, from the CPU side, it is just single thread in single core.



(a) 384×288 (b) 640×480

Figure 4.1: Disparity maps generated by both CPU and GPU

Table 4.1: The GPUs in the experiment

GPU name	Compute Capability	# of Multiprocessors	# of CUDA cores
Tesla C1060	1.3	30	240
GeForce GTX 280	1.3	30	240
GeForce 8800 GTX	1.0	16	128
Quadro FX 3700	1.1	14	116
GeForce 8400M GS	1.1	2	16

Furthermore (reference to [12, 14]), in order to get an exact executing time on GPU, we have to synchronize the CPU thread with the GPU by calling *cudaThreadSynchronize()* immediately before starting and stopping the CPU timer. Because many CUDA API functions are asynchronous; that is, they return control back to the calling CPU thread prior to completing their work. *cudaThreadSynchronize()* blocks the calling CPU thread until all CUDA calls previously issued by the thread are completed.

4.1.3 Experiment results of both CPU and GPU

In our experiments, we will use four pairs of image with different resolutions as the input. They are 384×288, 512×384, 640×480 and 1282×1110. We use the problem size N to represent their width information and use P as the number of multiprocessors. Then we will see the relation between N and P for the algorithm and the scalability of GPU in Table(4.2, 4.3 and 4.4) and Figure(4.2, 4.3, 4.4 and 4.5).

4.1.4 Discussion of results

a) From Table 4.2: We can see that only the green data satisfy the real time requirement. What we can predict is that, in current situation without modifying the

Table 4.2: Test results for the program - Time(P, N) (seconds)

P \ N	**384**	**512**	**640**	**1282**
1	1.23378	3.26514	6.54967	92.12514
2 (16)	0.34954	0.97243	2.07805	28.80753
14 (112)	0.03042	0.08353	0.17912	2.48310
16 (128)	0.02114	0.05922	0.12526	1.72098
30 (240-G)	0.01052	0.02828	0.05923	0.81743
30 (240-T)	0.00968	0.02605	0.05450	0.75095

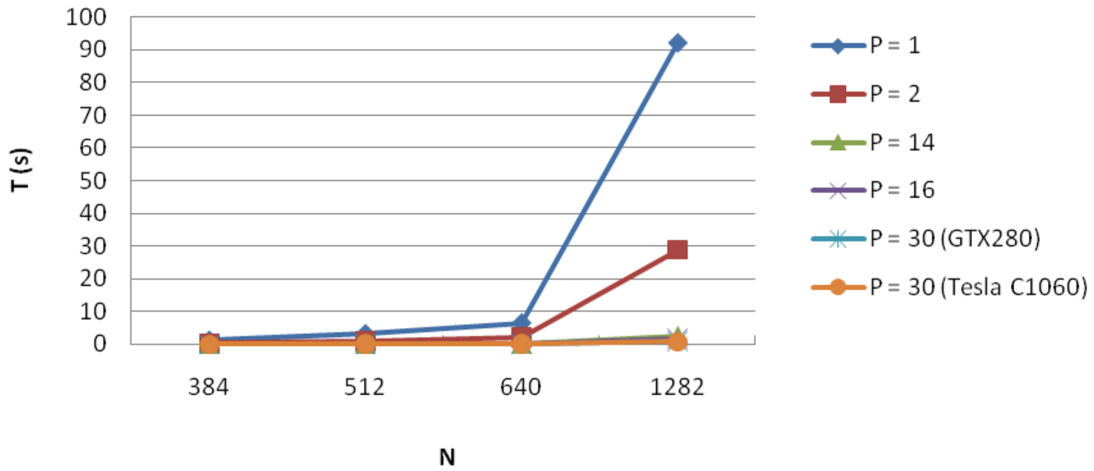


Figure 4.2: Illustration of Time(N, P fixed)

Table 4.3: Speedup (N, P fixed) = Time (384, P fixed) / T (N, P fixed)

N \ P	**1**	**2 (16)**	**14 (112)**	**16 (128)**	**30 (240-G)**	**30 (240-T)**
384	1	1	1	1	1	1
512	0.378	0.359	0.364	0.362	0.372	0.372
640	0.188	0.168	0.170	0.171	0.178	0.178
1282	0.013	0.012	0.012	0.012	0.013	0.013

code, more powerful GPU should be used for the larger problem size and this is the only way of achieving the real time requirement further. Additionally, the red

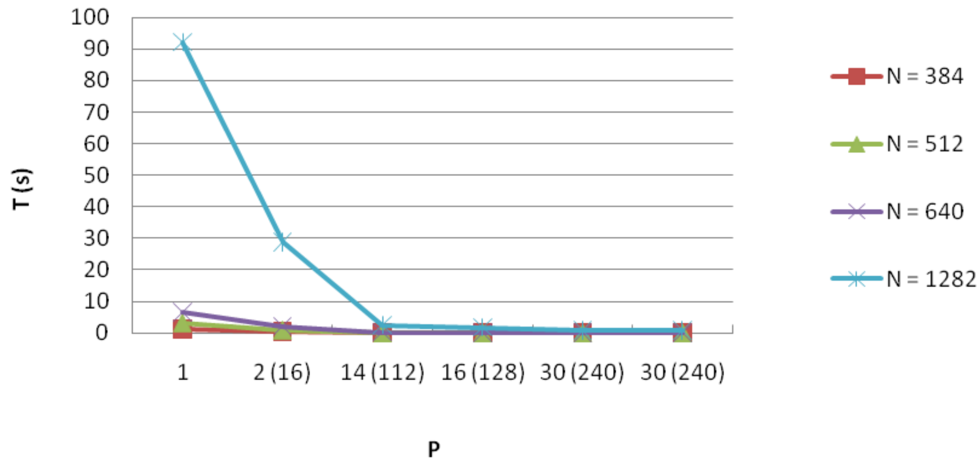


Figure 4.3: Illustration of Time(P, N fixed)

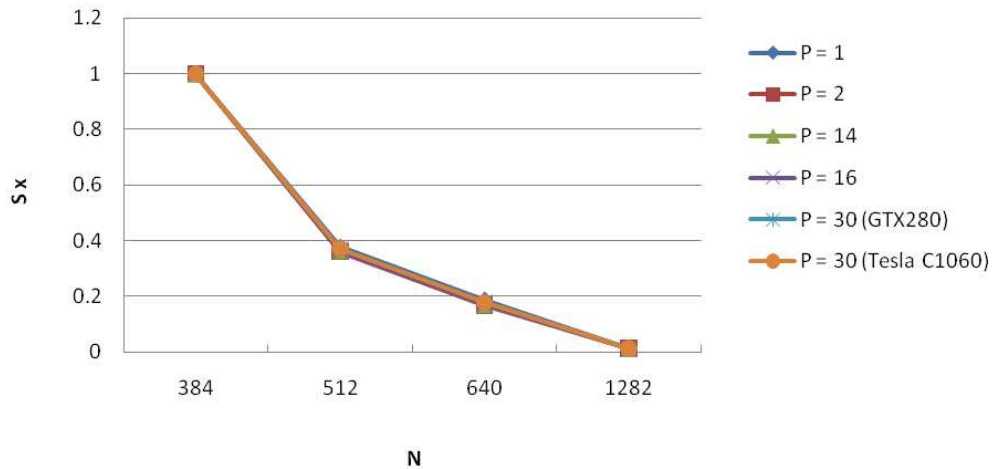


Figure 4.4: Illustration of Speedup(N, P fixed)

data is reasonably estimated based on the other data which are from experiment, because the stall of GPU happens when very large data is running on the weak laptop GPU and it is impossible to get the exact elapsed time of GPU.

b) T (N, P fixed): Totally, based on the graph, the result is what we expect. The time increases with the increment of problem size under fixed number of processors. Note that, with increasing the number of multiprocessors, the time decreases

Table 4.4: Speedup (P, N fixed) = T (1, N fixed) / T (P, N fixed)

P \ N	384	512	640	1282
1	1.00	1.00	1.00	1.00
2 (16)	3.53	3.36	3.15	3.20
14 (112)	40.56	39.09	36.56	37.10
16 (128)	57.63	55.14	52.29	53.53
30 (240-G)	117.28	115.46	110.58	112.70
30 (240-T)	127.46	125.34	120.18	122.68

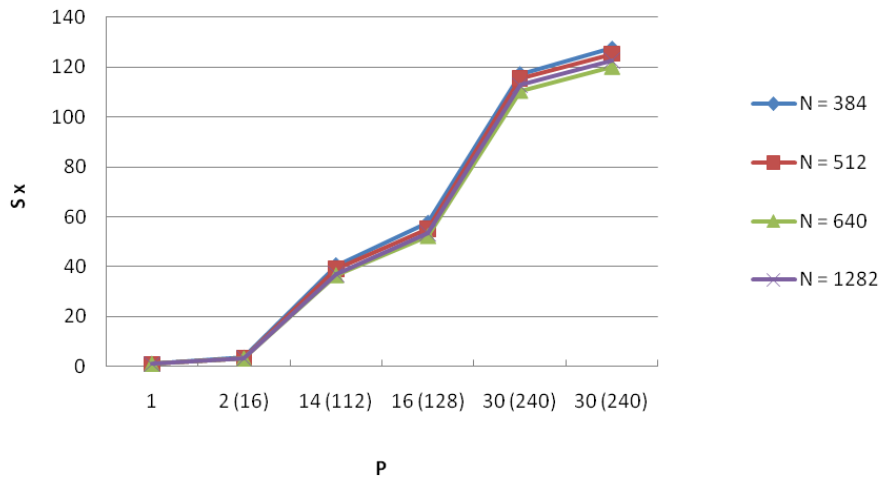


Figure 4.5: Illustration of Speedup(P, N fixed)

significantly. Although all the multiprocessors in GPU work parallel, there is no communication among them which is determined by the algorithm and the warp scheduling is zero-overhead in each of the multiprocessors in GPU. Additionally, only a few synchronization in the algorithm and a few block scheduling overhead, thus totally, without communication overhead, with a little synchronization among different threads and with some block scheduling overhead but not much, we can see a significant performance improvement with increasing the number of multiprocessors in GPU.

c) **T (P, N fixed):** As we see from the graph, for the fixed problem size, when we use more multiprocessors, naturally more time for computation will be decreased. Note that, if the problem size is larger, then the effectiveness for decreasing the

time is more obvious.

d) Speedup (N, P fixed): The result is normal. If we do not change the hardware, there is no speedup when we increase the problem size. In fact, the trends of performance decrement while increasing the problem size in different hardware are almost the same from the graph.

e) Speedup (P, N fixed): The truth from the graph is that, whatever the problem size is small or large, the speedup trends with increasing the number of the multiprocessors are almost the same, which is that more multiprocessors result in more speedup. Furthermore, we can see that the speedup value is less than the number of CUDA cores. However, it does not mean that, for one GPU, the upper bound speedup to CPU it can achieve is less than or equal to the number of CUDA cores it has.

Here we can extract two questions:

1. How much speedup potential to CPU can one GPU achieve?
2. Among different GPUs, what is the scalability for them?

4.1.5 Analysis of the speedup upper bound for one GPU comparing to one CPU

a) From algorithm level

Based on the time complexity analysis in Section 3.4.3 of both CPU and GPU, we can see that the GPU speedup can reach that:

$$AlgorithmSpeedup = \frac{O(n^2 \times l \times m^2)}{O(l \times m^2)} = O(n^2) \tag{4.1}$$

In Equation 4.1, it is assumed that every pixel of the image can be generated synchronous. For example, if the expected image resolution is 384×288, then the speedup achieved will be equal to the product of width and height of the image which is around 10^5 times ideally.

b) From hardware specification level

From the theory analysis above, although there is a large speedup potential, because of the hardware specification constraints, such as the number of multiprocessors in GPU, bandwidth between CPU and GPU, frequency difference between CPU and GPU and etc., all of these hardware features make achieving the theory speedup potential level impossible in actual applications.

In an empirical way, taking two main factors for consideration, which are the theoretical peak GFLOPS and the memory bandwidth of both CPU and GPU, we could

have an equation to calculate the GPU speedup potential to a CPU roughly. The empirical equation is:

$$PotentialSpeedup = \frac{GFLOPS_{GPU} \cdot MemoryBandwidth_{GPU}}{GFLOPS_{CPU} \cdot MemoryBandwidth_{CPU}} \times \tag{4.2}$$

In Equation 4.2, GFLOPS is a way of measuring the calculation power of both CPU and GPU, in addition Memory bandwidth determines the r/w operation speed. As most of the time, r/w operations and calculating operations exist together, so we can simply use the product of their speed to get the total speed of the whole operation. With this equation, we could easily skip mass of hardware details and reasonably calculate the GPU speedup potential to the CPU we used in experiments.

Table 4.5: Overview of GPU speedup potential to CPU

	CPU(Xeon5110)	G8400M	Q3700	G8800	G280	TC1060
Mem_bandwidth (GB/s)	5.3	9.6	51.2	86.4	141.6	102.4
Peak_performance (GFLOP/s)	6.4	38.4	420.0	518.4	933.1	933.1
Potential_c (x)	1	11	630	1313	3873	2801

Here the equations used for calculating CPU and GPU memory bandwidth and peak performance are:

$$Mem_bandwidth(GPU) = \frac{memFreq(MHz) \times memWidth(Byte) \times 2}{10^9} = GB/sec \tag{4.3}$$

In Equation 4.3, *2* is due to the double data rate, which means data is transferred on both the rising and falling edges of the clock signal within computer bus operations. Take Quadro FX 3700 as an example, memFreq is 800 MHz, memWidth is 256 bits (32 Bytes), and thus its memory bandwidth is:

$$Mem_bandwidth(Q3700) = \frac{800 \times 10^6 \times 256 \times 2}{10^9 \times 8} = 51.2GB/sec$$

For the CPU memory bandwidth in our experiment, it uses PC2-5300 which is Fully Buffered DDR2 SDRAM and its peak (theoretical) transfer rate runs at 5.33 GB/s.

In addition, the peak performance is calculated in the following equation both for CPU and GPU:

$$Peak = \#core \times coreFreq \times flop_issue_rate \tag{4.4}$$

In our case, the parameters of both CPU and GPU are in Table 4.6.

In our CPU based solution, the program is purely sequential and there is only one CPU core running for it, thus in Table 4 the CPU peak performance considered is just one sixteenth of that in Table 4.6.

Table 4.6: CPU & GPU hardware parameters

Hardware	CPU(Xeon)	G8400M	Q3700	G8800	G280	TC1060
#core	4	16	112	128	240	240
coreFreq(GHz)	1.600	0.800	1.250	1.350	1.296	1.296
flop_issue_rate	4	3	3	3	3	3
Peak (GFLOP/s)	25.6	38.4	420.0	518.4	933.1	933.1

c) From real implementation case

Although there is a high speed up potential in a GPU, it is not easy to fully exert its capability. Actually, in our real implementation case, only less than 10% capability is activated except the weakest GPU. Here we use Ec to represent the efficiency of practical implementation on GPU to that on CPU. It defines how much capability of GPU is activated for one implementation case comparing its CPU version implementation. It can be calculated by Equation 4.3.

$$Ec = \frac{RealSpeedup}{Potential_c} \times 100\% \tag{4.5}$$

For Equation 4.5, we can get the average *RealSpeedup* value from Table 4.4 and get the *Potential_c* from Table 4.5; then we could get the efficiency of our implementation case which is showed in Table 4.7:

Table 4.7: '*Ec*' measurements based on local stereo matching algorithm

	G8400M	Q3700	G8800	G280	TC1060
RealSpeedup (x)	3	38	55	114	124
Potential_c (x)	11	630	1313	3873	2801
Ec (%)	**30.6**	**6.1**	**4.2**	**2.9**	**4.4**

We can see from Table 4.7 that, the efficiency of our implementation case on GPUs to CPU is 3%∼6% which is very low. In fact, there are several factors constrain activating the full capability of GPU, some can be easily overcome but some cannot and they are:

1. Whether the algorithm is implemented with work efficient?

   Actually, the work overhead happens when the input data is prepared in different memory spaces, for instance, data transferring between host memory and device (global) memory, data transferring between global memory and shared memory on GPU and etc.

2. Whether the maximum number of threads, which can run simultaneously on each of GPU multiprocessors, is fully activated in real implementation to hide global memory latency?

There are three factors can determine the number of active threads on each GPU multiprocessor finally: number of threads per block, shared memory usage per block and number of registers in use per thread. It is beneficial to make a trade-off among them to get a high parallelism on GPU.

3. In the usage of shared memory, if there are bank conflicts, they will make parallel threads executed serially, thus reducing bank conflicts can improve the performance.

4. As global memory accessing has the longest latency on GPU, so reducing the times of global memory accessing can help save the total latency.

5. Thread blocks are scheduled among different multiprocessors by GPU itself. It can produce some overhead.

6. It is not certain that, warps in different multiprocessors run synchronously. Because a common GPU is responsible for monitor controlling and computation at the same time, thus the time for computation is mixed with some monitor task. It means that monitor control could distribute some performance of GPU.

7. In practical case, it is hard to achieve CPU and GPU theoretical memory bandwidth and peak performance, because theoretical value only happens when the right operations with right sequence running on the platform at the right time.

Thus, the factors above make the low efficiency of application running on GPU comparing with running on CPU. In spite of low efficiency between GPU and CPU, because GPU has massive parallelism, it can still make the application achieve a significant performance improvement comparing with CPU.

4.2 Evaluate the scalability among different GPUs

We can also clarify this problem from two aspects: one is from hardware specification and another is from real implementation case. The situation is simpler and closer to the truth when we discuss the performance of the implementation among different GPUs than discussing that between GPUs and CPU. Because the GPUs in our hands have almost the same hardware architecture and they are only different from the amount of resource and the compute capability with each other, for example they have different number of multiprocessors, different upper bounds of warps per multiprocessor and etc; so we can easily get the insight of scalability of different GPUs.

a) From hardware specification

In our hands, there are five NVIDIA graphic cards which are showed in Table 4.1. We will select the weakest one GeForce 8400M GS as the reference (GPU_ref) and compare it with others from memory bandwidth and peak performance. Here we will

use an equation, similar to Equation 4.2, to calculate the potential speedup among different GPUs:

$$PotentialSpeedup = \frac{GFLOPS_{GPU} \cdot MemoryBandwidth_{GPU}}{GFLOPS_{GPU_ref} \cdot MemoryBandwidth_{GPU_ref}} \times \quad (4.6)$$

The calculations of Memory bandwidth and GFLOPS are the same as Equation 4.3 and Equation 4.4 respectively. And also we can easily get these two values from Table 4.5, then we have Table 4.8.

Table 4.8: Overview of speedup potential among different GPUs

	G8400M	Q3700	G8800	G280	TC1060
Mem_bandwidth (GB/s)	9.6	51.2	86.4	141.6	102.4
Peak_performance (GFLOP/s)	38.4	420.0	518.4	933.1	933.1
Potential_g (x)	1	58	121	359	259

b) From real implementation case

From Table 4.8, GPUs peak performance and memory bandwidth are theoretically calculated, thus the potential speedup values of different GPUs in Table 4.8 are also theoretical ones. In practical cases, as declared previously, the theoretical values are just adapting to specific situations, such as specific datas, specific sequence of specific operations at specific time and so on. Empirically, after optimization, the real performance can achieve 70%∼80% of theory value ideally in some cases. While in another cases, because some specific features are inherent to the algorithm or application, real performance can just achieve 60% of theory value or even less.

In our experiments, in order to measure other GPUs speedup values comparing to the reference GPU; we can just calculate them with the following equation:

$$RealSpeedup(GPU2GPU) = \frac{RealSpeedup(GPU)}{RealSpeedup(GPU_ref)} \times \quad (4.7)$$

In Equation 4.7, *RealSpeedup(GPU)* and *RealSpeedup(GPU_ref)* are both from Table 4.4, and *GPU_ref* is GeForce 8400M GS which has only two multiprocessors. Then we have Table 4.9.

We could illustrate Table 4.9 with Figure 4.6:

In addition, we will use '*Eg*' to represent the efficiency of our implementation case running on different GPUs. *'Eg'* defines the ratio of real speedup to theory speedup of one GPU when comparing to the reference GPU. It can be computed with:

$$Eg = \frac{RealSpeedup(GPU)}{Potential_g} \times 100\% \quad (4.8)$$

Table 4.9: RealSpeedup(GPU2GPU)

GPU / N	G8400M	Q3700	G8800	G280	TC1060
P	**2**	**14**	**16**	**30**	**30**
Potential_g	1	58	121	359	259
384	1	11.	14	29	30
512	1	11	14	31	31
640	1	11	14	31	32
1282	1	11	14	31	32

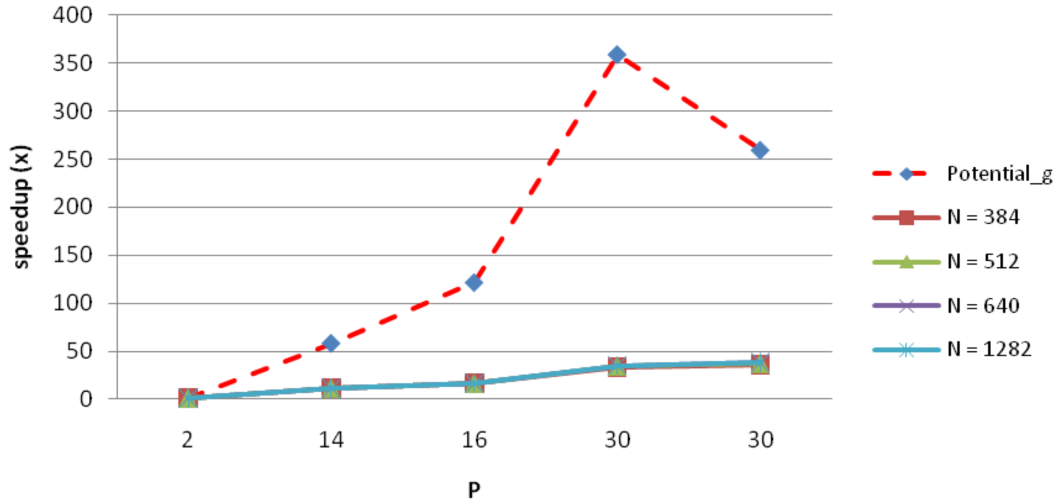


Figure 4.6: Scalability of different GPUs

Table 4.10: '*Eg*' measurements based on SAD local stereo match algorithm

	Q3700	**G8800**	**G280**	**TC1060**
RealSpeedup (x)	11.6	16.5	34.5	37.5
Potential_g (x)	58	121	359	259
Eg (%)	**19.8**	**13.6**	**9.6**	**14.5**

In Equation 4.8, *Potential_g* is from Table 4.8 and *RealSpeedup(GPU)* is the average speedup of different problem size cases of one GPU from Table 4.9.

When the real implementation is running on different GPUs without any modification, we can get some insight of the GPUs scalability. The positive aspect is that it can obtain more speedup with upgrading the hardware accordingly. For the negative aspect, from Figure 4.6, it illustrates that the gap between potential speedup and real speedup becomes larger and larger with increasing the GPU computing capability and from Table 4.10 we see that the GPU speedup efficiency stays at a low level which is less than 20%. The reasons behind it are almost the same as those of the end of Section 4.1.5. Another thing we should note is that, the potential speedup of GTX280 is better than that of Tesla C1060, but in our real test Tesla C1060 behaves better than GTX280. One reasonable explanation is that, because Tesla C1060 is not a normal graphic card with display but the one with a pure computing capability, in spite of somewhat weaker potential speedup than GTX280, Tesla C1060 can work more efficient on computing task than GTX280.

In summary of Section 4.1 and Section 4.2, we can find that our current local stereo matching algorithm implementation on GPU is not optimal one, but it can scale with the hardware which means its performance can be improved naturally by increasing the hardware computing capability. From the increasing gap between potential speedup and real speedup in Figure 4.6 and the low efficiency from Table 4.7 and Table 4.10, it shows that there is still large space for optimization to reduce the gap and increase the efficiency. Additionally, if the implementation scalability is improved, we can get more beneficial, such as more performance improvement, when we scale the hardware.

Thus, the rest of the work is to find the task bottle neck and then optimize it. We will design experiments in the following Section 4.3 to profile the task and find where the problems or optimization places are.

4.3 Profiling the GPU based implementation

Before one starts profiling the implementation, it is supposed to select the platform and the case of specific problem size first. Here in the experiment, Quadro FX3700 (compute capability 1.1) will be used as the platform, because first it possesses such enough capability that all the cases of different problem sizes can run on it; Secondly, it is the second weakest GPU in hands, if one finds the bottle neck of the task and do optimization for the GPU, then when the optimized code runs on more powerful GPUs, it will certainly get more speedup according to the scalability of GPUs discussed in Section 4.2.

For the case selection about specific problem size, the one 640×480 will be considered. First as we see from Table 4.2, it is not in real time and still has space for further optimization, so it is significant to find its bottleneck. Secondly, this problem size is very common in real application, if it can satisfy the real time requirement, then it will have wide application prospect.

In order to get the insight of local stereo matching algorithm implementation on GPU, we have the following experiment.

4.3.1 Experiment design

In the experiment, the applied measurement tool is *CUDA Visual Profiler*. It can measure the task execution time, the amount of resources used during the kernel function running on GPU, such as the size of the used shared memory per block, the number of registers used in each thread and etc. Based on the data measured with *CUDA Visual Profiler*, we can find the bottle neck of the task from data analysis.

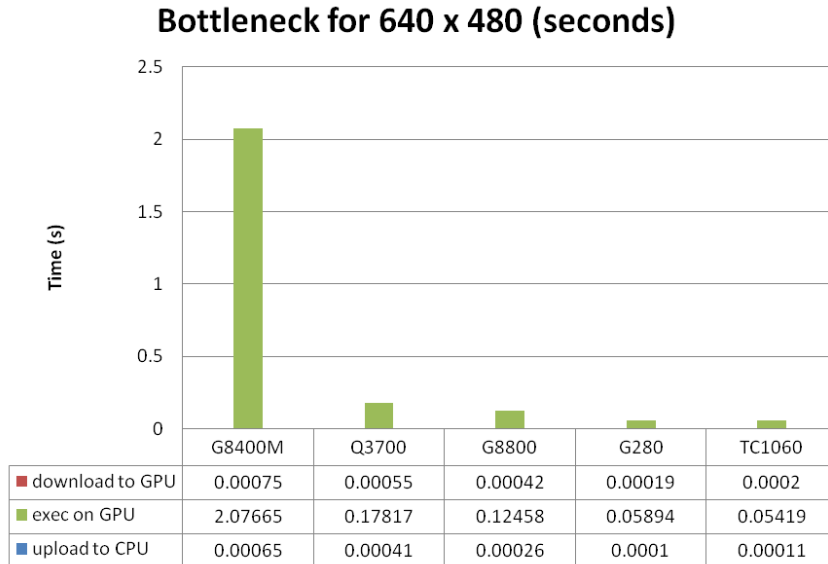
The procedure of the experiment is not complicated, we just need to compile the CUDA code and generate the execution file, and then we launch it on the selected GPU with its *CUDA Visual Profiler*. After that, the profiler will generate the profiler output table which contains all the necessary information taking place on that platform.

In addition, the profiler can just measure the time of data transfer between host (CPU) and device (GPU) and that of task executed on device (GPU). If one would like to know how much time each step of the task spends, he / she should design the experiment (him) herself. For example, one can use the difference of the executing time of the task between with the specific step and without it to obtain the actual time spent by the step.

4.3.2 Task measurement and Analysis

(i) Bottleneck of the whole task

For 640×480 case, after measuring the time of data transfer between CPU and GPU and the time of the kernel function running on different GPUs, one can get an overview of performance distribution among them from the following figures.



	G8400M	Q3700	G8800	G280	TC1060
download to GPU	0.00075	0.00055	0.00042	0.00019	0.0002
exec on GPU	2.07665	0.17817	0.12458	0.05894	0.05419
upload to CPU	0.00065	0.00041	0.00026	0.0001	0.00011

Figure 4.7: Profiling of the GPU based stereo matching alg. (I)

From Figure 4.7, because the data transfer time between CPU and GPU is so tiny that it is not displayed well. However, we can easily calculate the corresponding

percentage of each part taking up the whole task, and it is illustrated in Figure 4.7 in a more obvious way:

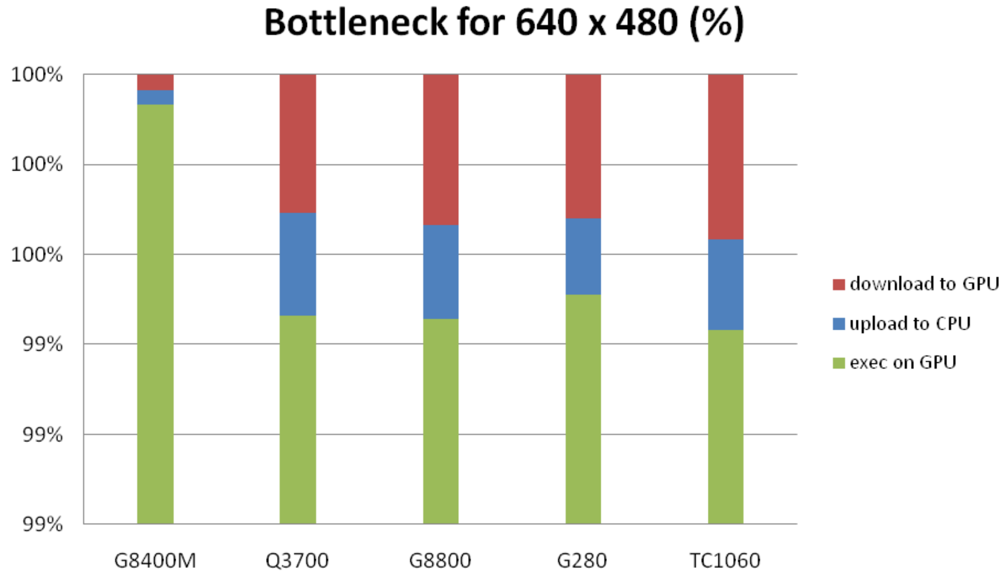


Figure 4.8: Profiling of the GPU based stereo matching alg. (II)

As seen from Figure(4.7 and 4.8), for all platforms, the whole task bottleneck is not data transfer between host and device but the execution on device. Then the rest experiment will just concentrate on this case (640×480) running on Quadro FX3700 GPU, because it is enough to represent the behaviours on other GPUs, and additionally, other relevant reasons has been explained at the beginning of Section 4.3.

Analysis

One can easily find that, to 640×480 case, whatever the GPU it is, the time of data transfer between host and device is always less than 1 millisecond. In addition, comparing the execution part, the percentage of data transfer between CPU and GPU is less than 1% in total so that it can be reasonably ignored. The results from the figures demonstrate that the way of putting the whole algorithm on GPU is correct, because it effectively avoids accessing the host frequently. It is known that the memory bandwidth between CPU and GPU is 8 GB/sec (PCIe×16 Gen2) which is far below the memory bandwidth on GPU such as 51.2 GB per second for Q3700.

(ii) GPU resource usage from CUDA prof.

After profiling the task on Q3700 platform (compute capability 1.1), we can get the resource usage information of the kernel function executed on the device in Table 4.11.

The multiprocessor occupancy is the ratio of active warps to the maximum number of warps supported on a multiprocessor of the GPU. In the selected device (computability 1.1), the max number of warps supported on each multiprocessor is 24. Occupancy is determined by the number of threads per block, size of used shared memory per

Table 4.11: Kernel function resource usage

#Block per Grid	#Threads per block	Static_shared_mem per block	#regs per thread	Occupancy	#active warp
40×30	16×16	5460	18	0.333	8 / mp

#instructions	gld_uncoalesced	gst_uncoalesced	Total global access	#instr / global
48399546	6627504	88064	6715568	7.3

Branch	Divergent branch
7005652	58870

block including both dynamic one and static one, and the number of registers used per thread, because these tree kinds of resource are limited on each multiprocessor. In our case, dynamic shared memory is not used.

In addition, "*#instr / global*" means the average number of instructions executed for every one global memory access. It equals *#instructions* divided by total number of global access which is the sum of global load (*gld*) times and global store (*gst*) times. In our case, all global accesses are uncoalesced and no coalesced ones.

Analysis

From the data above, one could know something:

Firstly, the current occupancy of GPU is 33.3% and the maximum warps of each multiprocessor are 24. Thus, the number of active warps on each multiprocessor is equal to the product of multiprocessor occupancy and the maximum warps per multiprocessor which is 8 displayed in Table 4.11.

Secondly, each multiprocessor on GPU implements zero-overhead warp scheduling which means that once the operands of next instruction for the specific warp are ready, the warp is eligible for execution. In addition, although one global access is time consuming which is about 200-cycle memory latency on the device (1.0 or 1.1), because of zero-overhead warp scheduling and the support for parallel execution of multiple warps, the global memory latency is possible to be hidden. For example, in our case, it needs 4 clock cycles to dispatch the same instruction for all threads in a warp in the GPU with computability 1.1. It is known from Table 4.11 that every one global memory access needs 7.3 instructions in average and now each multiprocessor has 8 active warps running in parallel. Thus time spent on executing 7.3 instructions on 8 warps is:

$$time = \frac{4\,cycle}{instr} \times \frac{7.3\,instr}{warp} \times 8warp = 223.6\,cycles > 200\,cycles$$

It is enough to fully tolerate 200-cycle memory latency! However, one should note that it cannot improve performance by simply enhancing the occupancy such as changing the granularity of the threads block, and the effect of granularity will be discussed in the next part.

Thirdly, because there are just un-coalesed accesses to global memory seen from the table, these will be very costly. In order to know un-coalesed accessing pattern clearly, we will explain how it happens and how it affects the memory bandwidth in "*Effective memory bandwidth*" part.

Lastly, branch or divergent branch is caused by the flow control statement (e.g. *if, switch, for, while*) and it significantly affects the total number of instructions executed for specific warp, because branches make the threads of the same warp have different execution paths which must be serialized. Generally, avoiding different execution paths in the same warp could enhance the performance.

(iii) Granularity of the threads block

In this part, we will get an insight about the relation among the number of threads per block, multiprocessor occupancy and kernel function performance (CPU time) when executing on the device. After measuring the task through CUDA visual profiler, we have Table 4.12:

Table 4.12: Granularity of the thread block

#Threads per block	**Static_shared_mem per block (bytes)**	**#regs per thread**	**Occupancy %**	**Warp serialized**	**Exec time (seconds)**
8×8	2644	18	41.7	24817509	0.19234
10×10	3252	18	50.0	33967079	0.24173
12×12	3924	18	41.7	25926330	0.20547
14×14	4660	18	29.2	26433219	0.23632
16×16	5460	18	33.3	0	0.17725
18×18	6324	18	45.8	18895667	0.21407
20×20	7252	18	54.2	14870430	0.19831

Additionally, their relation can be illustrated through Figure 4.9:

Based on Table 4.12 and Figure 4.9, it is easily seen that the best performance taking place when the threads block size is 16×16. At 16×16, there are no serialized warps and it spends the least time even if its multiprocessor occupancy is not the highest.

Analysis

Based on Figure 4.9, it is easily to conclude that high occupancy does not mean high performance, for example, at 16×16, its occupancy is not the highest but its performance is the best among these cases, however 20×20 obtains the highest occupancy while its performance is optimal. In addition, during the range less than 14×14, high occupancy even corresponds to low performance. The reason behind it is that although the occupancy is increased, at the same time it leads to other factors which are harmful to the performance.

In our case, the harmful factor driven by the configuration of threads block is the serialized warps. They are caused by the shared memory bank conflicts which can make

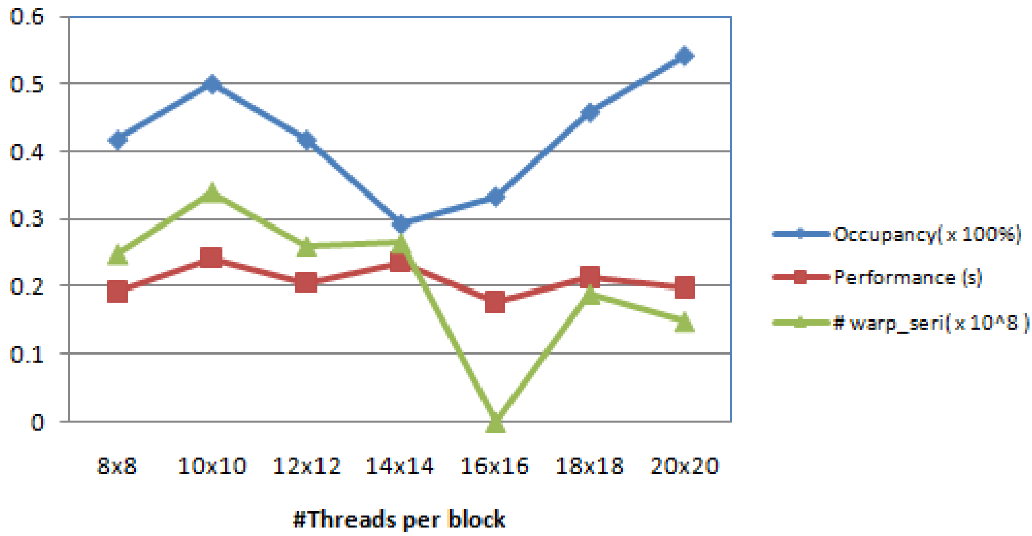


Figure 4.9: Illustration of the granularity of the block

parallel warps executed sequentially. The results in Figure 4.9 demonstrate the effect of the serialized warps.

In Figure 4.9, the number of serialized warps has almost the same trend with the performance which means that more serialized warps make worse performance, however, when the number of serialized warps drops to 0, the kernel function has the best performance.

In addition, how do shared memory bank conflicts happen in the cases above? Can we avoid them? In order to give the answers of these two questions, one should know what shared memory bank conflict is. In fact, in order to obtain a high memory bandwidth, shared memory is divided into several banks (memory modules) with equal sized (32-bit for the devices with computability 1.x) width for each one so that shared memory can be accessed simultaneously. One memory bank can serve one thread once, thus if more than one threads access the same bank simultaneously, then there will be conflicts, which is also called the bank conflicts, among the threads. In order to avoid bank conflicts, the treads in a warp will be serialized.

For the cases above, the cause of their bank conflicts is from the array mode of threads block. Firstly, the computability of the GPU (Q3700) in experiment is 1.1 and its shared memory has 16 memory banks. The shared memory block, accessed by its threads block, spreads on the banks in row wise. Figure 4.10 and Figure 4.11 explain how bank conflicts happen in the cases of threads block with configuration 8×8 and 20×20, and the rest of cases have the similar explanation to 8×8 and 20×20 for the reason of their bank conflicts.

In the case of 8×8 threads block, as the threads block width is shorter than the number of banks, it makes each of the first 8 banks accessed by 2 threads in half warp simultaneously and then the bank conflicts are present.

In the case of 20×20 threads block, the threads block width is longer than the

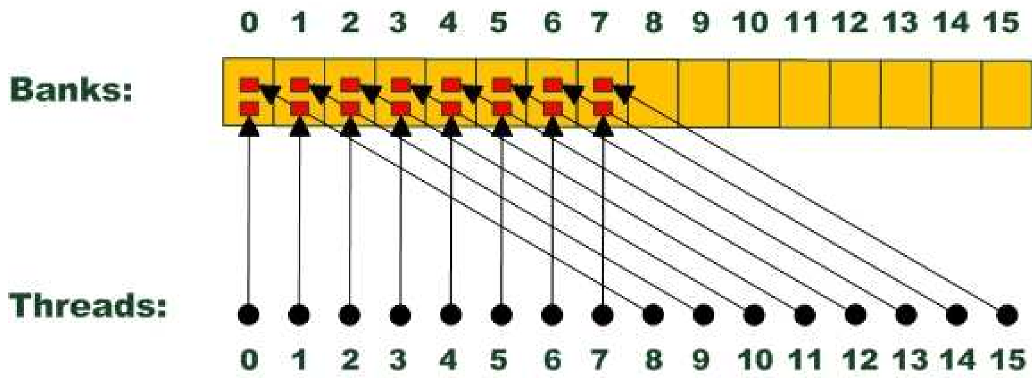


Figure 4.10: Bank conflicts of 8×8 threads block

number of banks, thus the bank conflicts take place in the end of each row of the threads block with the beginning of the next row. Figure 4.11 illustrates the situation.

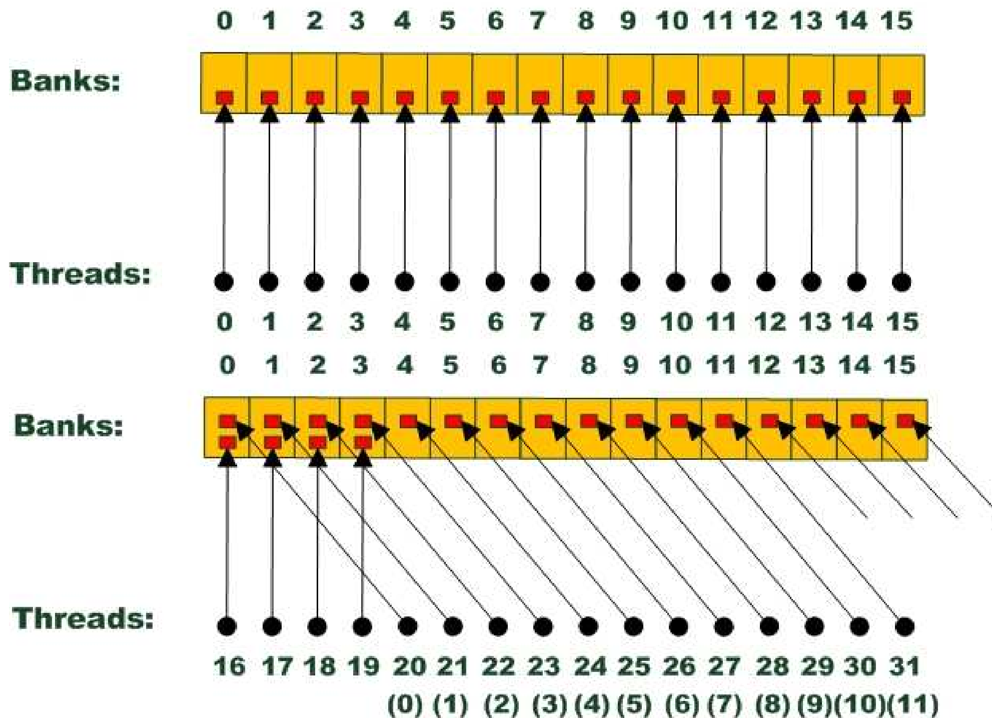


Figure 4.11: Bank conflicts of 20×20 threads block

Above all, one can conclude that only if the thread block width is equal to or multiple times of the number of banks (16 in this case), then there will no bank conflicts and the threads in a warp will not be serialized so that the kernel function can maintain its performance. For the device used, the width 16 or 32 is suitable. We will use 16 as the threads block width, because it is more fitted to the number of banks which is 16. Until now, the rest parameter still needed to be determined is the height of the thread block.

There are several alternatives for the height. Firstly the height should be an even number, as warps (32 threads per warp) are scheduling units in streaming multiprocessor of GPU; if the height is odd number, there will be half idle threads in some warps. Additionally, the upper bound of height should less than or equal to 28 based on the CUDA occupancy calculator and its lower bound should be 6 considering when the fixed size (11×11) window for computing SAD on the boundary of the image.

Based on the following experiment results, one could find the optimal combination of width and height to the threads block.

Table 4.13: Granularity of fixed width block

#Threads per block	**Static_shared_mem per block (bytes)**	**#regs per thread**	**Occupancy %**	**Warp serialized**	**Exec time (seconds)**
6×16	3380	18	37.5	0	0.16270
8×16	3796	18	50.0	0	0.15694
10×16	4212	18	41.7	0	0.16114
12×16	4628	18	50.0	0	0.15881
14×16	5044	18	29.2	0	0.18571
16×16	5460	18	33.3	0	0.17696
18×16	5876	18	37.5	0	0.17693
20×16	6292	18	41.7	0	0.17434
22×16	6708	18	45.8	0	0.17234
24×16	7124	18	50.0	0	0.17181
26×16	7540	18	54.2	0	0.17601
28×16	7956	18	58.3	0	0.17698

And Figure 4.12 illustrates Table 4.13.

As seen from the experiment results, first, it demonstrates that the threads block width fixed at 16 can promise no bank conflicts as the number of serialized warps right now is 0. In this situation, the optimal performance is never 16×16 setting for the threads block. At this time, all cases with different heights of threads block can be divided into two cliques, one is when the height is larger than and equal to 14 and another is when the height is less than 14. Generally, the latter cliques (fine grains) have better performance than the previous ones (coarse grains). Also the best performance is present at the height 8.

One could find that, in coarse grains, the performance keeps almost constant with the increment of occupancy. While, in fine grains, the occupancy is related to the performance, and the results show that higher occupancy is with better performance.

Therefore, the optimal size of threads block for the kernel function is that the width is 16 and the height is 8.

(iv) Effective memory bandwidth

Bandwidth is the rate at which data can be transferred. It is one of the most important gating factors for performance. To measure performance accurately, it is

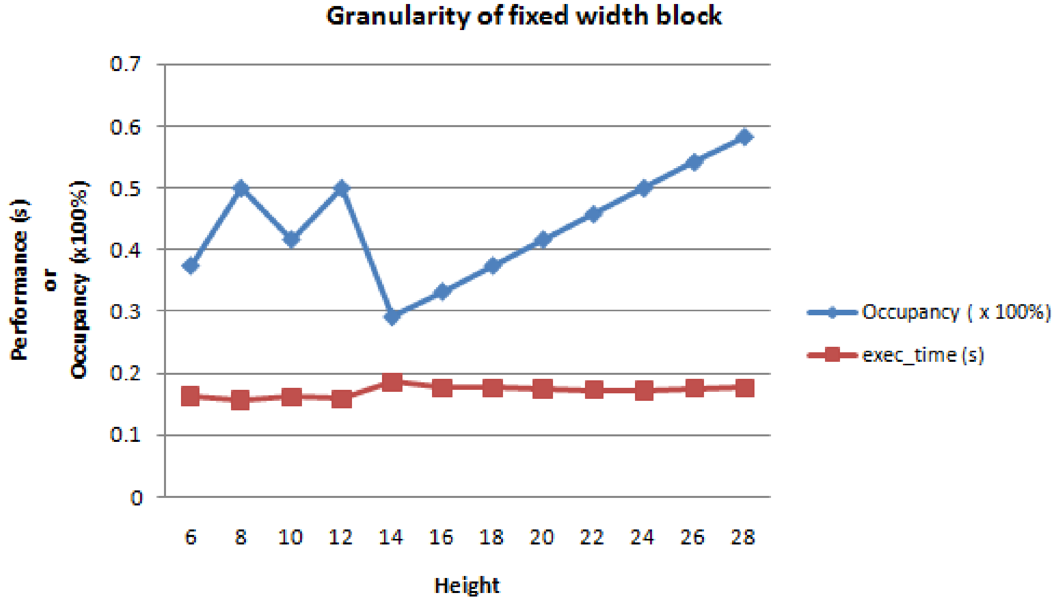


Figure 4.12: Granularity of fixed width block

useful to calculate theoretical and effective bandwidth. When the latter is much lower than the former, design or implementation details are likely to reduce bandwidth, and it should be the primary goal of subsequent optimization efforts to increase it. Device theoretical bandwidth is calculated with Equation 4.3, however its effective bandwidth is calculated by timing specific program activities and by knowing how data is accessed by the program. It is given by Equation 4.9:

$$Effective_bandwidth = \frac{B_r + B_w}{10^9 \times time} \tag{4.9}$$

Here, the effective bandwidth is in units of GB/sec, B_r is the number of bytes read from global memory per kernel, B_w is the number of bytes written to global memory per kernel, and ***time***, spent by the kernel function on device, is given in seconds.

In our case, the theoretical bandwidth is 51.2 GB/sec calculated before and the effective bandwidth is calculated below:

Firstly, in our case, the number of threads per block is 16×16, thus from Table 4.12,

$$time = 0.17725\, seconds.$$

Secondly, the image resolution in experiment is "640×480" and the image format is based on pgm (each pixel of the image is one byte). The kernel function only writes global memory once for storing the disparity value to global memory. Thus,

$$B_w = 640 \times 480 = 307200\, bytes$$

Furthermore, the global memory loading happens when the kernel function copies the two images information from global memory to shared memory. For the left or reference image, it is only read once from global memory. For another (right) image,

based on the implementation strategy, the times read by kernel function is equal to the disparity range determined by setting and in our case it is 55. In addition, the bytes of each image read by the kernel is actually more than the original image size, because the shared memory resolution used for one block is larger than that of threads block which is explained in the part "*shared memory management*". Based on "*shared memory management*", for a 16×16 thread block, it actually reads 26×26 bytes from global memory, and for each image it has 40×30 blocks got from *cudaprof*. Thus, the total number of bytes read from global memory should be:

$$B_r = (26 \times 26) \times (40 \times 30) \times (1 + 55) = 45427200\, bytes$$

Therefore,

$$Effective_bandwidth = \frac{45427200 + 307200}{10^9 \times 0.17725} \approx 0.258\, GB/sec$$

Analysis

From the result above, one could find that the achieved effective memory bandwidth on GPU is very low in real implementation. And we will use the following equation to evaluate how efficiently we apply the memory bandwidth which is offered by the hardware (GPU).

$$E_{mem_bw} = \frac{Effective_bandwidth}{Theoretical_bandwidth} \times 100\% \tag{4.10}$$

In Equation 4.10, *Theoretical_bandwidth* is calculated through Equation 4.3 and, for the device used in the experiment, it is 51.2 GB/sec. Thus, with Equation 4.10, one could have:

$$E_{mem_bw} = \frac{0.258\, GB/sec}{51.2\, GB/sec} \times 100\% \approx 0.5\%$$

Based on the efficiency, it means that the implementation does not effectively use the high memory bandwidth on GPU and it could be preliminarily estimated that there is still a large space to improve the usage of the memory bandwidth on GPU.

As a matter of fact, from the data in table 10, one could get some clews. The CUDA visual profiler shows that all of the global memory access are un-coalesced which means that, usually the kernel function can complete the data transferring by accessing the corresponding memory segment once, but in an un-coalesced pattern, it does it with many times. Obviously, un-coalesced global memory accessing is very costly.

How does un-coalesced global memory access happen in the implementation? The explanation will be given in the following part:

Firstly, for the GPU (compute computability 1.1) used in the experiment, its global memory is made up of fixed size memory segments and each one is 64 bytes. Every memory segment consists of sixteen 32-bit (4 bytes) words.

Secondly, based on the cognition to global memory construction (reference to [24]), the coalesced accessing mode can be achieved when the k-*th* thread in a half warp

Figure 4.13: Coalesced global memory accessing mode

(16 threads in total) aligns with the k-*th* word in a memory segment, however not all threads need to be present. The coalesced mode can be illustrated by Figure 4.13.

The Advantage of coalesced accessing mode is that, all of the threads in a half warp can be fed within one transaction of the memory segment.

Thirdly, if at least one of the thread in a half warp does not access the corresponding word in a segment, such as out of order case or misaligned case, then the un-coalesced accessing to global memory will happen and it leads to a separate transaction requested by each thread in a half warp which are 16 transactions in total for one segment. Obviously, un-coalesced accessing mode increases the times of global memory accessing extensively which will be very costly. Therefore, it is wise to avoid un-coalesced accessing to global memory as many as possible. Figure 4.14 illustrates the cases of un-coalesced accessing modes.

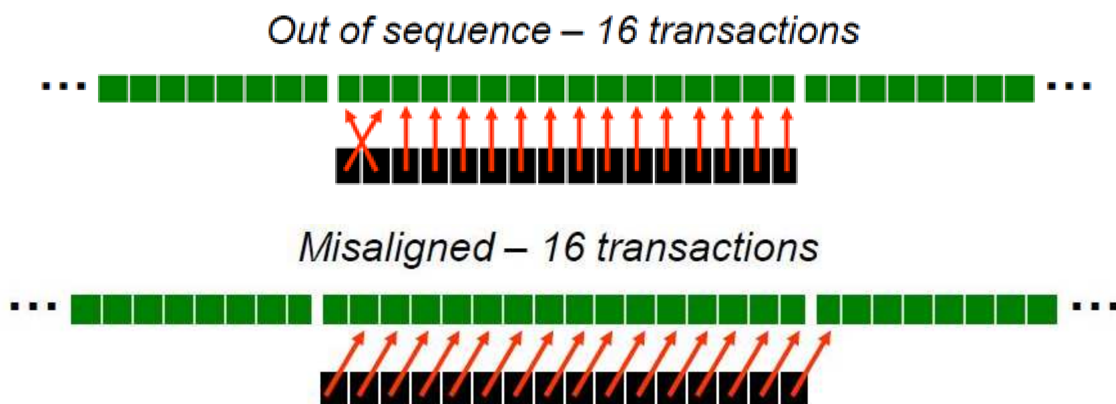


Figure 4.14: Uncoalesced global memory accessing modes

After understanding the reason of un-coalesced global memory accessing, now it is easy to explain how un-coalesced accessing mode happens in the real implementation. As described before, the input images of the algorithm are grey scaled images (.pgm format), thus each pixel size of the grey scaled image is one byte. In addition, based on the implementation itself, one thread accesses one pixel, so it accesses one byte. When the input image stores in global memory, every four pixels can hold one 32-bit word which is the basic element of memory segment. At this time, four threads access one word and 16 threads access the first four words of one memory segment for sure. The situation is illustrated by Figure 4.15.

Based on the situation in Figure 4.15, obviously, a half warp is impossible to be

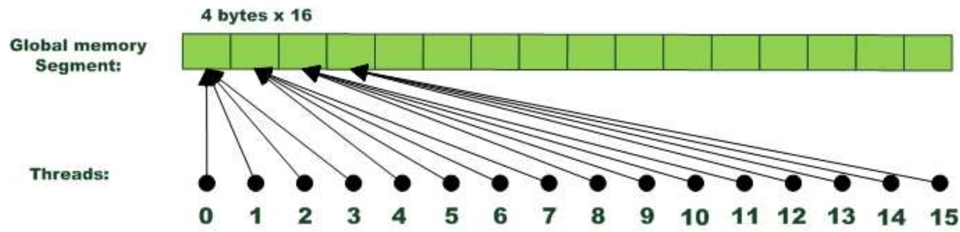


Figure 4.15: The real situation in the implementation

aligned with one memory segment. Furthermore, during accessing the shifted target image in the algorithm, the threads in a half warp are also misaligned with the memory segment. Therefore un-coalesced accessing to global memory in the real implementation is inevitable.

Fortunately, there are some ways to improve current situation. For example, using texture cache fetch instead of global memory load could avoid global memory accessing with some extent and save the time of reading data from the input images as texture cache is on chip which means it is more closed to the multiprocessors. For un-coalesced writing to global memory, it is inherent in programming, it is not easy to make it coalescable. In addition, replacing grey scaled input image with its color version (32 bits or 4 bytes per pixel) can apply coalesced global memory accessing mode better, because in this way, the threads in a half warp can be aligned with the word in a memory segment.

(v) Bottleneck of the execution part running on GPU

Based on previous experiment results and analysis, one can know that for the whole task, the part of data transferring between host and device can be ignored as it just takes up less than 1% of the total time, and the execution part running on GPU is the bottle neck of the whole task. For the execution part, it is still not clear about the detail of construction of the time for each step in the algorithm. Because the global memory is accessed in an un-coalesced way which is very costly, it is natural to divide the execution part into two; one is the time spent on global memory load and store, another one is the pure computation including both SAD (sum of absolute difference) and WTA (winner takes all) on each thread.

In order to know how much time is spent on global memory load and store, the computation of WTA and SAD could be removed temporarily and then the measured execution time for the modified kernel function is that of global memory accessing. Furthermore, the difference of the time between whole execution part and global memory accessing is the cost for the pure computation. In this way, one can get the data in Table 4.14.

One could get an insight of the percentage for each part in the above table through Figure 4.16.

Analysis

Regardless of data transferring between host and device, from the previous analysis about the effective memory bandwidth, one finds that the time spent on un-coalesced

Table 4.14: Performance distribution of each part (seconds)

Host2Device	Device2Host	Global_mem accessing	WTA&SAD
0.00055	0.00041	0.03447	0.14370

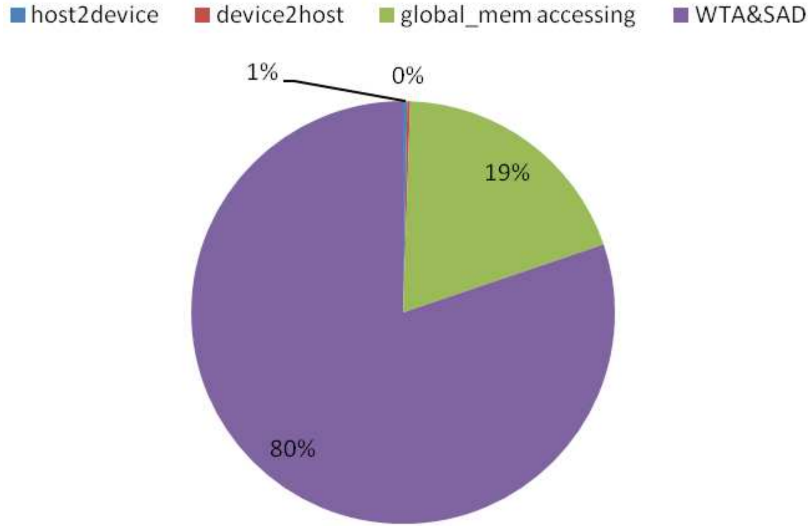


Figure 4.16: Overview of the performance distribution

global memory accessing should be much. As a matter of fact, from the data above, it demonstrates the estimation about un-coalesced global memory accessing, because it consumes 19% of the total time. However, it is still not the most time consuming part, as the time cost of the pure computation is four times of that of global memory accessing. Thus, although global memory accessing is somewhat expensive, the pure computation of WTA and SAD is the real bottle neck of the execution part on GPU which takes up 80% of the total execution time and the optimization on WTA and SAD can bring more significant improvement of the performance than that on global memory accessing.

Because all of operations in one thread are totally sequential and the computation of WTA and SAD is threefold iterative operation, time complexity of computing WTA and SAD is equal to its work complexity which is $O(l \cdot m^2)$ calculated before. "l" is the disparity range in computing WTA and "m^2" is the size of the fixed window size for computing SAD. They both are the inherent features of the algorithm and its implementation. Based on the discussion of global accessing, it is possible to improve it without changing the algorithm. However, for the computing part, it is not easy to reduce its time cost just by code transformation unless the algorithm itself can be modified or optimized correspondingly.

Actually, there exists a solution to reduce the computation without affecting the

accuracy of the results. This will be explained in detail in the chapter of *Further Optimization on GPU*.

4.3.3 Discussion and conclusion

Until now, the profiling for the task is completed. After that, one can have a deeper insight of the whole task. From the experiments, the problems that impede the performance have been found, and some possible optimizations are provided.

In summary, the problems existed and the corresponding solutions are extracted from the topics above:

(i) Bottleneck of the whole task

The result shows that data transferring between host and device has little effect on the performance. The decision of running the whole algorithm on GPU is correct and it can maximize parallel execution.

(ii) GPU resource usage

The data indicates that, firstly, the current number of active warps on each multiprocessor is enough to hide the global memory latency. In addition, there is only un-coalesced global memory accessing and the number of branch in all of the warps is somewhat large. Thus the possible optimization should be focus on improving the way of global memory accessing and decreasing the number of branches in warps if possible.

(iii) Granularity of the threads block

Based on the experiment, multiprocessor occupancy has no direct relation to the performance. And 16×16 threads block is not the optimal choice. The optimal one is $8 times 16$ for the device with compute capability 1.0 or 1.1 and 6×32 for device 1.2 or 1.3. To the fine grain of the threads block, higher multiprocessor occupancy could achieve a little better performance.

(iv) Effective memory bandwidth

After getting to know the effective memory bandwidth of the real implementation, it is far away from the theoretical one and still needs being improved further. The problem is the un-coalesced way of global memory accessing. One possible optimization is to use texture fetch instead of global memory reading. However, un-coalesced global memory writing, which happens during writing the results into global memory, is inherent in programming, thus it is inevitable. Another solution is to change the format of grey scaled input image into its color version to make it stored in global memory more adaptable. Of course, the combination of these two solutions is also probable.

(v) Bottleneck of the execution part running on GPU

After splitting the execution part executed on GPU into global memory accessing and pure computation, pure computation shows more time consuming than current un-coalesced global memory accessing and pure computation is inherently determined by the algorithm itself. Therefore, the improvement of algorithm is the intrinsic way for the performance optimization.

4.4 Conclusion

In fact, for the original implementation of the algorithm, it achieved the real time requirement in some cases. For example, the green data in Table 4.2 represents the real time cases, but others are not in real time. Based on the experiments and analysis, one could know that there is still space to improve the performance, thus more data could turn green in Table 4.2. The moderate goal for the further optimization is that one can make 640×480 input images running in real time on Q3700 GPU.

After experiment, the most important result is that the possible optimization has been found and they are:

1. Apply the optimal configuration for the threads block which is 8×16 to the device with compute capability 1.0 or 1.1.
2. Reducing the number of branch and divergent branch if possible.
3. Algorithm improvement on the computation of WTA and SAD if possible.
4. Use texture cache fetch instead of global memory loading.
5. 2D array arrangement for threads block.
6. Change the grey scaled image with its color version to make it more fitted to the memory segment of global memory.

Therefore, the next chapter will complete the provided optimization to see how fast we can achieve for the application, and then fully answer the questions that whether GPU is suitable for the development of stereo vision application and whether it is possible to make the built stereo vision system run in real time.

GPU Optimization 5

The contributions of this chapter are:

1. Further optimize the GPU based stereo matching algorithm and make more cases with the GPU based solution run in real time or near real time. Especially, when the block size for computing SAD is down to 5×5, the block matching method could run at **128.5 frames per second** in theory with the image size 640×480 and 50 pixels disparity range on GTX 280.
2. Demonstrate that GPU is an ideal platform for further developing the stereo vision system based on block matching method to make it run in real time or near real time.
3. (To do) Implement a GPU based 3D sensing part with the develop GPU based stereo matching algorithm.

Also in this chapter, the following research questions are answered:

1. Is GPU suitable for the development of stereo vision application?
2. Is it possible to make the built stereo vision system run in real time?

5.1 Principles of the optimization in CUDA

From CUDA Best Practices 3.0 [12], it provides some recommendations for the optimization on CUDA application. In our case, high-priority and medium-priority recommendations are the main concentration and they are:

High-priority:

H1. Maximize parallel execution.
H2. Use the effective memory bandwidth as the metric of performance and optimization benefits.
H3. Minimize data transfer between the host and the device.
H4. Ensure coalesced accessing to global memory whenever possible.
H5. Minimize the use of global memory.
H6. Avoid different execution paths with the same warp.

Medium-priority:

M1. Access to shared memory without bank conflicts.
M2. Use shared memory to avoid redundant transfers from global memory.
M3. Maintain approximately 25% occupancy as a minimum

M4. The number of threads per block should be a multiple of 32 threads for optimal computing efficiency and facilitating coalescing.
M5. Use fast math library whenever speed is more important than precision.

For H1, it has been discussed in Section 3.4.2(b) and implemented for our stereo matching algorithm.

Additionally, H2 will be continually applied in this chapter and its computation is based on Equation 4.9.

For H3, firstly, the data transfer between the host and the device only happens when input images are download from CPU to GPU and the output image is uploaded from GPU to CPU. Secondly, based on the experiment results in Section 4.3.2(i), data transfer between the host and the device has little effects on the performance. Thus, H3 will not be considered in the afterwards optimization.

For medium-priority recommendations, in Section 4.3.2(iii), it was found that how to avoid bank conflicts while accessing shared memory and how to configure the threads block to get an optimal computing efficiency. Furthermore, based on the original implementation which applies 16×16 threads block configuration, it is done by using shared memory explained in Section 3.4.5(a) and its occupancy of the multiprocessor from Table 4.11 is 33.3%. Therefore, from M1 to M4, they are already achieved; however, as M5 is not necessary in our specific implementation which has no complicated calculation, so M5 will not be considered too.

Until now, the rest of the recommendations are from H4 to H6. Actually, based on the conclusion of Chapter 4, the possible optimization places are matched with H4 to H6, thus the optimization work in this chapter will follow the results of chapter 4 and H4 to H6, and be implemented on the device Quadro FX 3700 for 640×480 images so that the results after optimization could be compared with the original result of 16×16 in Table 4.12 and Section 4.3.2(iv) which are also based on Quadro FX 3700 with 640×480 images.

As a start point, adopting the time and effective memory bandwidth as metrics, Table 5.1 presents the performance of the original implementation "Kernel 1" which is the case with 16×16 threads block for 640×480 images. When there is a new improvement with the optimization method in the rest sections of this chapter, we will call the improved kernel as "Kernel i (the applied method)". For instance, in this chapter, there will be five kernels, the later the better. They are:

- Kernel 1: the original solution.
- Kernel 2: with new threads block configuration.
- Kernel 3: with branch reduction method.
- Kernel 4: with algorithm improvement of row based accumulation reduction.
- Kernel 5: with CUDA array for 2D data storage.

For simplification, in the rest of this chapter without specific declaration, the experiment results are all for 640×480 images on the device Quadro FX 3700 with fixed parameters of mask size 11 and disparity range 55 from stereo matching algorithm. Also, time and effective memory bandwidth (calculated by Equation 4.9) are adopted as the metrics of performance.

Table 5.1: The original implementation performance

Case	**Time (seconds)**	**Effective Memory Bandwidth (GB/sec)**
Kernel 1 (16x16)	0.17725	0.258

5.2 Threads block configuration improvement

From the discussion about the granularity of the threads block in Section 4.3.2(iii), the results showed that 16×16 threads block was not the optimal case, even if it accessed to shared memory without bank conflicts and could make an efficient use of the threads in one multiprocessor. In fact, the best performance took place at the case of 8x16 threads block configuration, as it had finer granularity than that of 16×16 threads block configuration to make the multiprocessor have higher occupancy.

Thus, the first optimization for the CUDA program is to set the threads block from 16×16 to 8×16 and then we could see the result after improvement in Table 5.2.

Table 5.2: Optimization from threads block configuration

Case	**Time (seconds)**	**Effective Memory Bandwidth (GB/sec)**	**Step Speedup**	**Cumulative Speedup**
Kernel 1 (16x16)	0.17725	0.258	1	1
Kernel 2 (8x16)	0.15694	0.291	1.13×	1.13×

In Table 5.2, "*Step Speedup*" and "*Cumulative Speedup*" are used to compute how much performance enhancement got from the optimization method and they can be calculated by the following formulas:

$$Step_Speedup = \frac{Bandwidth_i}{Bandwidth_{i-1}} \times = \frac{Time_{i-1}}{Time_i} \times \tag{5.1}$$

$$Cum_Speedup = \frac{Bandwidth_i}{Bandwidth_1} \times = \frac{Time_1}{Time_i} \times \tag{5.2}$$

5.3 Branch or Divergent branch reduction

Usually, flow control statements (*if, switch, do, for, while*) can increase the number of branches and divergent branches if branch conditions are explicitly or implicitly related to the threads. "*explicit relation*" represents that the control flow is directly determined by the thread ID, while "*implicit relation*" means the control flow is determined by the specific result in the thread which will be further explained in Section 5.4. The increment of branches will increase the number of executed instructions and consequently reduce the instruction throughput of the GPU. In addition, if the threads of the same warp diverge caused by the branch, then different execution paths must be serialized,

consequently increasing the total number of instructions executed for this warp. Thus, if one can reduce the number of branches or divergent branches, the application performance will be improved.

In our case, the place of reducing branches is from the computation of SAD in a fixed size window around the pixel to be computed.

The original SAD computation is:

```
int e_row = threadIdx.y + MASK;
int e_col = threadIdx.x + MASK;

for(int j = threadIdx.y; j < e_row; j++){
    for(int i = threadIdx.x; i < e_col; i++)
        res += abs(Mds[j][i] - Nds[j][i]);
}
```

It is easily seen that, in the original SAD computation, the '*for*' statement condition is explicitly related to thread ID. Thus, it will increase the number of branches in the kernel. Actually, the accumulation is in the whole fixed window and just determined by the size of the window, so the computation is independent to the thread itself. With this consideration, the original SAD computation could be implemented in the following way.

The improved SAD computation is:

```
for(int j = 0; j < MASK; j++){
    for(int i = 0; i < MASK; i++)
        res += abs(Mds[threadIdx.y+j][threadIdx.x+i] - Nds[threadIdx.y+j
            ][threadIdx.x+i]);
}
```

In order to see the effect of the improvement, the difference of original SAD and improved SAD are shown in Table 5.3.

Table 5.3: The results of branch reduction from CUDA profiler

State	**# of Branch**	**# of Divergent Branch**	**Instructions**	**Time (seconds)**	**Speedup**
Orginal SAD	7025252	78474	49241645	0.15694	1.00×
Improved SAD	1053412	78474	24088026	0.08355	1.88×

From Table 5.3, the number of branch and the executed instructions in improved SAD are both significantly less than those of original SAD, so that the performance is enhanced. In addition, as each thread of the same warp does the same computation in a fixed range, so there are no divergent branches in a warp.

With branch reduction, the whole optimization result is shown in Table 5.4.

Table 5.4: Optimization from branch reduction

Case	Time (seconds)	Effective Memory Bandwidth (GB/sec)	Step Speedup	Cumulative Speedup
Kernel 1 (16x16)	0.17725	0.258	1	1
Kernel 2 (8x16)	0.15694	0.291	1.13×	1.13×
Kernel 3 (branch reduction)	0.08355	0.547	1.88×	2.12×

5.4 Algorithm improvement

From Figure 4.16, it is known that the bottleneck of the implementation is actually from the computation of WTA and SAD not from global memory accessing. Thus, if one could reduce the time of SAD or WTA computation, then the performance of the parallel stereo matching algorithm will be improved.

Based on the design discussion in Section 3.4.2(b), the SAD and WTA computation are in one thread, thus they are sequential and cannot run parallel. Fortunately, there is a method can save the time of computing SAD without affecting the accuracy of the final result and the inspiration is from Tangfei's work [21]. The method is that:

Firstly, in a disparity range, different disparity has different SAD value and stereo matching algorithm is responsible to find the minimum SAD value and record its disparity value.

Secondly, once the minimum SAD is found, there is no need to accumulate all the AD (absolute difference) in the block when computing other SAD. The reason is that, now that other SAD from non optimal disparity is larger than the minimum one, so the accumulation from part of its block is probably larger than the minimum SAD. Thus, when computing non-minimum SAD, it could stop earlier to avoid unnecessary computation. The mathematical demonstration is published in Tangfei's work [21].

In real implementation, there are three possibilities to achieve the idea above and they are element based, column based and row based accumulation.

5.4.1 Element based accumulation

Element based solution is illustrated in Figure 5.1.

As shown in Figure 5.1, element based accumulation is that, for computing SAD in a block, once it accumulates one element of the block, then compare the intermediate SAD with the minimum SAD. If the intermediate SAD is larger than the minimal one, then stop accumulating the rest of elements of the block and start computing next block SAD.

With element based accumulation, its result from CUDA profiler is shown in Table 5.5.

From Table 5.5, element based accumulation led to a lot of branch and divergent branch, and generated a large amount of instructions, consequently the performance descended. As every element accumulation makes a comparison between current SAD

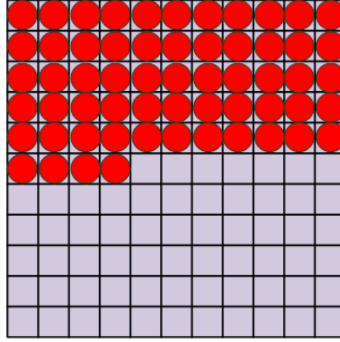


Figure 5.1: Illustration of element based accumulation

Table 5.5: Result of element based accumulation from CUDA profiler

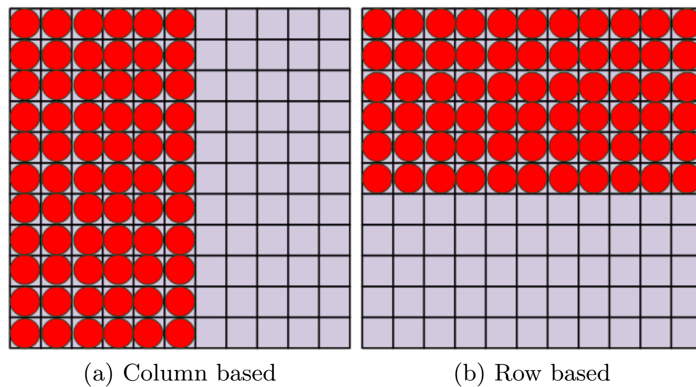
State	# of Branch	# of Divergent Branch	Instructions	Time (seconds)	Speedup
Kernel 3	1053412	78474	24088026	0.08355	1.00×
Element based	7564030	672048	46953608	0.13828	0.60×

and the minimal SAD to determine whether to stop accumulating or not and the minimal SADs of the threads are different from each other which are implicitly related to the threads, thus the threads of the same warp diverge and the number of executed instructions is increased.

The result of Table 5.5 demonstrated that element based accumulation is not a good choice.

5.4.2 Column based and Row based accumulation

Comparing to element based accumulation, the column and row based accumulation are illustrated in Figure 5.2.



(a) Column based (b) Row based

Figure 5.2: Illustration of column based and row based accumulation

As show in Figure 5.2, column (or row) based accumulation compares the intermediate SAD with the minimal SAD when every column (or row) accumulation is over. Thus the number of branch and divergent branch brought by column (or row) based accumulation is *O(n)* obviously less than that of element based accumulation which is $O(n^2)$.

With column (or row) based accumulation, their results from CUDA profiler is shown in Table 5.6.

Table 5.6: Results of column based and row based accumulation from '*cudaprof*'

State	# of Branch	# of Divergent Branch	Instructions	Time (seconds)	Speedup
Kernel 3	1053412	78474	24088026	0.08355	1.00×
Column based	1302330	191906	21961034	0.07222	1.16×
Row based	1285852	182602	18786280	0.06886	1.21×

An interesting result from Table 5.6 is that, the number of branch and divergent branch is increasing but the number of executed instruction and the performance are decreased and improved respectively. The reason is that, as column (or row) based accumulation could bring less branch (or divergent branch) than element based accumulation, therefore the number of increased instructions is less than the computation saved by column (or row) based accumulation and then in total the number of instructions is actually decreased.

In addition, it is easy to find that row based accumulation is better than column based accumulation from the results of Table 5.6. Another reason is that row based accumulation has better spatial locality than column based accumulation, so it makes row based accumulation obtain better performance.

Above all, row based accumulation will be the optimization choice and its result is shown in Table 5.7.

Table 5.7: Optimization from row based accumulation

Case	Time (seconds)	Effective Memory Bandwidth (GB/sec)	Step Speedup	Cumulative Speedup
Kernel 1 (16x16)	0.17725	0.258	1	1
Kernel 2 (8x16)	0.15694	0.291	1.13×	1.13×
Kernel 3 (branch reduction)	0.08355	0.547	1.88×	2.12×
Kernel 4 (row based accu)	0.06886	0.664	1.21×	2.57×

5.5 Efficient management of various memories

From the result of Figure 4.16, it shows that global memory accessing of the implementation takes up 19% of the total performance. Although global memory access is not the bottleneck of the task, its influence to the whole performance cannot be ignored. For this reason, if the global memory accessing could be further improved, it will be beneficial to the total performance. Also, the results from Table 4.11 reflect that there are a large amount of un-coalesced global memory accessing, thus there is still some space for optimizing the usage of global memory.

In fact, generally speaking, there are two possible ways in making a better use of global memory. One is the recommendation H5 in Section 5.1 which is to minimize the use of global memory. Another one is the recommendation H4 which is to ensure coalesced accessing to global memory whenever possible. These two ways are discussed in the following sections.

5.5.1 Minimizing the use of global memory

Based on Section 3.4.5(a), the use of global memory only happens when the threads load the image data from global memory to shared memory and store the disparity results from the multiprocessors to global memory. As the global memory writing is inherent in the program, thus it is inevitable. However, for global memory reading, there is a way to minimize the times of accessing to global memory, which is to apply the cache of read-only texture memory space to load the image data in device memory instead of directly reading the data from global memory.

From Figure 3.5 in Chapter 4, it shows the hierarchy of the memory on the device, as texture cache is closer to the multiprocessor, so if the cache is hit, texture fetch will spend less time than global memory read. A texture fetch costs one device memory read only on a cache miss. In addition the texture cache is optimized for 2D spatial locality, so threads of the same warp that read texture addresses that are close together will achieve best performance. This feature of texture cache is very suitable to our implementation that reads the image data.

Based on Figure 3.5, constant memory is also cached. As constant memory space, which is 64KB, is too limited to store the whole images data and constant cache does not obtain suitable features for our implementation, thus constant cache for loading the image data is not considered.

Furthermore, as 2D CUDA array is a opaque memory layout optimized for texture fetching, so it will be used to store the image data in device memory.

With the use of texture cache and 2D CUDA array, their effects from CUDA profiler are present in Table 5.8.

From Table 5.8, it shows that, with texture cache, global memory load is replaced by *texture cache hit/miss* and the times of accessing to device memory are significantly reduced so that the performance is totally improved. Additionally, with CUDA array, it enhances the texture cache hit rate, but the total time almost has no changes. Until now, with texture cache and CUDA array, global memory read is minimized.

Table 5.8: Effects of texture cache and CUDA array

Case	Branch	Divergent branch	Instructions	Gld un-coalesced	Gst un-coalesced	Time (seconds)
Row based accu	1285852	182602	18786280	9149868	87808	0.06886

Case	Branch	Divergent branch	Instructions	Tex cache hit	Tex cache miss	Gst un-coalesced	Time (seconds)
Texture	1271924	168014	18610599	1382089	25545	87808	0.06179
CUDA array	1271924	168014	18610605	1383666	24229	87808	0.06186

5.5.2 Coalesced accessing to global memory

From Table 5.8, although global memory read is minimized, global memory write is still not improved, especially it is un-coalesced accessing which is costly. From Section 4.3.2(iv), un-coalesced accessing to global memory is caused by the mismatch between pixel size (1 byte) and the element size (4 bytes) of the global memory segment shown in Figure 4.15, so the threads from half warp cannot be aligned with the memory segment. An immediate way to solve this problem is to expand the pixel size to 4 bytes each which means the images are no longer grey scaled but colourful. If the images become colourful, then the threads of half warp could be aligned with the memory segment when accessing the image data and the situation is shown in Figure 5.3.

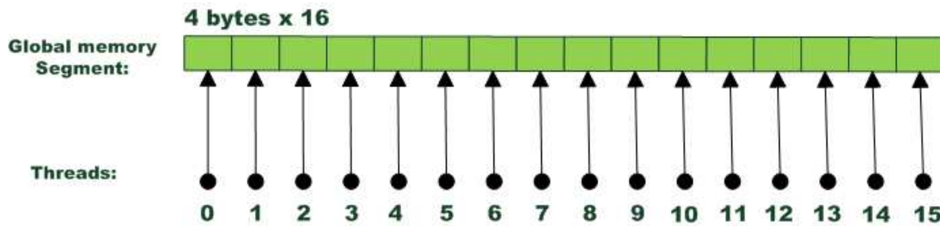


Figure 5.3: Color based accessing to memory segment

The effect of using color version images could be shown in Table 5.9 from CUDA profiler.

Table 5.9: The effects when using color images as the input

Case	Branch	Divergent branch	Instructions	Tex cache hit	Tex cache miss	Gst un-coalesced	Time (seconds)	Speedup
CUDA array	1271924	168014	18610605	1383666	24229	87808	0.06186	1.00×
Color images	1386264	179847	21282483	1205511	202349	11532 (coalesced)	0.07435	0.83×

From Table 5.9, it is easily seen that, the times of global memory writing are significantly reduced and they are coalesced. However, as the pixel size of color images is larger than that of grey scaled images, so the computation on color images is more costly than grey scaled images and also the time spent on data transferring between

host and device for color images is more than that of grey scaled images. Therefore, color images just make the global memory accessing more efficient with the increased expense of computation from other aspects. One could conclude that it is not a wise way to use color images instead of grey scaled images in our case.

Above all, the effective optimization method in this section are the use of texture cache and CUDA array, the optimization result is shown in Table 5.10.

Table 5.10: Optimization result from management of various memories on device

Case	**Time (seconds)**	**Effective Memory Bandwidth (GB/sec)**	**Step Speedup**	**Cumulative Speedup**
Kernel 1 (16x16)	0.17725	0.258	1	1
Kernel 2 (8x16)	0.15694	0.291	1.13×	1.13×
Kernel 3 (branch reduction)	0.08355	0.547	1.88×	2.12×
Kernel 4 (row based accum)	0.06886	0.664	1.21×	2.57×
Kernel 5 (CUDA array)	0.06186	0.739	1.11×	2.87×

Until now, all the optimizations for the GPU are completed. The results demonstrated that the optimization possibilities provided in the conclusion of Chapter 4 are effective except using the color version of the input images instead of grey scaled images.

5.6 Final results and Conclusion

Based the optimized GPU program, when it runs on different GPUs for the cases of different image size, one could get an overview of its performance. The results are shown in Table 5.11 and Table 5.12 respectively.

Table 5.11: Final results of the improved program - Time(P, N) (seconds)

P \ N	**384**	**512**	**640**	**1282**
1	1.23378	3.26514	6.54967	92.12514
2 (16)	0.10605	0.26352	0.65013	7.28144
14 (112)	0.01138	0.02715	0.06247	0.71293
16 (128)	0.00892	0.02726	0.05205	0.62388
30 (240-G)	0.00469	0.01214	0.02649	0.30854
30 (240-T)	0.00457	0.01183	0.02578	0.29969
GTX295 (60)	0.00485	0.01256	0.02656	0.31481
GTX480 (Fermi)	0.00288	0.00694	0.01611	0.18297

Table 5.12: Improved Speedup(P, N) = T(1, N fixed) / T(P, N fixed)

P \ N	**384**	**512**	**640**	**1282**
1	1	1	1	1
2 (16)	11	12	10	12
14 (112)	108	120	104	129
16 (128)	138	119	125	147
30 (240-G)	263	269	247	298
30 (240-T)	270	276	254	307
GTX295 (60)	254	260	246	292
GTX480 (Fermi)	428	470	406	503

The GPU based stereo matching algorithm in Table 5.11 is with the block size 11×11 and more than 50 disparity range for computing SAD which is costly, actually 5×5 block is also suitable for the images sizes less than or equal to 640×480. With 5×5 block size and 50 disparity range, the GPU based stereo matching could run much faster, for example, for 640×480 images, the GPU based program just spend **7.78 ms** which is **128.5 fps** in theory on GTX280 !

Comparing Table 5.11 with Table 4.2, there are more cases (the green data) could run in real time, CUDA program made the stereo matching algorithm, which is impossible to run in real time on CPU, run in real time on GPU. From Table 5.12, except the laptop GPU (P = 2(16)), other desktop GPUs accelerate the CPU based stereo matching algorithm for more than 100 times. Thus, the results of both Table 5.11 and Table 5.12 demonstrate the high potential of using GPU to speed up data parallel algorithm, and also demonstrate that GPU is suitable for further development of stereo vision application and is possible to make the built vision system run in real time.

In addition, from the results of Table 5.11, with GTX280 or Tesla C1060, the stereo matching is running faster than that from Point Grey stereo camera shown in Table 3.1, thus if the stereo matching algorithm used in Point Grey stereo camera is replaced with the GPU based stereo matching, the performance of the built 3D sensing part before will be improved and the GPU based 3D sensing part is shown in Figure 5.4.

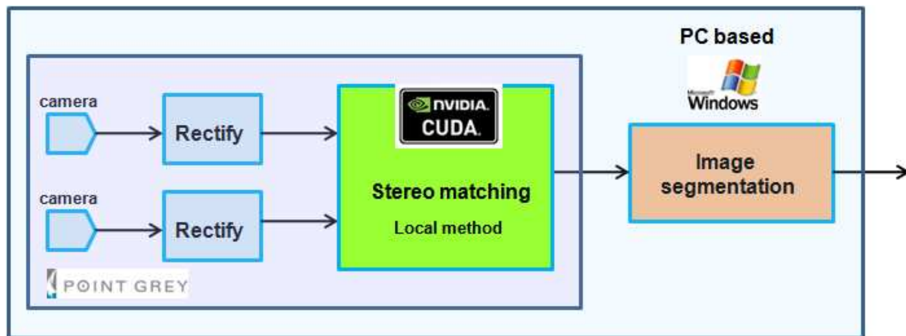


Figure 5.4: GPU based 3D sensing part

Conclusion 6

6.1 Summary

Until now, all of the work has been done. Based on the results from previous chapters, one could get an insight of the developed 3D sensing part. Actually, the efforts to the project in previous chapters are clarified as below:

From Chapter 1, a survey for the kernel of the developed 3D sensing part, which is the stereo matching algorithm for computing the depth information, is made. Through this survey, one could get an overview of the progress in stereo vision area, such as the basic concepts in stereo vision, the specific features of different methods and the state of the art stereo matching algorithm etc. Additionally, in consideration of the project context and motivation, the range (3D sensing part of the Semi-autonomous Robot Arm System) and the research goals of this Msc project were put forward. After that, the implementation of the project is carried on in the following chapters.

In Chapter 2, the prototype of 3D sensing part was implemented with COTS (commercial-off-the-shelf) components which satisfied the constraints from robot arm. Simply speaking, it was the combination of stereo matching algorithm and image segmentation algorithm. Furthermore, with importing ROI (Region Of Interest) into 3D sensing part, 3D sensing part could be applied in a more complex environment.

In Chapter 3, the prototype was profiled from speed aspect. The results showed the bottleneck of the system which needed further improvement afterwards. GPU was selected for improving the developed 3D sensing part.

In Chapter 4, through porting the stereo matching component on GPU, its performance was enhanced comparing its CPU based solution. After that, GPU based stereo matching was profiled to see whether it could be further accelerated and the results showed the possible optimization places.

Lastly, in Chapter 5, based on the results in Chapter 4, the recommendations for further optimizing GPU based stereo matching algorithm were verified. The improved GPU program had a significant performance enhance and could be used in real time for most test cases. The result demonstrated the GPU based stereo matching algorithm was faster than the original stereo matching component in 3D sensing, and also demonstrated the potential of GPU for further development of 3D sensing part. Finally, a GPU based 3D sensing part was implemented.

Totally speaking, the research goals provided in the first chapter are all achieved and the results are offered as the essential reference for future work of developing the vision system for the Semi-autonomous Robot Arm System.

6.2 General conclusion

Throughout the work, it is seen that image segmentation based on disparity map generated by usual stereo vision system can not only successfully extract the 3D information of objects on a plane, but also has high potential improvement. One kind of improvement from speed aspect is implemented in this project which is applying GPU capability of massive data parallelism. At the same time, though the implementation of GPU based stereo matching algorithm, one could see that GPU is suitable for the development of specific stereo vision system in our case.

However, we also notice that, as the GPU is a desktop platform, it consumes significant power, thus currently GPU is hard to be used in embedded application. If we want to consider bring our application to embedded system area, we should consider other alternatives for speed up the application.

6.3 Recommendations for future work

Although the primary research goals are achieved, there is still a lot of work to do through observing and analysing the results of the experiments. The recommendations for future work are mainly from two aspects as below:

1. From function aspect:

Through the observation of experiments, the quality of the disparity map, generated by currently used stereo matching algorithm, mainly depends on the parameters setting and the environment which cannot be little texture. Thus, in order to generate disparity map more reliably, it is better to use a more robust stereo matching algorithm instead of the current one, for example, it could consider using a global stereo matching algorithm.

In addition, for segmentation component, as it directly extracted the blobs after thresholding the disparity map, thus the noise information will also be extracted so that the procedure of labelling the blobs is not stable. Therefore, in order to get stable labels for the blobs, after thresholding, it is better to clean up the thresholded image first by means of morphological operation.

2. From speed aspect:

As in this master project, it just accelerated the stereo matching component with GPU, in order to make 3D sensing part faster, other component should also be accelerated. Based on the investigation, rectification component in 3D sensing could also run very fast on GPU if it uses texture memory, thus one could consider porting rectification part on GPU. In addition, for segmentation part, although it is somewhat more complex, one could profile it first to find the most time consuming part. If the most time consuming part in segmentation algorithm has data parallelism possibly, then one could accelerate this part with GPU too.

Another obvious result is that, the grabbing speed of currently used stereo camera is so slow that it could make the application run at 16Hz at most. Thus, replacing current camera with a faster one can also speed up 3D sensing part.

Lastly, coming back to currently developed GPU based basic stereo matching algorithm, in order to accelerate it further, one possible way is to run a separable box filtering based method on GPU.

With the above recommendations, 3D sensing part could be more robust and faster.

Bibliography

[1] Aaron F. Bobick and Stephen S. Intille. Large occlusion stereo. *International Journal of Computer Vision*, 33:181–200, 1999.

[2] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient belief propagation for early vision. In *In CVPR*, pages 261–268, 2004.

[3] A. Hosni, M. Bleyer, M. Gelautz, and C. Rhemann. Local stereo matching using geodesic support weights. pages 2093–2096, 2009.

[4] Kuk jin Yoon and In So Kweon. Adaptive support-weight approach for correspondence search. *IEEE Trans. PAMI*, 28:650–656, 2006.

[5] T. Kanade. Development of a video-rate stereo machine. pages I:549–557, 1994.

[6] Jae Chul Kim, Kyoung Mu Lee, Byoung Tae Choi, and Sang Uk Lee. A dense stereo matching using two-pass dynamic programming with generalized ground control points. In *Proceedings IEEE International Conference on Computer Vision and Pattern Recognition, Vol. II*, pages 1075–1082. IEEE Computer Society, 2005.

[7] David Kirk/NVIDIA and Wen mei Hwu. *Slides of Course ECE498AL*. University of Illinois in Urbana-Champaign.

[8] Andreas Klaus, Mario Sormann, and Konrad Karner. Segment-based stereo matching using belief propagation and a self-adapting dissimilarity measure. In *ICPR '06: Proceedings of the 18th International Conference on Pattern Recognition*, pages 15–18, Washington, DC, USA, 2006. IEEE Computer Society.

[9] Vladimir Kolmogorov and Ramin Zabih. Computing visual correspondence with occlusions via graph cuts. In *International Conference on Computer Vision*, pages 508–515, 2001.

[10] Annika Kuhl and Technische Universität Ilmenau. Comparison of stereo matching algorithms for mobile robots, 2004.

[11] Stefano Mattoccia. Stereo vision: algorithms and applications. April 2009.

[12] NVIDIA Corporation, Santa Clara, CA 95050. *CUDA Best Practices Guide*, February 2010. Version 3.0.

[13] NVIDIA Corporation, Santa Clara, CA 95050. *CUDA Programming Guide*, February 2010. Version 3.0.

[14] NVIDIA Corporation, Santa Clara, CA 95050. *CUDA Reference Manual*, February 2010. Version 3.0.

[15] M. Okutomi and Takeo Kanade. A locally adaptive window for signal matching. In *Proceedings of the Third International Conference on Computer Vision (ICCV '90)*, pages 190–199, December 1990.

[16] Point Grey Research Inc. *TRICLOPS Software Development Kit (SDK)*, 2003. Version 3.1.

[17] Point Grey Research Inc., 8866 Hudson Street, Vancouver, BC, Canada. *FlyCapture API Programming Reference*, October 2006. Version 1.6.

[18] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *INTERNATIONAL JOURNAL OF COMPUTER VISION*, 47(1–3):7–42, April 2002.

[19] J. Sun, H.Y. Shum, and N.N. Zheng. Stereo matching using belief propagation. page II: 510 ff., 2002.

[20] Hai Tao, Harpreet S. Sawhney, and Rakesh Kumar. A global matching framework for stereo computation. *Computer Vision, IEEE International Conference on*, 1:532, 2001.

[21] Tangfei Tao, Ja Choon Koo, and Hyouk Ryeol Choi. A fast block matching algorthim for stereo correspondence. pages 38 –41, sep. 2008.

[22] Olga Veksler. Stereo matching by compact windows via minimum ratio cycle. In *In ICCV, vol. I*, pages 540–547, 2001.

[23] Olga Veksler. Fast variable window for stereo correspondence using integral images. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, pages 556–561, 2003.

[24] Peng Wang. *OpenCL Optimization*. NVIDIA Corporation, October 2009. GPU Technology Conference.

[25] Wikipedia. Epipolar geometry. http://en.wikipedia.org/wiki/Epipolar_geometry, May 2010.

[26] Wikipedia. Least squares. http://en.wikipedia.org/wiki/Least_squares, September 2010.

[27] Wikipedia. Ransac. http://en.wikipedia.org/wiki/RANSAC, July 2010.

[28] Q.X. Yang, L. Wang, and R.G. Yang. Real-time global stereo matching using hierarchical belief propagation. page III:989, 2006.