Circuits and Systems
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ens.ewi.tudelft.nl/

CAS-2011-03

M.Sc. Thesis

High-Quality, Real-Time HD Video Stereo Matching on FPGA

Guanyu Yi B.Sc.

Abstract

Stereo matching is an important computer vision technique, which extracts the depth information of the scene by matching a pair of stereo images. It has numerous applications, such as view-point interpolation, 3DTV, object detection, etc. In the past decades, many algorithms have been proposed to improve the matching quality or to increase the speed. Due to the high computational complexity, it is still quite challenging to attain high matching-quality at real-time speed.

In this thesis, we propose a hardware design of stereo matching, which is capable of producing high-quality disparity maps at real-time speed. A high-quality stereo matching algorithm is efficiently implemented and hardware-oriented optimized, attaining huge speedup by parallel computing. The whole algorithm is implemented in a single EP3SL150 FPGA. The experimental results show that our design is capable of matching high-definition videos at real-time speed, i.e. 60 frame per second at 1024×768 resolution. In terms of matching quality, our design is among the leading real-time methods, evaluated in the Middlebury stereo benchmark. As an application of the stereo matcher, we also build up a depth-scaling system for 3DTV, working together with a view synthesis module. The SoC system synthesizes high-quality virtual views at real-time speed.

TU Delft

Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

High-Quality, Real-Time HD Video Stereo Matching on FPGA

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Microelectronics

by

Guanyu Yi B.Sc.
born in Jilin, China

This work was performed in:

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

Delft University of Technology

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 **"High-Quality, Real-Time HD Video Stereo Matching on FPGA"** by **Guanyu Yi B.Sc.** in partial fulfillment of the requirements for the degree of **Master of Science**.

Dated: 2011-August-29

Chairman: _____
prof.dr.ir. A.J. van der Veen

Advisors: _____
dr.ir. Gauthier Lafruit

_____
dr.ir. Rene van Leuken

Committee Members: _____
dr.ir. Georgi Kuzmanov

_____

Abstract

Stereo matching is an important computer vision technique, which extracts the depth information of the scene by matching a pair of stereo images. It has numerous applications, such as view-point interpolation, 3DTV, object detection, etc. In the past decades, many algorithms have been proposed to improve the matching quality or to increase the speed. Due to the high computational complexity, it is still quite challenging to attain high matching-quality at real-time speed.

In this thesis, we propose a hardware design of stereo matching, which is capable of producing high-quality disparity maps at real-time speed. A high-quality stereo matching algorithm is efficiently implemented and hardware-oriented optimized, attaining huge speedup by parallel computing. The whole algorithm is implemented in a single EP3SL150 FPGA. The experimental results show that our design is capable of matching high-definition videos at real-time speed, i.e. 60 frame per second at 1024×768 resolution. In terms of matching quality, our design is among the leading real-time methods, evaluated in the Middlebury stereo benchmark. As an application of the stereo matcher, we also build up a depth-scaling system for 3DTV, working together with a view synthesis module. The SoC system synthesizes high-quality virtual views at real-time speed.

Acknowledgments

This thesis work is performed at Imec (Leuven, Belgium) with support from the Circuit and System (CAS) group of Delft University of Technology (Delft, the Netherlands). I would like to take this opportunity to thank everyone who has helped me during my MSc thesis year.

First of all, I want to thank Dr. Gauthier Lafruit, Dr. Francesco Pessolano and Prof. Alle-Jan van der Veen for their agreement to provide and offer me the chance to perform my MSc thesis project in Imec. Their rich experience and scientific insights have led the entire search and development into the correct direction - dedicated hardware implementation of high performance dynamic programming stereo matching.

In the period of my MSc thesis year, I thank Ke Zhang for his excellently in developing the stereo matching algorithms with line-based dynamic programming and region growing. Ke also guided me into the stereo vision world and inspires me much on algorithm principles, computing data flow and hardware implementation. I thank Christine Lin in Imec-Taiwan for her advices on ASIC orientated hardware design. I thank Geert Vanmeerbeeck and Eddy De Greef for their collaboration on constructing hardware and software environments. I would like to thank Prof. Rene van Leuken for his invaluable suggestions for design optimization and improvements as well as all the hours he has spent on proofreading my thesis. I thank CK Liao and Josh Tu in Imec-Taiwan for their advices on algorithm improvements and synthesis tools advanced usage. I also would like to thank Gauthier Lafruit, Francesco Pessolano and Peter Lemmens for their outstanding management.

Special thanks are given to Gauthier Lafruit, Prof. Tian-Sheuan Chang and Hsiu-Chi Yeh. Without their support and contributions out 3D depth range adjustment system cannot work properly. Gauthier generates the idea of this system according to the 3D TV trend, and his ambition and enthusiasm is exactly the engine and power of the entire research and development. Gauthier guides, trusts and supports me no matter on the thesis work and life, and I faithfully appreciate his help offered over the whole year. Prof. Chang has proposed the view synthesis kernel and its implementation which is another key kernel in our system, and Hsiu-Chi Yeh has implemented the DDR2 SDRAM access scheduler which plays an important role in accessing DDR2 SDRAM with zero latency for frame-based storage. I am grateful to their collaboration on building our system.

The study and research in Delft and Leuven in the current two years is like a piece of music, which is composed by the melodies in TU Delft and Imec. The notes of international talented people make the melodies graceful, and the rhythms of my around friends make the melodies vivid. My wife and parents is exactly the main theme to make the melodies continue. The music will not be complete without anyone, and I appreciate encountering them in my life.

Finally I thank all my thesis defense committee members including Prof. Alle-Jan van der Veen, Prof. Rene van Leuken, Dr. Gauthier Lafruit and Prof. Georgi Kuzmanov.

Guanyu Yi B.Sc.
Delft, The Netherlands
2011-August-29

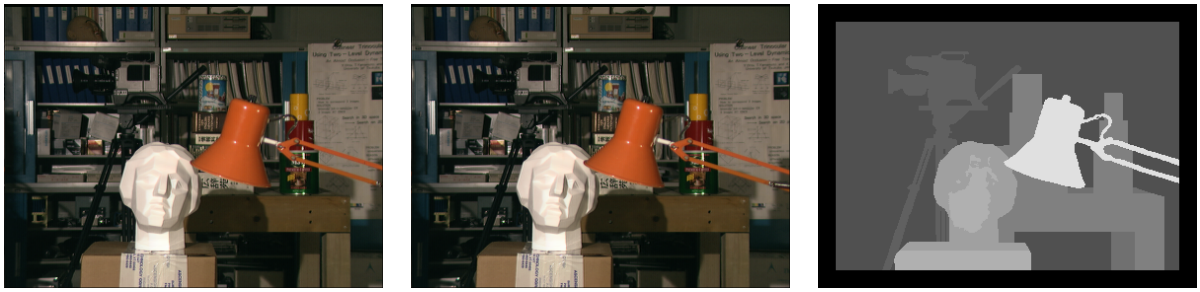
Contents

List of Figures

List of Tables

1 Introduction

Stereo matching has traditionally been one of the most attractive research topics in computer vision domain, and is still being paid attention to from many researchers. The main function of stereo matching is to obtain depth information from a pair of images or videos whose corresponding pixels have displacement, that varies according to the depth of objects from the camera pair. The displacement in number of pixels is called Disparity. Normally disparity spans a certain range, called disparity range, which is an important parameter of any stereo matcher. The calculated depth map is often displayed as gray-scale intensities, as shown in Figure 1.1. Stereo matching extracts



(a) Left camera image (b) Right camera image (c) Ground left truth disparity map

Figure 1.1: Tsukuba disparity map

depth information for each pixel, and actually that is the 3D information from image or video pair. Therefore, the depth information can be further used in 3D applications, such as HD-3DTV, 3D reconstruction, virtual reality and so on.

A standard to evaluate and compare the results of different stereo matching algorithms is Middlebury stereo benchmark evaluation[38]. In this website evaluation, the generated depth maps is compared with the ground truth depth maps. The not identical depth map pixel values in the comparison will be considered as incorrect matched pixel, therefore the average error rate of the entire depth map are produced. The pictures shown in Figure 1.1 are the Tsukuba standard image pair and truth disparity map from the Middlebury stereo correspondence algorithms evaluation website.

1.1 Motivation

The main motivation of this thesis research and design is to implement real-time stereo matching on HD-3DTV with an FPGA prototype to pave the way for an ASIC implementation.

3DTV is an exciting technology developed since the late-1890's, which is now leading a new trend in the consumer household entertainment. The main difference between

3DTV (stereoscopic TV) and 2DTV (regular TV) is that the former uses depth information, thus the image will appear to consist of solid 3D objects and persons located in 3D space. The TV then looks like a true window to the real world and gives the audience a new experience of entertainment.

3DTV is becoming more and more popular, and it is estimated that by the end of 2015 there will be 22.2 million homes across 53 countries watching 3D programming. Therefore people will take more and more time on watching 3DTV. Then the problem comes: is it harmful to people who watches 3D programs for a long time?

When looking to 2D programs people's eyes are focused on the subject, while the visual cortex of the brain analyzes the retinal disparities and then fuses the 2D images into 3D, as shown in Figure 1.2.

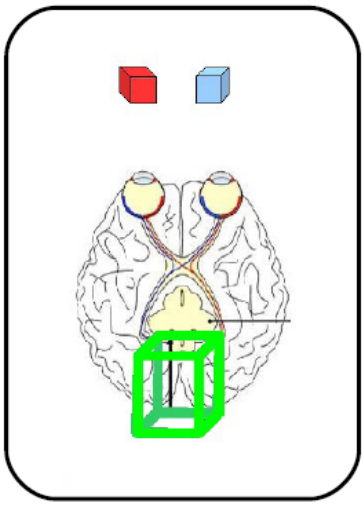


Figure 1.2: Eyes see in 2D

When looking to 3D programs people's eye-sight are parallel with 2.5 inch apart, which is shown in Figure 1.3. However, when objects are even further away with having left and right images too far apart, it will make the eyes diverging, which will cause eye strain and nausea. There are also other disparities that may cause eye strain and headache, for example, vertical disparity of the stereoscopic images of an object might cause one eye to track upward and the other downward, which probably is caused by imperfect lenses of dual cameras, as shown in Figure 1.4.

The eye strain and headache happens due to the mismatch between the 3D-display (eye focus) and real object locations in watching 3D video. Therefore, Imec proposed 'Personalized 3D Depth Range Adjustment', which is a nice solution to the problem, as shown in Figure 1.5. With this solution, users can easily modify the 3D depth range using a remote controller to satisfy personal visual comfort, furthermore, reduce eye strain headache and nausea.

Moreover, the most comfortable depth is also relative with the screen size, viewing distance and display type, which also makes it necessary to modify the 3D depth range according to the viewers' personalized visual comfort.

The proposed personalized 3D depth range adjustment system totally contains suc-

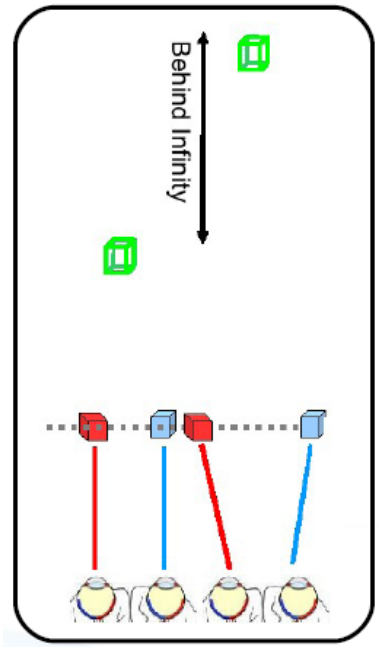


Figure 1.3: Eyes see in 3D

Figure 1.4: Dual cameras

cessively image acquisition, camera calibration, image rectification, stereo matching, DDR interface construction, view interpolation and video display. This thesis work focuses on the stereo matching part, which is the major processing bottleneck discussed in the following section.

1.2 Target of Stereo Matching

In the stereo videos, there should be one pixel in the right frame corresponding to a pixel in the left frame, which is the basis of stereo matching. The main task of stereo matching is to find the corresponding pixels (stereo pair). However, the stereo pairs in raw stereo videos taken from stereo cameras are often not on the same specific pixel line, as shown in Figure 1.6, which means that under this condition it has to complete a 2D search. Consequently, image rectification technology is imported to reduce the 2D search into 1D search, as shown in Figure 1.7, meaning that *Epipolar Geometry* is applied onto the stereo frames to perform image rectification, where the stereo pairs

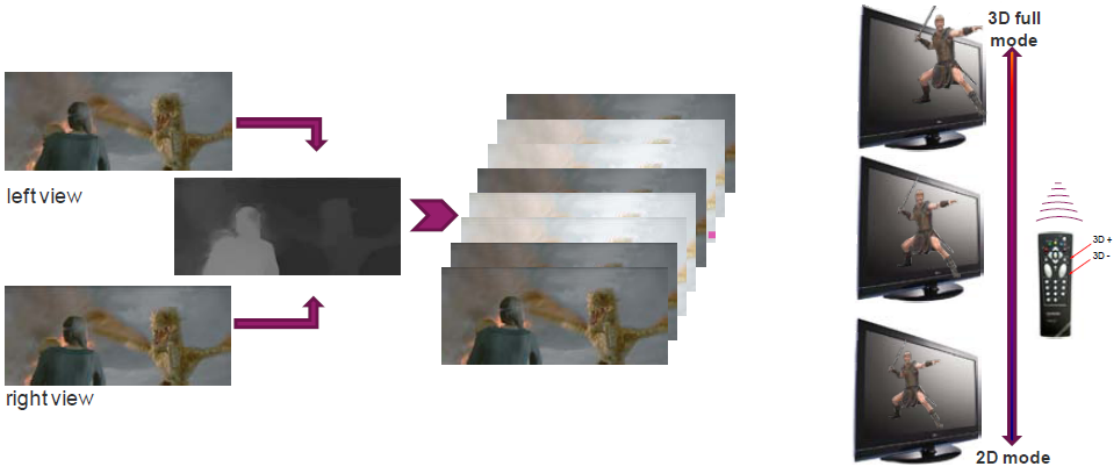


Figure 1.5: Imec proposed personalized 3D depth range adjustment

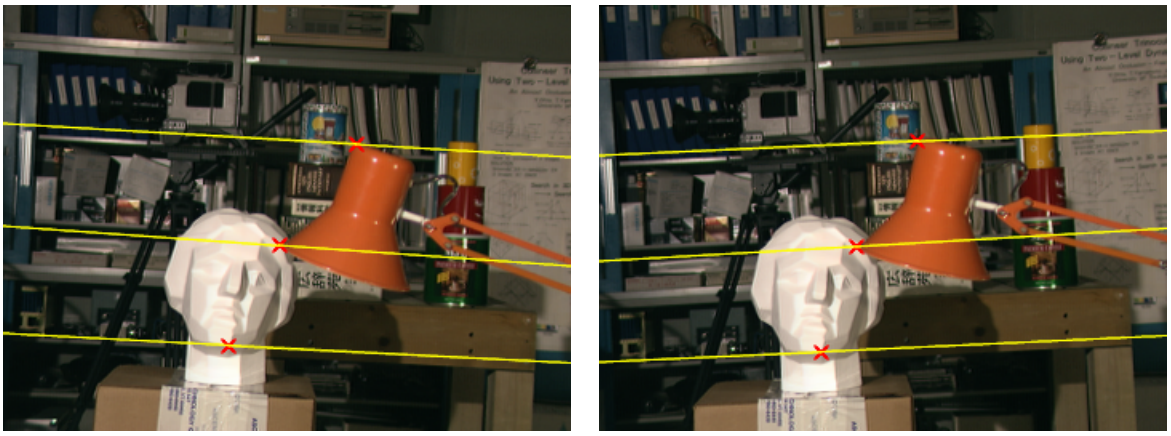


Figure 1.6: Not rectified frames and not aligned corresponding lines

can be matched on one specific line, called epipolar line.

In Epipolar Geometry, some essential concepts in stereo matching are shown in Figure 1.8. The object point P corresponds to the pixel p from the left frame and the pixel p' from the right frame. O_t is the target optical center (left camera) and O_r is the reference optical center (right camera). The distance between the two optical centers is defined as the *Baseline* represented by B, which is parallel with by epipolar lines, and the length f from frame plane to B is the focal distance. In the case shown in Figure 1.8, the distance from P to B is defined as the depth Z which is the actual distance between cameras and object point, and the distance from p to p' is defined as the disparity $d = (X_t - X_r)$, which is the displacement of one stereo pair's locations in the stereo frames. According to Figure 1.8 and considering the similar triangles (PO_rO_t and Ppp'), we obtain the relationship between depth Z and disparity d:

$$\frac{B}{Z} = \frac{(B + X_r) - X_t}{Z - f} \quad \Rightarrow \quad Z = \frac{B \cdot f}{X_t - X_r} = \frac{B \cdot f}{d} \tag{1.1}$$

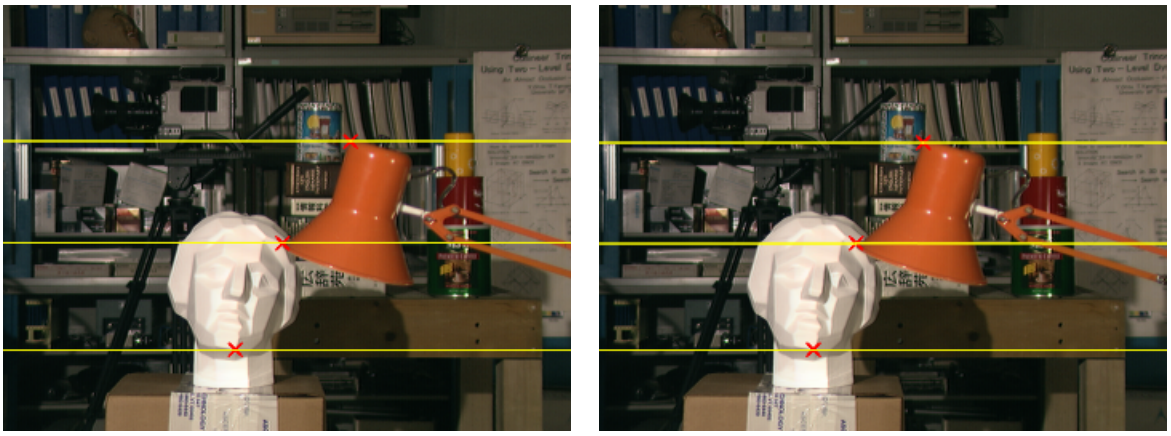


Figure 1.7: Rectified frames and epipolar lines

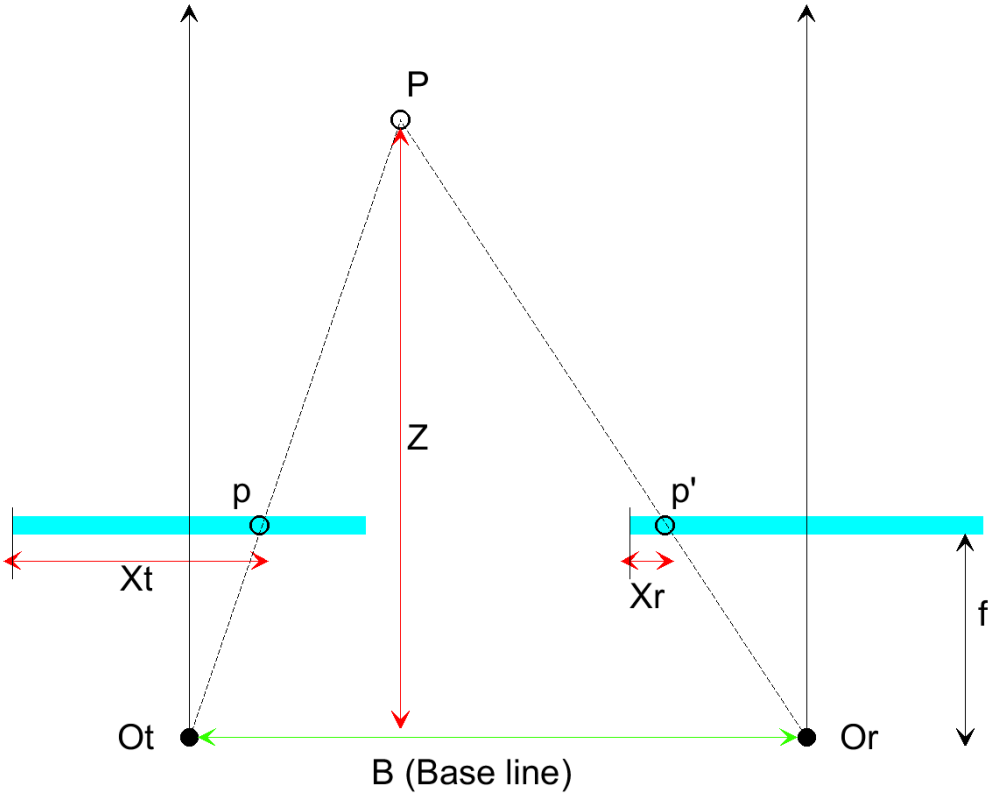


Figure 1.8: Basic concepts of stereo matching in epipolar geometry

The equation above basically demonstrates that the more deep the object lies, the less disparity it has between the stereo pair in the input videos, and this principle is also the basis of stereo matching.

Image rectification is a very important procedure before stereo matching, because of its reduction on correspondence searching complexity from 2D into 1D. Since this

thesis work is mainly about the subject of stereo matching, the technology of image rectification will not be discussed further, and in the following discussion we will take for granted all the input stereo videos has already been rectified according to epipolar geometry.

1.3 Problem Definition

Last section lists the processing stages, in which stereo matching is the most time consuming part. Recently, only a marginal number of implementations could meet satisfactory quality with the constraint of real-time application, because of the high computational complexity of the stereo matching. For example, a three camera viewpoint interpolation prototype system[32] proposed by Sharp Corporation in 2009 could only meet a couple of frames per second for low-resolution images. The bottleneck of stereo matching has to be solved for real-time and high-quality interpolation results, particularly for our 3D depth range adjustment implementation, which should run at high frame rate and high definition - $1920\times1080@60FPS$.

Nowadays, a number of platforms such as multi-core CPU, GPU, DSP, FPGA and ASIC could be utilized for stereo matching design and implementation. At the same time, a lot of algorithms corresponding to each platforms have also been developed, however, almost none of them could satisfy both real-time and high-quality requirements. For instance, an implementation proposed by Zhang et al.[45] on GeForce8800 GTX GPU in 2009 meets high quality with an averaged Middlebury benchmark error rate 7.65%, but only reaches 12 frames per second with 450×375 resolution. In contrast, an FPGA implementation proposed by Jin et al.[20] in 2010 achieves 230 frames per second with 640×480 resolution, however, the accuracy reduces too much with averaged Middlebury benchmark error rate of 17.24%.

1.4 Solutions and Contributions

We propose a hardware friendly high-accuracy dynamic programming algorithm and a complete 3D depth range adjustment real-time system to meet the requirements. The high-accuracy algorithm is based on *Accurate and Efficient Stereo Processing by Semi-Global Matching and Mutual Information* proposed by Heiko Hirschmuller[16] and *Cross-Based Local Stereo Matching Using Orthogonal Integral Images* proposed by Zhang et al.[44] The former paper provides a basic dynamic programming method for stereo matching calculation, while the later paper offers a consistency check and refine method for post processing. We merge them together to create a new *Scan-line Optimization Dynamic Programming* algorithm for stereo matching. The whole design of this prototyping system, including image data synchronization, stereo matching, DDR interface, view interpolation and video display, is implemented on a single EP3SL150 FPGA from Altera.

Compared with Lu's implementation[47] which only uses a cross-based local stereo matching algorithm without dynamic programming on the same FPGA, our new implementation improved the averaged Middlebury benchmark error rate from 8.2% to

6.6%. Moreover, our new implementation supports high-definition video, which means it can support the resolution of 1920×1080@60FPS instead of 1024×768@60FPS in Lu's implementation.

We make the following main contributions, separated into two aspects, releasing a new algorithm and implementing the system on FPGA:

releasing a new algorithm that:

- Preserves high stereo matching accuracy with 6.6% averaged Middlebury benchmark error rate.
- Modifies *Mini-Census Transform* proposed by Chang et al.[4] to improve robustness near the object edges.
- Generates more smooth boundary depth maps due to scan-line optimization dynamic programming.
- Creates consistent real-time video using timing consistency over successive frames.

Implementing the algorithm on FPGA that:

- Builds real-time stereo matching system to generate high-definition video on the resolutions of 1024×768@60FPS and 1920×1080@60FPS.
- Constructs a parallel and pipelined hardware architecture to meet the requirements of timing and resource usage.
- Makes the system structure independent on the disparity range. We have implemented on the FPGA EP3SL150 with maximum disparity range of 16, 32 and 64.
- Discarding the avalon interface used in Lu's implementation[47] to prepare a stereo matching core without industry IP for ASIC platform.
- Provides a complete system which could use the imported stereo videos to generate depth map videos (results from stereo matcher) or anaglyph videos (results from viewpoint interpolater) on standard DVI/HDMI monitors.

We utilize the DE3 board developed by Terasic to implement the whole system design. The board includes the EP3SL150 FPGA and a lot of peripherals, such as DDR2 SDRAM, USB and SD card. Two DVI extension boards are used to receive the stereo videos and transfer output videos to the monitor. In the setup system shown in Figure 1.9, the JTAG link is used to burn the configuration file compiled from PC into FPGA; DDR2 link is used to connect DDR2 SDRAM to save frames for timing consistency and viewpoint interpolating; HSTC interface is used to connect DVI extension boards for videos in and out. The main procedure is first configuring FPGA, and then the stereo videos comes into FPGA by dual link cable and two DVI extension boards, after that the output videos processed by FPGA goes out into the monitor from one DVI extension board.

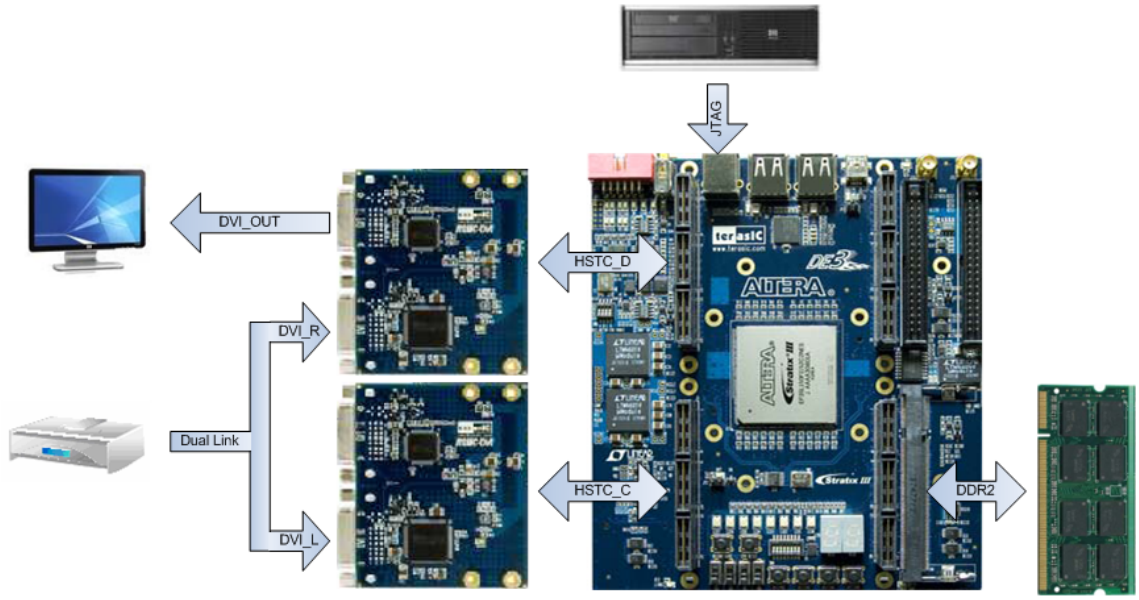


Figure 1.9: Complete setup system with FPGA on DE3 board

1.5 Overview of Chapters

In this thesis work, Chapter 1 gives a brief introduction about our problem, solution and contribution. Chapter 2 introduce the stereo matching instances and our algorithm. Chapter 3 presents a dynamic programming algorithm and corresponding hardware modules design using a top-down approach. Chapter 4 evaluates the hardware algorithm and stereo matcher design, and compares our results with other state-of-the-art related implementations. Chapter 5 presents the complete real-time system design using stereo matcher as a block on FPGA and introduces other important kernels, such as view synthesis and DDR2 SDRAM access scheduler. Chapter 6, as the final chapter, gives the summary and conclusion of our thesis work and provides relative suggestions and developments for the future design and application.

2 Stereo Matching Algorithm

Approximately 20 years ago, stereo matching algorithms began to fast develop, and nearly all these different algorithms have been generally evaluated by Scharstein and Szeliski[35], who classified those algorithms into two main kinds: local and global algorithms. Local algorithm is the short name from local area based stereo matching algorithms, which emphasizes the cost of every pixel is aggregated from a local area formed by its neighboring pixels. While global algorithm stands for global optimization based stereo matching algorithm, which emphasizes the cost of every pixel is extracted from a frame or a line by some minimum cost selection method.These two kinds of algorithms are both robust for they both generate the cost by referring to other pixels, but compared with local algorithms, global algorithms often take more computational complexity and more storage resources.

We take the advantages of both local and global algorithms to generate our own algorithm in this thesis work. It is efficient for us to use local algorithms to deal with consistency check and refine processing which is less important than the stereo matching core part but has almost the same computational complexity as the core, while it is accurate for us to utilize global algorithms to perform stereo matching core part (dynamic programming part) to provide as precise results as possible. We can hence take the advantage from local algorithm to reduce computational complexity and save storage resources, and from global algorithm to obtain accurate stereo matching results.

Section 2.1 categories the related existing instances according to various hardware platforms. Section 2.2 presents the our stereo matching algorithm, including computation flow, cross-based local algorithm and scan-line optimization dynamic programming global algorithm. Section 2.3 lists the targets of our stereo matching design and implementation on FPGA, and introduces briefly the benefits of FPGA platform and implementation.

2.1 Related Existing Stereo Matching Instances

As mentioned before, the challenge of stereo matching is to solve the complex computation while keeping the highly accurate results. Based on such challenge a lot of methods approaching real-time and efficient stereo matching computation have been developed in the past two decades, while recently, some of them have obvious achievements towards either computational speed or matching accuracy. They will be shown in the following paragraphs by categories of different implementation platforms.

CPU implementation is the first prototyping platform for different algorithms developed, and its main disadvantage is the processing speed, but till now there are a lot of aggregation methods being able to accelerate its processing speed. An efficient segmentation-based cost aggregation strategy for local algorithm was proposed

by Tombari et al.[37], which achieves 384×288@5FPS with a disparity range 16 ($D_{max} = 16$) on an Intel Core Duo 2.14GHz; and it only reaches 450×375@1.67FPS with $D_{max} = 60$. A dynamic programming algorithm proposed by Salmen et al.[34] achieves 384×288@5FPS with $D_{max} = 16$ on a 1.8GHz CPU platform. Later a method of combining a multi-level adaptive technique with a multi-grid approach proposed by Kosov et al.[24] achieves 450×375@3.5FPS with $D_{max} = 60$. Another local algorithm of variable cross algorithm and orthogonal integral image technique for accelerating the aggregation over irregularly shaped regions proposed by Zhang et al.[44] achieves 384×288@7.14FPS with $D_{max} = 16$ and 450×375@1.21FPS with $D_{max} = 60$. Although the processing speed performance of CPU is pretty bad, it has the advantage of very good matching accuracy, which is often less than 10% average error rates.

GPU implementation is another platform for various algorithms development. Yang et al.[42] proposed a global optimizing real-time algorithm (hierarchical belief propagation) on a GeForce 7900 GTX GPU, which achieves 320×240@16FPS with $D_{max} = 16$; and its average error rate is 7.69%. Wang et al.[40] proposed an algorithm based on adaptive aggregation and dynamic programming, achieving 320×240@43FPS with $D_{max} = 16$ on an ATI Radeon XL1800 GPU; but its average error rate is increased a little to 9.82% compared with Yang's work. Zhang et al.[46] implemented a variable cross algorithm and an orthogonal integral image technique on GeForce GTX8800 GPU, which reached 384×288@100.9FPS with $D_{max} = 16$ and 450×375@12FPS with $D_{max} = 60$ and error rate 7.60%. In current years, Humenberger at el.[17] improved the Census Transform on GeForce GTX 280 GPU and reached 450×375@105.4FPS with $D_{max} = 60$ and error rate 9.05%. GPU implementations perform well till now, but it is not the suitable solution for highly integrated embedded systems because of the GPU's high power and memory consumption.

DSP implementation is also investigated for applying stereo matching algorithms. Chang et al.[3] propose a 4×5 jigsaw matching template and parallel processing method to improve stereo matching performance on a VLIW DSP, which reaches 384×288@50FPS with $D_{max} = 16$ and 450×375@9.1FPS with $D_{max} = 60$, however, the average error rate is high to above 20%.

The implementation types mentioned above are all software based platform, so that the computing logic and data paths related with hardware resource and utilization cannot be changed and configured, which means they do not have the potential for optimizing the existing algorithms. Moreover, these kinds of implementations all require very high clock frequency and data throughput. Compared with these obvious disadvantages, dedicated hardware design is taken into consideration for performance revolution recently. A high performance stereo matching algorithm with mini-census transform and adaptive support weight, as well as its corresponding real-time VLSI architecture was proposed by Chang et al.[4], which is implemented with UMC 90nm ASIC and achieves 352×288@42FPS with $D_{max} = 64$. Jin et al.[20] designed a complete pipeline hardware architecture with census transform and sum of hamming distances, which achieves 640×480@230FPS with $D_{max} = 64$ and average error rate 17.24%. Seen from these two hardware implementations, their bottleneck is the on-chip memories which limits the image resolutions, although they have notable performance. Zhang et al.[47] implemented variable cross algorithm on FPGA, which achieves 1024×768@60FPS

with $D_{max} = 64$ and average error rate 8.2%. Zhang's work is excellent on both computational speed and matching accuracy aspects, but the algorithm he uses is a local algorithm, therefore it still has space for matching accuracy improvement. We list and compare all of the above implementations with ours in Chapter 4.

2.2 Our Stereo Matching Algorithm

Most stereo matching algorithms generally perform the following steps or a subset thereof, according to the taxonomy proposed by Scharstein and Szeliski[35]:

1. Matching cost computation
2. Cost aggregation
3. Disparity computation/optimization
4. Disparity refinement

For local algorithms, they often follow the steps as 1→2→3→4 with a simple Winner Takes All (WTA) strategy. However, for global algorithms, they often follow the steps as 1(→2)→3→4 to skip step 2 with global or semi-global reasoning. In our algorithm, dynamic programming is a kind of global one, so we will omit step 2 and describe other steps in the following subsections.

2.2.1 Matching Cost Computation

As introduced in the previous sections, the stereo pairs are matched in 1D along the epipolar line. In fundamental stereo matching cost computation method, the corresponding points in the reference frame to the points in the target frame are searched pixel by pixel along the epiploar horizontal line within a certain disparity range, as shown in Figure 2.1. In this fundamental searching method, the disparity range is

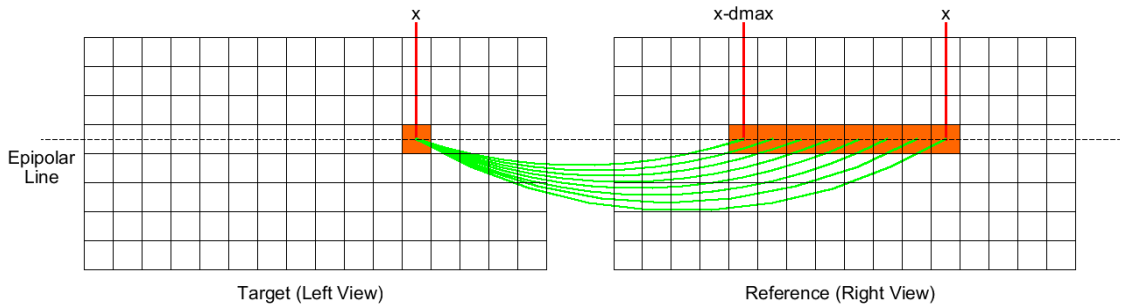


Figure 2.1: Fundamental stereo matching computation method

$[0 \dots d_{max}]$, while $d_{max} = 7$ and total disaprity levels $D_{max} = 8$. According to Equation 1.1, the disparity range d depends on the depth Z, the length of baseline B and the focal length f. For a stereo pair, estimating the matching cost is defined as evaluating the probability of each match in the disparity range, which means the matching cost is

inverse to the probability. In the fundamental pixel by pixel method, the matching cost could be defined as *Absolute difference*, *Squared Difference*, *Hamming Distance* and so on, no matter the frames are gray or RGB colored.

The absolute difference matching cost is measured by Equation 2.1;

$$Cost(x, y, d) = |I(x, y) - I'(x - d, y)| \tag{2.1}$$

The squared difference matching cost is defined by Equation 2.2;

$$Cost(x, y, d) = (I(x, y) - I'(x - d, y))^2 \tag{2.2}$$

In the equation above, I refers to the gray-scale intensity of a pixel from the target frame, while I' from the reference frame, and they have the relationship with the disparity d as the equation shows. In the basic example, the stereo pairs are determined by the minimum cost value as the $Winner$ in WTA strategy, and the corresponding d is the chosen disparity of the pixel $p(x, y)$ in the target frame.

The hamming distance matching cost is what we use in our algorithm, which is normally obtained from converted frames. The conversion procedure is performed by converting the gray-scale intensity of each pixel into some kind of bit vectors. In our algorithm we modify the $Mini - CensusTransform$ proposed by Chang et al.[4] to create our own Census Transform to do the conversion, and more detailed of the converting algorithm are introduced in Chapter 3.

2.2.2 Disparity Computation/Optimization

This step is to search the best disparity assignment which minimizes a cost function over the whole or subset of the whole stereo pair. We use a kind of semi-global algorithm - scan-line optimization dynamic programming - to perform this procedure. This algorithm searches for the best path within the *Disparity Space Image (DSI)* over the stereo pair in one line using the pixel-based matching cost, as shown in Figure 2.2.

The parameter $\{p_j | j \in [0, W-1]\}$ is defined as the pixels in one row of the target frame, where W is the image width, and in this example $W = 16$. The chosen disparity $d(p_j)$ is in a range of $[0, D-1]$, and $D = 8$ in this example. The matching cost of p_j at each disparity d is $C(j, d)$, therefore, we obtain a $W \times D$ cost matrix, as shown in Figure 2.2. The purpose of dynamic programming is to find an *optimal path* in the matrix with following two stages.

First stage is updating the matrix from left to right ($j = 1$ to $j = W - 1$) in one line according to Equation 2.3:

$$C(j, d) = C(j, d) + \min_{d' \in [0, D-1]} \{C(j-1, d') + S(|d - d'|)\} \tag{2.3}$$

The cost term $C(j, d)$ is to measure how well the assignment fits to the stereo pair, while the smoothness term $S(|d - d'|)$ is to penalize the disparity variations and large variations are only allowed at depth borders, where d' is the disparity of the previous pixel. In the detailed case, the smoothness cost $S(|d - d'|)$ has different computational equations depending on different models. For the Potts model, the smoothness cost is

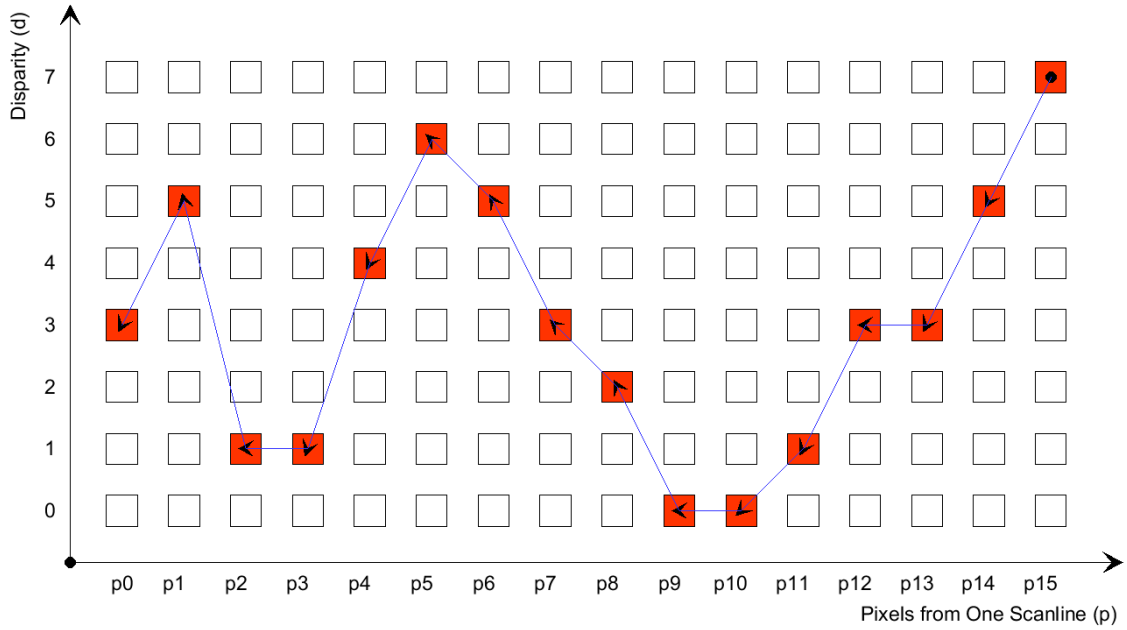


Figure 2.2: Scan-line optimization dynamic programming

computed as Equation 2.4:

$$S(|d-d'|) = \begin{cases} 0, & d = d' \\ C, & d \neq d' \end{cases} \tag{2.4}$$

We see from Equation 2.4 that the smoothness cost of the Potts model depends only on whether the disparity from the previous pixel is the same as the disparity from the current pixel, so that the smoothness cost only has two possible values. For linear model the smoothness cost is shown in Equation 2.5:

$$S(|d-d'|) = \alpha|d-d'| \tag{2.5}$$

We see that smoothness cost of the linear model depends on the difference between the disparity from the current pixel and each disparity from the previous pixel, so that the smoothness cost has D possible values, which is much more complex than the Potts model on implementation especially for a parallel structure. The main target for this first stage is utilizing Equation 2.3 to establish links between (j, d) and $(j-1, d^*)$, where d^* is the chosen minimum d'.

Second stage is to track the so-obtained matrix from right to left to get the optimized path. First, we obtain the entry point at the end of each line $(W-1, d(p_{W-1}))$ according to Equation 2.6:

$$d(p_{W-1}) = \arg \min_{d' \in [0, D-1]} C(W-1, d') \tag{2.6}$$

Starting from the chosen disparity of the entry point we traverse backward (from $j =$

$W-1$ to $j=0$) along the path built in the first stage, assigning a disparity at each pixel, as shown in Figure 2.2. Then we obtain the disparity of every pixel and further the whole depth map.

The computational complexity of the first stage is $O(D^2)$ utilizing a parallel structure according to Equation 2.4. Computational complexity directly affects hardware resource usage, especially logic combinational circuit usage, so it is better to reduce it by some optimization method. There is an envelope algorithm proposed by Pedro F. Felzenszwalb et al.[11] to reduce the computational complexity from $O(D^2)$ to $O(D)$, however, this algorithm is computed in a serial way, which means we have to adopt a serial structure to implement it. Considering our main bottleneck of implementation is the calculation speed not the hardware resource, it has more benefit for us to choose the parallel structure in the trade-off between hardware computational speed and hardware resource usage.

2.2.3 Disparity Refinement

There is a limitation about disparity estimation, because there may not always exist correspondences for every pixel in the target frame, such as *Half-Occlusion* pixels shown in Figure 2.3. In Figure 2.3, the pixels of the camera legs in target frame of Tsukuba

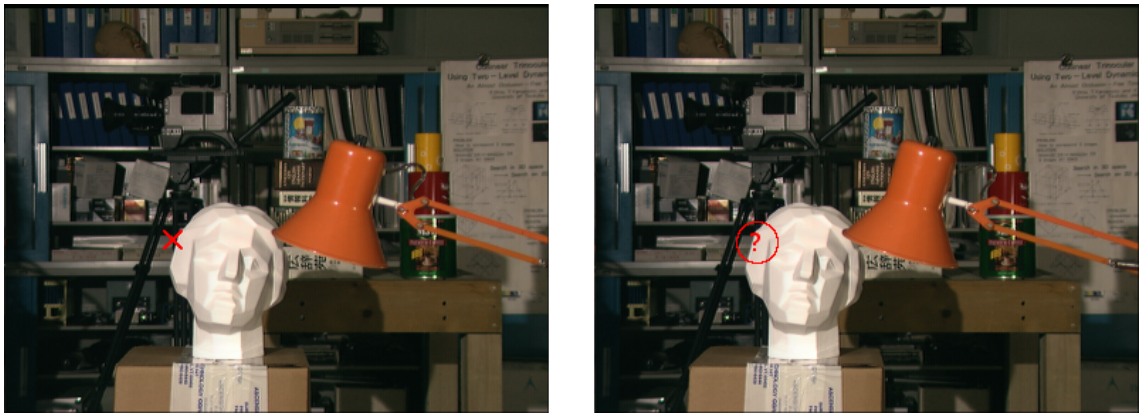


Figure 2.3: Half-occlusion point in tsukuba

have no corresponding pixels in reference frame, because they are occluded by some front objects, and we called them half-occlusion pixels. In this case, *Consistency Check* procedure is often utilized to check these areas and make up them consistent. Before consistency check, it is necessary to make sure the depth map from the reference frame is also generated by the stereo matcher, then we could perform a target frame consistency check (correcting left depth map with right depth map) and a reference frame consistency check (correcting right depth map with left depth map) to fix detected mismatches by surface fitting or distributing neighboring valid disparity estimates[35].

After the consistency check, the disparity map is not nice enough for use, therefore some additional techniques are utilized to improve the reliability and accuracy of disparity map, such as *Sub-Pixel Estimations*[36] or *Disparity Voting*[28]. We choose disparity voting in a support region for our system, and the support region is based

on luminance difference before dynamic programming, which we introduce in detail in Chapter 3. Later after voting, it often has a final procedure of median filter to reduce the salt and pepper noise and make the disparity map more smooth.

Our dynamic programming algorithm can generate the high-quality of depth maps with the trade-off of increasing the computational complexity, however the computational complexity can be implemented and optimized on FPGA platform, therefore it is possible to create a real-time and high-quality stereo matcher on FPGA using our algorithm, and we have successfully performed it, whose evaluation is shown in Chapter 4.

2.3 Design Targets

For our total system has both real-time and accuracy requirements, we need to consider several aspects to suit our design targets on FPGA platform:

1. Real-time implementations

   Our design should at least process 60 frames per second with configurable resolutions including high-definition resolution.

2. Accurate matching

   Our design should achieve less than 7% average error rate

3. Implementation complexity

   Our implementation should utilize only basic arithmetic blocks such as adders and comparators, and not employ complex blocks such as multipliers and dividers.

4. Memory usage

   Our FPGA prototyping implementation should reduce the memory utilization as low as possible for aiming at further ASIC implementation, and in detail the usage should not be over 50% of total memory on FPGA.

5. Implementation without IP

   Our architecture should be built on transparency models without IP for aiming at ASIC implementation and customers.

The advantage of FPGA or ASIC platform is the excellent flexibility in architecture design which decides the logic and data path. Therefore it provides the potential optimization and possible most efficient design environment. According to the related research and investigation, it is the customized hardware design that could lead us to the most suitable approach for our dynamic programming algorithm and parallel pipelined architecture. We choose the FPGA prototyping platform because of its highly configurable properties and great hardware resources for our huge parallel structure. Our FPGA board is Terasic DE3, which contains an Altera EP3SL150 FPGA core chip and a lot of peripherals such as JTAG UART port, DDR2-SDRAM, DVI extension borad, usb port, SD card port etc.

Stereo Matching Algorithm Hardware-Oriented Optimization and Implementation

3

This chapter discusses in detail the algorithm we adopt for stereo matching, which is a mixture of a global[16] and a local algorithm[44]. The global one is applied on dynamic programming part, while the local one is utilized on refinement part. We do some modification on our algorithm to make it more hardware friendly, such as modifying census transform, changing architecture, reconfiguring parameters etc. For our whole hardware architecture, we take the pipeline structure to obtain the best data processing throughput across all the blocks under the pixel rate clock frequency, which means in such clock frequency there is one pixel going into the stereo matcher while there is one depth result coming out of the stereo matcher every clock cycle. That whether the processing is in scan-line order or not depends on our hardware critical path, which we discuss in detail in later sections.

We will discuss our algorithm modification and hardware architecture design in a top-down approach. Section 3.1 introduces the complete parallel and pipeline data processing structure of our stereo matcher. Section 3.2 introduces the algorithm modification of the hardware model compared with the software model. Section 3.3 introduces our design of stereo matcher function block one by one.

3.1 Parallel and Pipeline Architecture of Stereo Matching

The basic stereo matching processing is a nesting loop on pixels and disparities, which is shown in Figure 3.1. Obviously the disparity loop of computing costs is the bottleneck of an effective computation, and it is also this loop that makes stereo matching very slow, especially when the stereo videos have large disparities. However, the sequential data flow has advantages: on one hand it saves hardware resources because every disparity of each pixel has to reuse the same resources, on the other hand it is a friendly structure for algorithm optimization and data compression. Consequently, such sequential structure is widely used in software based platform for not real-time application.

The content is different in configurable hardware platforms, rich in hardware resources and flexible architecture building, so that it is not necessary to reuse the hardware resources as the sequential structure does. We hence unroll the computational loop and adopt a parallel architecture as shown in Figure 3.2. Seen from the parallel structure, we unroll the disparity loop for each pixel and compute the corresponding costs independently at the same time in parallel to speed up our total computation procedure. The number of parallel paths is the maximum disparity, which is unlimited in the algorithm, but limited by the total hardware resources on FPGA board.

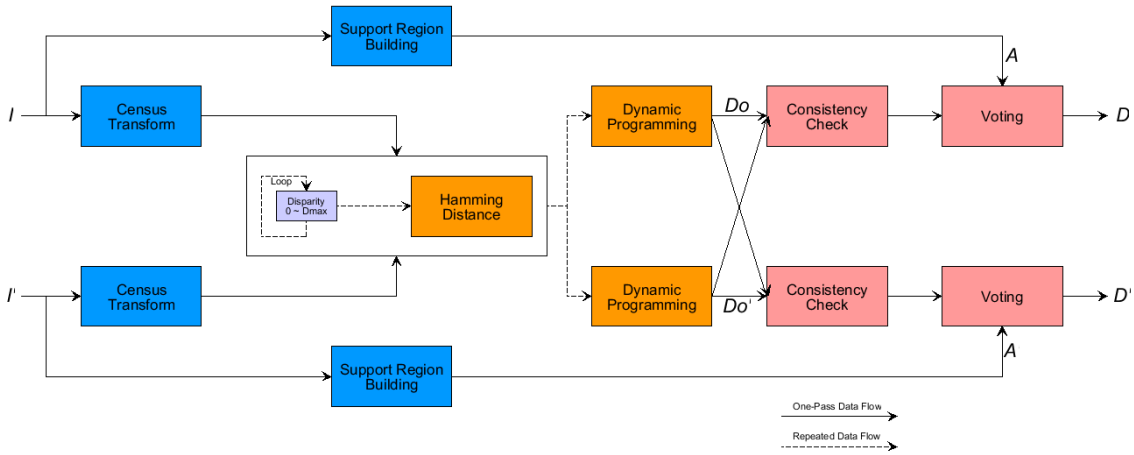


Figure 3.1: Sequential stereo matching data flow

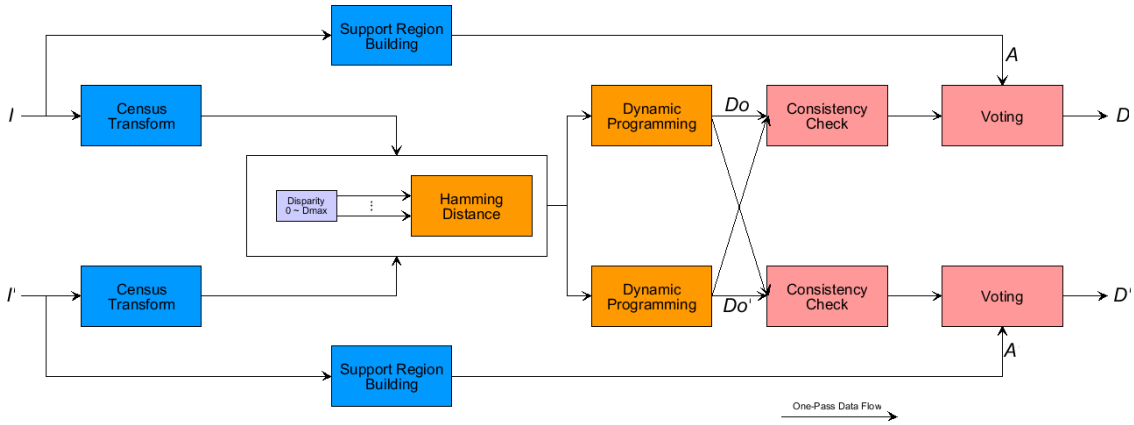


Figure 3.2: Parallel stereo matching data flow

The structure shown is only for stereo matching part, and actually the total system is more complex including synchronization, RGB-YUV converter, stereo matcher, DDR scheduler, viewpoint interpolater, anaglyph creator, etc., which is discussed in Chapter 5. The total system is also used for stereo matching verification, and we are assuming here the other parts work fine to provide an ideal environment of pipeline processing. Under such conditions, the functions of stereo matcher is shown in Figure 3.3. Take Tsukuba as an example, the rectified stereo gray-scale frames go into the stereo matcher, and the data width of pixels from left and right frames are 16 bits. After median filter, support region builder and census transform, the 84 bits data of transformed census vectors and support region arms from left and right frames are passed to reorder for further processing. Reorder, raw cost scatter, dynamic programming, post reorder and disparity output logic are main function blocks of dynamic programming; they work out the depth map together using scheduled dynamic programming algorithm which is hardware and resource usage friendly. The 64 bits data generated by the scheduled

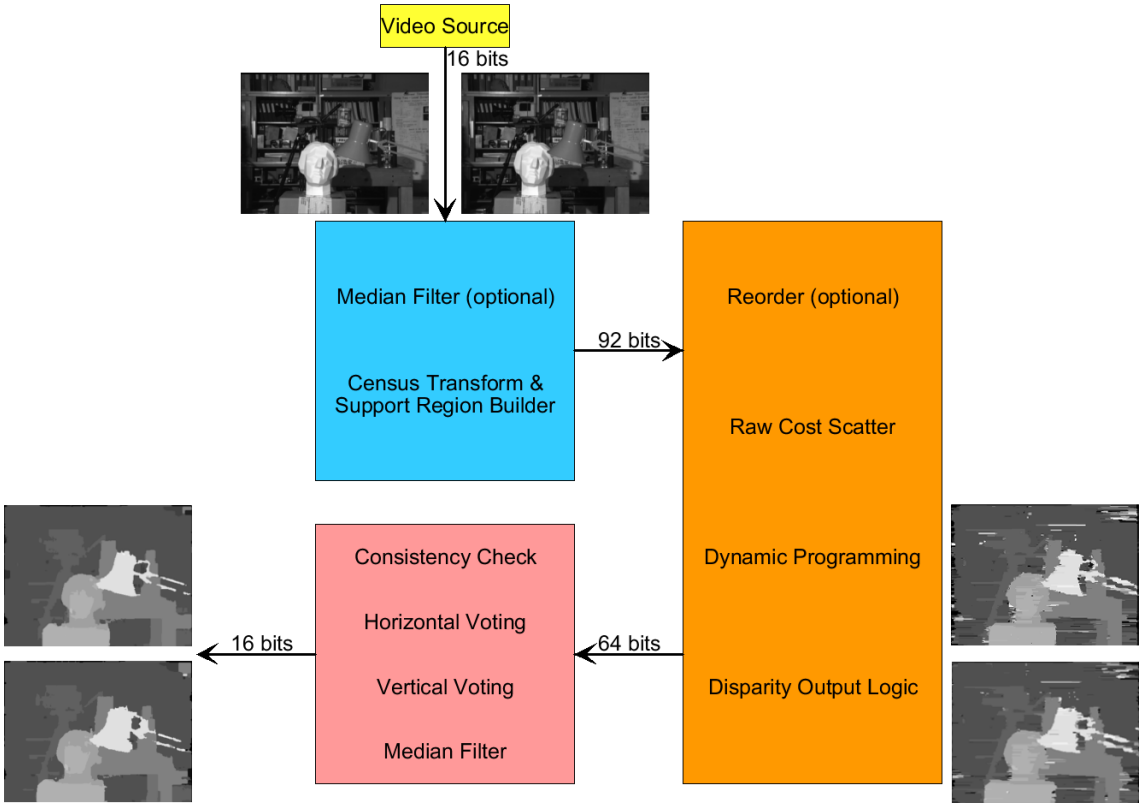


Figure 3.3: Pipelined stereo matching functions

dynamic programming goes into the refinement procedure, which is mainly used to make up the raw depth map, i.e. check and correct occlusion parts and voting for most frequently used disparity. After that the left and right depth maps are generated as the luminance 16 bits data. The total stereo matching is configurable for run-time parameters using control registers, which is convenient for setting resolution, maximum disparity and some threshold used in algorithm.

In the following section, we discuss the modification from the original algorithm of the software model to the hardware friendly algorithm. These modifications change the functions of algorithm because of hardware limitation or optimization, and we provide the comparison and evaluation of each modification.

3.2 Algorithm Modification from Software to Hardware

The software model is suitable for algorithm development or research, because it is a very flexible platform to make your thought into reality model without considering anything else. However, for the hardware model it is often considered more, such as computation complexity, hardware limitation and hardware optimization, so there are a lot of differences between software and hardware model. The disparity loop unrolling introduced in the last section is also a type of difference, but that doesn't change the main function of algorithm. In this chapter, what we discuss is called a modification,

which is actually changing the function to be more hardware friendly.

3.2.1 Vertical Cost Aggregation Modification

The software model proposed by Zhang et al.[44] computes the four direction arm lengths of each pixel to construct the support region for voting, which we discuss in detail in Section 3.3. The support region is shown in Figure 3.4. The vertical

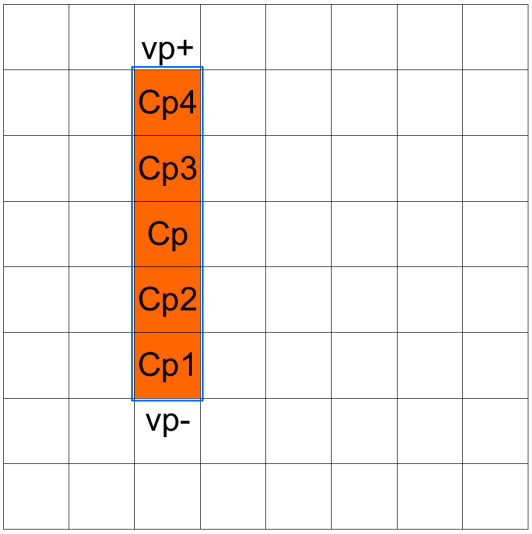


Figure 3.4: Vertical cost aggregation in support region

cost aggregation is one step before dynamic programming processing, which is used to include more vertical relative luminance values to improve the quality of dynamic programming. In Figure 3.4, the aggregation area is the shadowed rectangular, whose number is a parameter limited by hardware resources, and after aggregation the total value need to be weighted by aggregated pixel numbers:

$$C(p,d) = \frac{\sum_{i=-v_p^-}^{v_p^+} C(p+i,d)}{v_p^+ + v_p^-}, \quad d \in [0, d_{max}] \tag{3.1}$$

In hardware implementation, building vertical windows will cost lots of line buffers, whose number depends on the vertical span of the window size, which we introduce in Section 3.3. The default vertical span for vertical cost aggregation is 5, so it has to spend 4 line buffers for that. Moreover, the structure we would like to implement on FPGA is parallel, which means it will cost at least $4 \times D_{max}$ line buffers for vertical cost aggregation. Assuming the frame width is 1024 and the cost is 3 bits using mini-census transform[4], the total on-chip memory usage on vertical cost aggregation will be $D_{max} \times 12Kbits$. For a classical stereo matcher, the maximum disparity D_{max} should be at least 64, so that it will cost $768Kbits$ on-chip memory totally, which is not a small memory usage in our FPGA board.

Considering the point above, the hardware implementation doesn't adopt vertical

cost aggregation, and utilizes the concatenation census vector instead with the completely same result, which we are going to introduce next.

3.2.2 Census Transform Vector Modification

3.2.2.1 Census Transform

Census transform is a kind of non-parametric local algorithm, which depends on the local pixel luminance values in a relative local window, and not on the pixel luminance itself, as shown in Figure 3.5. The direct result of census transform is that the

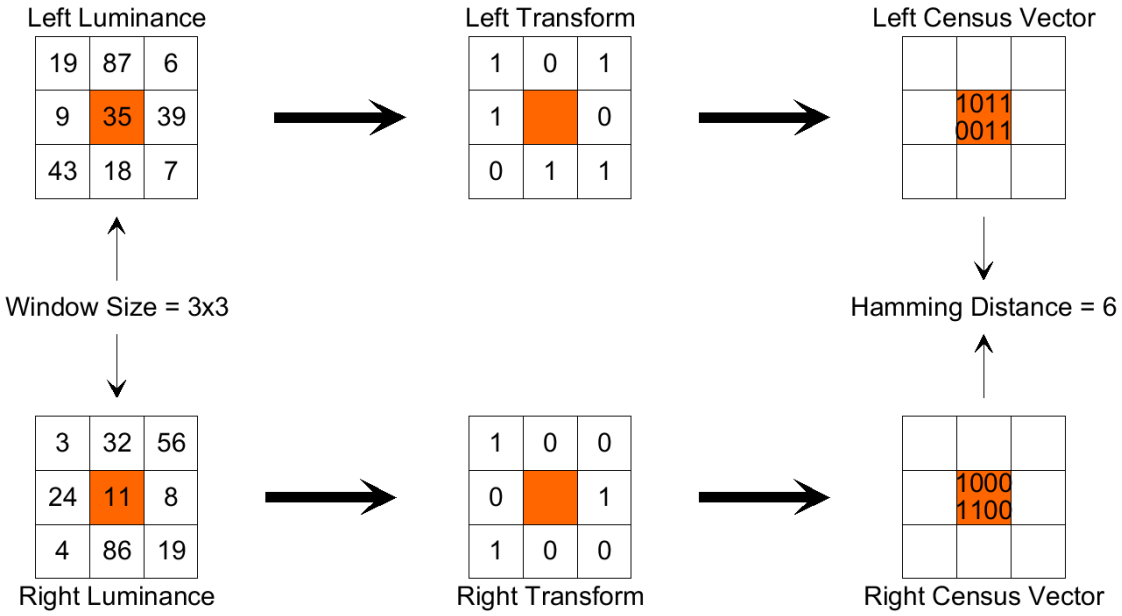
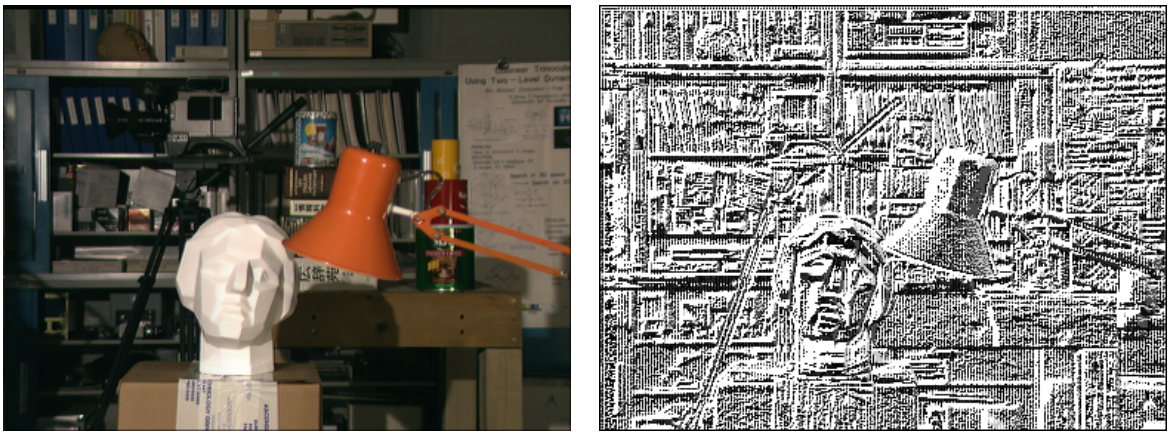


Figure 3.5: Census transform in a 3×3 window

transformed vector attaches the structure information, as shown in Figure 3.6.

The main advantages of the census transform is increasing the robustness near the edge of objects where depth discontinuities occur. The stereo videos from stereo cameras often have gain and bias variations, and in such conditions the census transform is also good to reduce the effects of these variations. The main disadvantage of census transform is the original information attached to each pixel is weakened, but for our scan-line dynamic programming algorithm, the structure information is more important to increase the area smoothness than the luminance value itself. Another disadvantage is that the census transform often generates a long vector consuming many data, which depends on the window size and sampling pattern. For instance, a 3×3 window shown in Figure 3.5 will generate 8 bits census vector, however, a 5×5 window will generate 32 bits vector. The increase of vector length is exponential to the increase of window size when referring to all the neighboring pixels, which will consume too much hardware resources, so it is necessary to choose a hardware-friendly window size and census sampling pattern. In this thesis work, we take the window size by 5×5 for a single



(a) Tsukuba original image (b) Tsukuba census transform vector

Figure 3.6: Tsukuba census transform in a 3×3 window

central pixel, while take the sampling pattern by mini-census transform, which we introduce in detail next.

3.2.2.2 Census Transform Hardware Friendly Modification

The census transform hardware friendly modification has two separated parts, one is the sampling pattern selection, the other is the census vector concatenation optimization to replace the vertical cost aggregation.

In this work, we use the mini-census transform sampling pattern proposed by Chang et al.[4], as shown in Figure 3.7. Utilizing this mini-census transform, the luminance

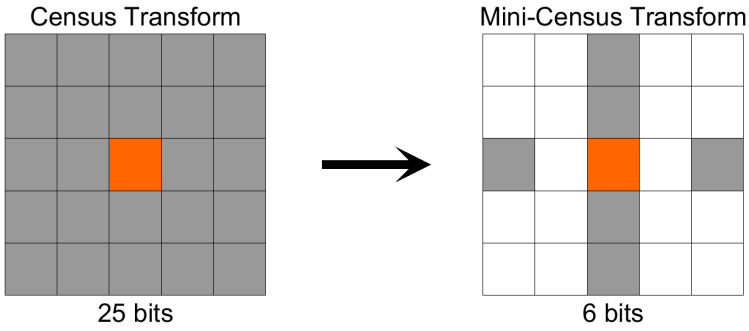


Figure 3.7: Mini-census transform sampling pattern

value of each pixel is converted into a 6 bits vector, which saves very much data width compared with the full neighbor sampling pattern (32 bits). Moreover, the mini-census consumes less data width of raw cost which is the result of the following function block. For instance, the maximum raw cost extracted from luminance difference is 255 (8 bits), while the one extracted from the mini-census vector is only 6 (3 bits). So the data width for raw cost is significantly reduced using mini-census transform vector.

For the memory hungry vertical cost aggregation block mentioned in Section 3.2.1, we could also use census vector concatenation method instead:

$$\mathcal{CT}(v_1) + \mathcal{CT}(v_2) + \mathcal{CT}(v_3) + \mathcal{CT}(v_4) + \mathcal{CT}(v_5) = \mathcal{CT}(v_1 \& v_2 \& v_3 \& v_4 \& v_5) \tag{3.2}$$

In Equation 3.2 $\mathcal{CT}$ is the census transform calculation. Because the raw cost calculation of the census vector is hamming distance calculation, the cost aggregation procedure is the hamming distance aggregation, which could be replaced perfectly by computing the hamming distance of concatenation census vector once according to the property of the hamming distance. This equivalent replacement is shown in Figure 3.8.

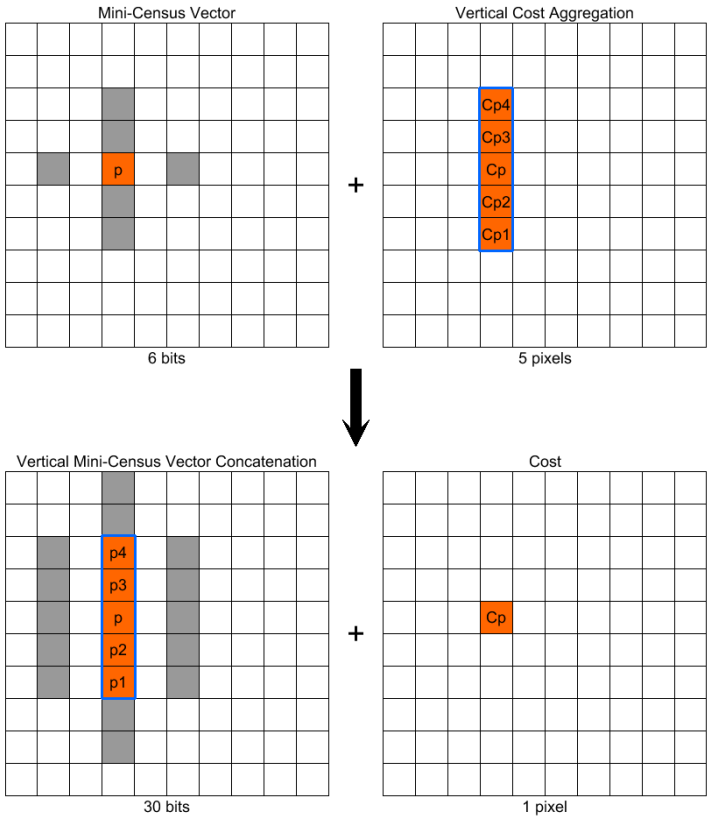


Figure 3.8: Replacement of vertical cost aggregation with census vector concatenation

3.2.3 Parameters and Multipliers Modification

Stereo matching is a complex system, and there are a lot of parameters configurable for it. In the software model performed by C++, a number of weighted or floating point parameters are applied to the stereo matching, furthermore, many multipliers as the attachment are adopted for setting them. However, those floating point parameters and multipliers are not friendly to our hardware implementation at all, which also obey our original target of building a simple system for ASIC. So the hardware friendly modification here is to convert weighted parameters to unweighted ones, and use the

shifter and adder to build a basic multiplier, thus we could control the priority of more complex multiplier structure or more accurate result, which is a resource-accuracy trade-off.

In the following section, we divide the total algorithm by blocks and present them together with the related hardware design. We also provide every possible parameters setting, corresponding latency and on-chip memory usage for reference of each block. The latency computation is based on the required delay and pipeline stages, and the former is related with block functions such as line buffer delay and read latency, while the latter is related with timing analysis of each block to schedule critical paths. The on-chip memory usage from each block is useful for further memory optimization in ASIC design.

3.3 Stereo Matching Design

The high level architecture of stereo matcher is shown in Figure 3.9, including data flow, data width and colored function segment sets. The yellow set is the image source and sink, where the luminance values go into stereo matching while depth map (displayed by luminance) comes out. Considering left and right frames and depth maps of real-time video, the throughput of stereo matching is 16 bits at 65MHz pixel rate ($1024\times768@60FPS$) or more. The green set is some optional segments which could help to improve the performance when encountering particular bottlenecks, e.g. the median filter in the beginning could remove noise if the input stereo video is imperfect, and the reorder could improve critical path problems when the disparity range is pretty high. Adding or removing these optional segments will not effect the final result if there are not such bottlenecks. The blue set is pre-processing part for stereo matching, for the census transform generates census vectors prepared for dynamic programming while the support region builder generates arm lengths and 2D regions prepared for voting. The orange set is the core processing part: the dynamic programming. Its incoming is concatenated census vector from left and right census transform, while its outgoing is fundamental depth map for refinement. The pink set is the refinement part, which makes up the fundamental depth map to generate the final left and right depth map. The gray set is synchronization part used to bypass the arm lengths and synchronize them with corresponding pixels in order to make sure the correct information arrives at the correct place at the correct time.

3.3.1 Median Filter

The median filter is utilized to filter speckle noise such as 'salt and pepper' noise, as shown in Figure 3.10. If a 3×3 window is applied, the abstract function of a median filter can be seen in Figure 3.11. The input data stream of median filter is luminance value from original stereo video in scan-line order, and after being processed the noise filtered luminance value comes out also in scan-line order synchronized with the input pixel rate. The left and right data flows are processed independently using two parallel median filters. The implementation of median filter should be divided into two main parts: one is building a window, the other is obtain the median value in this window.

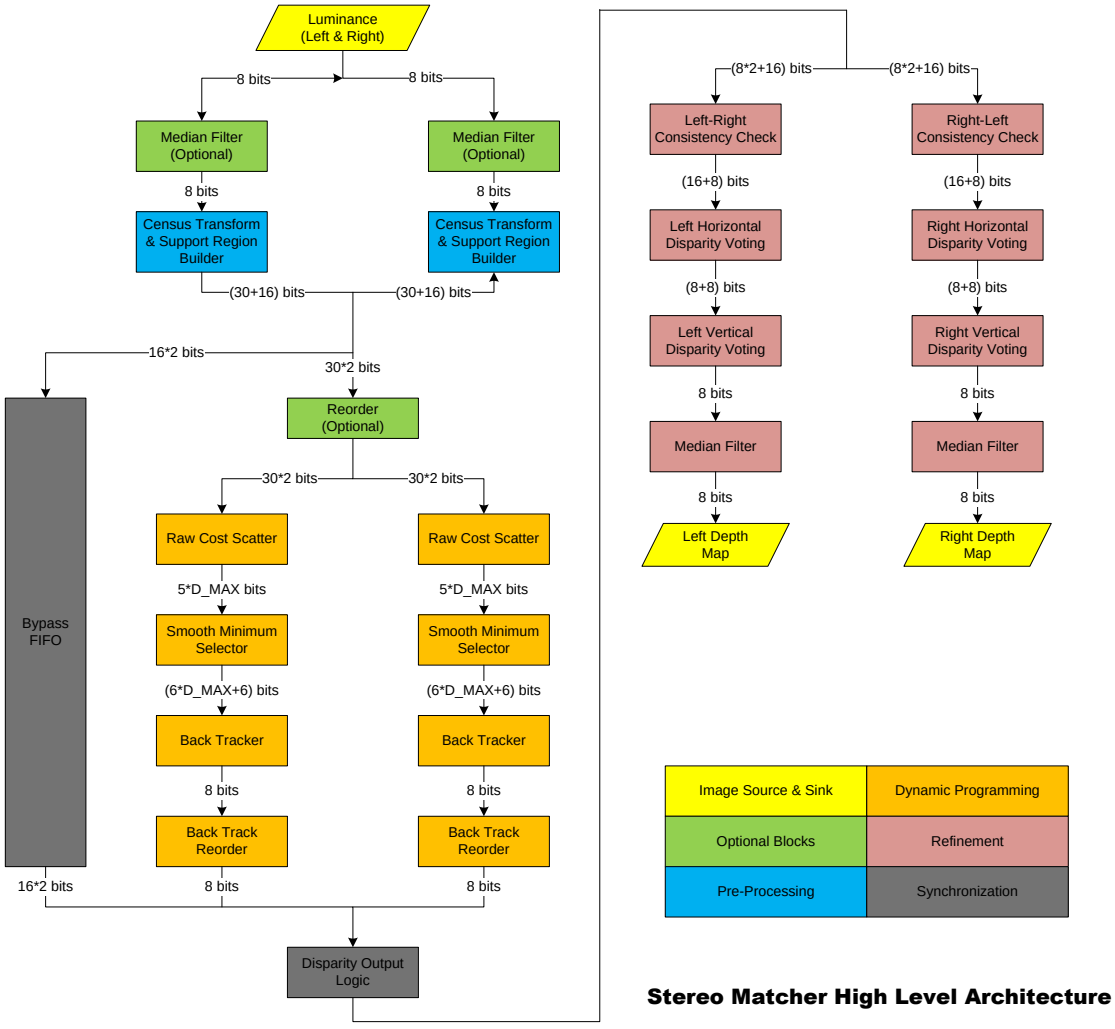


Figure 3.9: Stereo matcher high level architecture

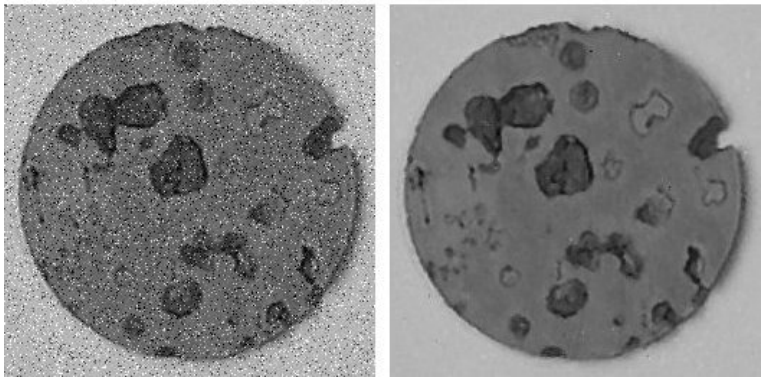


Figure 3.10: Noise removed by median filter

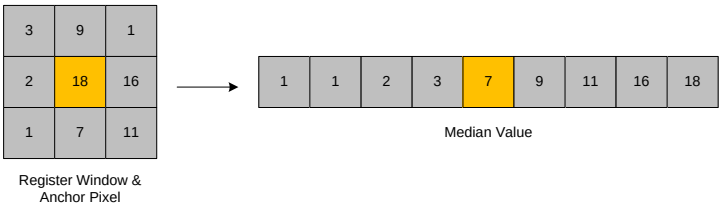


Figure 3.11: Median filter function

In this thesis work, the window size of our median filter is 3×3, and in order to keep a continuous data flow we use the sliding window pixel by pixel, as shown in Figure 3.12. The 3×3 window slides over the complete frame and generates a new 3×3 pixel

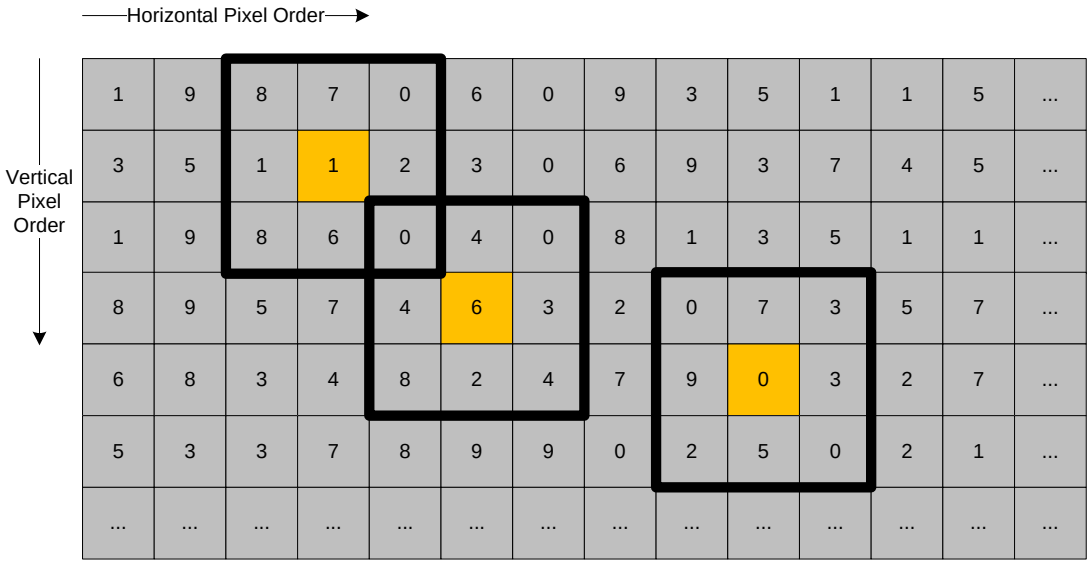


Figure 3.12: Sliding window in median filter in scan-line order

array in every valid pipeline cycle, and the boundary pixels are skipped over the sorting part therefore not filtered. It is necessary to utilize line buffers to save pixels in whole lines in order to construct the vertical part in a sliding window, as well as shift registers to build the horizontal part. The detailed RTL architecture is shown in Figure 3.13.

For building a 3×3 window, 2 line buffers of on-chip ram and 3×3 shift register array are used. The 2 line buffers are used to store 8 bits luminance of 2 lines corresponding to the top 2 lines of the window, and the depth of a line buffer depends on the maximum possible frame width (FW), e.g. the depth of the line buffer we used is 1024 so that any frame width lower than 1024 could be supported by this line buffer. The register array is adopted to save luminance value of the window for further use, where *WinReg4* saves the anchor pixel. The registers near the input are for balancing the read latency of on-chip ram (2 cycles). The counter cyclically counts ($0 \sim FW-1$) to read data out and write data in by providing read and write addresses for line buffers. The delay line

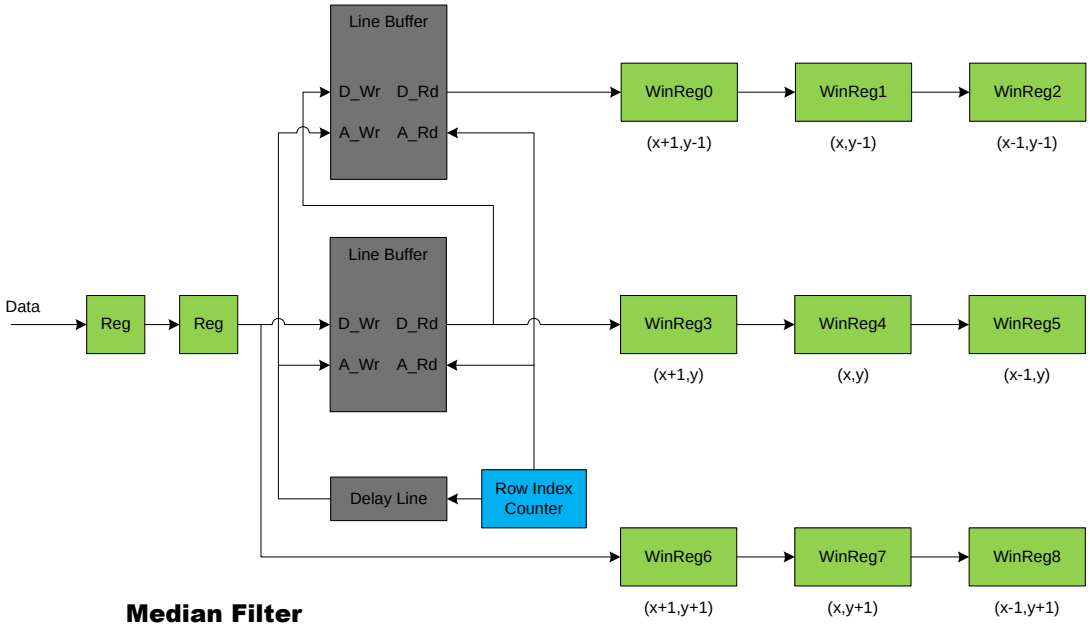


Figure 3.13: Median filter RTL architecture

is utilized for avoiding to overwrite the unread data in the same line buffer.

After building sliding window, the 9 luminance values in it go into a module to obtain their median value. The RTL architecture utilized by us is proposed by Vega-Rodriguez et al.[39], as shown in Figure 3.14. Totally 19 comparators 38 multiplexers

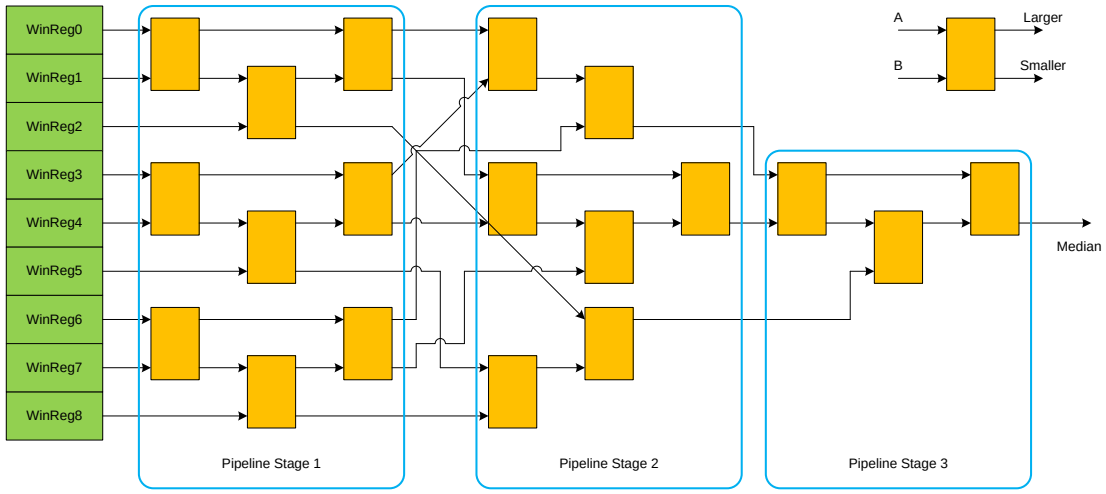


Figure 3.14: Median filter sorting

are used in this structure to sort the median value, and the basic segment shown in the top-right corner is composed of 1 comparator and 2 multiplexers for 2 unsigned inputs. The whole procedure is divided into 3 pipeline stages, which means this sorting

segment has 3 cycles latency.

The total latency (the valid cycles used to obtain output from input) and memory usage for median filter are shown in Table 3.1. The latency is the valid cycles used to

	Latency (cycles)	Memory usage (bits)
Line Buffers	FW	$2\times2\times FW\times8$
Input Registers	2	0
Buffer Window	2	0
Sorting Array	2	0
Total	$FW+6$	$32\times FW$

Table 3.1: Median filter latency and memory usage

obtain output from input for each block, while the memory usage includes the parallel median filter, and in the next blocks the memory usage calculated in the tables includes all the parallel segments, not the single one. The complete quantitative reports about the latencies and memory usage of every stereo matcher segments are listed in Chapter 4.

3.3.2 Census Transform & Support Region Builder

According to our concatenated census transform shown in Figure 3.8, the input luminance is transformed into a 30 bits vector considering vertical concatenation, which needs also a sliding window. The implementation of the census transform is also divided into two steps: one is constructing sliding window using line buffers and shift register array, the other is obtaining the census vectors by comparing neighboring pixels with vertical center anchor pixels according to the mini-census transform pattern and concatenating them. The first step uses the same architecture with median filter to build the sliding window, while the second step just applies comparators onto the register array obtained in the first step. Moreover, the first step shares the hardware resources with support region builder, because they share the same sliding window.

The support region builder is used to build a region with similar luminance for refinement, so it does not contribute to the dynamic programming. The key procedure of building a support region for an anchor pixel p is to determine its arm length in four directions, as shown in Figure 3.15. The intersection pixel of two orthogonal lines is called the anchor pixel, for which the support region is building. In the figure the anchor pixel p is the intersection of $V(p)$ and $H(p)$. Each anchor pixel has four direction arm lengths, i.e. $\{h_p^-, h_p^+, v_p^-, v_p+\}$ for p are the arm length of the left, right, up and down directions. The support region building depends on the luminance difference between the pixels in the two orthogonal lines and the anchor pixel, therefore the largest span in four directions is determined as:

$$r^* = \max_{r\in[1,L]}(r \prod_{i\in[1,r]} \delta(p,p_i)) \tag{3.3}$$

In Equation 3.3, r^* is the largest span; pixel $p_i = (x_p - i, y_p)$ and constant L is the maximum allowed arm length, which is a configurable parameter of support region

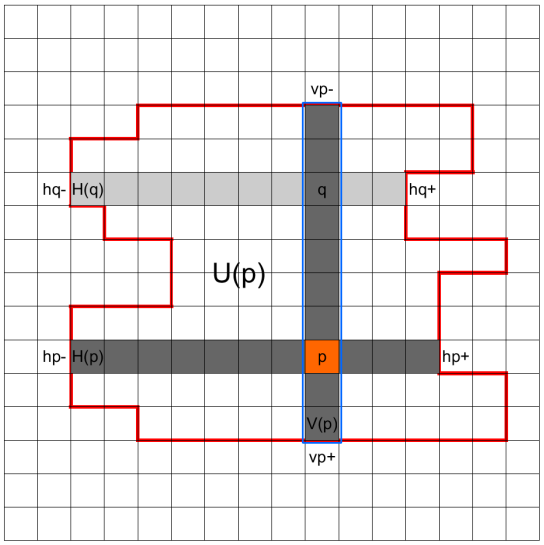


Figure 3.15: Support region of the anchor pixel p

builder; $\delta(p, p_i)$ is the luminance difference between pixel p and p_i based on another configurable parameter τ, which controls the luminance difference degree:

$$\delta(p_1, p_2) = \begin{cases} 1, & |Y(p_1) - Y(p_2)| \leq \tau \\ 0, & otherwise \end{cases} \tag{3.4}$$

The parameter τ has various performance implications according to the applications and image texture. According to the four arms provided, the total horizontal span and vertical span of anchor pixel p are:

$$\begin{cases} H(p) = \{(x, y) \,|\, x \in [x_p - h_p^-, x_p + h_p^+], y = y_p\} \\ V(p) = \{(x, y) \,|\, x = x_p, y \in [y_p - v_p^-, y_p + v_p^+]\} \end{cases} \tag{3.5}$$

Therefore, assembling all the horizontal arm lengths of the pixels on the vertical $V(p)$ builds the complete support region $U(p)$ of pixel p. The built arm lengths then passes dynamic programming by a bypass FIFO to prepare for voting.

The implementation of the support region builder is also divided into two main parts: one is the building window and the other is comparing candidates with the central pixel in the window to obtain the arm lengths.

The first window building step is similar with the window building of the median filter, and the number of required line buffers depends on the parameter L. The maximum arm length L needs $2{\times}L$ line buffers to provide a $(2{\cdot}L + 1) \times (2{\cdot}L + 1)$ window, which is composed of shift register arrays. The optimized algorithm takes $L = 15$, so the hardware implementation has to select 30 line buffers to build a $31{\times}31$ window which is used by both the support region builder and the census transform. The second step of these two segments are processed in parallel.

The second step is computing every $\delta(p, p_i)$ in each direction in parallel by comparing every p_i with p simultaneously, and then passing the result into parallel priority encoders

to work out the largest span of each direction. The boundary problem occurs when the central pixel is near the boundary, e.g. the anchor pixel is in the last line of one frame. In such condition the window includes the pixels not belonging to the current frame, and the row counter and the column counter are adopted to avoid those incorrect pixels being processed.

The latency of census transform and support region builder is determined by the line buffers and operation pipeline stages, and the detailed value is listed in Table 3.2. The configurable parameters are listed in Table 3.3, and the effect of adjusting parameters is discussed together in Chapter 4.

	Latency (cycles)	Memory usage (bits)
Line Buffers	$L \times FW$	$2 \times 2 \times L \times FW \times 8$
Input Registers	2	0
Buffer Window	L	0
Comparators + Encoders	2	0
Total	$L \times FW + L + 4$	$32 \times L \times FW$

Table 3.2: Census transform and support region builder latency and memory usage

Parameters	Current Value	Description
L	15	maximum support region arm length
τ	15	luminance difference threshold

Table 3.3: Census transform and support region builder configurable parameters

3.3.3 Reorder

There is a loop for scan-line optimized dynamic programming, which cannot be unrolling to a parallel structure. The reorder kernel provides loop unrolling preparation if the critical path of that loop cannot meet the pixel rate frequency. It is optional because in most cases the maximum disparity is not too large and the pixel rate frequency is not too high. However in some special applications or conditions such as stereo matching for 1080p video with maximum disparity 256 or more, reorder is a necessary kernel performing static scheduling to improve the critical path with the cost of increasing on-chip memory utilization. In principal this loop unrolling is a space-time trade-off.

In this design, the reorder is used after census transform to change the pixel (census vector) flow order from scan-line order into line-based rotational order, as shown in Figure 3.16. In this figure four lines are reordered, which means three other pipeline stages are inserted into the loop, and each of these three stages is the point of other loops, so the four loops are unrolled by each other. Pixel order is changed into this line-based rotational order because dynamic programming uses pixel loop for each line.

The implementation of reorder is line buffers and counters. Assuming the number of reordered lines is r, which is a configurable parameter to control the unrolling stages, and r also influences the following design including the loop in dynamic programming.

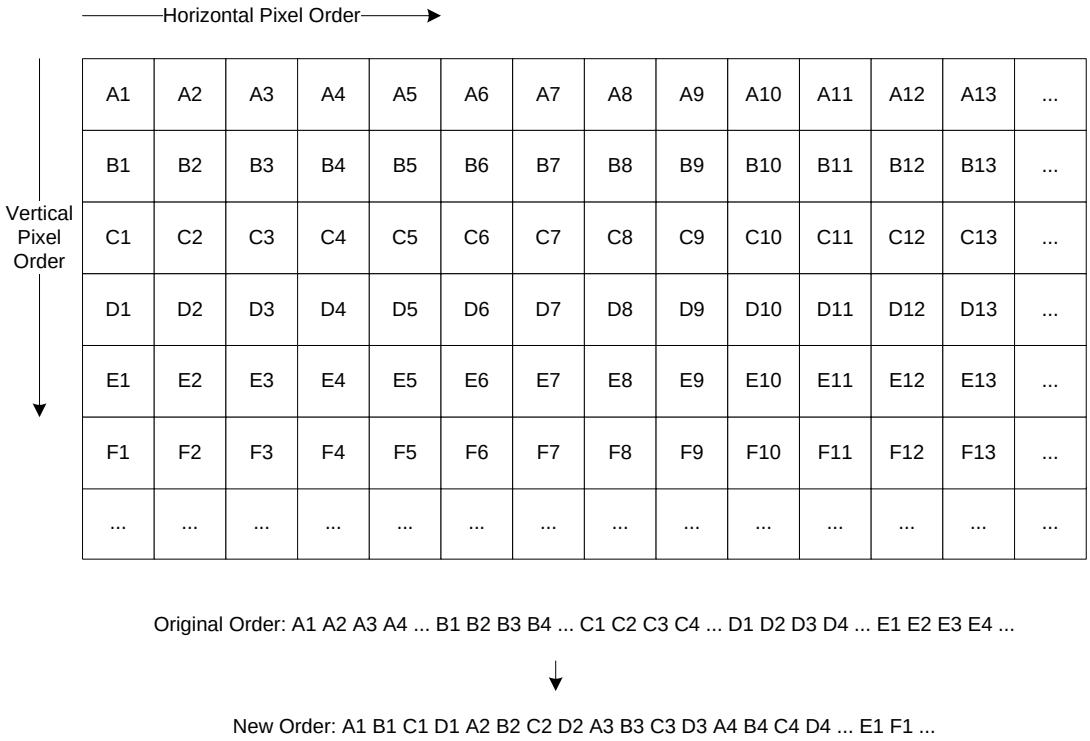


Figure 3.16: Reordered order of pixels

Then the number of line buffers is $2r$ to provide ping-pong buffering, and the depth of the line buffers depends on the frame width, while the length of them is determined by the width of census vector. The counters are the key segments used to write and read pixels in the line buffers to change the data flow order, therefore the writing counter is in scan-line order in the front buffer, while the reading counter is in line-based rotational order in the back buffer.

The latency of reorder is determined by the line buffers and r, while the memory usage depends on the line buffers, r and data width of census vector (DW_CV), and the detailed value is listed in Table 3.4, The configurable parameters are listed in Table 3.5, and $r = 1$ means currently reorder is not used in our system, thus there is no latency and memory usage for this kernel.

	Latency (cycles)	Memory usage (bits)
Line Buffers	$r \times FW$	$2 \times r \times FW \times 2 \times DW\_CV$
Buffers Reading	2	0
Total (for $r > 1$)	$r \times FW + 2$	$4 \times r \times DW\_CV \times FW$
Total (for $r = 1$)	0	0

Table 3.4: Reorder latency and memory usage

Parameters	Current Value	Description
r	1	reordered line number

Table 3.5: Reorder configurable parameters

3.3.4 Raw Cost Scatter

The raw cost scatter is used to calculate all the raw costs between left frame and right frame pixels in the disparity range, thus the disparity parallel structure starts with this kernel. It prepares for dynamic programming, and its structure is influenced by the reorder parameter r, as shown in Figure 3.17. We assume in Figure 3.17 the maximum

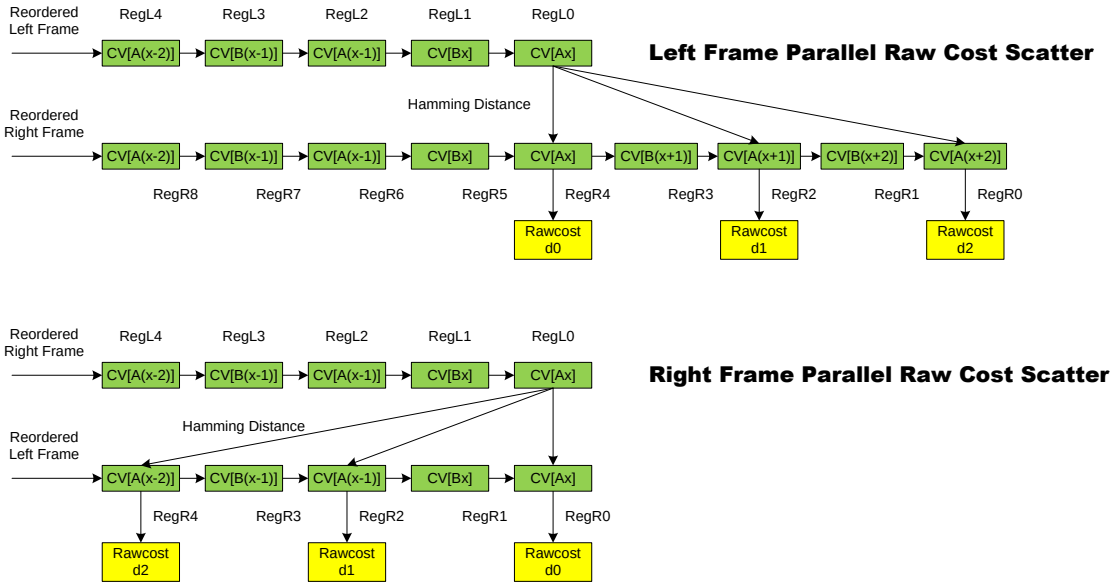


Figure 3.17: Raw cost scatter RTL architecture

disparity $D_{max} = 3$ and reordered lines $r = 2$. The raw cost kernel generates costs for all disparities in parallel, and the structures are different for left and right frame because the searching ranges for left and right frame pixels are in different direction i.e. left searching direction for target frame pixels and right searching direction for reference frame pixels, but the principle of them are the same, and in this section the left frame parallel raw cost scatter is taken as the instance. The input data stream for this module is the reordered census vector $cv_p(x, y)$ from left frame pixel $p(x, y)$ and $cv_{p'}(x', y')$ from right frame pixel $p'(x', y')$. Because the x' is in the searching range from x to $x - (D_{max} - 1)$ in the left direction of x, the $cv_p(x, y)$ goes into a $r \times (D_{max} - 1)$ length shift register array, while the $cv_{p'}(x', y')$ goes into a $2r \times (D_{max} - 1)$ length shift register array for parallel hamming distance calculation. After a certain pipeline cycle, the raw costs for $p(x, y)$ for all disparities from 0 to $(D_{max} - 1)$ is obtained:

$$C(p, d) = HammingDistance(cv_p(x, y), cv_{p'}(x - d, y)), \quad d \in [0, D_{max} - 1] \tag{3.6}$$

The first $r \times D_{max}$ shift registers for left and right census vectors input in the left frame parallel raw cost scatter is for synchronization to make sure the pipeline latency is the same with right frame parallel raw cost scatter, because for the anchor pixel $p'(x', y')$ in the right frame the x is in the searching range from x' to $x' + D_{max}$ in the right direction of x'.

The synchronized final latency of raw cost scatter is determined by D_{max} and r, as shown in Table 3.6. In addition, the neighboring pixels continuity check prepared

	Latency (cycles)	Memory usage (bits)
Shift Registers	$r \times (D_{max} - 1)$	0
Hamming Distance Calculation	2	0
Total	$r \times (D_{max} - 1) + 2$	0

Table 3.6: Raw cost scatter latency and memory usage

for dynamic programming is performed in this kernel using a configurable parameter continuity threshold th_c, which is listed in the dynamic programming parameter table.

3.3.5 Dynamic Programming

3.3.5.1 Smooth Minimum Selector

Smooth minimum selector is the first step of dynamic programming:

$$C(j, d) = C(j, d) + \min_{d' \in [0, D_{max}-1]} \{C(j-1, d') + S(|d - d'|)\} \tag{3.7}$$

It is used to construct every smooth paths of every pixels in each disparity by configuring the smoothness term. There is a loop in the Equation 3.7 that the cost of current pixel $C(j, d)$ is determined by the sum of the cost of previous pixel in every disparity $C(j, d')$ and smoothness term $S(|d - d'|)$. In this real-time implementation, we adopt the pixel rate frequency which is $65MHz$ for $1024 \times 768@60FPS$, so that in one cycle we need to perform the function of $C(j, d) + \min_{d' \in [0, D-1]} \{C(j-1, d') + S(|d - d'|)\}$, which includes adders and comparators, as shown in Figure 3.18. It is disparity parallel structure for calculation, so all the array signals include the corresponding signals in all disparities.

For the Potts model design, the upper part is the main loop composed by adder array, minimum selector array and registers, and the minimum selector has a tree structure, as shown in Figure 3.19. The coming in data stream for this tree structure minimum selector is all costs in disparity range, and for each cost a disparity value as a property is attached. After $\log_2 D_{max}$ levels of comparison, the final minimum cost as well as its attached disparity is obtained in the output. Therefore in one pixel rate cycle, dynamic programming performs one loop including two levels of adders and $\log_2 D_{max}$ levels of comparators. In normal cases, the D_{max} is not very large, e.g. $D_{max} = 64$ in our system meets the most applications, and then this loop could be completed in one cycle. However for special applications which require a very large disparity range, e.g. $D_{max} = 256$ or more, the reorder kernel needs to be adopted to unroll this loop into more clock cycles, and correspondingly more pipeline stages need to be designed in the loop to fit the unrolling pixel order. From the Synplify report, our worst path

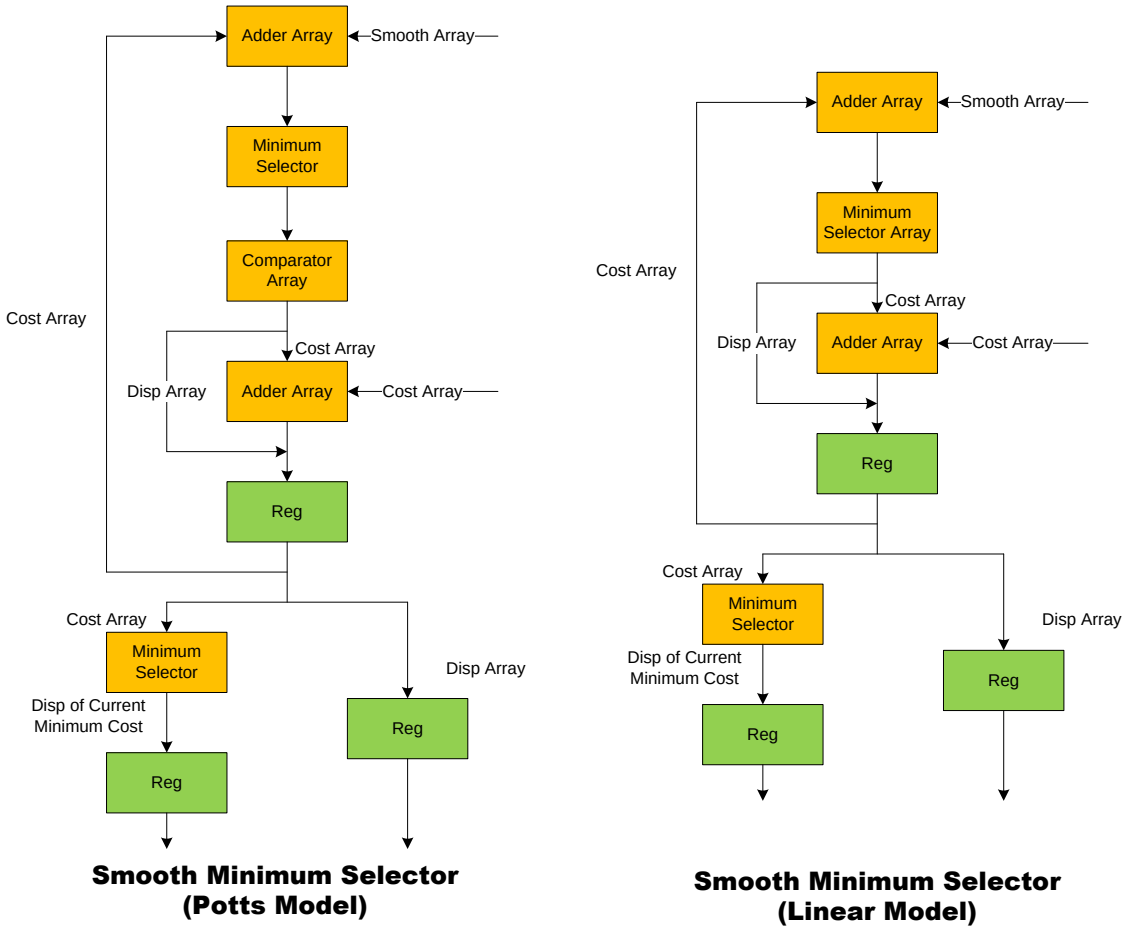


Figure 3.18: Smooth minimum selector RTL architecture

is this loop whose propogation time is 13.812ns, which means the estimated frequency of our stereo matcher is 72.4MHz. Currently the stereo matcher operates on 65MHz pixel rate frequency for $1024\times768@60FPS$, therefore only one pipeline stage (register) in the loop is enough for operation and we don't need the reorder kernel to optimize the critical path.

For the linear model design, the basic principle of the loop is the same, and the only difference is the minimum selecting part, because their smoothness term is different. The smoothness term for the linear model has D_{max} different values for all disparities $d' \in [0, D_{max} - 1]$ in Equation 3.7, so that D_{max} parallel tree structure minimum selectors are used to implement the equation, which includes D_{max}^2 comparators, and the computational complexity is $O(D^2)$.

The smoothness term for the Potts model only has two 2 different values for all disparity $d' \in [0, D_{max} - 1]$, so that we optimize the parallel structure into a more serial one, which is firstly using a tree structure to select minimum cost and then starts the parallel comparison with original values for $d' \in [0, D_{max} - 1]$:

$$C(j,d) = C(j,d) + \min_{d' \in [0, D_{max}-1]} \{C(j-1, d'), \min_{d' \in [0, D_{max}-1]} \{C(j-1, d')\} + C\} \quad (3.8)$$

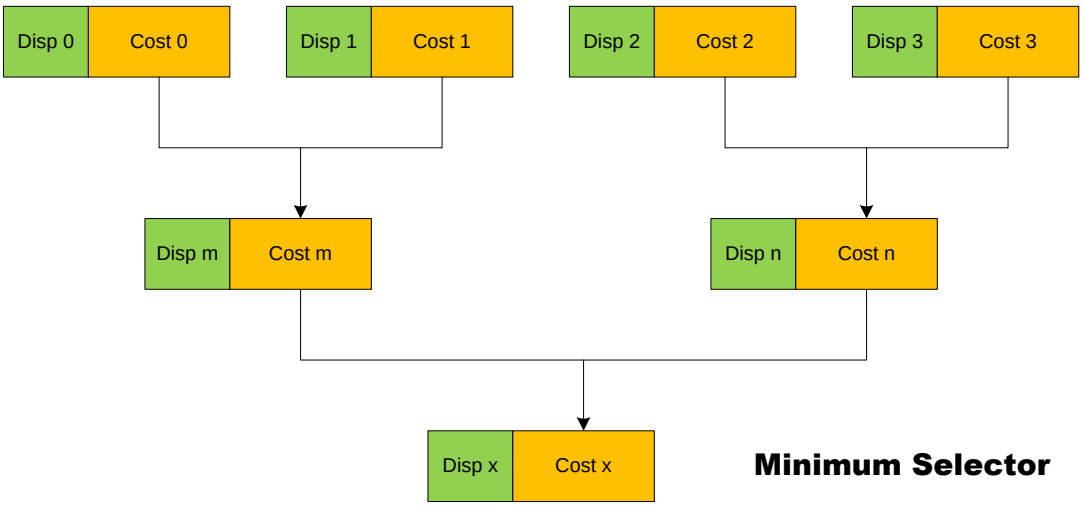


Figure 3.19: Minimum selector tree structure

This modified implementation only includes $2 \cdot D_{max}$ comparators with the computational complexity $O(D)$, however, it will increase the length of critical path by adding a level of comparators, which is a time-resource trade-off. In our system, the modified longer critical path can still meet the pixel rate clock cycle, so that we adopt this optimized implementation to save hardware resources. The constant C is a parameter for smooth cost in the Potts model and depends on the continuity threshold th_c that checks neighboring pixel continuity. Its main function is to reduce the smoothness near the boundary.

The bottom part is preparing for the second step of dynamic programming and is completely the same for the Potts model and linear model, which is used for outputting disparity matrix and selecting the minimum cost in the disparity range of every pixel. This minimum cost is only used in the last pixel of each line, because the second step of dynamic programming is a back tracking procedure along the saved disparity matrix, which means every disparity selection of each pixel depends on the previous selected value, and this value provides the beginning value of this tracking.

The latency of smooth minimum selector depends on the pipeline stages designed in the whole calculation procedure and there is no memory usage in smooth minimum selector, as shown in Table 3.7, and the parameters of smooth minimum selector is listed in Table 3.8.

	Latency (cycles)	Memory usage (bits)
Loop	1	0
Minimum Cost Selector	1	0
Total	2	0

Table 3.7: Smooth minimum selector latency and memory usage

Parameters	Current Value	Description
th_c	15	neighboring pixels continuity check threshold
C	25(continued)/5(discontinued)	smooth cost for Potts model when $d \neq d'$

Table 3.8: Smooth minimum selector configurable parameters

3.3.5.2 Back Tracker

Back tracker is the second stage of dynamic programming:

$$d(p_{W-1}) = \arg \min_{d' \in [0, D-1]} C(W-1, d') \tag{3.9}$$

It is used to track the disparity matrix obtained by the first stage backward to fetch a smooth disparity path for one line. In the first stage every disparity of every pixel in one scan-line has a corresponding smooth disparity of the previous pixel. This disparity matrix is saved in the back tracker and is tracked from the minimum cost disparity of the last pixel in one scan-line, which is also obtained in the first stage. So in this kernel a lot of on-chip memory is used to save the disparity matrix and minimum cost disparity, and the RTL architecture of back tracker is shown in Figure 3.20.

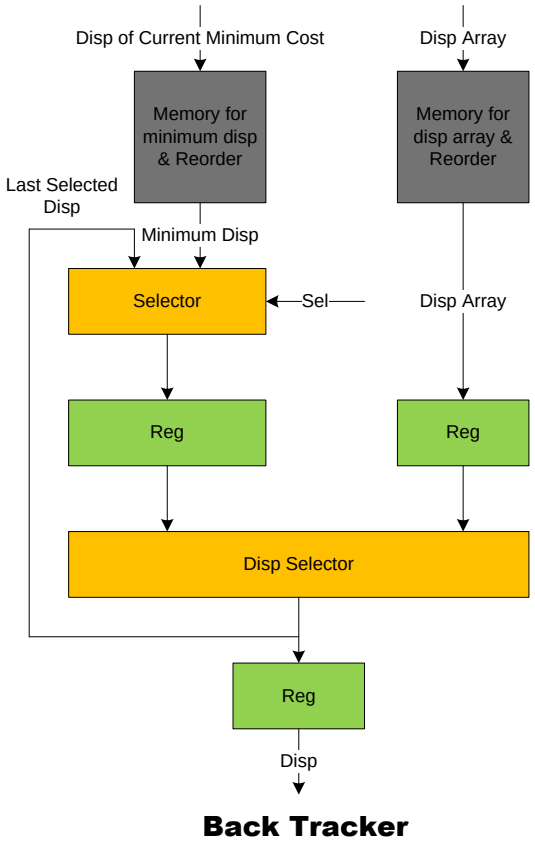


Figure 3.20: Back tracker RTL architecture

The top two on-chip memory is used to store the results of smooth minimum selector, where the left RAM is to store the minimum costs of every pixel and the right RAM is to save the disparity matrix for tracking. In the left RAM, only the last pixel of each line refers to its value, however the minimum costs of a whole scan-line is stored to easily keep synchronization with the right RAM using the same address counter. Another function of these two RAMs is to change back the modified pixel order into backward scan-line order because the procedure of back tracking is tracking from the last pixel of each line to the first pixel of this line using ping-pong buffering. The selector under the left RAM is a multiplexer to choose between minimum cost disparity at the last pixel of each line and last selected smooth disparity at the other pixels of each line. Here it is another loop for back tracking one after another to select the smoothest path, and the input is not the unrolled order but the normal scan-line order, so that it has only one pipeline stage in the loop. Thereafter, disparity selector chooses the next smooth neighboring pixel disparity according to the disparity matrix and last selected disparity to start the next cycle. The bottom register is used to store the last selected disparity as well as the minimum disparity of the last pixel of each line, because the latter is not chosen by the disparity selector but the selector under the minimum disparity memory. Accordingly for the last pixel of each scan-line, the selector also offers its result to the bottom register to keep this minimum disparity.

The latency of the back tracker depends on FW, r and the pipeline stages, and there is a lot of memory usage in this kernel, which is determined by FW, D_{max} and r, as shown in Table 3.9. There is no configurable parameters for back tracker.

	Latency (cycles)	Memory usage (bits)
Buffers	$r \times FW$	$2 \times 2 \times r \times FW \times (D_{max} \times \log_2 D_{max} + \log_2 D_{max})$
Buffers Reading	2	0
Tracking Selection	1	0
Total	$r \times FW + 3$	$4 \times r \times (D_{max} \times \log_2 D_{max} + \log_2 D_{max}) \times FW$

Table 3.9: Back tracker latency and memory usage

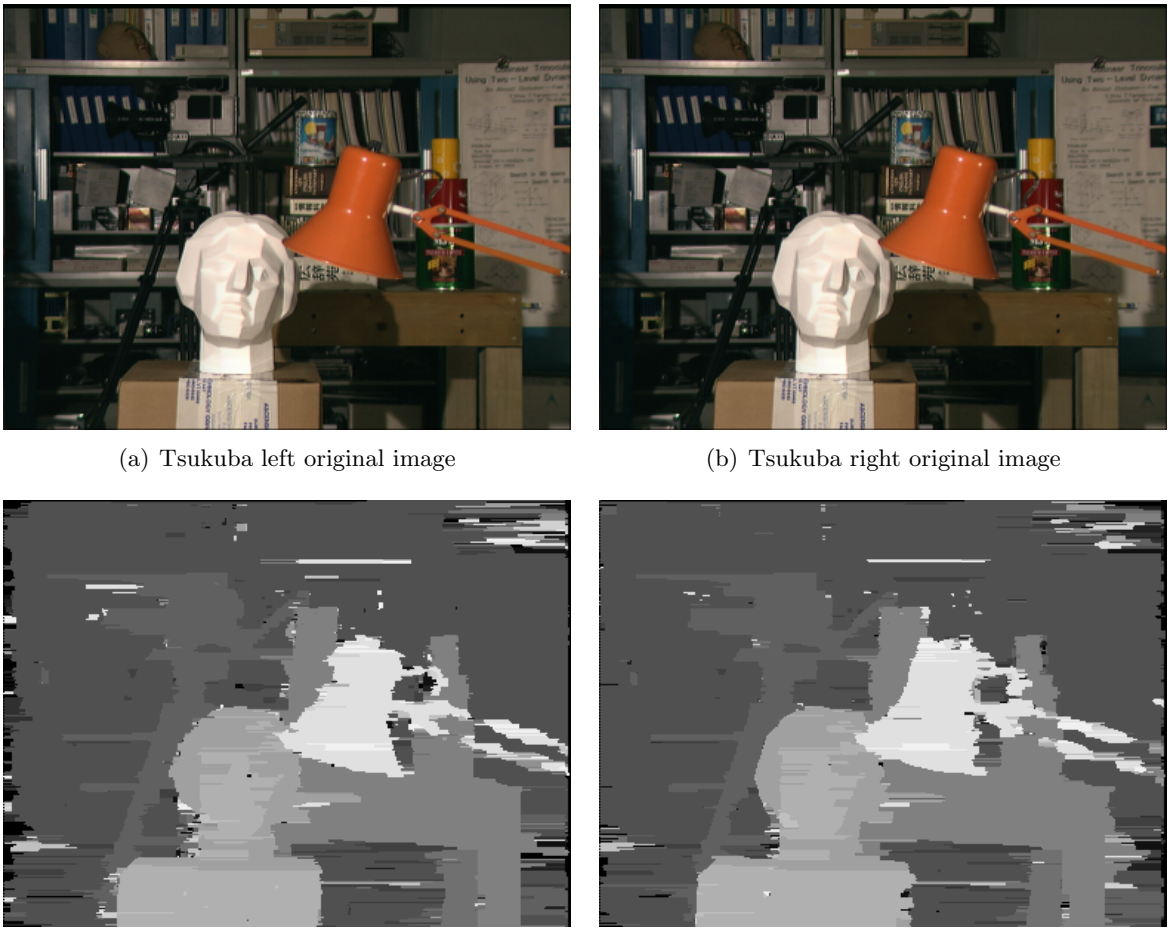
3.3.5.3 Back Track Reorder

The output of the back tracker is the backward order of scan-line, and this back track reorder is used to change the backward order back into forward order in one scan-line using ping-pong buffering. The implementation is just line buffers and counters, which is almost the same with reorder. But unrolling has been terminated in the back tracker block, so the parameter r has no influence with the latency and memory usage in back track reorder, as shown in Table 3.10.

The result of the back track reorder are the left and right depth maps of dynamic programming, as shown in Figure 3.21. We see from the depth maps there are some stripe noise and point noise, and the former is produced by the incorrect matching of this scan-line optimized dynamic programming, while the latter is generated by the incorrect matching in the occlusion areas. The quality of the depth maps is improved in refinement by removing these two kinds of noise.

	Latency (cycles)	Memory usage (bits)
Buffers	FW	$2\times2\times FW\times\log_2 D_{max}$
Buffers Reading	2	0
Total	$FW+2$	$4\times\log_2 D_{max}\times FW$

Table 3.10: Back track reorder latency and memory usage



(a) Tsukuba left original image

(b) Tsukuba right original image

(c) Tsukuba dynamic programming left depth map

(d) Tsukuba dynamic programming right depth map

Figure 3.21: Tsukuba left and right depth maps generated by dynamic programming

3.3.6 Bypass FIFO and Disparity Output Logic

The arm lengths generated in the support region builder is not used in dynamic programming, and bypass FIFO is used to pass the eight arm lengths (32 bits) for both left and right frame pixels by dynamic programming for refinement. Actually, the bypass FIFO is a two port RAM without read address and write address, but read enable and write enable instead following the first in first out order.

The latency of bypass FIFO mainly depends on the sum of reorder latency, raw cost

scatter latency and dynamic programming latency:

$$\begin{aligned} l_{fifo} &= r\times FW + 2 + r\times(D_{max} - 1) + 2 + 2 + r\times FW + 3 + FW + 2 \\ &= r\times(2\times FW + D_{max} - 1) + FW + 11 \end{aligned} \tag{3.10}$$

The memory usage of the bypass FIFO is determined by its latency, which is the number of data it needs to store at least, as shown in Table 3.11.

	Latency (cycles)	Memory usage (bits)
FIFO	l_{fifo}	$l_{fifo}\times 32$
FIFO reading	1	0
Total	$l_{fifo} + 1$	$l_{fifo}\times 32$

Table 3.11: Bypass FIFO latency and memory usage

Bypass FIFO has no write latency, which is the same with reorder, so that the input data for both of them is synchronized. However bypass FIFO has one cycle of read latency when the read enable signal goes high, and back track reorder has no read latency when its valid signal goes high, therefore the output data is not synchronized. Disparity output logic is used to synchronize them by increasing one cycle latency for the data from back track reorder. Its implementation is just using shift registers, and its latency and memory usage is listed in Table 3.12. After the disparity output logic, the synchronized data is then used for refinement.

	Latency (cycles)	Memory usage (bits)
Shift Registers	2	0
Total	2	0

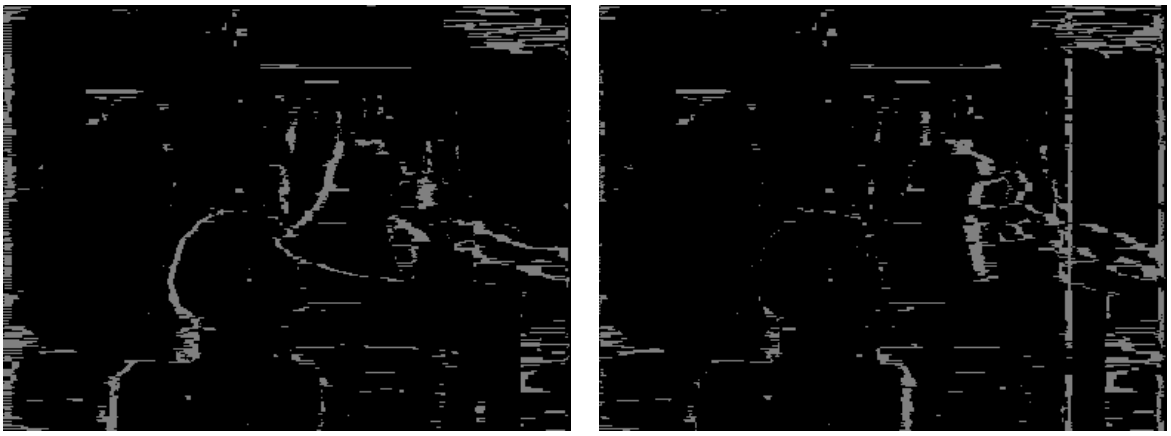
Table 3.12: Disparity output logic latency and memory usage

3.3.7 Refinement

Refinement is the kernel sets making up the depth maps generated from dynamic programming. It takes the left and right depth maps as well as the arm lengths of left and right pixels to remove the stripe and point noises, and finally generates the smoother left and right depth maps. It mainly includes consistency check, horizontal voting, vertical voting and median filter.

3.3.7.1 Consistency Check

The stereo frames have occlusion areas both for left and right images, and we can show these areas using occlusion map, as shown in Figure 3.22. In the occlusion maps, the light areas are occlusion parts and are not reliable, thus the disparities in these areas are not reliable either. The function of consistency check is to detect these occlusion areas and modify them with the closest reliable disparities. The adjustment of reliable disparity is given in Equation 3.11. In addition, the consistency check also makes up



(a) Tsukuba left occlusion map (b) Tsukuba right occlusion map

Figure 3.22: Tsukuba left and right occlusion maps

the corresponding unreliable boundaries (left boundary of left depth map and right boundary of right depth map) by replacing them with the nearest reliable disparities in the same scan-line.

$$\begin{cases} |D(x,y) - D'(x - D(x,y), y)| \leq th_{cc}, & left \\ |D'(x,y) - D(x + D'(x,y), y)| \leq th_{cc}, & right \end{cases} \tag{3.11}$$

The RTL architecture of the consistency check for both left and right is shown in Figure 3.23. The difference between left and right consistency check is that the left and

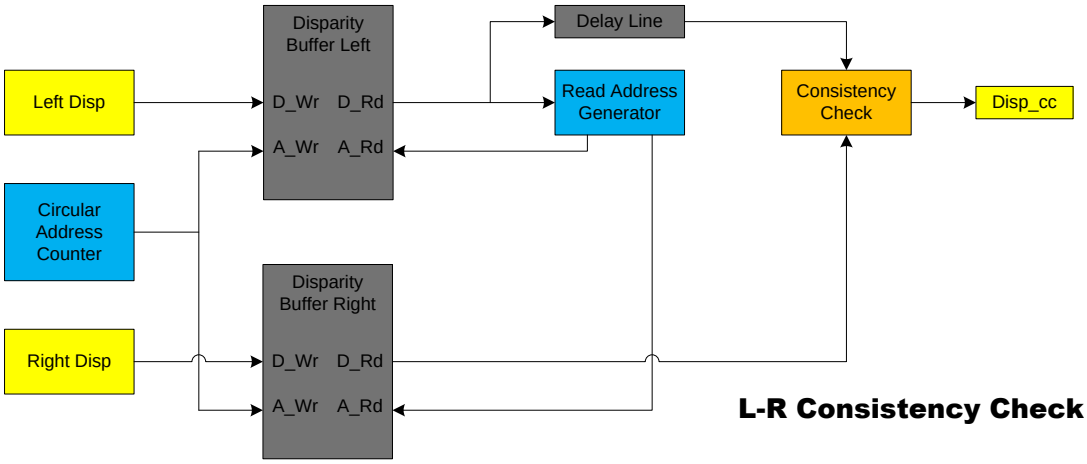


Figure 3.23: Consistency check RTL architecture

right input is exchanged, and we take the left one as the instance. The circular address counter generates addresses to write left and right disparities into disparity buffers. The length of disparity buffers must not be shorter than D_{max} to make sure $x - D(x,y)$ can be obtained. The read address generator fetch $D(x,y)$ from left disparity buffer

and then uses the fetched value to obtain $D'(x - D(x, y), y)$ from right disparity buffer. However, this serial procedure generates a reading time difference between these two disparity buffers, and the delay line is used to synchronize them for consistency check. The consistency check segment is composed of comparators and multiplexers to realize the comparison and replacement as well as update the closest reliable disparity. In addition, the arm lengths for voting is bypassed by buffers and shift registers.

The latency and memory usage of the consistency check depends on D_{max} and implemented pipeline stages, as shown in Table 3.13. The parameters of the consistency check is shown in Table 3.14.

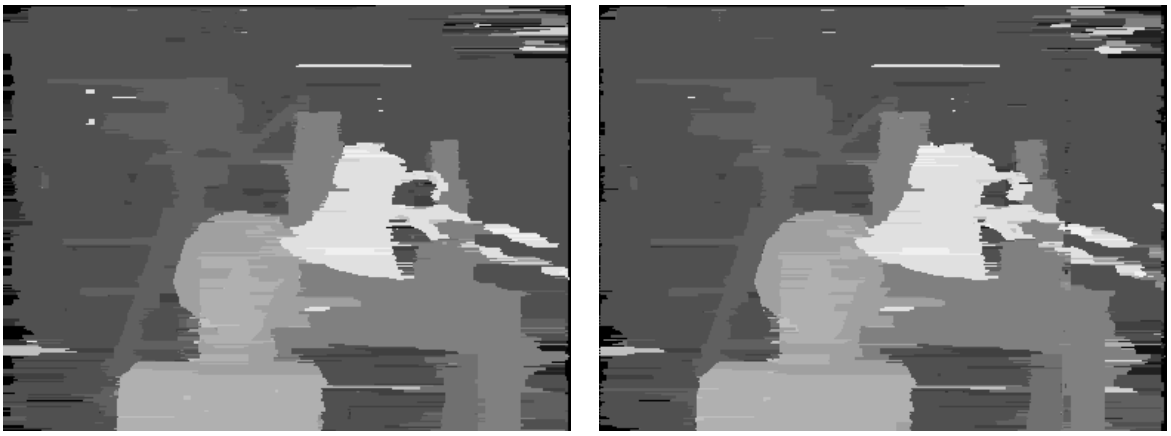
	Latency (cycles)	Memory usage (bits)
Disparity Buffers	D_{max}	$2\times2\times D_{max}\times \log_2 D_{max}$
Buffers Reading	2	0
Delay Line	2	0
Consistency Check	1	0
Bypass Buffer	$D_{max}+5 \quad (parallel)$	$D_{max}\times32$
Total	$D_{max}+5$	$(4\times \log_2 D_{max}+32)\times D_{max}$

Table 3.13: Consistency check latency and memory usage

Parameters	Current Value	Description
th_{cc}	2	reliability check threshold

Table 3.14: Consistency check configurable parameters

The depth maps refined by the consistency check is shown in Figure 3.24. Compared



(a) Tsukuba consistency check left depth map

(b) Tsukuba consistency check right depth map

Figure 3.24: Tsukuba left and right depth maps after consistency check

with the depth maps generated by dynamic programming in Figure 3.21, its quality is significantly improved by removing point noises caused by incorrect matching in occlusion areas. However the stripe noise is still there, and its removal is performed by voting.

3.3.7.2 Voting

The main function of voting is to select the most frequently used disparity in the support region where pixels have similar luminance. In our system it is used to remove the stripe noise to improve the quality of depth maps. Normally the voting is performed to build a histogram φ_p over all the disparities from 0 to $D_{max} - 1$ in a 2D region $U(p)$, which is called 2D voting, as shown in Equation 3.12 and Figure 3.25.

$$D(p) = \arg\max \varphi_p(d), \quad d \in [0, D_{max} - 1] \tag{3.12}$$

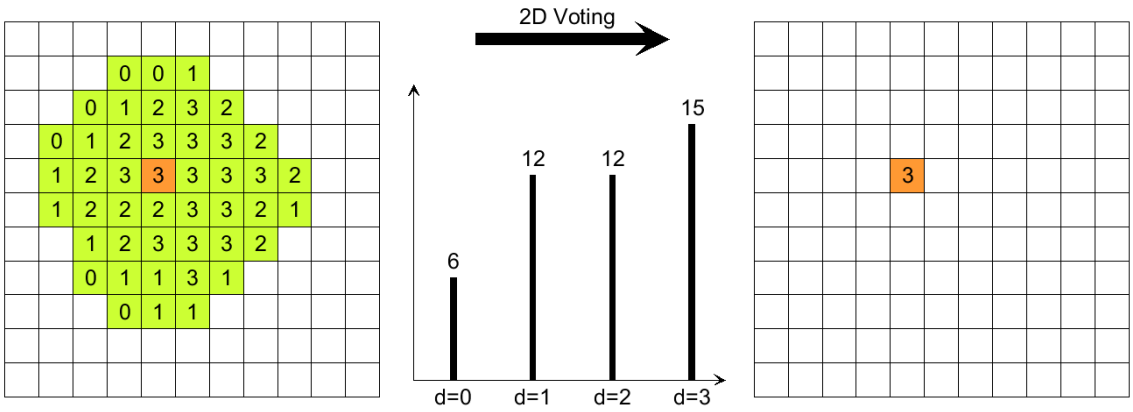


Figure 3.25: 2D Voting

However, building a 2D region histogram is not hardware friendly, which consumes a lot of memory resources and combinational logic resources because of parallelism. Therefore, we replace the 2D voting with a more serial 2×1D voting to save hardware resources and simplify the logic design, as shown in Figure 3.26. This simplification

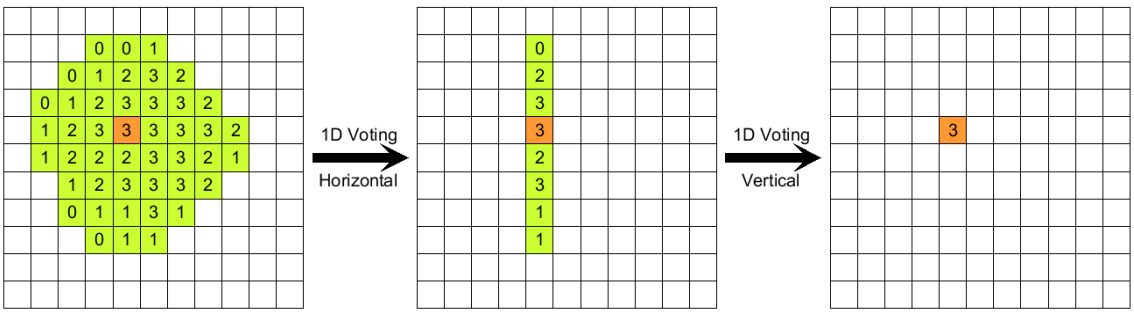


Figure 3.26: 2×1D Voting

divides voting into two separate votes in horizontal and vertical direction, which makes the voting design much simpler and generates not worse result than 2D voting, and the comparison is shown later.

3.3.7.3 Horizontal Voting

Horizontal voting is used to select the most frequently used disparity of one pixel in the horizontal arm length searching range. This kernel is composed by D_{max} parallel segments corresponding to all disparities, whose implementation is based on comparators, counters and registers, as shown in Figure 3.27. This segment is for disparity

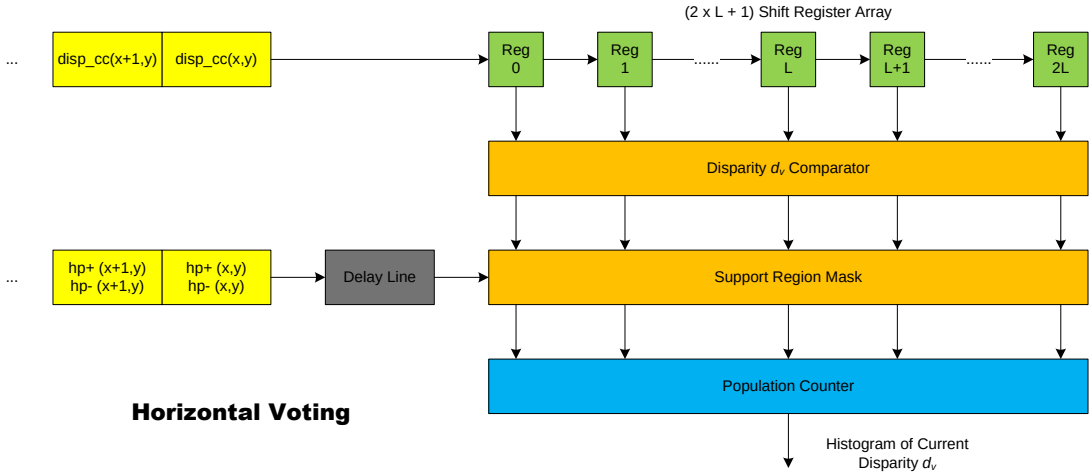


Figure 3.27: Horizontal voting RTL architecture

d_v. The shift registers store all the disparities in horizontal direction, and the range is $2 \times L + 1$ where L is the maximum arm length. Then all the disparities are compared with d_v to check whether they are equal or not to assign a corresponding flag value '1' or '0'. The support region mask is used to filter out the flag values out of the range of horizontal arm length, and the left candidates are passed to a parallel counter to compute the number of flag values '1', therefore the histogram value of disparity d_v is obtained. Then all the parallel histogram values corresponding to all the disparities are compared by a tree structure completer to select the largest one as the horizontal voting result. The delay line composed by shift registers is used to synchronize the horizontal arm lengths with the anchor pixel stored in Reg L to make sure the current arm lengths are the corresponding one.

The latency and memory usage of horizontal voting is listed in Table 3.15.

	Latency (cycles)	Memory usage (bits)
Shift Registers	L	0
Registers reading	1	0
Population Counters	2	0
Tree Structure Completer	4	0
Output Registers	1	0
Delay Line	$L + 1$ $(parallel)$	0
Total	$L + 8$	0

Table 3.15: Horizontal voting latency and memory usage

3.3.7.4 Vertical Voting

Vertical voting is used to choose the most frequently used disparity of one pixel in the vertical arm length searching range. Its main operation procedure is almost the same with horizontal voting, but one significant difference is the vertical voting needs many line buffers to provide disparities in vertical direction, because the disparities comes in scan-line order. The RTL architecture of vertical voting implementation is shown in Figure 3.28. $2\times L$ line buffers are used to provide vertical window, which is the same

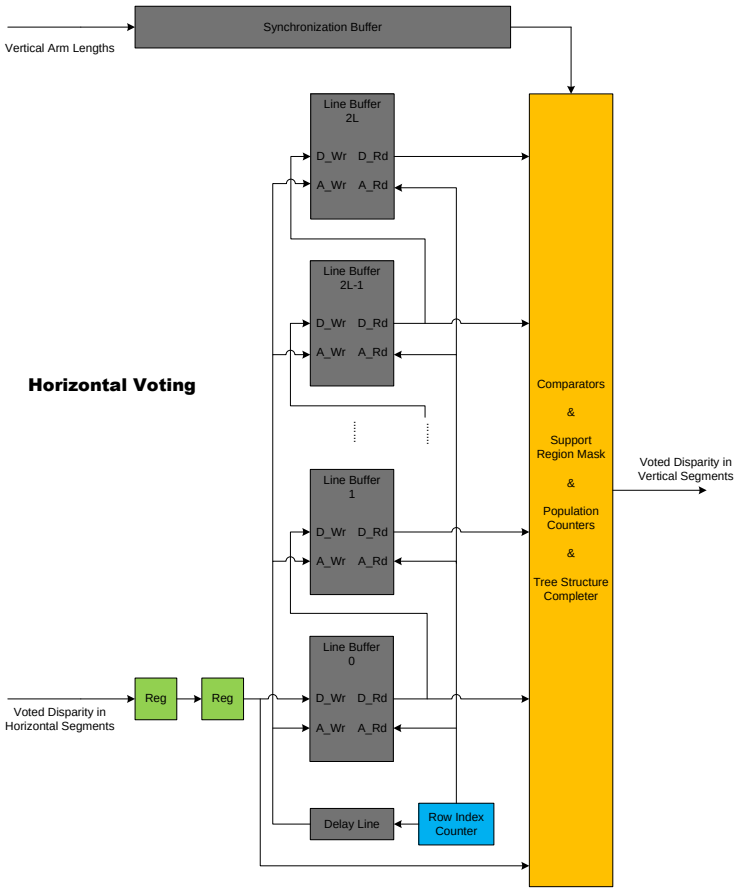


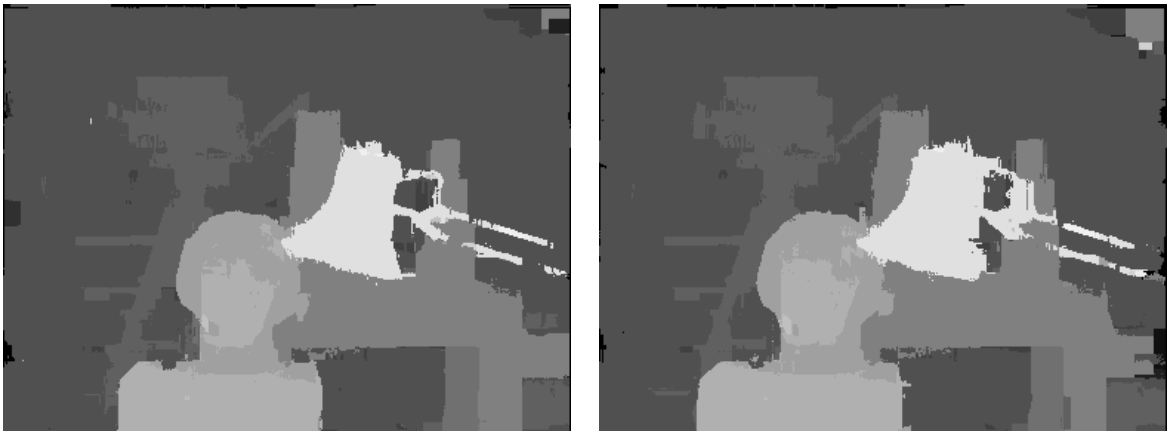
Figure 3.28: Vertical voting RTL architecture

structure with median filter architecture, and the processing part after fetching vertical disparities is the same with horizontal voting, including comparators, support region mask, population counters and tree structure completer. This vertical voting kernel requires a lot of memory as line buffers and the synchronization buffers which provides the delay of the synchronization between the anchor pixel stored in line buffer L and corresponding vertical arm length.

The latency and memory usage of vertical voting is listed in Table 3.16. The depth maps refined by horizontal voting and vertical voting are shown in Figure 3.29. Compared with the depth maps after consistency check in Figure 3.24, their stripe noises are removed by this 2×1D voting. Although the normal voting is performed

	Latency (cycles)	Memory usage (bits)
Line Buffers	$L \times FW$	$2 \times 2 \times L \times FW \times \log_2 D_{max}$
Buffers Reading	2	0
Population Counters	2	0
Tree Structure Completer	4	0
Output Registers	1	0
Synchronization Buffer	$L \times FW + 2$ $(parallel)$	$2 \times L \times FW \times 8$
Total	$L \times FW + 9$	$(16 + 4 \times \log_2 D_{max}) \times L \times FW$

Table 3.16: Vertical voting latency and memory usage

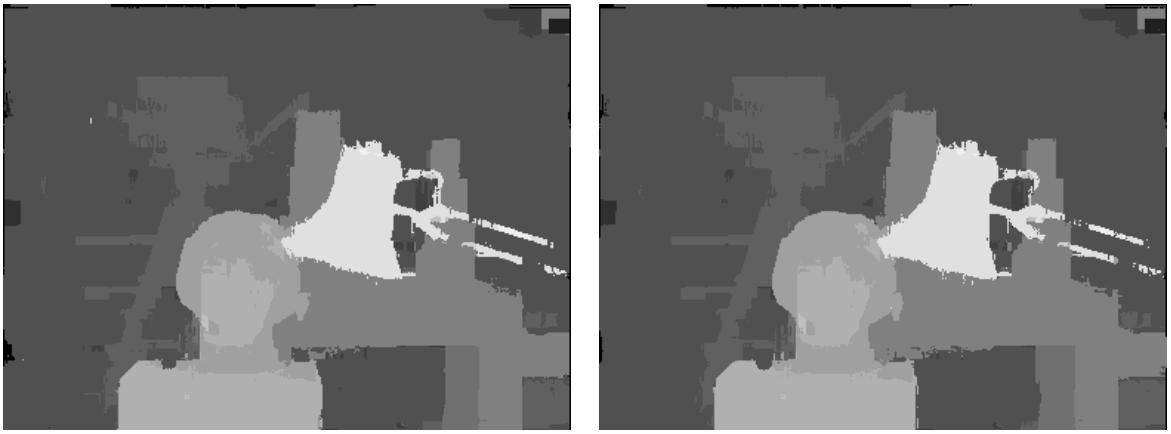


(a) Tsukuba voting left depth map

(b) Tsukuba voting right depth map

Figure 3.29: Tsukuba left and right depth maps after voting

in 2D region, this 2×1D voting is not worse than 2D voting, and their comparison is shown in Figure 3.30.



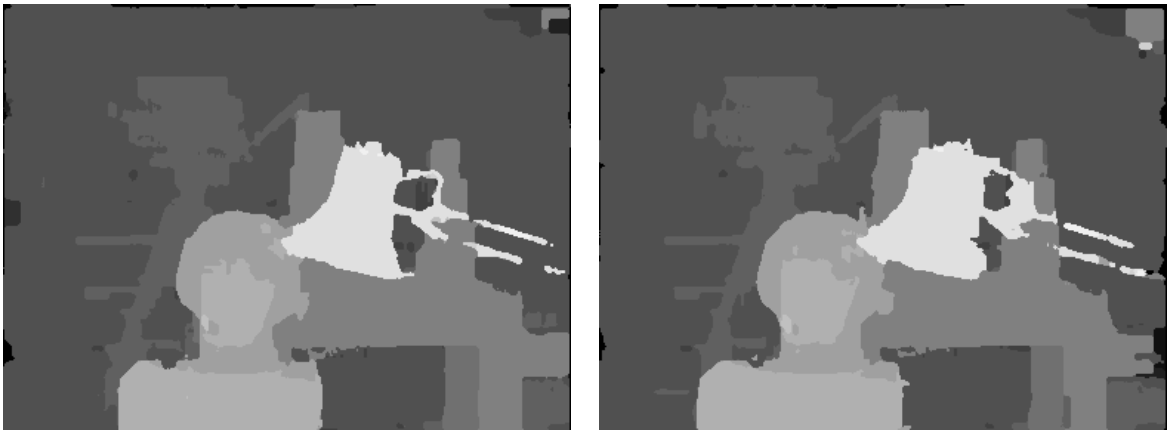
(a) 2×1D voting (error rate : 2.49%)

(b) 2D voting (error rate : 2.56%)

Figure 3.30: Comparison of 2×1D voting and 2D voting

3.3.7.5 Median Filter and Final Depth Map

The median filter in refinement is used to remove the spike noise in the depth maps refined by voting in order to increase the quality and reliability of depth maps. It is exactly the same with the median filter in 3.3.1, and it is the final step of refinement. The final depth maps are shown in Figure 3.31.



(a) Tsukuba final left depth map (error rate : 2.42%)

(b) Tsukuba final right depth map

Figure 3.31: Tsukuba final depth maps

Stereo Matching Proposed Hardware Design Evaluation

4

In this chapter we evaluate the modified hardware-friendly stereo matching algorithm and the implementation of stereo matcher on FPGA. The detailed comparison with state-of-the-art stereo matching implementations is presented. Section 4.1 presents hardware algorithm evaluations according to parameters configuration in each main function segments. Section 4.2 introduces stereo matcher implementation evaluations, including resources and timing. Section 4.3 compares our stereo matcher with other state-of-the-art implementations on every aspects.

4.1 Evaluation of Hardware Algorithm

Evaluating the modified hardware-friendly algorithm is mainly configuring parameters of each functional segments in the stereo matcher, including the support region builder, dynamic programming and refinement. All the algorithm configurable parameters are listed in Table 4.1. In the following parts, we finetune these parameters separately to

Segments	Parameters	Current Value	Description
SR Builder	L	15	maximum support region arm length
	τ	15	luminance difference threshold
DP	th_c	15	neighboring pixels continuity check threshold
	C	25(c.)/5(d.)	smooth cost for Potts model when $d \neq d'$
Refinement	th_{cc}	2	reliability check threshold

Table 4.1: Hardware algorithm configurable parameters

check their influence with the Middlebury stereo benchmark average error rate[38].

4.1.1 Support Region Builder Parameters

The configurable parameters of support region builder are listed in Table 4.2.

Segments	Parameters	Current Value	Description
SR Builder	L	15	maximum support region arm length
	τ	15	luminance difference threshold

Table 4.2: Support region builder configurable parameters

The parameter L controls the maximum support region size. By increasing it, the support region includes more neighboring pixel information, but has also more possibility to contain incorrect pixels and consumes more line buffers, and vice versa. The Middlebury separated and average error rate of four standard pictures are shown

in Table 4.3, and the curve of average error rate following the parameter L is shown in Figure 4.1. According to the chart we set $L = 15$.

L	Error Rate				
	Tsukuba	Venus	Teddy	Cones	**Average**
11	2.72	0.22	6.43	4.14	**6.69**
12	2.60	0.22	6.39	4.18	**6.65**
13	2.50	0.21	6.38	4.22	**6.63**
14	2.47	0.20	6.37	4.28	**6.61**
15	2.42	0.20	6.38	4.32	**6.60**
16	2.42	0.19	6.39	4.44	**6.63**
17	2.49	0.19	6.43	4.59	**6.72**
18	2.51	0.19	6.47	4.72	**6.78**
19	2.55	0.20	6.46	4.83	**6.82**

Table 4.3: Middlebury error rate and maximum support region arm length L

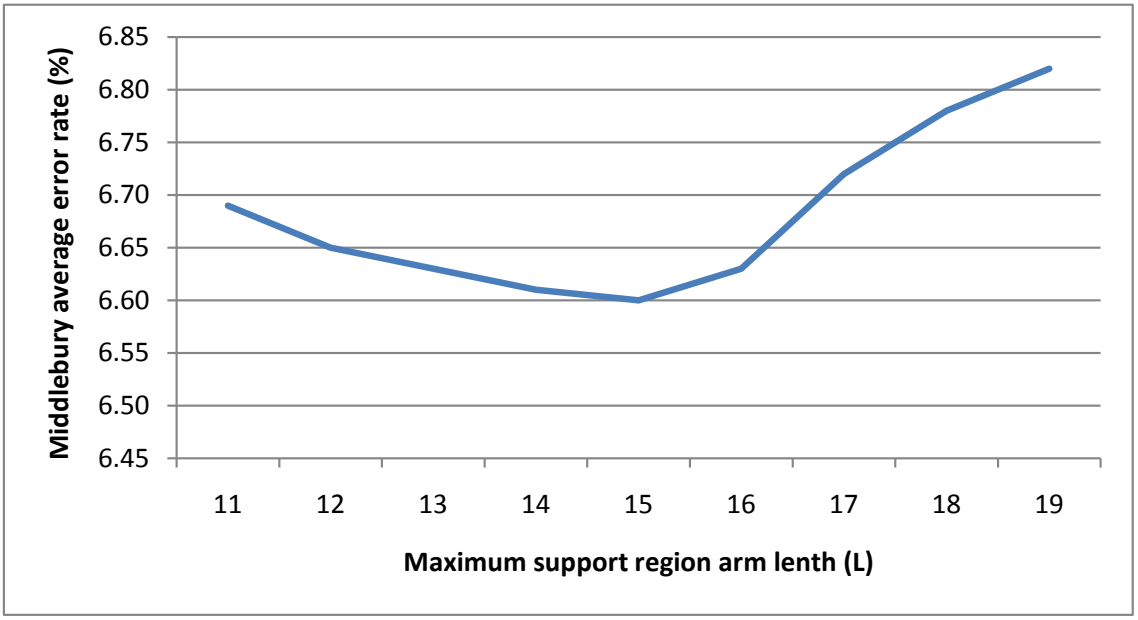


Figure 4.1: Middlebury error rate and maximum support region arm length L

The parameter τ is also used to control the support region, however it does not control the region size, but rather the candidate pixels. It sets a luminance difference threshold and builds the region consisting of similar luminance pixels. Therefore increasing this threshold, the support region includes more neighboring pixels and texture information but also includes more incorrect pixel values, and vice versa. The error rate following the change of parameter τ is shown in Table 4.4 and Figure 4.2. According to them we set $\tau = 15$.

The curves of L and τ is similar for their similar function, which is building the support region for refinement.

τ	Error Rate				
	Tsukuba	Venus	Teddy	Cones	**Average**
11	2.45	0.27	6.33	4.28	**6.65**
12	2.47	0.24	6.37	4.24	**6.63**
13	2.43	0.22	6.41	4.26	**6.62**
14	2.41	0.21	6.39	4.31	**6.60**
15	2.42	0.20	6.38	4.32	**6.60**
16	2.44	0.19	6.42	4.47	**6.67**
17	2.50	0.20	6.46	4.56	**6.77**
18	2.51	0.19	6.44	4.63	**6.78**
19	2.53	0.20	6.46	4.71	**6.83**

Table 4.4: Middlebury error rate and luminance difference threshold τ

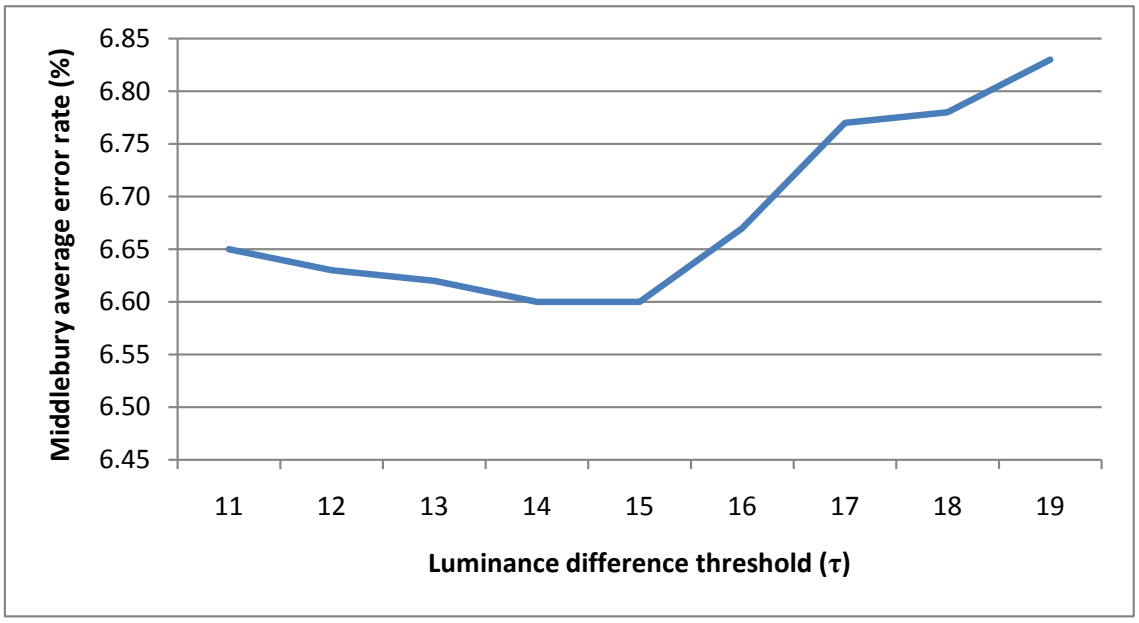


Figure 4.2: Middlebury error rate and luminance difference threshold τ

4.1.2 Dynamic Programming Parameters

The configurable parameters of dynamic programming are listed in Table 4.5.

Segments	Parameters	Current Value	Description
DP	th_c	15	neighboring pixels continuity check threshold
	C	25(c.)/5(d.)	smooth cost for Potts model when $d \neq d'$

Table 4.5: Dynamic programming configurable parameters

These two parameters are adopted to constrain the smoothness degree when choosing the smooth path in dynamic programming. If the smoothness degree is high, the continuous disparities (i.e. flat areas) processing is better. However, if the smoothness

degree is low, the discontinuous disparities (i.e. boundaries) processing is better. The ratio of continuous smooth cost and discontinuous smooth cost is 5, which is used to distinguish the continuous and discontinuous disparity areas, and the fine-tuning of this ratio has only slight difference on the result of differentiation, therefore we skip this unimportant parameter. Because the ratio is fixed, we use the discontinuous smooth cost as the index. The error rate versus the parameter th_c is shown in Table 4.6 and Figure 4.3, and the parameter C in Table 4.7 and Figure 4.4. According to the charts, we set $th_c = 15$ and $C = 25(c.)/5(d.)$.

th_c	Error Rate				
	Tsukuba	Venus	Teddy	Cones	**Average**
11	2.54	0.18	6.36	4.30	**6.63**
12	2.42	0.18	6.35	4.31	**6.61**
13	2.46	0.18	6.33	4.28	**6.60**
14	2.32	0.18	6.39	4.41	**6.61**
15	2.42	0.20	6.38	4.32	**6.60**
16	2.38	0.19	6.43	4.31	**6.62**
17	2.49	0.19	6.38	4.33	**6.65**
18	2.47	0.21	6.36	4.34	**6.64**
19	2.52	0.21	6.43	4.33	**6.71**

Table 4.6: Middlebury error rate and neighboring pixels continuity check threshold th_c

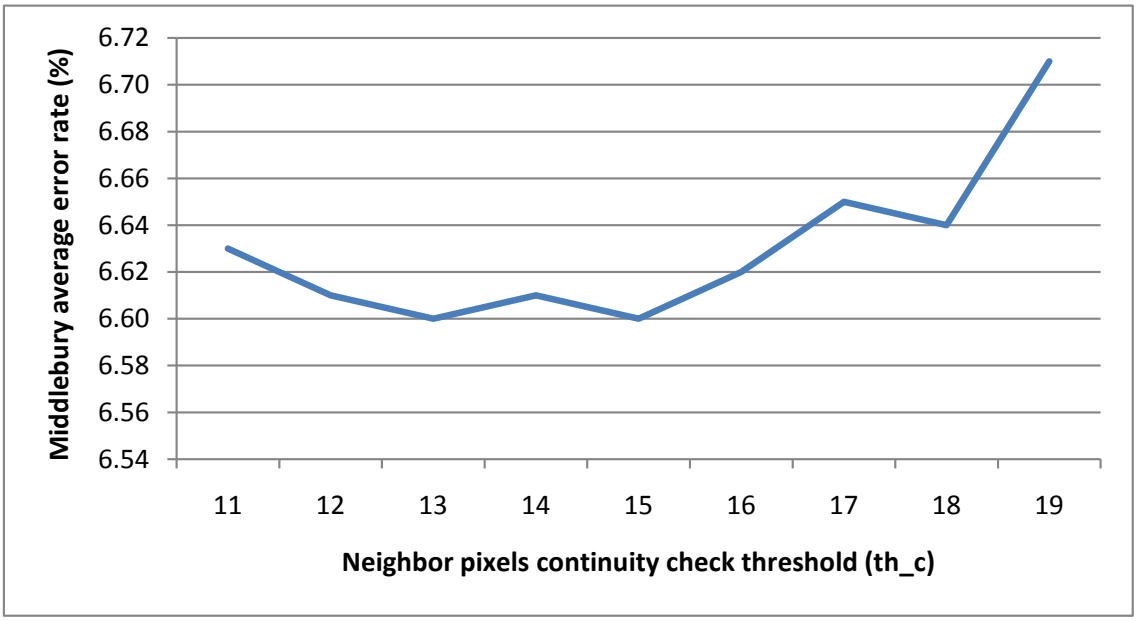


Figure 4.3: Middlebury error rate and neighboring pixels continuity check threshold th_c

Both increasing the smooth cost and decreasing the continuity check threshold can increase the smoothness of the optimized path, so these curves are correspondingly similar because of their similar function.

C	Error Rate				
	Tsukuba	Venus	Teddy	Cones	**Average**
1	2.56	0.21	6.79	4.38	**6.82**
2	2.39	0.23	6.60	4.22	**6.67**
3	2.44	0.21	6.43	4.26	**6.63**
4	2.36	0.21	6.42	4.32	**6.62**
5	2.42	0.20	6.38	4.32	**6.60**
6	2.44	0.18	6.34	4.36	**6.62**
7	2.57	0.17	6.23	4.43	**6.60**
8	2.54	0.18	6.22	4.51	**6.61**
9	2.63	0.19	6.22	4.54	**6.67**

Table 4.7: Middlebury error rate and and Potts model discontinuous smooth cost C

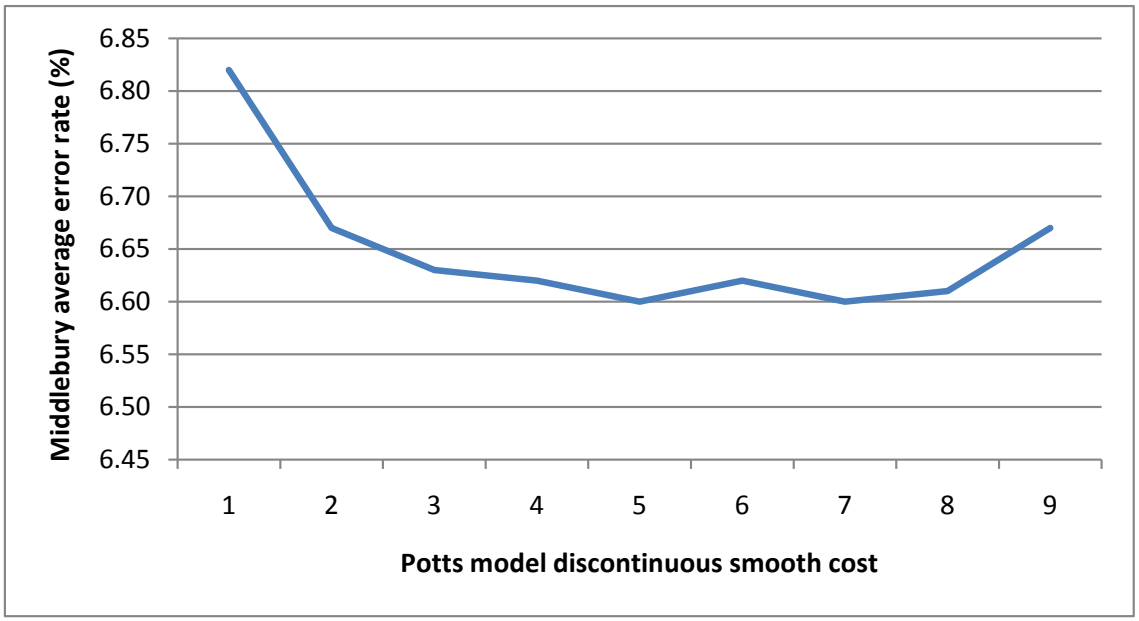


Figure 4.4: Middlebury error rate and Potts model discontinuous smooth cost C

4.1.3 Refinement Parameters

The configurable parameters of refinement are listed in Table 4.8.

Segments	Parameters	Current Value	Description
Refinement	th_{cc}	2	reliability check threshold

Table 4.8: Refinement configurable parameters

This parameter th_{cc} is utilized to obtain the occlusion map by judging disparity differences between stereo pairs. Because the disparity maps obtained by dynamic programming is based on the cost of neighboring pixels information, the disparities of corresponding pixels may have a slight difference, and if we take them as the incorrect

corresponding instances, we will obtain a worse result. Therefore this th_{cc} is set to increase the fault tolerance of stereo matching, and the error rate versus the parameter th_{cc} is shown in Table 4.9 and Figure 4.5. According to the chart, we set $th_{cc} = 2$.

th_{cc}	Error Rate				
	Tsukuba	Venus	Teddy	Cones	**Average**
1	2.57	0.20	6.37	4.42	**6.68**
2	2.42	0.20	6.38	4.32	**6.60**
3	2.38	0.19	6.39	4.33	**6.61**
4	2.54	0.23	6.39	4.35	**6.75**
5	2.54	0.24	6.46	4.36	**6.80**

Table 4.9: Middlebury error rate and reliability check threshold th_{cc}

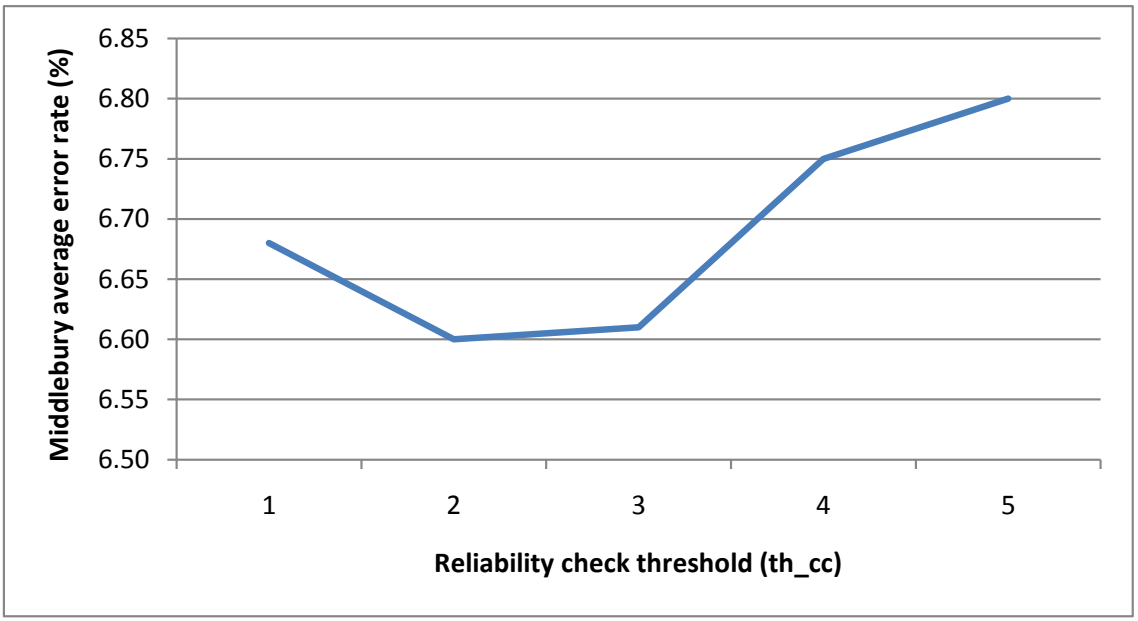
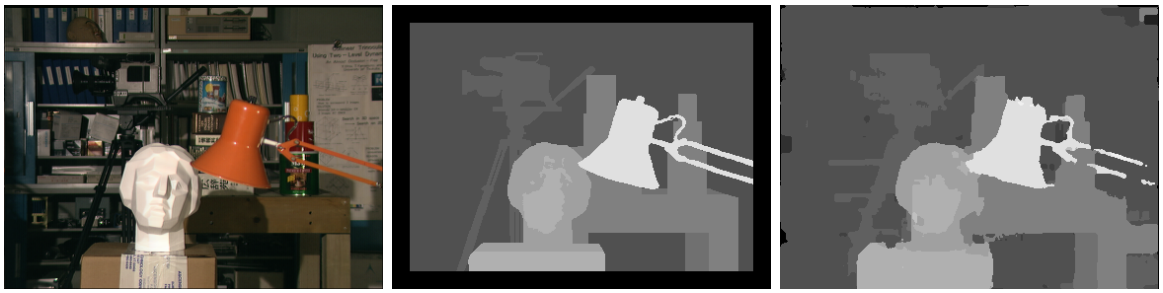


Figure 4.5: Middlebury error rate and reliability check threshold th_{cc}

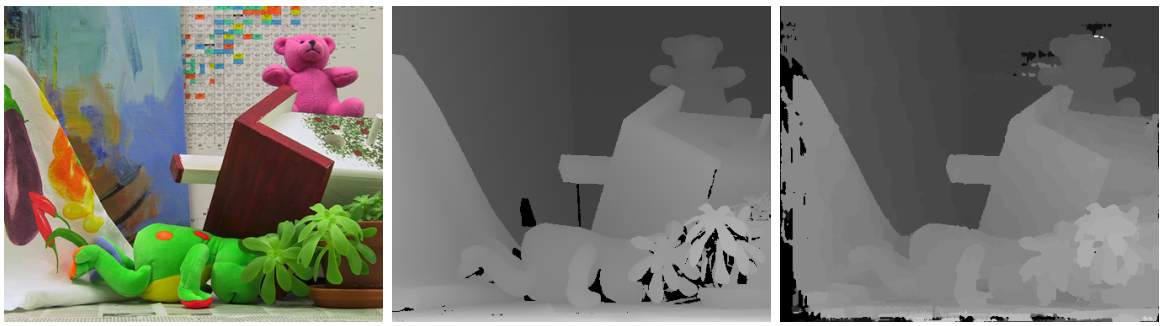
4.1.4 Final Middlebury Results

In a sense, the quality of stereo matching based on difference parameters depends on the texture of stereo images, therefore in the tables above tuning a certain parameter generates different error rate trends for different images. The four Middlebury standard images have various texture characters, i.e. complex objects in Tsukuba, bevel edges in Venus, arc-shaped edges in Teddy and sharp edges in Cones, so that the average Middlebury error rates reflects the common quality of the stereo matching in normal cases. Our final Middlebury results is shown in Figure 4.6.

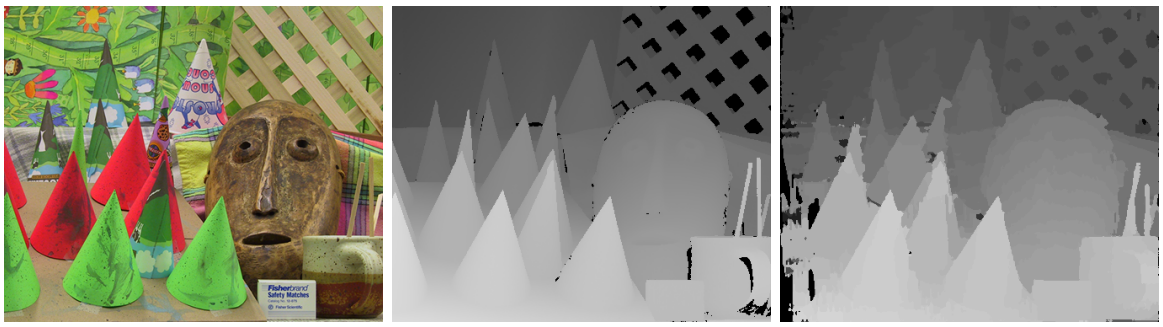


(a) Tsukuba (b) Truth disparity (c) Error rate 2.42%

(d) Venus (e) Truth disparity (f) Error rate 0.20%



(g) Teddy (h) Truth disparity (i) Error rate 6.38%



(j) Cones (k) Truth disparity (l) Error rate 4.32%

Figure 4.6: Final Middlebury benchmark results

4.2 Evaluation of Stereo Matching Implementation

4.2.1 FPGA Hardware Resources Utilization

Our implementation platform - FPGA EP3SL150 belongs to Altera 65nm Stratix III L family, and the basic building block of logic in the Stratix III architecture, the adaptive logic module (ALM), provides advanced features with efficient logic utilization. Each ALM contains a variety of look-up table (LUT) based resources that can be divided between two combinational adaptive LUTs (ALUTs) and two registers. With up to eight inputs to the two combinational ALUTs, one ALM can implement various combinations of two functions. One ALM can also implement any function of up to six inputs and certain seven-input functions. The high-level block diagram of Stratix III ALM is shown in Figure 4.7[8].

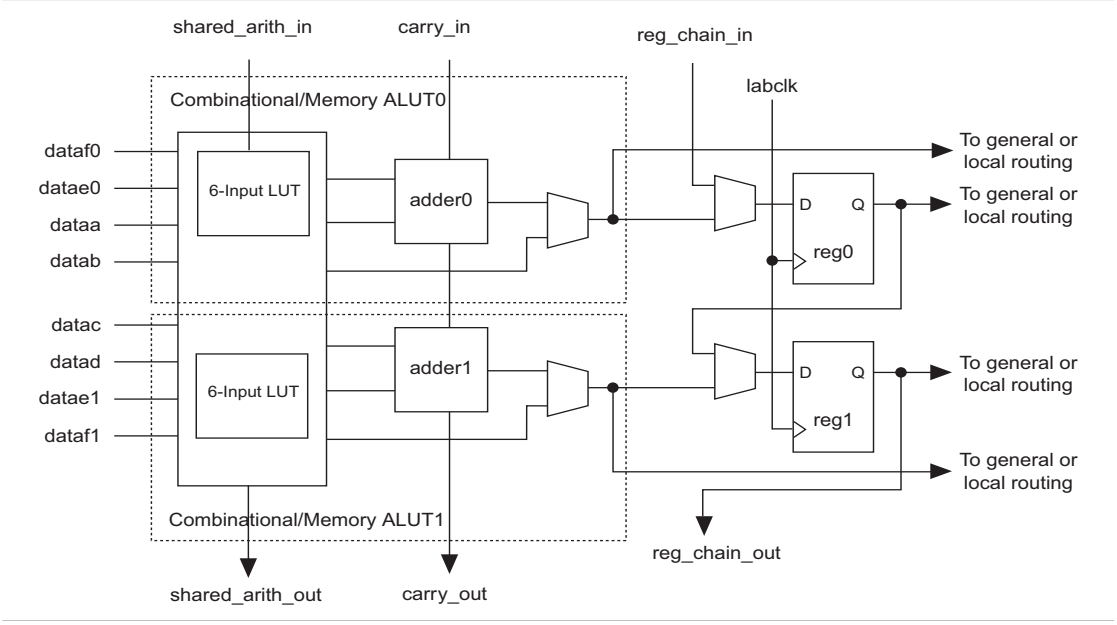


Figure 4.7: High-level block diagram of Stratix III ALM

The memory blocks on EP3SL150 FPGA are divided into two main kinds: memory logic array blocks (MLABs) and dedicated memory blocks (BRAMs). MLABs is a superset of the ALM including SRAM memory, so that it has all ALM characters. It is distributed over the entire FPGA, therefore it is very suitable for small delay lines and shift registers. There are two kinds of BRAMs, M9K and M144K, including 9K SRAM bits and 144K SRAM bits correspondingly. Both of them support the single port and dual port mode, and they are suitable for general purpose memory applications, i.g. Bypass FIFO, line buffer etc. The aspect ratios of all these memory blocks are listed in Table 4.10[7].

Based on the architecture of ALM and block memory, the resource utilization of our stereo matcher ($D_{max} = 64$) main kernels is listed in Table 4.11. The Table 4.11 lists the resource utilization when $D_{max} = 64$, however, the utilized hardware resources depends on the disparity range because the parallelism of dynamic programming is D_{max}. We

Feature	MLABs	M9K Blocks	M144K Blocks
Maximum Performance	600MHz	580MHz	580MHz
Aspect Ratios	16×8	$8K\times1$	$16K\times8$
	16×9	$4K\times2$	$16K\times9$
	16×10	$2K\times4$	$8K\times16$
	16×16	$1K\times8$	$8K\times18$
	16×18	$1K\times9$	$4K\times32$
	16×20	512×16	$4K\times36$
		512×18	$2K\times64$
		256×32	$2K\times72$
		256×36	

Table 4.10: EP3SL150 on-chip memory features

$D_{max} = 64$	**Combinational ALUTs**		**Memory ALUTs**		**Dedicated Logic Registers**		**Block Memory Bits**	
	Total: 113600	Util.	Total: 56800	Util.	Total: 113600	Util.	Total: 5630976	Util.
CT & SRB	2916	2.57%	0	0%	1451	1.28%	498704	8.86%
Bypass FIFO	24	0.02%	0	0%	24	0.02%	131072	2.33%
Raw Cost Scatter	6065	5.34%	0	0%	3991	3.51%	2672	0.05%
Dynamic Programming	16717	14.72%	0	0%	4452	3.92%	1630208	28.95%
Disparity Output Logic	9	0.01%	0	0%	38	0.03%	0	0%
Consistency Check	622	0.55%	1024	1.08%	1429	1.26%	0	0%
Horizontal Voting	10782	9.49%	0	0%	9168	8.07%	624	0.01%
Vertical Voting	10602	9.33%	0	0%	8704	7.69%	753664	13.38%
Median Filter	448	0.39%	0	0%	389	0.34%	32768	0.58%
Stereo Matcher	49376	43.46%	1024	1.80%	29703	26.15%	3049712	54.16%

Table 4.11: EP3SL150 hardware resource utilization summary

have also implemented $D_{max} = 16$ and $D_{max} = 32$ separately, and their corresponding resource utilization is shown in Figure 4.8. Because the computational complexity of dynamic programming is $O(D)$, which is the main logic and memory consumption part, so that the disparity scalability of the entire FPGA hardware resource is almost

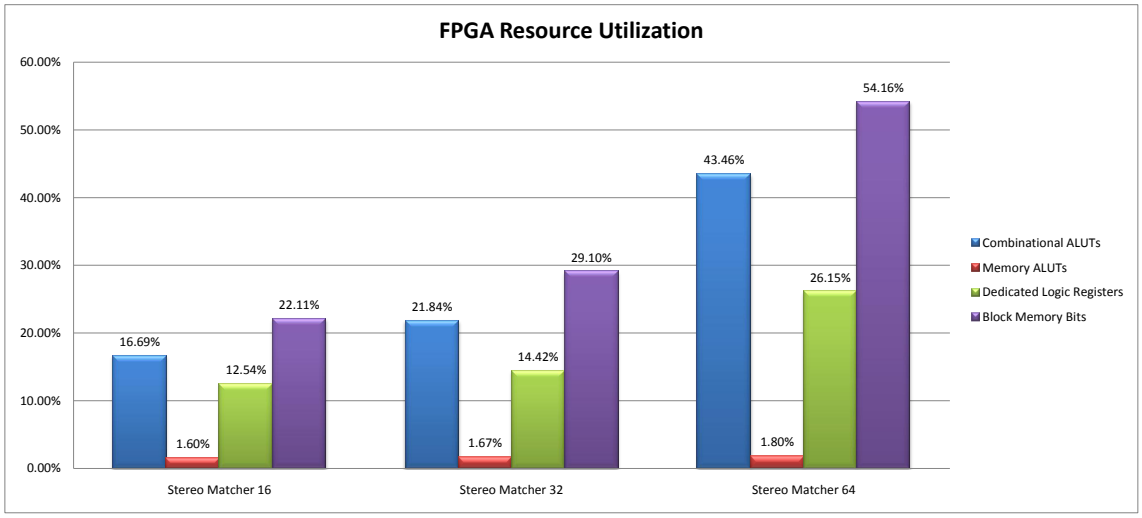


Figure 4.8: FPGA resource utilization of various disparities

linear, as shown in Figure 4.9. In practice, D_{max} is determined by the depth of objects

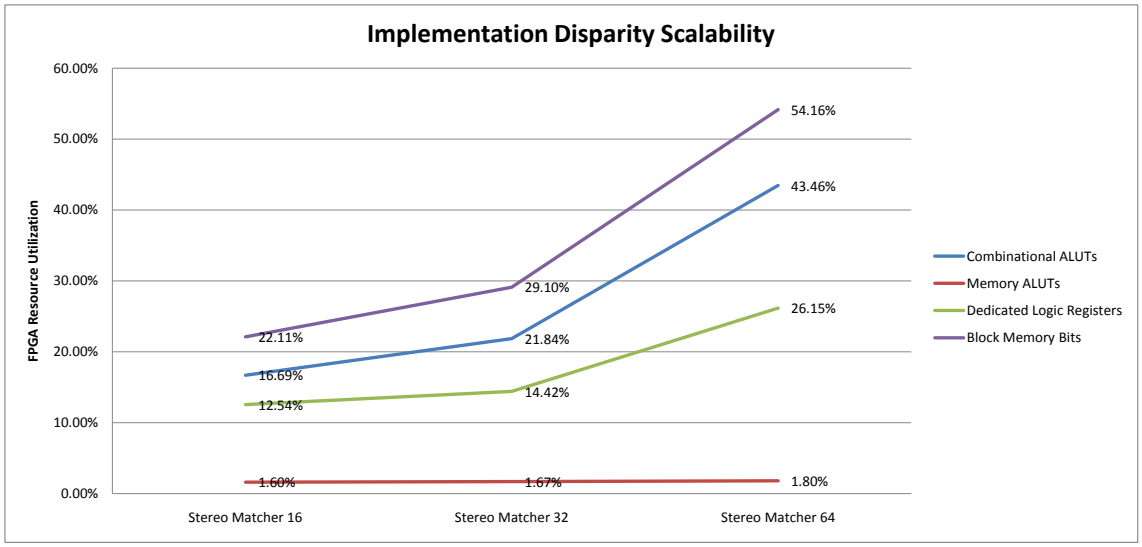


Figure 4.9: Stereo matcher implementation disparity scalability

and baseline length varies according to Equation 1.1, hence changing with various applications and stereo camera setup. Apart from D_{max}, the frame size also determines the resource utilization, especially for the memory consumption, which is shown in the memory usage tables in Chapter 3.

4.2.2 Real-Time Performance

The stereo matcher supports the real-time video in/out function because of its pipeline architecture, and the latency of the entire stereo matching is listed in Table 4.12, taking the current parameters and $D_{max} = 64$.

Kernels	Pipeline Latency	Frame Resolution	
		1024×768	**1920×1080**
CT & SRB	$15\times FW + 19$	15379	28819
Raw Cost Scatter	65	65	65
Dynamic Programming	$2\times FW + 7$	2055	3847
Disparity Output Logic	2	2	2
Consistency Check	69	69	69
Horizontal Voting	23	23	23
Vertical Voting	$15\times FW + 9$	15369	28809
Median Filter	$FW + 6$	1030	1926
Total	$33\times FW + 200$	33992	63560

Table 4.12: Stereo matcher 64 pipeline latency summary

However the real-time video source is not just pixels but blanking according to VGA signal timing format, as shown in Table 4.13, where the pixel clock required by each frame resolutions are also listed. From the report of Synplify, the worst path of stereo

Resolution	**800×600**	**1024×768**	**1280×720**	**1280×1024**	**1920×1080**
Refresh Rate (Hz)	75	60	60	60	60
Pixel Clock (MHz)	49.5	65	74.2	108	182.5
H Sync Pulse	80	136	80	112	696
H Front Porch	16	24	72	48	32
H Back Porch	160	160	216	248	32
V Sync Pulse	2	6	5	3	11
V Front Porch	1	3	3	1	22
V Back Porch	21	29	22	38	22

Table 4.13: VGA signal timing format for various resolutions

matcher located in the loop of the first step of dynamic programming, whose estimated frequency is 72.4MHz. Therefore under the condition of running pixel clock on FPGA, our real-time stereo matcher works on a resolution not higher than 1024×768 with 33 lines and 200 pixels latency without reorder optimization, which has already fulfill the requirement of real-time high-definition video.

4.3 Comparison with Related Instances

As mentioned in Chapter 1, the Middlebury stereo benchmark evaluation website[38] provides the error rate of *all* the pixels in the entire depth map. In addition, Middlebury evaluation also provides *nonocc* and *disc* error rate results, which stands for the non-occlusion areas and discontinue areas, and the former is always located near occluded regions that is the most tricky areas for the stereo matching algorithm.

The benchmarked evaluation of our implementation and some other related instances are listed in Table 4.14. However, the Middlebury benchmark evaluation does

Stereo Matching Error Rates (%)													
Images	*Tsukuba*			*Venus*			*Teddy*			*Cones*			***Average Bad Pixel Rate***
Image Size	*384×288*			*434×383*			*450×375*			*450×375*			
Disparity Range	*16*			*20*			*60*			*60*			
Evaluation	*nonocc*	*all*	*disc*	*nonocc*	*all*	*disc*	*nonocc*	*all*	*disc*	*nonocc*	*all*	*disc*	
Our Method	**2.42**	**2.78**	**10.9**	**0.20**	**0.47**	**2.07**	**6.38**	**11.7**	**16.1**	**4.32**	**10.6**	**11.3**	**6.60**
VariableCross[44]	1.99	2.65	6.77	0.62	0.96	3.20	9.75	15.1	18.2	6.28	12.7	12.9	**7.60**
RealtimeBFV[46]	1.71	2.22	6.74	0.55	0.87	2.88	9.90	15.0	19.5	6.66	12.3	13.4	**7.65**
RealtimeBP[42]	1.49	3.40	7.87	0.77	1.90	9.00	8.72	13.2	17.2	4.61	11.6	12.4	**7.69**
Chang et al. 2010[4]	N/A	2.80	N/A	N/A	0.64	N/A	N/A	13.7	N/A	N/A	10.1	N/A	**N/A**
Zhang el al. 2010[47]	3.84	4.34	14.2	1.20	1.68	5.62	7.17	12.6	17.4	5.41	11.0	13.9	**8.20**
FastAggreg[37]	1.16	2.11	6.06	4.03	4.75	6.43	9.04	15.2	20.2	5.37	12.6	11.9	**8.24**
OptimizedDP[34]	1.97	3.78	9.80	3.33	4.74	13.0	6.53	13.9	16.6	5.17	13.7	13.4	**8.83**
RealtimeVar[24]	3.33	5.48	16.8	1.15	2.35	12.8	6.18	13.1	17.3	4.66	11.7	13.7	**9.05**
RTCensus[17]	5.08	6.25	19.2	1.58	2.42	14.2	7.96	13.8	20.3	4.10	9.54	12.2	**9.73**
RealTimeGPU[40]	2.05	4.22	10.6	1.92	2.98	20.3	7.23	14.4	17.6	6.41	13.7	16.5	**9.82**
Jin et al. 2010[20]	9.79	11.56	20.29	3.59	5.27	36.82	12.5	21.5	30.57	7.34	17.58	21.01	**17.24**
Chang et al. 2007[3]	21.5	21.7	48.7	16.5	17.8	29.9	26.3	33.6	35.1	24.2	32.4	31.0	**N/A**

Table 4.14: Stereo matching algorithm Middlebury benchmark evaluation

not include the processing speed evaluation although it provides the fair comparison between various algorithms. Therefore the concept of million disparity evaluation per second (MDE/s) is imported to measure the processing speed of different systems implementation, as computed in Equation 4.1.

$$MDE/s = frame\ width \times frame\ height \times D_{max} \times FPS \tag{4.1}$$

The processing speed comparisons between our proposed implementation and other reported implementations are listed in Table 4.15.

Stereo Matching Implementation Processing Speed				
Algorithm	**Implementation**	**D_{max}**	**Frame Rate**	**MDE/s**
Jin et al. 2010[20]	FPGA Virtex-4	64	640×480@230	4521
Our Method	**FPGA Stratix III**	**64**	**1024×768@60**	**3019**
Zhang el al. 2010[47]	FPGA Stratix III	64	1024×768@60	3019
RTCensus[17]	GPU Geforce GTX 280	60	450×375@105.4	1067
Chang et al. 2010[4]	ASIC UMC 90nm	64	352×288@42	272
RealtimeBFV[46]	GPU GeForce GTX 8800	64	450×375@12	129
Chang et al. 2007[3]	DSP TMS320C64x	60	450×375@9.1	92
RealTimeGPU[40]	GPU Radeon XL1800	16	320×240@43	53
RealtimeVar[24]	CPU Pentium 2.83GHz	60	450×375@3.5	35
RealtimeBP[42]	GPU GeForce GTX 7900	16	320×240@16	20
FastAggreg[37]	CPU Core Duo 2.14GHz	60	450×375@1.67	17
OptimizedDP[34]	PC 1.8GHz	60	450×375@1.25	13
VariableCross[44]	CPU Pentium IV 3.0GHz	60	450×375@1.21	13

Table 4.15: Stereo matching implementation processing speed evaluation

The three FPGA platform implementations are all fully pipelined designs, therefore there frame rates are not limited by the computing pipeline itself. Zhang's work[47] and our proposed implementation both achieve real-time high-definition performance and better stereo matching accuracy than Jin's work[20]. Furthermore, our implemen-

tation has higher accuracy than Zhang's work, as shown in Table 4.15. In addition our implementation consumes less hardware resources than Zhang's work, as shown in Table 4.16, where the comparison is performed by percentage because both Zhang's implementation and our implementation are using the same FPGA Stratix III EP3SL150.

	Combinational ALUTs	Memory ALUTs	Registers	DSPs	SRAM
Our Method	**44%**	**2%**	**26%**	**0%**	**54%**
Zhang el al. 2010[47]	52%	36%	84%	67%	67%

Table 4.16: Hardware resource utilization between our implementation and Zhang

For FPGA implementations the limitation is hardware resources, e.g. on-chip memories and combinational ALUTs, and for CPU and GPU implementations the limitation is serial computation procedure, therefore FPGA platforms often have higher processing performance at lower clock speed, while CPU and GPU platforms often have more highly accurate results. Our implementation achieves very high frame rates compared with most of the other implementations as well as highly accurate results. To the best of our knowledge, our implementation has achieved the fastest processing performance on high-definition images with high accuracy.

A Real-Time View Synthesis System with Stereo Matching on FPGA

5

The system implemented on FPGA is our proposed *3D Depth Range Adjustment System*. The technique used for adjusting the depth range is depth scaling, where a new left and right stereo image pair is calculated and rendered on the (anaglyph or polaroid glasses) 3D display. The original left and a new right image corresponding to an intermediate viewpoint is created, reducing the 3D effect. Consequently depth scaling needs the depth information between stereo pairs, requiring a stereo matching technique to create a pixel/object displacement/re-positioning which is inverse proportional to depth. The original RGB stereo frames have to be stored in a DDR2 SDRAM and its scheduler is adopted to provide a large storage space and data exchange synchronization. The processing pipeline therefore consists of view synthesis, stereo matcher and a DDR2 SDRAM access scheduler which will be further described in this chapter.

The structure of this chapter is organized as follows: Section 5.1 presents the on-chip system architecture based on the Stratix III FPGA[8] and introduces its function kernels briefly. Section 5.2 presents the view synthesis kernel. Section 5.3 introduces the DDR2 SDRAM access scheduler and estimates our DDR2 SDRAM bandwidth usage.

5.1 Real-Time System Architecture on FPGA

The real-time on-chip system architecture includes stereo matcher, view synthesis, DDR2 SDRAM access scheduler and other processing modules, as shown in the dashed line rectangle in Figure 5.1. The function of each segment in this system is as following:

- VI Adaptor/Sync makes the output pixels start at the first pixel of one frame as well as synchronize the left and right pixels.
- RGB2YUV422 is the function block which convert RGB value of one pixel into YUV422 format.
- Stereo Matcher processes the luminance (Y) value from RGB2YUV422 and the current frame luminance, previous frame luminance and previous frame depth map to generate current frame depth map with timing consistency which makes the video more consistent between current frame depth map and previous frame depth map. Finally the stereo matcher generates left and right depth map.
- VS Adaptor (View Synthesis Adaptor) is used between the stereo matcher and view synthesis to synchronize the depth maps and original YUV frames. Because the depth map of stereo matcher is separated by blanking signals according to VGA signal timing format. This VS Adaptor is used to adjust the timing of depth map to suit the requirement of view synthesis.

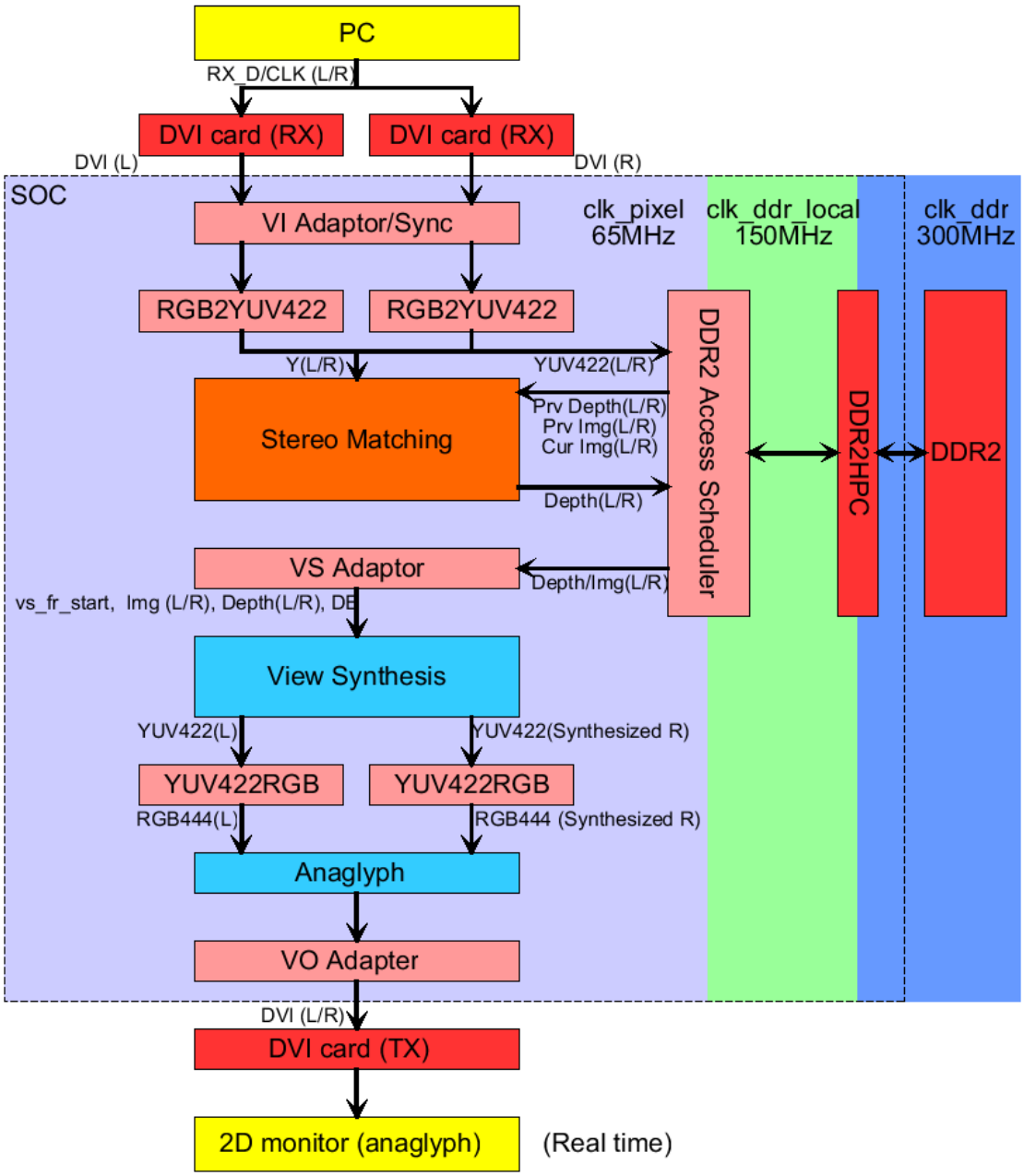


Figure 5.1: On-chip system architecture

- View Synthesis takes the depth maps from stereo matcher, original YUV frames from DDR2 and some extra control signals from VS Adaptor to generate the 3D depth range adjustable interpolated frames, and the scaling range can be configured by buttons on the FPGA board.

- DDR2 SDRAM access scheduler is used as a mini custom DMA to provide our on-chip system a platform to write and read multiple stream data simultaneously without delay. It takes original image value as well as depth maps and offers

image value and depth maps simultaneously.

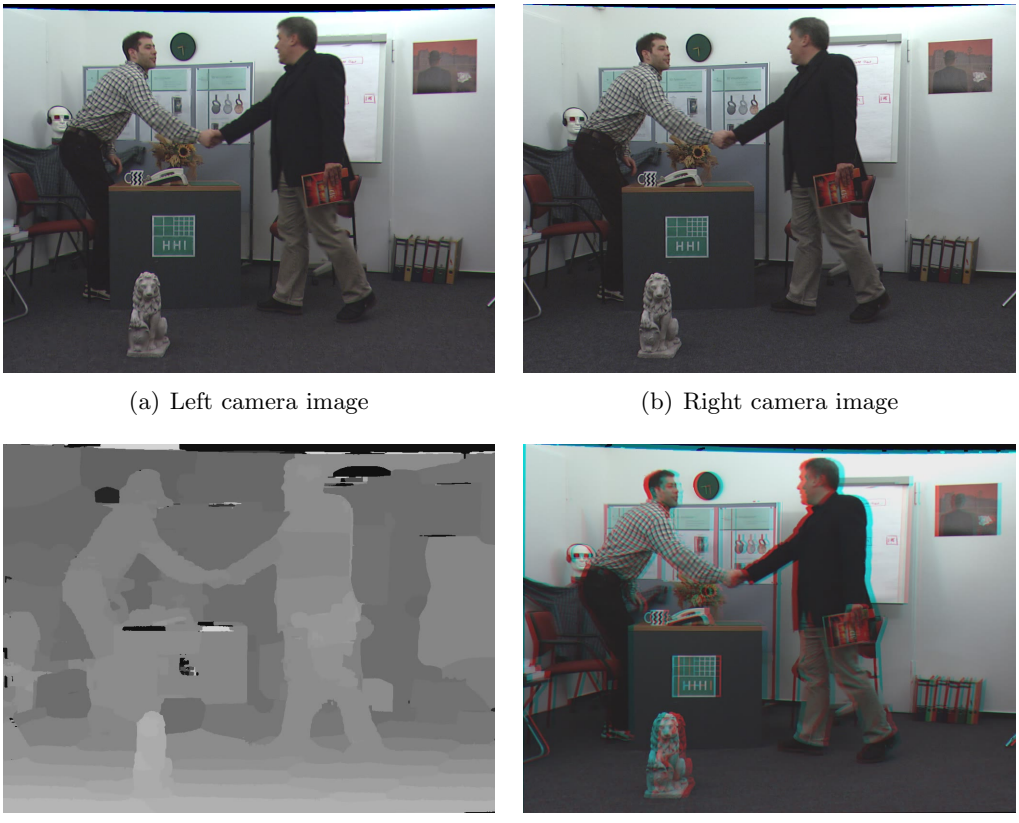
- YUV422RGB is used to convert YUV422 format video signals back into RGB signals for further use.

- Anaglyph is the converter that changes RGB video signals to anaglyph 2D image, which shows the 3D effect with red-cyan glasses.

- VO Adaptor generates the video timing signals for display.

There are various clock domains in our on-chip system, as also shown in Figure 5.1. The main kernels run at 65MHz to meet the real-time video requirements. The interface between DDR2 SDRAM access scheduler and DDR2 high performance controller[6] operates at 150MHz (half rate of DDR2) to improve the critical path. Currently our bandwidth usage is less than a quarter of the total available DDR2 bandwidth of 300 MHz maximum clock rate.

Our final depth map and interpolated anaglyph frame together with original left and right input frames are shown in Figure 5.2 for the typical HHI test sequences.



(a) Left camera image

(b) Right camera image

(c) Depth map

(d) Interpolated anaglyph image

Figure 5.2: Depth map and interpolated anaglyph frame with original frames

5.2 View Synthesis

View synthesis acts the role of adjusting the 3D depth range in our system, and its design and implementation is performed by NCTU and Imec-Taiwan. The high level structure of view synthesis is shown in Figure 5.3. Left and right camera parameters

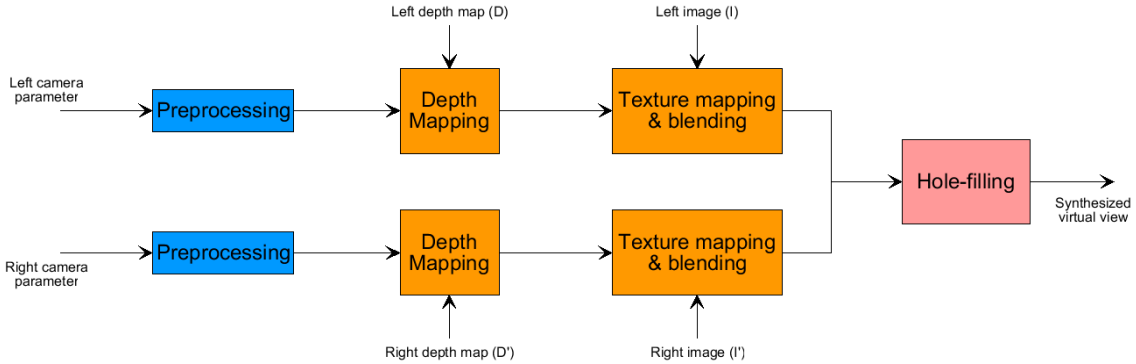


Figure 5.3: View synthesis architecture

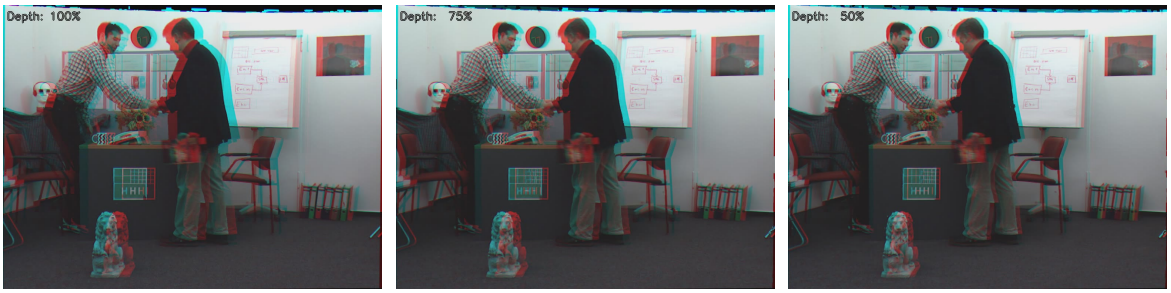
are used to warp the depth maps obtained from stereo matcher, and to generate the interpolation frame towards the target viewpoint by using pixel color (texture) information. Pixels around the object edges are often in occlusion region, which could only be seen from either the left image or the right one, and they are handled by the last hole-filling module.

The view synthesis is used to configure the 3D depth range from full 3D (anaglyph 100%) to full 2D (anaglyph 0%) gradually based on the depth map generated by stereo matcher, as shown in Figure 5.4.

5.3 DDR2 SDRAM Access Scheduler

There are three levels in hierarchical storage system, including register level, on-chip SRAM level and off-chip SDRAM level. Registers are the fastest storage units which promise zero latency for both reading and writing, therefore in our system they are used for holding the time critical data, and sometimes it is convenient using shift registers as short FIFOs. The on-chip SRAMs are slower than registers but support large data storage, thus they are used as line buffers and long bypass FIFOs in the system. Compared with them, off-chip SDRAMs are the slowest ones but provide very huge data storage, so that in our system they are used as image buffers and depth map buffers.

The off-chip SDRAM we used is the external DDR2 SDRAM which is a 1GB DDR2-667 dual-rank SO-DIMM module. The DDR2 SDRAM Access Scheduler allows the whole system kernels to get access to DDR2 SDRAM easily and simultaneously with zero delay, and it behaves like a huge on-chip memory with the pre-fetch technique. However, because SDRAM has internal access latencies, the DDR2 SDRAM cannot work as efficient as the on-chip RAMs. This kernel is implemented by Hsiu-Chi Yeh



(a) Anaglyph 100% (b) Anaglyph 75% (c) Anaglyph 50%

(d) Anaglyph 25% (e) Anaglyph 12% (f) Anaglyph 0%

Figure 5.4: 3D depth range adjustment from full 3D to full 2D

who is also from Delft University of Technology for MSc thesis in Imec, and the detailed high-level architecture of DDR2 SDRAM access scheduler is shown in Figure 5.5. It serves several reading and writing tasks simultaneously:

- Writing the left and right original images in YUV422 format
- Reading the left and right previous frame images and current frame images in luminance as well as the left and right previous depth maps for timing consistency in stereo matching
- Writing the left and right depth maps generated by the stereo matcher
- Reading the left and right current frame images in YUV422 format as well as the left and right current depth maps for view synthesis

The arbiter here is a complex manager to balance the timing and throughput of each task to make sure there is no conflicts between tasks. The DDR2 SDRAM is working at 300MHz which is high enough for the data transfer in our system, and the interface between scheduler and DDR2HPC is working at 150MHz which is half the rate of DDR2 SDRAM frequency to reduce power consumption. Therefore the data reading and writing is much faster than the SoC working frequency 65MHz, and this feature makes it possible to transfer a large mount of data at the same time. However there is efficiency problems when transferring the data. The worst efficiency is frequently switching between short reads and short writes, which causes the memory address access the same row on every transfer[6]. Therefore it is efficient to utilize the technique of

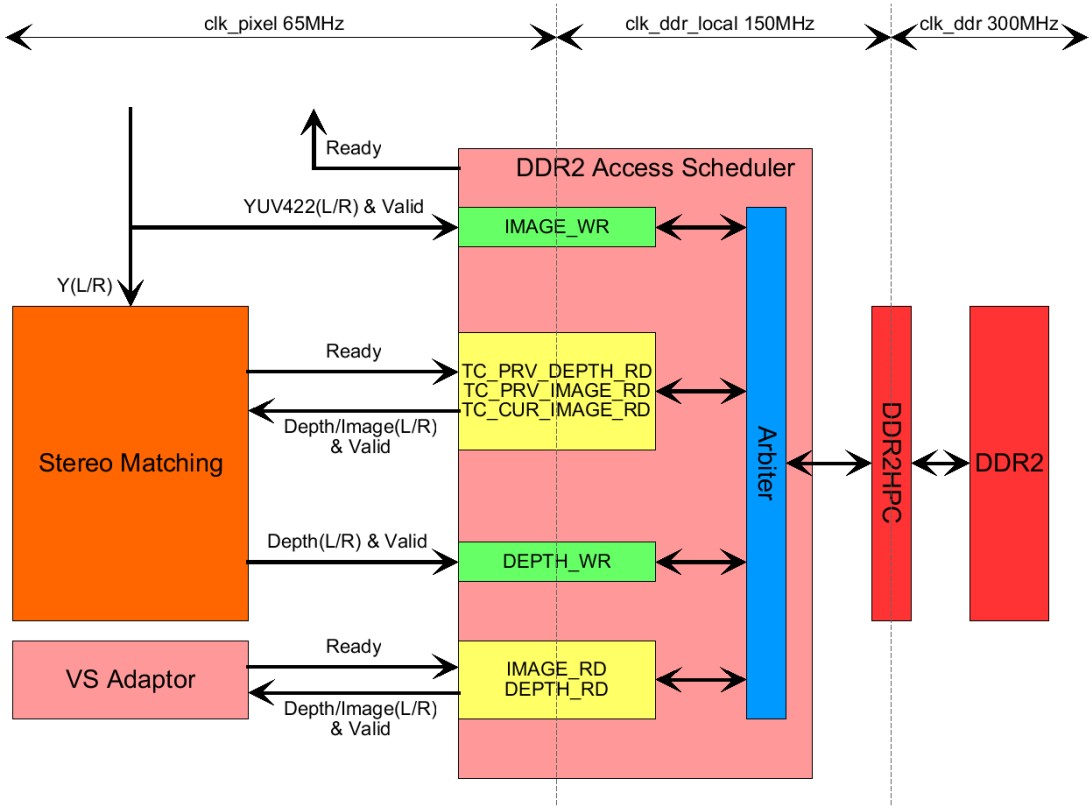


Figure 5.5: DDR2 SDRAM access scheduler architecture

long read and write bursts to continually access the address space. This architecture utilizes that technique, for DDR2 access scheduler packages the multiple stream data in its buffers and bursts writing them to DDR2, meanwhile it pre-fetches stream data from DDR2 in burst package and readies to serve any block requests, realizing long read and write bursts.

DDR2 SDRAM supports twice higher bus speed as well as dual rate of the bus speed caused by transferring data at both the rising and falling clock edge, therefore the DDR2 SDRAM provides the bandwidth in Equation 5.1.

$$Bandwidth = memory\ clock\ rate \times bus\ clock\ multiplier \times dual\ rate \times burst\ data\ width \tag{5.1}$$

Assuming the data burst is 64 bits at one time, and the memory clock frequency is 300MHz, then the theoretical maximum data bandwidth is:

$$Bandwidth = 300M/s \times 2 \times 2 \times 64bits = 76.8Gbits/s \tag{5.2}$$

In our system architecture, the writing and reading blocks are listed above to consume bandwidth. The bandwidth consumption of one reading or writing buffer is:

$$\begin{aligned} Bandwidth_{R/W} = FPS \times required\ frame\ number \\ \times\ frame\ width \times frame\ height \times data\ width \end{aligned} \tag{5.3}$$

The *required frame number* is two for left and right frames. Therefore the YUV422 image reading and writing buffers consume the same bandwidth:

$$Bandwidth_{16bits} = FPS \times 2 \times frame\ width \times frame\ height \times 16bits \tag{5.4}$$

while the depth and luminance image reading and writing buffers consume the same bandwidth:

$$Bandwidth_{8bits} = FPS \times 2 \times frame\ width \times frame\ height \times 8bits \tag{5.5}$$

Assuming our real-time video is 1024×768@60FPS, then the results are:

$$\begin{cases} Bandwidth_{16bits} = 60/s \times 2 \times 1024 \times 768 \times 16bits = 1.51Gbits/s \\ Bandwidth_{8bits} = 60/s \times 2 \times 1024 \times 768 \times 8bits = 0.755Gbits/s \end{cases} \tag{5.6}$$

There are two YUV422 image buffers and five depth and luminance image buffers in all, so that the entire bandwidth usage of our system is:

$$\begin{aligned} Bandwidth\ usage &= \frac{2 \times Bandwidth_{16bits} + 5 \times Bandwidth_{8bits}}{Bandwidth} \\ &= \frac{2 \times 1.51Gbits/s + 5 \times 0.755Gbits/s}{76.8Gbits/s} = 8.85\% \end{aligned} \tag{5.7}$$

6 Conclusions and Future Work

6.1 Conclusions

In this thesis, an hardware-efficient stereo matching algorithm is proposed based on Dynamic Programming, Variable Cross and Mini-Census Transform algorithms. These algorithms are implemented on an FPGA based on a full pipeline architecture. We utilize the stereo matcher as a kernel together with View Synthesis kernel and DDR2 SDRAM Access Scheduler kernel to build a real-time 3D depth range adjustment system. The test results of the experiments demonstrate that our stereo matcher has reached high speed real-time processing with various video resolutions while keeping highly accurate results. In detail our stereo matcher achieves the known fastest processing speed for high-definition resolution $1024\times768@60FPS$ and preserving excellent Middlebury benchmark evaluation average error rate of 6.60%. Therefore our implemented stereo matcher on FPGA has fulfilled our original design targets set up in Section 2.4.

6.2 Chapters Summary and Contributions

The 3D depth range adjustment application system is our motivation for performing the stereo matching hardware algorithm and real-time implementation. This system is utilized to satisfy personal visual comfort when watching 3D TV for a long time, as presented in Chapter 1. The major contribution of this thesis is that we have scheduled the computational loop of Scan-Line Dynamic Programming and realized the parallel structure of this serial Dynamic Programming, while preserving excellent accuracy.

In Chapter 2, we presented various stereo matching algorithms and different implementation platforms, including high performance CPU, GPU, DSP, FPGA and ASIC. However these platforms have their own advantages and disadvantages, and are not suitable for both highly accurate and real-time requirements, especially for high-definition video processing applications. We take the algorithm of Dynamic Programming and the platform of FPGA for our solution. Dynamic Programming has high accuracy but complex loops and computations, while FPGA platform has fast processing speed but relative low accuracy, therefore we employ them together to compensate the disadvantages of each other.

Chapter 3 discussed the entire software-hardware co-design and implementation of stereo matching using a top-down approach. Firstly we introduce the parallel and pipeline processing architecture, which replaces the disparity loop serial architecture used in software implementation. Then the algorithm is modified to a more hardware efficient version in the aspects of vertical cost aggregation, census transform, parameters and multipliers. Then we go deep into the RTL level to discuss our design block by

block in the data flow order. In our entire design, the parallel structure is used to speed up the computation and the fully pipelined structure is to enable the highest throughput.

In Chapter 4, the entire 3D depth range adjustment system on FPGA was presented in detail. The implemented stereo matching is used as a kernel to provide depth maps for view synthesis to generate the viewpoint interpolated anaglyph frames, and DDR2 SDRAM access scheduler is adopted to provide the large storage space for frames and buffer between stereo matcher and view synthesis. The main function of view synthesis is to adjust anaglyph 3D effects from full 2D to full 3D gradually by adjusting the viewpoint interpolation scale, and this kernel is co-developed by NCTU and Imec-Taiwan. The main principle of DDR2 SDRAM access scheduler is pre-fetching the data into on-chip memory buffers and burst read/write those data from/to DDR2 in order to realize the zero latency access to DDR2 SDRAM, and this kernel is implemented by Hsiu-Chi Yeh who is also from Delft University of Technology for MSc thesis in Imec. Our final on-chip system implemented on FPGA demonstrates the function of 3D depth range real-time adjustment successfully with good video quality.

Chapter 5 evaluated the entire design and implementation in various aspects. We evaluate the modified hardware efficient algorithm by configuring all the parameters as the single variables and finally obtain a set of parameters with lowest Middlebury benchmark evaluation error rate. Evaluation of stereo matching implementation is performed in hardware resource utilization and real-time performance. After comparing with other state-of-the-art implementations, ours is a fastest stereo matcher for high-definition resolution $1024\times768@60FPS$ with excellent Middlebury average error rate 6.60%.

6.3 Possible Future Work

According to our current algorithm, implementation and evaluation, there are several possible plans for future improvements of the stereo matcher performance.

- The Dynamic Programming kernel consumes much on-chip memory to store the large disparity matrix for tracking, which will be a huge number of gate counts for ASIC. One possible method to reduce this burden is performing data compression by encoding and decoding according to the features of the selected Dynamic Programming model.

- The support region builder and vertical voting kernels use a lot of line buffers, and separated buffers will increase the difficulty of ASIC layout. We can use a large buffer to replace these separated buffers, however it needs the redesign of data flow and memory address counters.

- Current Dynamic Programming is not very robust to luminance bias although it has high accuracy, because the Dynamic Programming depends much more on the luminance information than the structure information. One possible solution is to optimize the style of the Dynamic Programming, such as dividing Dynamic Programming and orthogonal Dynamic Programming. Another possible solution

is to optimize the census transform sampling pattern to include more texture information and improve its weight, because the census transform increases the robustness to luminance deviation for Dynamic Programming.

Utilizing the real-time depth maps generated by our stereo matching processing, a number of other applications except the 3D depth range adjustment system can be developed and improved, such as free-viewpoint TV, virtual reality, object tracking, gesture detection, video games etc.

Bibliography

[1] K. Ambrosch, M. Humenberger, W. Kubinger, and A. Steininger. SAD-based stereo matching using FPGAs. *Embedded Computer Vision*, pages 121–138, 2009.

[2] M.Z. Brown, D. Burschka, and G.D. Hager. Advances in computational stereo. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pages 993–1008, 2003.

[3] N. Chang, T.M. Lin, T.H. Tsai, Y.C. Tseng, and T.S. Chang. Real-time DSP implementation on local stereo matching. In *Multimedia and Expo, 2007 IEEE International Conference on*, pages 2090–2093, 2007.

[4] N.Y.C. Chang, T.H. Tsai, B.H. Hsu, Y.C. Chen, and T.S. Chang. Algorithm and Architecture of Disparity Estimation With Mini-Census Adaptive Support Weight. *IEEE Transactions on Circuits and Systems for Video Technology*, 20(6):792–805, 2010.

[5] P.P Chu. *RTL hardware design using VHDL: coding for efficiency, portability, and scalability*. Wiley-IEEE Press, 2006.

[6] Altera Corporation. The Efficiency of the DDR & DDR2 SDRAM Controller Compiler. http://www.altera.com/literature/wp/wp_ddr_sdramefficiency.pdf, December 2004.

[7] Altera Corporation. TriMatrix Embedded Memory Blocks in Stratix III Devices. http://www.altera.com/literature/hb/stx3/stx3_siii51004.pdf, May 2009.

[8] Altera Corporation. Stratix III Device Handbook. http://www.altera.com/literature/hb/stx3/stratix3_handbook.pdf, March 2011.

[9] Altera Corporation. Video and Image Processing Suite User Guide. http://www.altera.com/literature/ug/ug_vip.pdf, May 2011.

[10] Ahmad Darabiha, Jonathan Rose, and W. James MacLean. Video-Rate Stereo Depth Measurement on Programmable Hardware. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, pages I203–I210, 2003.

[11] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Efficient Belief Propagation for Early Vision. Technical report, Department of Computer Science, Cornell University, 2004.

[12] A. Fusiello, V. Roberto, and E. Trucco. Efficient stereo with multiple windowing. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 858–863. Citeseer, 1997.

[13] M. Hariyama, T. Takeuchi, and M. Kameyama. VLSI processor for reliable stereo matching based on adaptive window-size selection. In *PROC IEEE INT CONF ROB AUTOM*, volume 2, pages 1168–1173, 2001.

[14] H. Hirschmueller, P.R. Innocent, and J. Garibaldi. Real-time correlation-based stereo vision with reduced border errors. *International Journal of Computer Vision*, 47(1):229–246, 2002.

[15] H. Hirschmueller and D. Scharstein. Evaluation of cost functions for stereo matching. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007.

[16] Heiko Hirschmuller. Accurate and Efficient Stereo Processing by Semi-Global Matching and Mutual Information. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2:807–814, 2005.

[17] M. Humenberger, C. Zinner, M. Weber, W. Kubinger, and M. Vincze. A fast stereo matching algorithm suitable for embedded real-time systems. *Computer Vision and Image Understanding*, 2010.

[18] H. Jeong and S.C. Park. Trellis-based systolic multi-layer stereo matching. In *IEEE Workshop on Signal Processing Systems*, pages 257–262, 2003.

[19] Y. Jia, X. Zhang, M. Li, and L. An. A miniature stereo vision machine (MSVM-III) for dense disparity mapping. In *Proceedings of the 17th International Conference on Pattern Recognition*, volume 1, 2004.

[20] S. Jin, J. Cho, XD Pham, KM Lee, S.K. Park, M. Kim, and JW Jeon. FPGA Design and Implementation of a Real-time Stereo Vision System. *IEEE Transactions on Circuits and Systems for Video Technology*, 20(1), 2010.

[21] Ratheesh Kalarot and John Morris. Comparison of FPGA and GPU implementations of Real-time Stereo Vision. In *Computer Vision and Pattern Recognition Workshops*, pages 9–15, 2010.

[22] T. Kanade, A. Yoshida, K. Oda, H. Kano, and M. Tanaka. A stereo machine for video-rate dense depth mapping and its new applications. In *CVPR*, page 196. the IEEE Computer Society, 1996.

[23] P. Kauff, N. Brandenburg, M. Karl, and O. Schreer. Fast hybrid block-and pixel-recursive disparity analysis for real-time applications in immersive tele-conference scenarios. In *Proceedings of WSCG*. Citeseer, 2000.

[24] S. Kosov, T. Thormahlen, and H.P. Seidel. Accurate real-time disparity estimation with variational methods. In *Advances in Visual Computing*, pages 796–807. Springer, 2009.

[25] M. Kuhn, S. Moser, O. Isler, F.K. Gurkaynak, A. Burg, N. Felber, H. Kaeslin, and W. Fichtner. Efficient ASIC implementation of a real-time depth mapping stereo vision system. In *Proceedings of the of the 46th IEEE Midwest International Symposium on Circuits and Systems*, pages 1478–1481. Citeseer, 2003.

[26] Chia-Kai Liang, Chao-Chung Cheng, Yen-Chieh Lai, Liang-Gee Chen, and Homer H. Chen. Hardware-Efficient Belief Propagation. In *Circuits and Systems for Video Technology*, pages 525–537, 2011.

[27] J. Lu, S. Rogmans, G. Lafruit, and F. Catthoor. High-speed stream-centric dense stereo and view synthesis on graphics hardware. In *IEEE 9th Workshop on Multimedia Signal Processing*, pages 243–246, 2007.

[28] Jiangbo Lu, Gauthier Lafruit, and Francky Catthoor. Anisotropic local high-confidence voting for accurate stereo correspondence. In *SPIE*, volume 6812, page 68120J, 2008.

[29] D. Masrani and W. MacLean. A real-time large disparity range stereo-system using FPGAs. *Computer Vision - ACCV 2006*, pages 42–51, 2006.

[30] Y. Miyajima and T. Maruyama. A real-time stereo vision system with FPGA. In *Field-Programmable Logic and Applications*, pages 448–457. Springer, 2003.

[31] John Morris and Georgy Gimel'farb. Dynamic Programming Stereo in Reconfigurable Hardware. Technical report, Department of Computer Science, The University of Auckland, August 2006.

[32] S. Mukai, D. Murayama, K. Kimura, T. Hosaka, T. Hamamoto, N. Shibuhisa, S. Tanaka, S. Sato, and S. Saito. Arbitrary view generation for eye-contact communication using projective transformations. In *the 8th International Conference on Virtual Reality Continuum and its Applications in Industry*, pages 305–306. ACM, 2009.

[33] Sungchan Park and Hong Jeong. Real-time Stereo Vision FPGA Chip with Low Error Rate. In *International Conference on Multimedia and Ubiquitous Engineering*, pages 751–756, 2007.

[34] J. Salmen, M. Schlipsing, J. Edelbrunner, S. Hegemann, and S. Luke. Real-Time Stereo Vision: Making More Out of Dynamic Programming. In *Computer Analysis of Images and Patterns*, pages 1096–1103. Springer, 2009.

[35] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47(1):7–42, 2002.

[36] Q. Tian and M.N. Huhns. Algorithms for subpixel registration. *Computer Vision, Graphics, and Image Processing*, 35(2):220–233, 1986.

[37] F. Tombari, S. Mattoccia, L. Di Stefano, and E. Addimanda. Near real-time stereo based on effective cost aggregation. *19th International Conference on Pattern Recognition*, pages 1–4, 2008.

[38] Middlebury University. Middlebury Stereo Evaluation - Version 2. http://vision.middlebury.edu/stereo/, September 2009.

[39] M.A. Vega-Rodriguez, J.M. Sanchez-Perez, and J.A. Gomez-Pulido. An FPGA-based implementation for median filter meeting the real-time requirements of automated visual inspection systems. In *the 10th Mediterranean Conference on Control and Automation*. CiteSeer, 2007.

[40] L. Wang, M. Liao, M. Gong, R. Yang, and D. Nister. High-quality real-time stereo using adaptive cost aggregation and dynamic programming. In *3D Data Processing, Visualization, and Transmission, Third International Symposium on*, pages 798–805, 2006.

[41] J. Woodfill and B. Von Herzen. Real-time stereo vision on the PARTS reconfigurable computer. In *IEEE Symposium on FPGAs for Custom Computing Machines*. Citeseer, 1997.

[42] Q. Yang, L. Wang, R. Yang, S. Wang, M. Liao, and D. Nister. Real-time global stereo matching using hierarchical belief propagation. In *The British Machine Vision Conference*, pages 989–998, 2006.

[43] R. Yang, M. Pollefeys, and S. Li. Improved real-time stereo on commodity graphics hardware. In *Conference on Computer Vision and Pattern Recognition Workshop*, pages 36–36, 2004.

[44] K. Zhang, J. Lu, G. Lafruit, R. Lauwereins, and L. Van Gool. Cross-based local stereo matching using orthogonal integral images. *IEEE Transactions on Circuits and Systems for Video Technology*, 19(7):1073–1079, 2009.

[45] K. Zhang, J. Lu, G. Lafruit, R. Lauwereins, and L. Van Gool. Real-time accurate stereo with bitwise fast voting on CUDA. *5th IEEE workshop on embedded computer vision, held in conjunction with ICCV*, 2009.

[46] K. Zhang, J. Lu, G. Lafruit, R. Lauwereins, and L. Van Gool. Real-time accurate stereo with bitwise fast voting on CUDA. *5th IEEE workshop on embedded computer vision, held in conjunction with ICCV*, 2009.

[47] Lu Zhang, Ke Zhang, Tian Sheuan Chang, Gauthier Lafruit, Georgi Krasimirov Kuzmanov, and Diederik Verkest. Real-time High-definition Stereo Matching on FPGA. In *the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 55–64. ACM, 2011.