REAL-TIME OPTICAL FLOW SENSOR DESIGN AND ITS APPLICATION ON OBSTACLE DETECTION

by

Zhaoyi Wei

A dissertation submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Electrical and Computer Engineering

Brigham Young University

August 2009

BRIGHAM YOUNG UNIVERSITY

GRADUATE COMMITTEE APPROVAL

of a dissertation submitted by

Zhaoyi Wei

This dissertation has been read by each member of the following graduate committee and by majority vote has been found to be satisfactory.

Date	Dah-Jye Lee, Chair
Date	James K. Archibald
Date	Bryan S. Morse
Date	Brent E. Nelson
Date	Doran K. Wilde

BRIGHAM YOUNG UNIVERSITY

As chair of the candidate's graduate committee, I have read the dissertation of Zhaoyi Wei in its final form and have found that (1) its format, citations, and bibliographical style are consistent and acceptable and fulfill university and department style requirements; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the graduate committee and is ready for submission to the university library.

Date

Dah-Jye Lee
Chair, Graduate Committee

Accepted for the Department

Michael J. Wirthlin
Graduate Coordinator

Accepted for the College

Alan R. Parkinson
Dean, Ira A. Fulton College of
Engineering and Technology

ABSTRACT

REAL-TIME OPTICAL FLOW SENSOR DESIGN AND ITS APPLICATION ON OBSTACLE DETECTION

Zhaoyi Wei

Department of Electrical and Computer Engineering

Doctor of Philosophy

Motion is one of the most important features describing an image sequence. Motion estimation has been widely applied in structure from motion, vision-based navigation and many other fields. However, real-time motion estimation remains a challenge because of its high computational expense. The traditional CPU-based scheme cannot satisfy the power, size and computation requirements in many applications. With the availability of new parallel architectures such as FPGAs and GPUs, applying these new technologies to computer vision tasks such as motion estimation has been an active research field in recent years. In this dissertation, FPGAs have been applied to real-time motion estimation for their outstanding properties in computation power, size, power consumption and reconfigurability.

It is believed in this dissertation that simply migrating the software-based algorithms and mapping them to a specific architecture is not enough to achieve good performance. Accuracy is usually compromised as the cost of migration. Improvement

and optimization at the algorithm level are critical to performance. To improve motion estimation on the FPGA platform and prove the effectiveness of the method, three main efforts have been made in the dissertation. First, a lightweight tensor-based algorithm has been designed which can be implemented in a fully pipelined structure. Key factors determining the algorithm performance are analyzed from the simulation results. Second, an improved algorithm is then developed based on the analyses of the first algorithm. This algorithm applies a ridge estimator and temporal smoothing in order to improve the accuracy. A structure composed of two pipelines is designed to accommodate the new algorithm while using reasonable hardware resources. Third, a hardware friendly algorithm is developed to analyze the optical flow field and detect obstacles for unmanned ground vehicle applications. The motion component is de-rotated, de-translated and postprocessed to detect obstacles. All these steps can be efficiently implemented in FPGAs. The characteristics of the FPGA architecture are taken into account in all development processes of these three algorithms.

This dissertation also discusses some important perspectives for FPGA-based design in different chapters. These perspectives include software simulation and optimization at the algorithm development stage, hardware simulation and test bench design at the hardware development stage. They are important and particular for the development of FPGA-based computer vision algorithms. The experimental results have shown that the proposed motion estimation module can perform in real-time and achieve over 50% improvement in the motion estimation accuracy compared to the previous work in the literature. The results also show that the motion field can be reliably applied to obstacle detection tasks.

ACKNOWLEDGMENTS

When I unintentionally opened the backup folder named *Brigham Young Univ.* which I created almost four years ago, the memory brought me back to the days when I was waiting in China to start my Ph.D. study in BYU. At that time, I was looking forward to but anxious about my future study at BYU, somewhere over 7,000 miles from my hometown in China. Four years later, pondering on the accomplishment I have achieved, I feel so lucky and grateful for the help I have received from many people.

First of all, I would like to thank my advisor Professor Dah-Jye Lee. Thanks to his knowledge and consideration, this four years turn out to be a great journey which I greatly appreciate. I am also grateful for Professor Lee's patience, encouragement. He helped me through some of the toughest times in my life. I cannot overemphasize the support I received from him. I also want to thank his wife Wendy for her hospitality and smile.

I sincerely thank my committee members. Thanks to Professor Nelson for leading me into the magic world of FPGAs, teaching me knowledge beyond the textbook and for the insightful discussions. Thanks to Professor Archibald and Professor Wilde for their welcoming discussions in the DARPA Grand Challenge competition. It was a very exciting experience. Thanks to Professor Morse for his wonderful teaching on computer vision in my first semester here at BYU. In that class, I really enjoyed the experience of applying what I learned to solve problems.

The internships at Broadcom were valuable opportunites for me to have hands-on experiences on real product development. I want to thank Val Johnson, Jorge Wong, Shawn Zhong, Jiajia Xu, Jack Ma, Qian Zhou, Rahul Sansi, Jeffrey Chiao and

Jing Xu for showing me the technologies behind a Digital TV. I enjoyed a lot working there with them.

I want to thank Elizabeth for her kind help on revising my dissertation. I also need to thank the lab members Wade Fife, Dong Zhang, Barrett Edwards, Spencer Fowers, Beau Tippetts, Kirt Lillywhite and Aaron Dennis for their constructive discussions.

I am grateful for having great parents like mine. I thank them for supporting me to pursue my dreams so that I could be who I am today. *Baba, Mama, Wo Ai Ni Men!*.

I am truly indebted to my wife Lihong and my son Wenfan. Thanks Lihong for your unselfish sacrifice made for the family. Thank you for standing behind me and supporting me for all these years. With your support, I have the courage and strength to face the challenges and strive for the future.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction to Motion Estimation

1.1.1 Motion Estimation

Motion perception and interpretation are important functions in the human vision system. With prior knowledge, human and other biological visual systems use motion to infer the structure of the environment, estimate egomotion, detect objects and so on. This theory was first discussed in Hans Wallach's pioneering work on motion direction perception published in his doctoral dissertation in 1935 [1].

In the human vision system, motion is the movement of the pattern of light projected upon the retina. Correspondingly, for computer vision and video processing tasks, when an imaging sensor takes multiple snap shots at a fixed short time interval, the three-dimensional (3D) scene is projected onto a two-dimensional (2D) sensor and saved as images. These images are highly temporally correlated. When perceived by a human being, his or her brain will fuse these images and form a sense of motion. Physically, for a fixed time interval, the motion of each pixel on the frame across time can usually be represented by a displacement vector $\mathbf{v} = (v_x, v_y)^T$ which reflects the velocity. If we can estimate all or part of the displacements, we can retrieve all or partial information about the 3D scene from these 2D images. One simple example of motion estimation application is shown in Figure 1.1. Two frames are captured at time t, and $t^{'}$, and frame t is superimposed on frame $t^{'}$ for viewing purposes. Across these two frames, the front tree is moving from left to right with a displacement of $\mathbf{v}$. The displacement is $\mathbf{v}^{'}$ for the tree in the back. Assuming we have the prior knowledge that the scene is static, and we calibrated the camera and know its movement from

time t to $t^{'}$, then we can estimate the distance from the trees to the camera by estimating $\mathbf{v}$ and $\mathbf{v}^{'}$ and reversing the image projection process.

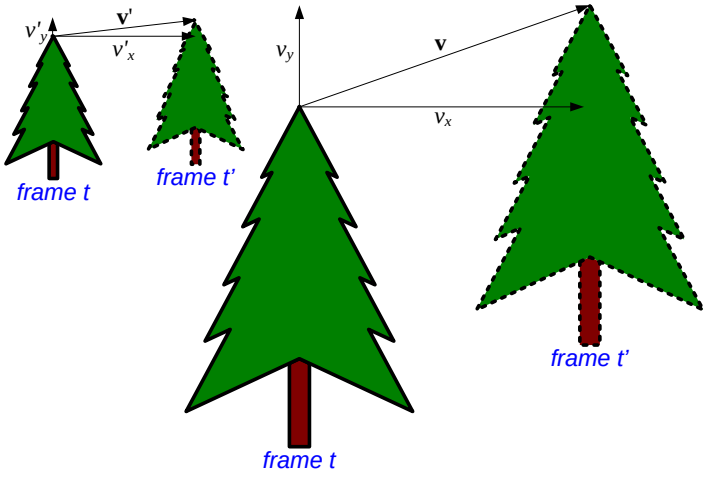


Figure 1.1: Motion estimation illustration

One important assumption made in most motion estimation algorithms is that the local brightness pattern is invariant over short time intervals. In other words, if the time interval is short enough, the brightness change of one pixel is only due to the relative motion between the object and the camera. Other effects such as ambient lighting change, self-lighting effect or reflectance can be neglected.

In computer vision and video processing tasks, motion is one of the most important features to describe an image sequence. In literature, motion estimation algorithms can be roughly divided into three categories:

- **Feature tracking**. Feature tracking algorithms include two stages: feature selection and feature tracking across frames. Pixels or regions with strong texture information are usually selected first. Then these pixels or regions are tracked across frames based on a certain rule to infer their motion. In [2], a 2-by-2 symmetric matrix was built using spatial derivative components. By

analyzing the matrix, uniform brightness areas, edges and corner points were indentified separately. Edges and corner points were good candidates for tracking algorithms while uniform brightness areas were neglected. The classical KLT feature tracker was proposed in [3, 4]. The feature tracking process was treated as an optimization process which minimized the residue error in a given window. A simple translation motion model was chosen over an affine motion model to prevent over-parameterization. In [5], the motion model was generalized to the affine motion model. Also a method for feature selection and a technique for monitoring features during tracking were proposed to take exceptions like occlusion into account.

- **Block matching**. Block matching algorithms are widely used in video compression and media processing for their efficiency. There are several factors that determine the performance of block matching algorithms. These factors include block size selection, distance metric, searching strategy, etc. Searching strategy is one of the most important factors in performance and efficiency and has been studied the most. In [6], a survey on block matching algorithms and hardware realization architectures was given. A widely used block matching algorithm was published in [7] which applied different searching strategies at different scales to speed up the matching process and to avoid being trapped in local maxima.

- **Optical flow**. Optical flow aims to measure the motion field from the apparent motion of the brightness pattern. Optical flow is widely used in 3D vision tasks such as egomotion estimation, Structure from Motion (SfM), motion segmentation, etc. Since the first optical flow algorithm was proposed by Horn and Schunck [8] in 1981, much work has been done in this area. However, research on optical flow algorithms with real-time performance and adequate accuracy for embedded vision applications just started. Development of an optical flow algorithm that meets these unique requirements is the main research focus in

this dissertation. Further details about optical flow algorithms will be presented in a later section in this chapter.

The separation of these three categories is sometimes ambiguous. Some specific algorithms may share the properties of more than one category. Therefore, it is to be noted that the above categorization is not absolute.

1.1.2 Challenges in Motion Estimation and Interpretation

The human vision system relies on years of stimuli and training to perceive and interpret motion. The perceived motion carries object and enviroment information and the human brain infers object and environment information from the motion information [9]. During the perception and interpretation process, parallel connections and parallelism exist in the primate brain [10]. Similarly, for motion perception and interpretation tasks in the computer vision context, we can conclude that there are two major challenges in motion estimation and interpretation:

- **Knowledge about object and environment**. In the human brain, experience or knowledge is very important to motion perception and interpretation. For example, while the observer is moving, the brain needs to estimate the egomotion (global motion in this case) and differentiate it from the local motion in order to tell the difference between the moving objects and the still objects. Without training, the brain will be confused by the still objects and moving objects because their projected images on the retina are all moving at the same time. Also, as in [9], the gait of an actor in dark background could be recognized by attaching bright spots to the joints of the actor. However, the static set of spots couldn't be understood by the observer. This experiment proved that motion is an important clue in recognition and the brain is trained to recognize certain patterns of motion.

- **Parallel calculation structure**. Motion estimation is computationally expensive. The complexity originates from two aspects: a large amount of data and complexity for calculating motion at each pixel. Firstly, one image frame usually

contains hundreds of thousands of pixels, and more than one frame is needed to estimate motion. This means that the motion estimation algorithms need to cope with millions of pixels in a very short time. Secondly, motion estimation aims at finding the minimum trajectory in the spaitotemporal image volume. To deal with noise and avoid error estimation, a large amount of operations are needed to estimate motion at a single pixel. The expensive computation cost of motion estimation algorithm is due to the combination of these two aspects. The traditional computation platform (general purpose processors) is serial in nature and is rarely fit for real-time motion estimation.

As to the first challenge, with current techniques, it's very difficult or nearly impossible to design a uniform system that can interpret motion without taking the specific application into account, i.e. a general motion interpretation system. For a specific task, assumptions or constraints are usually posed to simplify the underlying problem. In this dissertation, optical flow is applied to the obstacle detection task for unmanned ground vehicles (UGVs). By constraining the application area, the motion pattern can be substantially simplied. The simplified motion pattern is better suited for the available computation scheme. As to the second challenge, in recent years advances in different hardware architectures (e.g. Graphics Processing Unit (GPU), Field-Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs), etc.) make it possible to perform computation intensive tasks in real time. More and more algorithms that were not fit for real-time application are now feasible. The migration of the old algorithms onto the new architectures or the formation of new algorithms harnessing new architectures is attracting more attention. Because different architectures have different advantages, it requires extra effort to concurrently choose a suitable architecture and optimize the processing algorithm according to the specific architecture features.

1.1.3 Applications of Motion Estimation

Motion estimation can be applied to, but is not limited to, the following areas.

- **Video compression and enhancement**. A substantial difference between image compression standards (e.g. JPEG) and video compression standards (e.g. AVC) is that video compression techniques are performed in the spatial and temporal domain while image compression techniques are performed only in the spatial domain. In video compression, motion estimation is the key to reduce the temporal redundancy between frames and to achieve a high compression ratio. In the latest AVC (MPEG-4 Part 10) compression standard [11, 12], the enhanced motion compensation scheme is one of the most important factors improving AVC over the previous standards. Also in consumer electronics, motion estimation is widely used in noise reduction, frame rate conversion, deinterlacing and many other important video enhancement modules available in Digital TV, DVD player, etc [13].

- **Human motion analysis**. Human motion analysis [14, 15, 16] is applied in security surveillance, human computer interaction, video conferencing, athletic performance analysis, etc. No matter which analysis algorithm is applied, the prerequisite is the robust estimation of motion for different parts of the body. In [17], optical flow features were used to track a human arm and torso and were incorporated into kinematic and geometric models in order to study the human body in terms of the degrees of freedom. In [18], motion was integrated with segmentation and shapes to build a deformable model which can track human body parts.

- **Structure from Motion**. SfM refers to retrieving the shape, depth or 3D structure of the object, environment or egomotion of the camera by analyzing the motion across frames. SfM is a challenging task because image capturing is a noninvertible process of projecting a 3D signal onto a 2D image and the reconstruction process is ambiguous up to one parameter. Efforts were made in

order to require less priori/assumptions in reconstruction algorithms or improve the robustness to the outliers in motion estimation and calculation errors [19, 20, 21].

- **Vision-based navigation**. Using motion in navigation finds its inspiration in biological behaviors. In [22], it was reported that honeybees control their flight speed by regulating the image velocity in their eyes. If the visual texture was removed, it would drastically compromise honeybees' speed control because of the absence of image motion cues. Using motion estimation for Unmanned Aerial Vehicles (UAV) and UGV is attracting more attention [23, 24] because of the passive nature of vision-based sensors and the abundance of information it can provide. The applications of motion estimation in navigation include: egomotion estimation [25], landmark tracking [24], obstacle detection and avoidance [26] and so on. In this dissertation, obstacle detection is used as an example application of motion estimation algorithms.

In the following sections of this chapter, a literature review on motion estimation algorithms will be given first. Then the proposed algorithms and designs will be sketched and the dissertation will be outlined in the last section.

1.2 Optical Flow Algorithms

Optical flow algorithms were first proposed by Horn and Schunck [8] in 1981. Optical flow algorithms aim to measure motion field from the apparent motion of the brightness pattern. The basic assumption of optical flow algorithms is the **B**rightness **C**onstancy **C**onstraint (BCC) which can be formulated as

$$I(x, y, t) = I(x + \Delta x, y + \Delta y, t + \Delta t), \tag{1.1}$$

where $I(x, y, t)$ is the intensity of pixel (x, y) at frame t and $I(x+\Delta x, y+\Delta y, t+\Delta t)$ is the intensity of the corresponding pixel at frame $t + \Delta t$. By expanding Equation

(1.1) using Taylor series [8], we have

$$I(x, y, t) = I(x, y, t) + I_x v_x + I_y v_y + I_t + O$$
$$\Rightarrow \nabla I \cdot \mathbf{v} + I_t = -O, \tag{1.2}$$

where $\nabla I = (I_x, I_y)$ is the gradient, $\mathbf{v} = (\frac{\Delta x}{\Delta t}, \frac{\Delta y}{\Delta t})^T = (v_x, v_y)^T$ is the optical flow and O is a higher order residual which is often set to 0. The BCC assumes that image brightness changes are due only to motion in a short time. In other words, if the time interval between frames is small, other effects (such as changes in lighting conditions) causing brightness changes can be neglected.

In (1.2), to calculate the optical flow at one pixel, there are two variables to be resolved, i.e. v_x and v_y. BCC assumption only provides one constraint. Therefore, one or more constraints are needed to calculate the optical flow vector $\mathbf{v}$. Since 1981, many optical flow algorithms have been proposed. According to the way the extra constraint is made, these optical flow algorithms were roughly divided into the following categories as in [27, 28]:

1. **Differential methods**
2. **Frequency-based methods**
3. **Correlation-based methods**
4. **Tensor-based methods**

In this dissertation, all discussions will follow the above categorization. A brief introduction about each category will be given in the following section.

1.2.1 Differential Methods

Differential methods use the first or second order spatiotemporal derivatives as the second constraint. One classical first order differential method is proposed in

[3]. In this paper, a weighted Gaussian mask is applied to build the cost function

$$C(\mathbf{v}) = \sum_{(x,y)\in R} W(x,y)(\nabla I \cdot \mathbf{v} + I_t(x,y,t))^2, \tag{1.3}$$

where $W(x,y)$ is the mask to weight votes at different pixels in a local neighborhood R. Optical flow vector $\mathbf{v}$ is defined as the vector which minimizes the cost function $C(\mathbf{v})$. In [29], the weighting mask was generalized to a form of convolution and therefore separated the choice of weighting mask from minimizing Equation (1.3). Equation (1.3) can be represented in another form shown as

$$C(\mathbf{v}) = \big\langle (\nabla I \cdot \mathbf{v} + I_t(x,y,t))^2 \big\rangle, \tag{1.4}$$

where operator $\big\langle \cdot \big\rangle$ is used to represent the convolution which incorporates the information of pixels in a local neighborhood as in [29]. An analytic result [28] could be obtained after some deduction shown as

$$\begin{pmatrix} \langle I_x I_x \rangle & \langle I_x I_y \rangle \\ \langle I_x I_y \rangle & \langle I_y I_y \rangle \end{pmatrix} \begin{pmatrix} v_x \\ v_y \end{pmatrix} = - \begin{pmatrix} \langle I_x I_t \rangle \\ \langle I_y I_t \rangle \end{pmatrix}. \tag{1.5}$$

This algorithm is fit for hardware processing because the main calculations, such as convolution and derivatives calculation as in (1.5), can be efficiently implemented in hardware pipelines or other parallel processing structures. It has been one of the most popular algorithms which were optimized and implemented on the FPGAs. Details of this algorithm will be discussed further in the later chapters.

Besides the first order derivative algorithm shown above, second order derivative algorithms [8, 30, 31] were also devised. Assumptions on the continuity of image signal and its first order derivative were made in order to use the second order derivatives. However, image data are discrete signals and the derivatives are calculated using difference. In practice, second order derivative algorithms are usually more sensitive to noise than first order derivative algorithms and for further details, one may refer to the cited papers.

1.2.2 Frequency-based Methods

In frequency-based algorithms, motion in the spatial domain is extracted by transforming the image into the frequency domain and then measuring in the frequency domain. It was reported that frequency-based methods were able to estimate motion in some cases where correlation-based algorithms may fail.

As in [32, 33, 34], one important equation for frequency-based methods is shown as

$$\hat{I}(k_x, k_y, \omega) = \hat{I}_0(k_x, k_y)\delta(v_x k_x + v_y k_y + \omega), \tag{1.6}$$

where $\hat{I}_0(k_x, k_y, \omega) = \mathcal{F}[I(x, y, 0)]$ is the Fourier transform of the translating motion pattern, $\delta(\cdot)$ is a Dirac delta function, k_x, k_y are the spatial frequencies and ω is the temporal frequency. In Equation (1.6), it describes that the nonzero frequency components lie on the plane through the origin in the frequency domain. Different velocity-tuned filters were designed to extract the motion information. In [35], four Gabor filters tuned to different spatial orientations were used at each of the three temporal frequencies. This setting was applied to different spatial scales to adjust to different scales of spatiotemporal structures. In [32], the image sequence was represented by the responses of a series of spatiotemporal velocity-tuned linear filters which were bandpass and constant phase. The velocity was calculated from the local first-order behavior of constant *phase* surfaces. Fleet et al. [32] claimed that phase representation is relatively insensitive to the contrast variation, affine deformation or lighting condition changes which are more realistic in practice. The drawback of frequency-based methods includes filtering in the spatial and temporal domain which is very expensive in computation and memory consumption. Also it needs extra computation to transform the images into the frequency domain. In [36], Fleet and Langley proposed to use recursive Infinite Impulse Response (IIR) filters to ease both the computation and memory requirement.

1.2.3 Correlation-based Methods

Correlation-based methods, sometimes called matching methods, try to match a feature or a region to its corresponding part in another frame. The advantage of correlation-based methods is its less strict requirement on the frame number (typically two frames) compared to frequency-based or differential methods. For frequency-based methods, a certain number of frames is needed to satisfy the temporal sampling theorem. For differential methods, to obtain a good estimation on temporal derivative, usually more than two frames have to be used. Correlation-based methods are usually better fit for hardware implementation than others. Also correlation-based methods usually allow larger displacement between frames than the other algorithms.

In correlation-based methods, motion vector across frames is defined as a displacement. To estimate the displacement, correlation-based methods have a few important factors similar to the block matching algorithm used in video compression and media processing. These factors include:

- **Matching template selection**. One can simply define the to-be-matched candidate as a region such as a rectangular window centering at current pixel. The bigger the region is, the more resistant it is to noise and the less likely the algorithm will be trapped in local minima. The price for using a bigger region is its requirement for more computational power. A smaller region has better spatial resolution and decreases the chances of appearance of multiple motion patterns inside a region. Besides region matching, another possible solution is using a feature point as the candidate. This is valid when a sparse but reliable motion field is wanted in the application. Actually, for feature matching, it usually still requires neighborhood analysis. By restricting the matching candidates to feature points only, the amount of mismatches and the computation expense are decreased substantially.

- **Distance metric**. A distance metric is used to define the similarity between two regions or two features. For example, one can use the sum of squared difference (SSD) or the sum of absolute difference (SAD) as the metric or other

more complicated ones. The smaller the distance metric is, the more similar the two matches are. Also, the normalized correlation can be used to avoid biased caused by illumination differences. Other metics such as multiplication can be used, except that a bigger value means smaller distance between two regions.

- **Searching strategy**. Once the matching template and the distance metric are selected, a certain searching strategy is followed to find the best match. The brute force method is the most straightforward one and it searches every possible place inside the searching area. To speed up the searching process, optimized searching strategies were deployed [37, 38, 39] by posing certain assumptions on the local signal model.

In [37], Anandan proposed a coarse-to-fine matching algorithm using the Laplacian pyramid. By using a Laplacian pyramid, a rough range of motion vectors was estimated at the coarsest level and refined in the following levels. SSD was used as the distance metric in the paper. Subpixel motion vectors were computed by interpolating the SSD surface quadratically and finding the minimum match.

In [39], three consecutive frames I_{-1}, I_0 and I_1 were first filtered by band-pass filters. Two SSD images $SSD_{\{-1,0\}}$ and $SSD_{\{0,1\}}$, calculated from two frames pairs $\{I_{-1}, I_0\}$ and $\{I_0, I_1\}$, were then averaged to suppress the spurious noise in the SSD image. The averaged SSD image was used to calculate a probability distribution. Last, the motion vectors were obtained from the center of mass of the probability distribution with respect to each estimate.

1.2.4 Tensor-based Methods

As shown in [28], optical flow computation can be treated as an orientation estimation problem in the spatiotemporal volume. The spatiotemporal volume is a stack of time-sequential images one on top of the other as shown in Figure 1.2. According to the BCC in (1.1), the trajectory of one pixel across time can be treated as a line/curve of discrete voxels (pixel in 3D spatiotemporal domain) in the volume. Different from direction, orientation ranges between 0° and 180° with respect to the

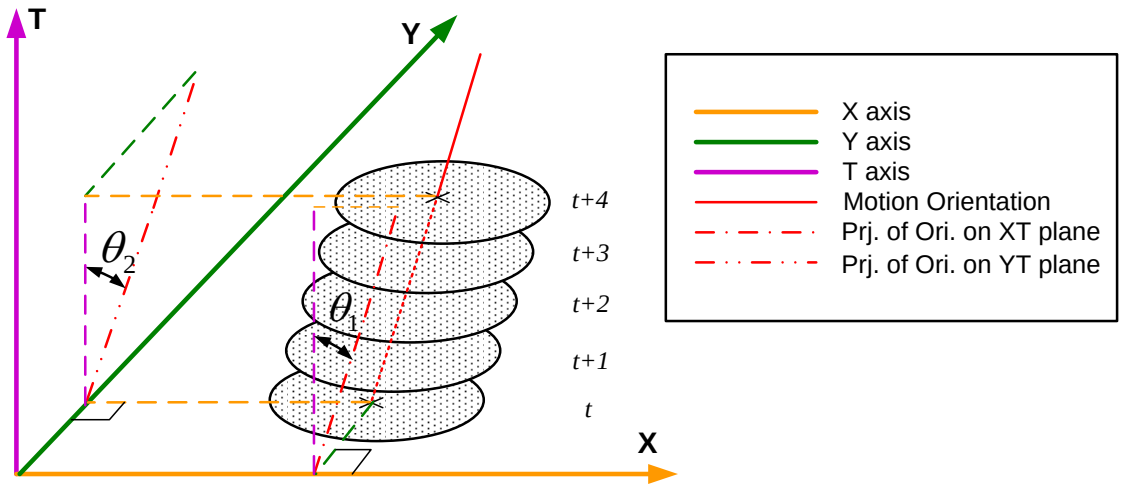


Figure 1.2: Orientation illustration

T axis. In other words, any 3D vector pairs **x** and $-\mathbf{x}$ have the same orientation. In Figure 1.2, across frames the dotted circle moves along the red line which connects the center of the circle and is marked as a cross. This red line is projected onto planes **XT** and **YT**. The angles between the projected red lines and **T** axis are denoted as θ_1 and θ_2. With some geometrical deduction, the optical flow is calculated as [28]

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix} = - \begin{pmatrix} \tan\theta_1 \\ \tan\theta_2 \end{pmatrix}. \tag{1.7}$$

Tensor is a way to represent orientation. In 1987, Bigün and Granlund [40] used tensor to represent local orientation. Since then, a lot of research has been done to apply tensor to optical flow estimation. In 3D spatiotemporal volume, tensor is a 3-by-3 symmetric matrix. Compared to the vector representation, tensor provides a powerful and closed representation of the local brightness structure. Higher level of analysis such as certainty analysis and neighborhood motion model analysis can be easier under the tensor framework. Different types of tensor have been developed to represent the local orientation. There are three types of tensors in the literature:

- **Quadrature tensor**. In [41], outputs of six symmetric quadrature filters were weighted and summed to build the quadrature tensor. These six filters operated in the frequency domain and each was tuned to one orientation in the 3D domain.

- **Gradient tensor**. In [40, 42], gradient vectors in a local neighborhood were weighted and summed to build the gradient tensor which was often called a structure tensor. Gradient tensor is a positive semidefinite n-by-n matrix. (For more properties of the gradient tensor, one can refer to [28].) Similar works have been published by Liu et al. [43] which incorporated affine motion model in the tensor estimation. In this dissertation, a gradient tensor based optical flow estimation algorithm is presented. Further details about this algorithm will be discussed in Chapter 3.

- **Polynomial tensor**. Local image signals can be projected onto second degree polynomials [44, 45] which gives sufficient information for orientation estimation. Similar to the gradient tensor, polynomial tensor is also positive semidefinite.

Either type of tensor maps local signal orientation in different ways. Their properties were evaluated and compared in more details in [46]. In this dissertation, the focus is on gradient mainly because it is more efficient for pipelined hardware structure and provides good accuracy.

1.3 Proposed Method and Dissertation Outline

One common limitation of optical flow is its computational power requirement. The calculation time of existing optical flow algorithms is typically on the order of seconds or even longer per frame. This long processing time thus prevents optical flow algorithms from many real-time applications such as autonomous navigation for unmanned vehicles. There are some hardware platforms available such as GPUs, FPGAs, etc. After comparison, an FPGA-based platform was selected for its low power, small size and increasing computation capability which are all essential for

embedded vision applications. There are a lot of optical flow algorithms described in the literature. However, most of them are running on general purpose processor platforms and have been developed without taking any specific hardware architecture into account. Therefore, these algorithms are usually not suitable for the FPGA implementation. In this dissertation, two optical flow algorithms are presented for the FPGA platform. The first one is a tensor-based optical flow algorithm that has been optimized specifically for a pipelined hardware structure. After this work, another algorithm is devised based on the previous algorithm to fully utilize the newer FPGA technology. It has been shown that the second algorithm produces drastically better accuracy on the test bench sequences. To maximize algorithm performance on a specific hardware architecture, e.g. FPGA, it is believed in this dissertation that effort must be made at the very first stage of algorithm design in order to utilize and accommodate for characteristics of the specific hardware architecture. By following the same methodology, an obstacle detection algorithm by analyzing the motion field output from the optical flow module is developed. This algorithm was also devised and optimized for the FPGA architecture so that it is possible to integrate the optical flow module and obstacle detection module into a single FPGA platform. The goal is to develop a standalone obstacle detection sensor.

The rest of this dissertation focuses on further discussion on the developed algorithms and FPGA modules. In Chapter 2, a detailed overview of current parallel computation architectures and the features of each architecture are first presented. These architectures include FPGAs, GPUs and Ambric processors. Besides the FPGA platforms, the optical flow algorithms have been implemented in GPU platforms and Massively Parallel Processor Arrays (MPPAs) by other researchers at Brigham Young University. Chapter 2 will review these works as well as the existing works on applying FPGAs to accelerate real-time computer vision and video processing algorithms. Finally, the hardware platforms which were used in our research are introduced.

Chapter 3 discusses the first optical flow algorithm developed. It begins with the motivation and algorithm formulation which is a good fit for full hardware pipeline

processing. Then the hardware structure is introduced including the architecture and functionality of each module. After that, specific optimization techniques which optimize the hardware implementation and their trade-offs are given. This is important especially because FPGA usually has very limited resources. Finally, the experimental results are given which show the effectiveness of the proposed design.

Chapter 4 proposes an improved optical flow algorithm based on a ridge estimator. In this design, we take advantage of the newer hardware platform which is equipped with a more powerful FPGA. To harness the later technology, new hardware architecture is proposed which lowers the system throughput but substantially improves the accuracy. The ridge regression algorithm improves the stability of motion estimation at the cost of slightly more hardware resources. The experimental results are given at the end of this chapter to show its improvement over the previous algorithm in Chapter 3. The resulting design is a standalone FPGA system which can be used as an optical flow sensor in practice.

Chapter 5 introduces the application of real-time optical flow sensor to obstacle detection tasks for small UGVs. First, the motivation for using optical flow sensor in obstacle detection is introduced. The vision-based scheme is compared to other solutions such as **LI**ght **D**etection **A**nd **R**anging (LIDAR) and sonar. Then the different stages of the algorithm are discussed one by one. The corresponding hardware modules in the system are presented afterwards. Finally, software simulation results and hardware simulation results are both given to show its performance.

Chapter 6 summarizes the work accomplished in this dissertation and its contributions. It also discusses the limitation of current design and proposes future research directions.

Chapter 2

Hardware Accceleration(Optimization) Techniques

2.1 Available Techniques

As video processing and computer vision algorithms are getting more and more complicated, in many applications the traditional general purpose processor (CPU) based solutions cannot satisfy the real-time computation, size or power requirements. In these years, many designs based on new architectures appear to fill the gap between the traditional CPU-based solution and various practical demands. These alternatives include ASICs (Application Specific Integrated Circuits), FPGAs, GPUs, DSPs, MPPAs, etc. It is interesting to observe that media processing has been a very active area [47, 48, 49] in verifying different architectures. The reasons for the popularity of media processing can be summarized as follows:

- The international standards such as JPEG, JPEG 2000, MPEG-2, and MPEG-4 contribute to the huge success of media compression applications. The applications need encoding or decoding equipment of low manufacturing cost and small size;

- Different applications have different requirements for the devices. No single solution can satisfy all of them. Tradeoffs are often made among power, size, cost, performance, development complexity and flexibility;

- Newer standards with better compression ratio and quality are being developed at the cost of more complex calculations. Better computation platform is needed to keep up with the evolution of industrial standards;

- Most common and computationally expensive operations in media processing such as motion estimation, Discrete Cosine Transform (DCT) can be substantially optimized using existing parallel architectures.

Although there is no standard in computer vision as there is in media processing, computer vision algorithms share many common characteristics with media processing as listed above. Therefore, when discussing the parallel architectures in this chapter, it refers to designs in the media processing area as well as computer vision.

Table 2.1 shows the comparison of different architectures. The first column shows different candidates of the comparison [50]. The second column *Performance* and the third column *Programmability* are usually in reverse relation. *Unit Cost* represents the manufacturing cost for each unit. *Power* represents the electrical power consumed by each architecture during run-time.

Table 2.1: Comparison of different architectures.

	Performance	*Programmability*	*Unit Cost*	*Power*
CPUs	Low	High	Low	High
ASICs	High	Low	Low	Low
FPGAs	Medium	Low-Medium	Medium	Medium
GPUs	High	High-Medium	Low	High
DSPs	Medium	High-Medium	Low	Low
MPPAs	Medium	High	Medium	Low

Flexible structures bring better performance while flexibility increases the difficulty in programmability. For example, the general purpose CPUs execute instructions in sequential order. Users only need to program functions and data objects and the compiler will translate codes into binary instructions understood by the processor. Development using CPUs is convenient from the design iteration perspective. An ASIC is a dedicated circuit for a particular application. ASICs are capable of handling complicated and intensive processing tasks. The design of ASICs is very complicated, and not only includes functional parts but also the timing, layout and

routing in the low level hardware. ASICs can only be verified through software-based simulation before taping out. Once taped out, circuits in ASIC cannot be changed and the cost for one taping out is millions to tens of millions of dollars. Although ASICs can achieve maximum performance, with the increasing design cost and short life cycle, alternatives such as FPGAs, GPUs, DSPs and MPPAs appeared to fill the gap between ASICs and CPUs. In the following section, FPGA and GPU techniques which are related to our research will be briefly introduced. Using different hardware architectures to accelerate computation utilize different types of parallelism such as data parallelism, or task parallelism existing in the data or algorithms.

2.1.1 FPGAs

In recent years, FPGAs are receiving more attention in the video processing and computer community [51, 52] for their good cost and performance traits. Two dominant FPGA manufacturers are Xilinx and Altera who together own over 80 percent of the market share. FPGA architectures speed up computation by aligning data in a hardware pipeline and processing them in parallel. The speedup ratio of a pipeline is proportional to the pipeline depth. At the same time, a deeper pipeline will have larger latency. An FPGA-based solution can achieve its maximum performance when all the computation can be fully pipelined and has enough throughputs to feed in and read out the data. There are some advantages and disadvantages in FPGA-base design:

- Different from CPU-based development, the development in FPGAs usually uses a Hardware Description Language (HDL) such as VHDL or Verilog. The synthesizer will translate the code into corresponding circuits and the placer and router will map the circuits onto the physical resources on the FPGA. Although higher level languages are available, they need support from higher level synthesis and the quality of synthesized codes is lower than the HDL. In FPGAs, circuit components include RAMs, Look-Up Tables (LUTs), registers, interconnection wires, etc. A designer needs to specify the behavior of the circuits at each clock cycle, the interconnection of each module, timing, synchronization

and other details. Therefore, development using FPGAs is significantly more time consuming than CPU-based design.

- Simulation of part of the circuits or the whole system is a very important stage in FPGAs design. Usually each module in the system is tested separately before it is incorporated into the whole system. System level simulation is available when a simulation model of each module is ready. This creates extra overhead in FPGA design.

- Unlike ASICs, circuits in FPGAs can be re-programmed after a bitstream is downloaded. This makes it possible for reconfigurable computing which allows switching circuits between different configurations during run-time. Also the design-verification cycle is much shorter than that of ASICs. On the other hand, to verify the correctness of the circuit's functionality, the bitstream needs to be downloaded to FPGAs. The feedback loop in FPGAs development is much longer than in software development.

- Some complicated algorithms are beyond the capability a single FPGA can provide. The configurable structure facilitates the design of scalable multi-FPGAs systems and this feature makes FPGAs a powerful tool for implementing algorithms of different complexity. At the same time, multi-FPGA schemes also raise the questions of how to partition the task and how to control data flow across multiple FPGAs [53, 54]. In this dissertation, a single FPGA system is the focus. But it is believed that multi-FPGA systems are an efficient solution for the resource limitation issue and designing algorithms in the multi-FPGA framework is an interesting and promising future direction to investigate.

- Tradeoffs are usually made to map the software prototype of an algorithm onto an FPGA architecture. In order to minimize the performance degradation caused by the tradeoffs, special concerns need to be taken into account beforehand. A few aspects (tips) are included as follows. *First*, with current

FPGA architecture, it's much more efficient to implement fixed-point operations than floating-point ones. If an algorithm cannot be primarily represented in fixed-point numbers, it may not be a good fit for current FPGA architectures. *Second*, performance of FPGA-based design is maximized in a fully pipelined architecture which sometimes is not feasible for algorithms in computer vision and video processing in practice. One example is the iteration operation which is widely used in optimization-type problems such as minimizing a cost function or finding optimal results. Iteration is a typical example which can dramatically decrease the pipeline throughput. If the iteration number is fixed, one possible solution is to uncouple the iteration and concatenate each loop at the cost of using more hardware resources. *Third*, in hardware implementation, operations such as division or trigonometric functions can be optimized by using LUTs which can be efficiently stored in block RAMs in FPGAs. By careful design, LUT-based operations can achieve good accuracy while using only a fraction of the resources if they are directly implemented. This is useful especially when logic resources are very limited on an FPGA chip.

2.1.2 GPUs

GPU is another popular architecture for computation-intensive tasks. GPUs were originally developed for rendering graphics for personal computers, game consoles or workstations. In recent years, they started being used for video processing and computer vision [55, 56] for its high performance and programming flexibility. ATI and NVIDIA are the two main manufacturers of high-end GPUs. In 2007, NVIDIA released the Compute Unified Device Architecture (CUDA) which is a parallel computing architecture supported by most recent NVIDIA GPUs. Coming with the CUDA architecture, the CUDA API was also released by NVIDIA to allow programming CUDA-compatible GPUs in the familiar C language. CUDA C bridges the gap between traditional C programming and the GPUs' parallel architecture and promotes the *General-Purpose computation on GPUs* (GPGPU) on NVIDIA GPUs.

As a parallel processing processor, each CUDA-compatible GPU is a manycore parallel system [57]. This architecture has the following features.

- The CUDA architecture is a hierarchical structure. The top physical level is an array of Streaming Multiprocessors (SMs). Under the top level, within each multiprocessor, there are eight Scalar Processor (SP) cores, shared memory and other modules. There is synchronization scheme inside a multiprocessor but not between multiprocessors. *Threads*, the basic processing unit, are grouped into *warps* to be executed by the SP cores inside each multiprocessor.

- Corresponding to the hierarchical processor structure, memory is organized in a hierarchical manner as well. In the CUDA architecture, there are different types of memories which include global memory, constant memory, texture memory, local memory and shared memory. Among them, the constant and texture memory are read-only. Each SP has its own local memory; shared memory is accessible by all the SPs in a multiprocessor and all multiprocessors can access global memory. Besides different accessibility, different memories have different latency. Managing memory in an efficient way is critical to the performance of GPU-based design.

- In [58], a few factors were concluded in order to achieve good performance on the GPUs. These include:

    1. Choose the right algorithm;
    2. Achieve high processor occupancy;
    3. Reduce the number of divergent threads and
    4. Use memory efficiently.

2.1.3 Others

Besides FPGAs and GPUs, there are some other popular hardware architectures available on the market. In this section, DSPs and MPPAs solutions will be briefly reviewed.

A DSP is a specialized microprocessor with optimized architectures for real-time signal processing. DSPs usually support optimized arithmetic operations like Multiply-ACcumulates (MACs), geometric transforms and special addressing modes to increase efficiency. DSPs have evolved from the early dedicated audio processors to the audio/video processors of today. Since DSP is a microprocessor, it is suitable for embedded, lower power constrained applications. Texas Instrument, Freescale, and Analog Devices are the main manufacturers for DSPs. In video processing, DSPs are often used as video codecs in the embedded system [59, 60].

An MPPA architecture is a single-chip, massively parallel processing array with hundreds of processor cores [61]. In [61, 62], an Ambric MPPA architecture containing over 300 programmable RISC processors with hundreds of distributed memories is discussed. The Reduced Instruction Set Computer (RISC) architecture is embodied in the most prevalent microprocessors architectures such as Advanced RISC Machine (ARM), Microprocessor without Interlocked Pipeline Stages (MIPS), Power Performance Computing (PowerPC), etc. With current techniques, the array of RISC processors can be fabricated onto a single ASIC chip. These processors can be programmed using common software languages and communicated through hardware channels.

In [63], Ambric MPPAs were applied to a JPEG codec. Lin et al. [59] used Ambric MPPAs to build a 64-channel digital high frequency linear array ultrasound imaging beamformer. An optical flow algorithm was implemented using the Ambric architecture in [64]. Further details about this design will be discussed in the next section.

2.2 Existing Designs

It is impossible to list all published works on image/video processing or computer vision algorithms on specialized hardware architectures in this dissertation. Instead, the ones which optimized the motion estimation especially the optical flow calculation using FPGAs, GPUs or other architectures will be listed.

2.2.1 FPGA-based Designs

There is not a long history of applying FPGAs to optical flow calculation. The recent fast advance in this field is due to the improvement of the FPGA's capability which made it feasible and affordable to implement complicated algorithms like optical flow algorithms in FPGAs.

A hardware architecture to estimate motion was designed for FPGAs or ASICs in [65]. This design applied an iterative spatiotemporal derivative based algorithm. Arribas and Maciá [66] implemented a correlation-based algorithm on an Altera FPGA. This design was able to calculate 25 fps (frames per second) of optical flow field for image size of 100-by-100 pixels. Martín [67] implemented the iterative algorithm in [8] on a Altera FPGA. which could process 256-by-256 images at 60 frames per second. The simulation flow including the module simulation and system simulation was introduced. This is important for FPGA development and is one of the reasons why the FPGA development cycle is longer than software. In [68], another correlation-based optical flow algorithm was implemented in FPGA. There were two configurations for the system, which could calculate 320-by-240 images at 840 fps or 640-by-480 images at 30 fps. A 7-by-7 window was compared with certain number of windows depending on configurations. A SAD function was used to measure the similarities between two windows for its simplicity. A PCI board equipped with a Xilinx XC2V6000 FPGA was used as the platform. Two different configurations occupied 71% and 84% of the FPGA individually.

Diáz [69] et al. implemented the classical Lucas and Kanade approach [3] and provided a tradeoff between accuracy and processing efficiency. This paper is also the first to provide the accuracy report on the Yosemite sequence to the best of my knowledge. The Yosemite sequence is a synthetic sequence used as one of the most popular benchmarks for testing optical flow algorithms performance. The way of evaluating both speed and accuracy is more scientific than evaluating speed only because speed is not the only gauge to measure the performance of an optical flow calculation design. Botella et al. [70] applied a Multichannel Gradient Model (McGM), inspired by a biological motion estimation model. A customizable architecture of a neuromorphic

optical flow algorithm was designed for FPGAs and ASICs device. This algorithm used an iterative scheme and it was composed of some Finite Impulse Response (FIR) filters, and Infinite Impulse Response (IIR) filters which could be efficiently implemented in hardware and some other computation stages. The throughput of this algorithm was over 2000Kpps (Kilo pixels per second) and could obtain 5.5° angular error and 12° standard deviation on Yosemite sequence.

2.2.2 GPU-based Designs

One advantage of GPU-based design is its inherent support for floating-point calculation. In comparison the design in FPGAs usually use fixed-point numbers. Data range and bit width are monitored through the pipeline to maintain desirable accuracy.

Strzodka and Garbe [71] used eigenvector analysis of the spatiotemporal structure tensor to estimate motion at each pixel. A quality indicator was also proposed to indicate the reliability of each estimation and a pseudocolor map was applied to visualize the velocities. The proposed design was implemented on graphics cards and could compute real-time optical flow on 320-by-240 images. In [72], Horn and Schunck's algorithm [8] was implemented on a CUDA GPU. The resulting GPU design offered around 2 times the execution speed compared to its CPU version. In [73], a phase-based optical flow algorithm was implemented using the CUDA platform. In this phase-based optical flow algorithm, the response of Gabor filters at different scales was used to construct the motion estimation. The convolution operations used in filter response calculation could be extensively optimized by the CUDA architecture. The system (NVIDIA 8800 GTX) was reported to have a 40-fold speed-up over the 2.4GHz Core 2 Quad processor (using a single core) or up to 120 at higher resolution. For a typical setting, this design could calcuate 640-by-512 images at 48.5 fps.

In [58, 74], the optical flow algorithms introduced in Chapter 3 and Chapter 4 were implemented on GPUs. A tensor-based optical flow algorithm was implemented on an NVIDIA 8800 GTX platform to compare the FPGA-based design in [74]. After

carefully adjusting the memory access and block size of the images, the proposed design could calculate 640-by-480 images at 150 fps. This is the fastest speed that has been described in the literature to the best of my knowledge. This high speed did not sacrifice algorithm accuracy; instead it was the result of optimizing the factors which were important to GPU performance. In [58], a ridge regression algorithm was implemented on GPU and a systematic comparison between the FPGA-based and the GPU-based implementation of the two optical flow algorithms was made.

A few interesting conclusions were obtained after comparing these two implementations. *First*, GPU-based implementation has higher productivity. The development time for GPU is substantially shorter than for FPGA on the same algorithms. For the GPU-based design, it was coded in CUDA C which is a superset of C language and could be compiled and tested instantaneously. For FPGA design, the feedback loop is much longer. When comparing the lines of source code, FPGA-based design is 5 to 9 times more complicated than GPU design. *Second*, the power consumption of a GPU platform (typically a few hundred watts) is much higher than FPGA platform (around 10 watts). For applications that have power and size constraints, an FPGA is a much better choice. Also the cost for installing a GPU platform (a few hundred dollars) is lower than FPGA (a couple of thousand dollars).

2.2.3 Other Designs

There were some other techniques besides FPGAs and GPUs which were applied to optimize the optical flow computation.

Correia and Campilho [75] proposed a design which could process the Yosemite sequence of 252-by-316 images in 47.8ms using a pipelined image processor. In [76], the Lucas and Kanade algorithm [3] was parallelized and implemented by a cluster with 8 biprocessor nodes (2.4 GHz). The nodes were connected using a Gigabit Ethernet switch. The proposed system was able to calculate 502-by-288 images at 25 to 30 fps. An Ambric MPPA [64] was used to implement a tensor based optical flow algorithm. Small Java programs were written for each processor in the array, and the interconnections of the processors defined the directions of the data flow. By

hierarchically organizing the processors and writing and reusing small programs for the processors, the programming work for the over 300 processors were found to be manageable in practice.

2.3 FPGA Platform

The hardware platforms used for this research are briefly introduced in this section. This is important because applying FPGAs as the computation platform is a highly application specific task. The FPGA platform is an important factor when comparing different solutions.

2.3.1 Xilinx XUP Board

A Xilinx XUP V2P (XUP) development system [77, 78] was used in the first part of the project. The XUP system is a general platform for introductory educational purposes or advanced research projects. The XUP board has various hardware resources which are listed as follows:

- Virtex-II Pro XC2VP30 FPGA. This FPGA has 30,816 logic cells, 2,448 Kb of block RAM and two PowerPC processors;
- One DDR DIMM slot which supports up to 2Gb of RAM;
- USB2 port for FPGA configurations only;
- 10/100 Ethernet MAC/PHY;
- XSGA video output port;
- Stereo audio with AC97 codec;
- PS/2 and RS-232 ports;
- SATA connectors for Gb serial I/0;
- User GPIO (4 DIP switches, 5 push buttons, 4 LEDs);
- Additional I/O through 60-pin headers;

- SystemACE chip which supports Compact Flash for FPGA configuration and data storage.

Xilinx provides reference designs, such as video decoder using VDEC, displaying the image files stored on a compact flash to an external display monitor, running a web server and edge detection used in image processing, etc. Overall, XUP platform is a powerful environment for learning FPGA or algorithm research using FPGA-based architecture. However, XUP system is a general purpose platform and it is not the best choice for some specific applications, e.g. UGV applications, because of its large size (50 square inches) and power consumption.

2.3.2 BYU Helios Platform

The Helios Robotic Vision platform [79] was developed in the Robotic Vision Lab at Brigham Young University to accommodate real-time computing on small autonomous vehicles. The Helios board is equipped with the following hardware resources:

- The Helios board is compatible with Xilinx Virtex-4 FX series FPGAs, up to the FX60. The Virtex-4 FX60 has 56,880 slices which is almost twice as many as the Virtex-II Pro XC2VP30 on the XUP V2P board, 4,176 Kb block RAM and two PowerPC processors;
- Up to 64 Mb SDRAM which runs up to 133 MHz at 32 bits;
- Up to 16 Mb flash memory;
- Supports 8, 16 or 32 Mb platform flash which allows power-on configuration of FPGA;
- Supports high-speed USB2 port which is controlled by a Cypress CY7C68014A USB peripheral chip;
- Additional 64 GPIO through 120-pin header;

- Onboard 1.2V, 2.5V and 3.3V power supplies and supports input voltage between 5-24V;
- Different daughter boards were developed for the Helios board which can be equipped with video DAC, dual camera inputs, wireless communications and other I/O devices.

The Helios board has almost all of the XUP board's features but is only the size of a small deck of business cards and consumes only around 2 to 5 watts of power when it is running at maximum capacity.

Chapter 3

Tensor Based Optical Flow Algorithm

A tensor-based optical flow algorithm has been developed and implemented on an FPGA. Details of this work are presented in this chapter. The resulting algorithm is accurate and specifically developed for pipeline hardware implementation. Through lessons learned from software simulations, a tradeoff is made between accuracy and hardware complexity to develop an optimal pipelined hardware structure to enable the algorithm to operate in real time. This design is able to process 640×480 images at 64 fps, which is more than adequate for most real-time robot navigation applications. This design has low resource requirements, making it suitable for small embedded systems. This algorithm is tested on synthetic sequences such as the Yosemite sequence and the Flower Garden sequence. Error analysis on the Yosemite sequence is also discussed. Besides synthetic sequences, this algorithm is also tested on a real image sequence to show its robustness and limitations. Limitations are analyzed and an improved scheme is then proposed. It is shown that with sufficient hardware resources, this algorithm can be further improved and the performance of this design can be significantly improved.

This chapter is organized as follows. In Section 3.2, the algorithm is formulated and the modifications are introduced. In Section 3.3, the hardware implementation design and tradeoffs made in the design are discussed. In Section 3.4, the performance analysis of the proposed design is presented. Discussions and further improvements on this algorithm are discussed in Section 3.6.

3.1 Algorithm Overview

Many optical flow algorithms have been developed in the last two decades. During this time, 3D tensor techniques have shown their superiority in producing dense and accurate optical flow fields [41, 44, 45, 80, 43]. 3D tensor provides a powerful and closed representation of the local brightness structure. For example, in [45] an accurate and fast tensor-based optical flow algorithm was modified for a FPGA hardware implementation. The hardware architecture of this implementation was adjusted to reach a desired tradeoff of accuracy for resource utilization.

The concept of tensor has been introduced briefly in Chapter 1. In this chapter, a hardware friendly tensor-based optical flow algorithm is presented and the implementation of this algorithm on the FPGA is discussed. The resulting design is accurate and well-suited for pipeline hardware implementation. It is able to process 640×480 images at 64 fps or faster, which is adequate for most real-time navigation applications. Based on the literature survey performed recently, this design described in [69] achieves a 30% accuracy improvement on the test bench image sequence and executes faster (19661 Kpps) than any previous work. It is also an efficient design that uses less FPGA resources than other similar work. Potential applications of this design include real-time navigation and obstacle avoidance of Unmanned Autonomous Vehicles (UAV) or other applications which require real-time optical flow computations. In this paper, the hardware structure of this design is described in more detail, additional analysis of its performance is provided for real image sequences, and the algorithm is extended to increase its robustness for use in the presence of significant platform jitter and image noise.

3.2 Algorithm Formulation

An image sequence $I(\mathbf{x})$ can be treated as volume data where $\mathbf{x} = (x, y, t)^T$ is the 3D coordinate, x and y are the spatial components, and t is the temporal component. According to the BCC mentioned in Chapter 1, object movement in spatial-temporal domain will generate brightness patterns with certain orientations.

The outer product $\mathbf{O}$ of the averaged gradient $\nabla\bar{I}(\mathbf{x})$ is defined as

$$\mathbf{O} = \nabla\bar{I}(\mathbf{x})\nabla\bar{I}(\mathbf{x})^T = \begin{pmatrix} o_1 & o_4 & o_5 \\ o_4 & o_2 & o_6 \\ o_5 & o_6 & o_3 \end{pmatrix}, \tag{3.1}$$

where

$$\begin{aligned} \nabla\bar{I}(\mathbf{x}) &= \sum_i w_i \nabla I(\mathbf{x}_i) \\ &= (\bar{I}_x(\mathbf{x}) \quad \bar{I}_y(\mathbf{x}) \quad \bar{I}_z(\mathbf{x}))^T \end{aligned} \tag{3.2}$$

and w_i are weights for averaging the gradient. Then the gradient tensor $\mathbf{T}$ can be constructed by weighting $\mathbf{O}$ in a neighborhood as

$$\mathbf{T} = \sum_i c_i \mathbf{O}_i = \begin{pmatrix} t_1 & t_4 & t_5 \\ t_4 & t_2 & t_6 \\ t_5 & t_6 & t_3 \end{pmatrix}. \tag{3.3}$$

$\mathbf{O}$ is smoothed to reduce the effect of the noise and decrease the singularity of the tensor matrix. The gradient tensor $\mathbf{T}$ is a 3×3 positive semi-definite matrix.

The optical flow $(v_x, v_y)^T$ is measured in pixels per frame. It can be extended to a 3D spatiotemporal vector $\mathbf{v} = (v_x, v_y, 1)^T$. For an object with only translation movement and without noise in the neighborhood, $\mathbf{v}^T\mathbf{T}\mathbf{v} = 0$. In the presence of noise and rotation, $\mathbf{v}^T\mathbf{T}\mathbf{v}$ will not be zero. Instead, $\mathbf{v}$ can be determined by minimizing $\mathbf{v}^T\mathbf{T}\mathbf{v}$. A cost function can be defined as

$$e(\mathbf{v}) = \mathbf{v}^T\mathbf{T}\mathbf{v}. \tag{3.4}$$

The velocity vector $\mathbf{v}$ is the 3D spatio-temporal vector which minimizes the cost function at each pixel.

The optical flow can be solved as

$$v_x = \frac{(t_6 t_4 - t_5 t_2)}{(t_1 t_2 - t_4^2)}, \tag{3.5}$$

$$v_y = \frac{(t_5 t_4 - t_6 t_1)}{(t_1 t_2 - t_4^2)}. \tag{3.6}$$

This algorithm is similar to the one discussed in [45] which assumes a constant motion model. Further deduction details can be found in [45].

An affine motion model is used to incorporate tensors in a neighborhood [45]. Pixels in a neighborhood are assumed to belong to the same motion model. In this design, the constant model is used which assumes the constancy of velocity in a local neighborhood in order to save hardware resources. The constant model performs almost as well as the affine motion model when operating in a small neighborhood.

3.3 System Design and Optimization

3.3.1 Data Flow

Data flow of the proposed algorithm is divided into five modules as shown in Figure 3.1. In the Gradient Calculation module, images are read from memory into the FPGA and aligned to calculate the gradient. The gradient components are then averaged in the Gradient Weighting module (Equation (3.2)). From the outer product of the averaged gradient components (Equation (3.1)), five outer product components can be obtained. These components are then further averaged to compute the gradient tensor (Equation (3.3)). Finally, these tensor components are fed into the Optical Flow Calculation module to compute the final result according to Equations (3.5) and (3.6) and then written back to memory. Of importance, there is no iterative processing in this design. Therefore, the computation process can be fully pipelined in hardware to improve its processing throughput and thereby enable real-time use.

In the block diagram of Figure 3.1, the connections between modules consist of only unidirectional data and a corresponding data valid signal. Once a set of data is generated in a module, it is registered into the downstream module for processing. At

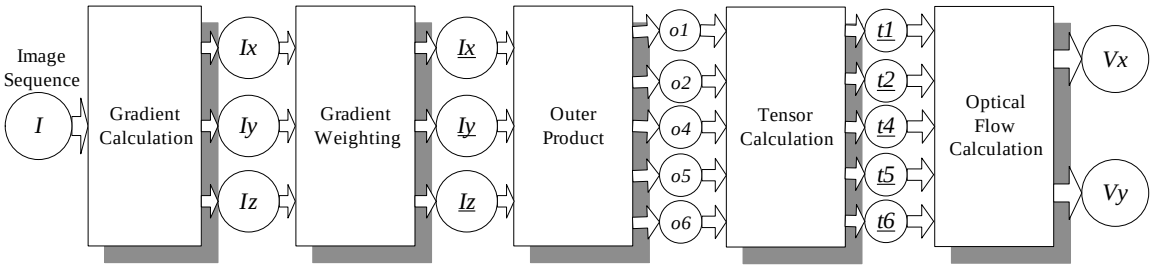


Figure 3.1: Data flow of the system (the signals with underscore are the weighted signals of those without underscore)

the end of the pipeline, the results are written back to memory for further processing, display, or analysis.

3.3.2 Tradeoffs between Accuracy and Efficiency

There are three major modules in the computation which are all convolution-like operations: 1. Gradient Calculation; 2. Gradient Weighting; and 3. Tensor Calculation. These three modules are critical to the algorithm performance and use most of the hardware resources. Therefore, different configurations of these modules were evaluated and compared to reach an optimal solution.

The Gradient Calculation module is a 1D convolution process while the Gradient Weighting and Tensor Calculation modules are both 2D convolution processes. Generally, a larger convolution kernel size will provide better accuracy but will require more hardware resources. Finding the optimal tradeoff between accuracy and efficiency requires finding the optimal convolution kernel sizes so that the design meets its performance requirements but with minimum hardware usage. Once the kernel sizes are decided, the corresponding optimal weights can be specified.

The design was simulated in MATLAB. These MATLAB simulations were designed to match the hardware down to the bit level. There were two purposes for this. First, the performance of the design could be evaluated precisely in order to get a satisfactory tradeoff between accuracy and efficiency. Second, the intermediate variables of the simulation could be used to verify the hardware during debugging. The Yosemite sequence with ground truth was used to evaluate the system performance.

A few tradeoffs are made to optimize the hardware design. They are discussed in detail in the following subsections.

Gradient calculation. Gradient calculation is the first step in the computation and errors from this step will propagate to subsequent modules. Therefore, a derivative operator with large radius is preferred in the design to get better derivative estimates. If the radius of the derivative operator is denoted as r, then $2r+1$ frames of data must be read from off-chip memory into the FPGA to calculate the temporal derivative for each frame in hardware because there is not enough on-chip memory in the FPGA to store these frames. If a larger derivative operator is used, the memory bandwidth required to retrieve the needed $2r+1$ frames goes up as well.

After comparison, a series-designed first-order derivative operator of radius 2 is chosen [29].

$$D = \frac{1}{12}(1 \;\; -8 \;\; 0 \;\; 8 \;\; -1). \tag{3.7}$$

Therefore, to calculate the derivative for one frame, five frames must be read from the memory.

Gradient weighting and tensor weighting. As shown in Figure 3.1, there are two weighting processes in the computation. Both weights w_i and c_i are given by the Gaussian function. The kernel is $n_1 \times n_1$ for gradient weighting and $n_2 \times n_2$ for tensor calculation. These weighting processes are essential to suppress noise in the image. These two processes are correlated, and so they are analyzed together to get an overall tradeoff.

Table 3.1 shows different possible combinations of the kernel size and the best accuracy each combination can provide. To obtain the results shown in the table, the algorithm was tested on the Yosemite sequence and the resulting accuracy was compared against the ground truth. Accuracy was measured in angular error. Resource utilization was estimated by the number of flip-flops used. The flip-flop utilization estimate can be calculated as

$$F = 2n_1 * b_1 * m_1 + 2n_2 * b_2 * m_2. \tag{3.8}$$

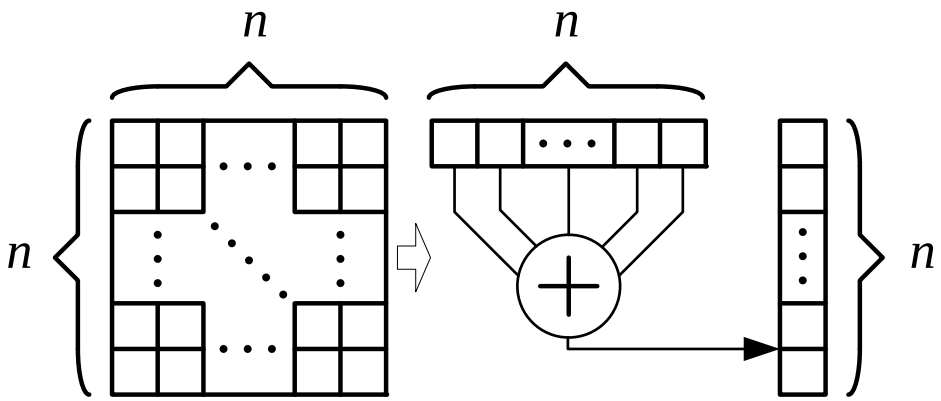


Figure 3.2: Simplification of 2-D convolution

F is related to the kernel sizes n_1 and n_2, input data bit width b_1 and b_2, and number of units m_1 and m_2. σ_1 and σ_2 are the standard deviations of the kernels.

Table 3.1: Configurations of weighting parameters and accuracy

n_1	n_2	F	σ_1	σ_2	Best Accuracy
3	5	1334	3.0	3.0	14.8°
5	3	1050	1.9	2.1	15.8°
7	3	1206	2.1	2.5	12.7°
9	3	1362	2.4	2.9	11.5°

Because the weights are constructed as Gaussian functions, in order to save hardware resources, a 2D weighting process is efficiently decomposed into two cascaded 1D convolutions as shown in Figure 3.2. After the 2D-to-1D simplification, the resource utilization becomes a linear rather than a quadratic function of n_1 and n_2 as shown in Equation (3.8).

Once the kernel sizes are fixed, different sets of standard deviation σ_1 and σ_2 are tried to find the best accuracy. σ_1 and σ_2 are both increased from 0.3 to 3 with an increment of 0.1. Finally, $n_1 = 7$ and $n_2 = 3$ are chosen for optimal performance and efficiency in the implementation.

Hardware optimization. Several hardware optimization steps are taken to achieve optimal hardware performance:

1. *Pipeline structure.* A heavily pipelined hardware structure is used to maximize the throughput. Once the pipeline is full, the hardware can produce a result for every clock cycle. At a 100MHz clock rate, the pipeline will be able to compute around 100 million pixels per second (325.5 frames per second at 640×480 resolution).

2. *Fast memory access.* Due to the pipelined hardware architecture used, the major system bottleneck is memory access. To overcome this, a specialized memory interface is employed. This will be discussed in more detail in the next section.

3. *Bit width trimming.* Fixed-point numbers are used in the design. To maintain computation accuracy and yet save hardware resources, the bit widths used are custom-selected at each stage of the processing pipeline. Figure 3.3 is an expanded version of Figure 3.1 that shows the bit widths used. Data are trimmed twice to save hardware resources, once after the Gradient Weighting module and once after the Tensor Calculation module. It can be observed from Equations (3.5) and (3.6) that a scaling factor applied to variables t_1-t_6 will not change the final results. Bit width trimming will not affect the relative ratio of these two equations.

4. *LUT-based divider.* A normal pipelined 32×32 bits hardware divider would consume too much hardware resource. Therefore, two LUT-based dividers are used in the Optical Flow Calculation module to decrease the latency and save hardware resources. These are used without any adverse impact on result accuracy.

3.3.3 System Architecture

This design was implemented on the Xilinx XUP V2P board [77, 78]. In the system diagram shown in Figure 3.4, the FPGA communicates with a host PC through both Ethernet and serial ports. Ethernet was used for high-speed data transfers,

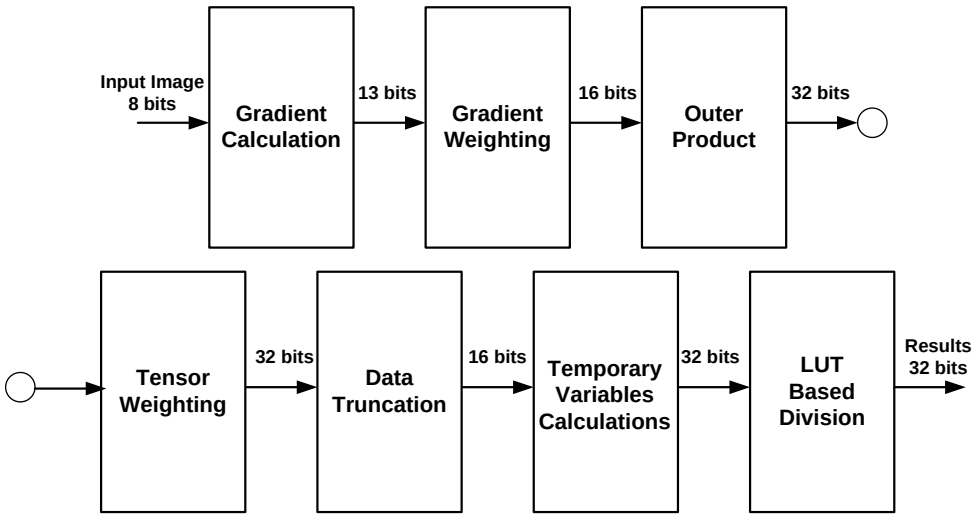


Figure 3.3: Bit width propagation

specifically the uploading of image sequences to hardware and the downloading of results. The UART was used for control and for outputting debugging information such as hardware status through the serial port. The Ethernet controller was hosted on the Xilinx PLB bus and the UART on the slower OPB bus.

The design was implemented using the Xilinx EDK tools. The most common structure of an EDK-based design is for the DDR memory controller to connect to the PLB bus like any other peripheral. An initial analysis of the memory bandwidth required for the computation (multiple frames must be fetched from memory for each computed result frame) showed that the PLB bus would likely become the performance-limiting component in the design. As a result, a multi-port memory controller (MPMC) architecture was chosen instead. The MPMC used was provided by Xilinx and is a dedicated DDR memory controller that provides four separate memory ports to the rest of the design.

As can be seen in Figure 3.4, one of those ports is connected to the PLB bus to provide PLB-based memory access. However, the optical flow core hardware (the pipeline of Figure 3.1) is directly connected to two of the remaining memory ports. This provides at least two benefits. First, memory accesses bypass the PLB bus, leaving it free for other communications within the system. Second, interfacing with

the MPMC is much simpler than interfacing with the PLB bus, resulting in a smaller and higher-performance optical flow core implementation. A final advantage of this architecture is that less buffering to communicate with memory is required in the optical flow core.

This design accepts 640×480 16-bit YUV images as input, which are read into the memory through the Ethernet interface. They are then read from the memory into the FPGA through the multi-port memory controller for processing. Prior to processing, the U and V components are removed and only the Y component i.e. intensity is processed.

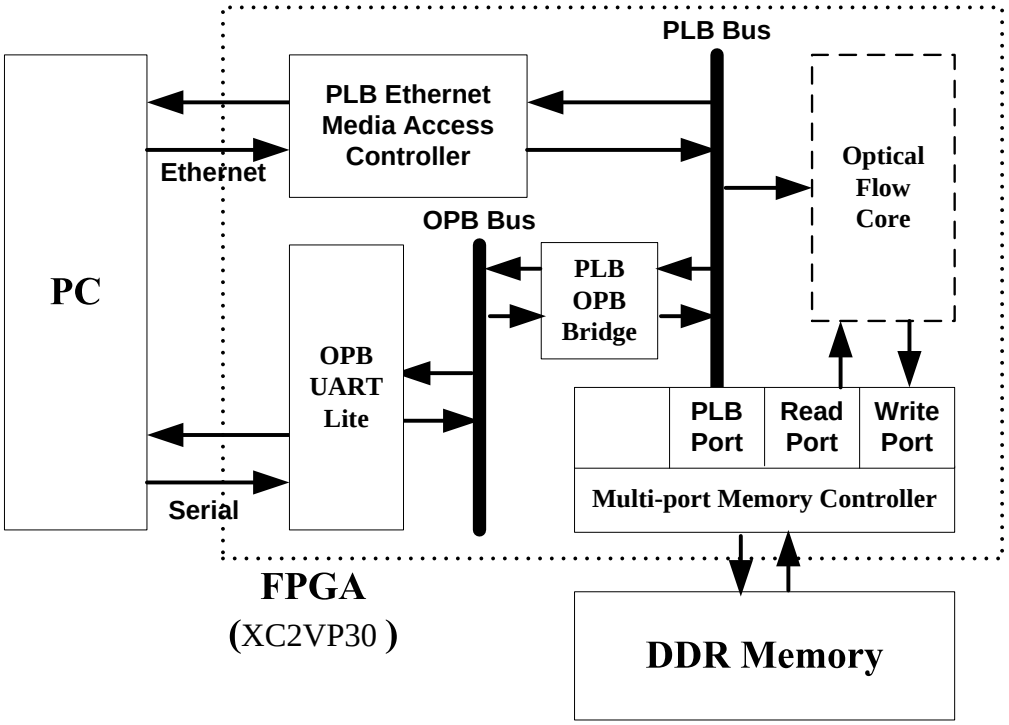


Figure 3.4: Tensor-based system diagram

3.4 Experimental Results

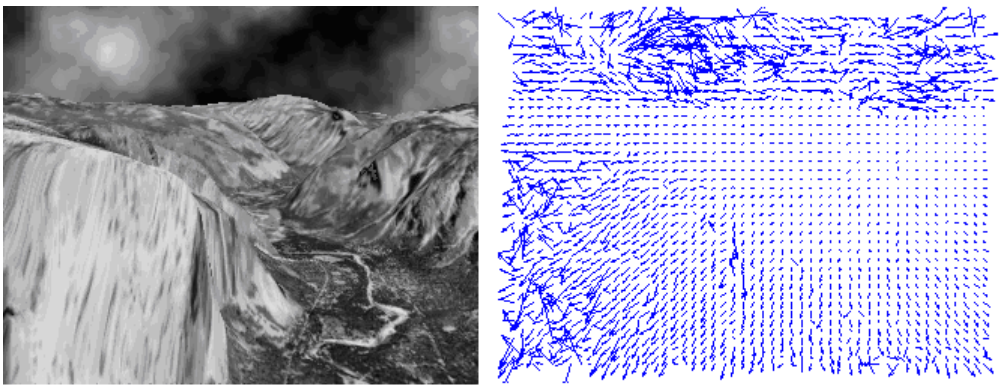
The system clock rate for our XUP board implementation is 100 MHz. It can process 640×480 frames at a rate of 64 frames per second. This processing speed is equivalent to about 258 fps of image size 320×240. As mentioned above, all three YUV components are read into the memory but only Y is processed. If only the Y component was read into the FPGA, the system speed could be further increased (nearly doubled) by reducing the memory access bottleneck. This optical flow design uses 10,288 slices (75% of the total slices available).

3.4.1 Synthetic Sequence

This design was tested on the Yosemite sequence and the Flower Garden sequence. The hardware for these two experiments was the same. The image sizes of these two sequences are smaller than 640×480, and so were padded with zeros before processing. However, the results are shown in the figures below in their original sizes.

Figure 3.5(a) shows one frame of the sequence and Figure 3.5(b) is the result from the hardware. The optical flow field in the sky region is noisy due to the brightness change across frames. The optical flow in other regions is less noisy except the lower left part because of the absence of texture. The average angular error was 12.9° and the standard deviation was 17.6°. The sky region was excluded from the error analysis like most of the other work found in the literature. In simulation, the division operations in the final step (Equations (3.5) and (3.6)) were in double precision. In the hardware implementation, a 17-bit LUT was used to implement the division operations. This accounts for the minor differences between the simulation and hardware results.

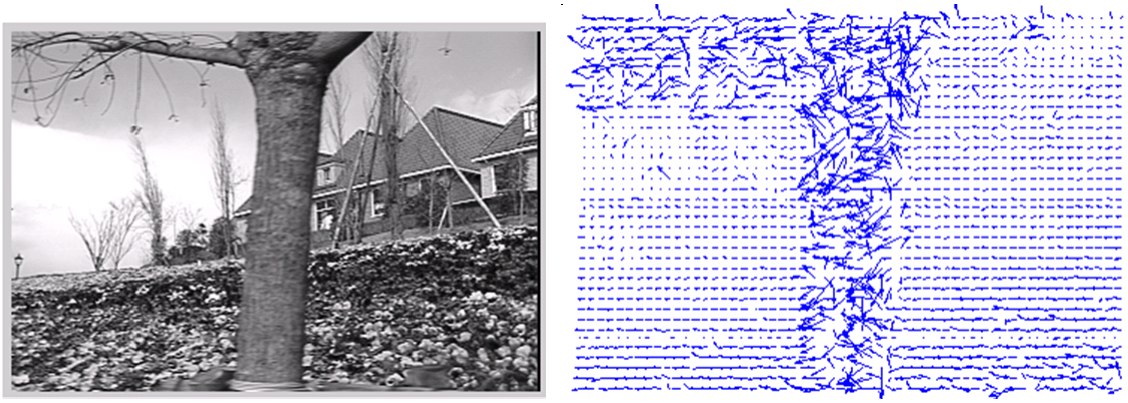
When the research was conducted, the only error analysis on optical flow implemented in an FPGA was given in [69], which also tested their design on the Yosemite sequence. Their average angular error was 18.3° and their standard deviation was 15.8°. Their design processed 320×240 images at 30 fps compared to 640×480 images at 64 fps in this design and used approximately 19,000 slices, almost twice as many as this design.



(a) 8th frame of Yosemite sequence (b) Optical flow field

Figure 3.5: The Yosemite sequence and the measured optical flow field

Figure 3.6 shows the results for the Flower Garden sequence. Optical flow values near the trunk boundary are quite noisy because the velocity within the neighborhood is not constant which does not meet the assumption shown in Equation (3.3). The optical flow inside a uniform motion region is more accurate than that along the motion boundary.



(a) 10th frame of Flower Garden sequence (b) Optical flow field

Figure 3.6: The Flower Garden sequence and the measured optical flow field

3.4.2 Real Sequence

Figure 3.7 shows results using a real image sequence with the motion field plotting on top of the original image. The image sequence was taken using a camera placed on top of a small toy truck. Between images, the truck was manually moved by a small distance to mimic real truck motion. The truck motion was toward the jig on the ground. The optical field for this sequence is much noisier than the synthetic sequence, especially in the far end background. The main reasons for this are as follows:

- The distances moved between frames were controlled by hand. As a result, the movement constancy between frames cannot be guaranteed;
- The time interval between frames was slightly too long and so lighting changes between frames were more than what the algorithm can tolerate;
- The camera used is a CMOS line exposure imager which introduces extra noise and artifacts into the images;
- There is little texture in the background regions.

This image sequence represents an extreme example of the types of sequences which may be encountered in a real-time machine vision environment. Thus, it has value in demonstrating the shortcomings of optical flow algorithms. However, the solution to this problem is mainly restricted by the limited hardware resources. To overcome this, stronger regularization techniques should be used to suppress the noise and smooth the motion field. The following improvements are thus investigated:

- **Increase the weighting mask size**. From Table 3.1, it can be seen that larger weighting window sizes produce better accuracy;
- **Incorporate temporal smoothing**. Temporal smoothing will significantly suppress the noise in the optical flow vectors. However, temporal smoothing can be costly to implement in hardware;

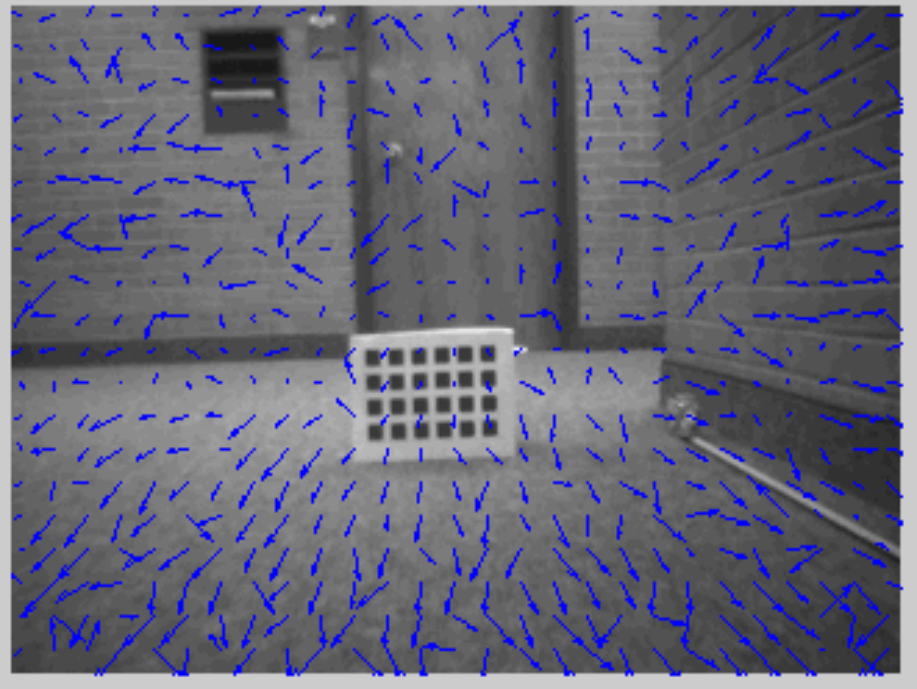


Figure 3.7: Initial result on real image sequence

- **Append more weighting processes**. One weighting process can be added after module 5 as shown in Figure 3.1;

- **Apply the biased least squares techniques**. When the tensor matrix is ill-conditioned, the produced optical flow can be severely distorted. If a weighting scheme is applied afterwards, this result will "pollute" the neighboring vectors. Therefore, biased least squares can be applied to solve this at the cost of a small bias in the result. Details on this approach can be found in [81].

These improvements were incorporated into the MATLAB model and simulated. Figure 3.8 shows the improved results. It can be observed that the generated optical field is much smoother and more accurate. The key point here is not to show the effectiveness of the new algorithm but to illustrate that with more hardware resources, the proposed design can be improved significantly.

The detail of this revision is not given here, but with the abundant hardware resources of the Helios platform that was developed in the Robotic Vision Lab, im-

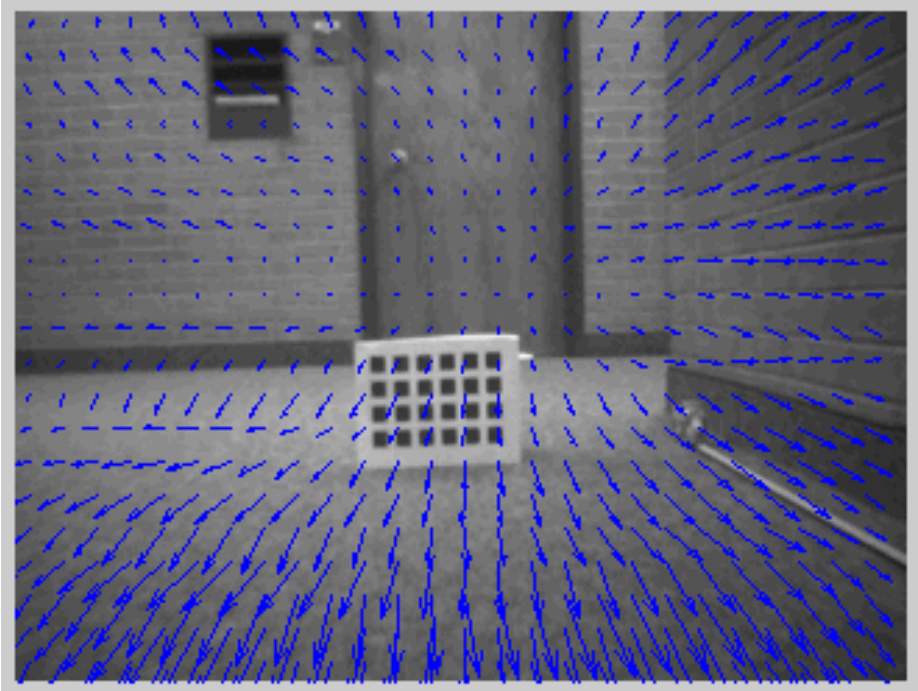


Figure 3.8: Improved result on real image sequence

plementation of the improved algorithm is possible. Further research work is needed to analyze these revisions and find a tradeoff between performance and hardware implementation complexity and achieve a feasible real-time result.

3.5 Futher Improvement

3.5.1 Motivation

A tensor-based optical flow algorithm has been implemented on an FPGA and presented in Section 3.2. This implementation is able to process 64 frames of 640×480 images per second. This computation contains two weighting processes and the weights are determined offline by finding the settings which give the best accuracy compared against the ground truth. In this section, a new algorithm is proposed that uses a cost function to adaptively determine the optimal weights. An efficient scheme is devised to determine the optimal weights faster and more accurately. To better fit this scheme to the hardware structure, an efficient tensor computation method is

proposed. Experimental results are also presented to show the effectiveness of the algorithm.

3.5.2 Formulation

Using a tensor to represent orientation has a notable advantage over scalars and vectors. Tensors not only estimate the orientation but also include the certainty of the estimation. Local brightness patterns are divided into different cases and eigenvalues of the structure tensor in each case are analyzed [28]. Three measures are defined: total coherency measure, spatial coherency measure, and corner measure. They are functions of the eigenvalues of the structure tensor. Middendorf et al. [82] used eigenvalue analysis to divide the optical flow field into five categories for motion segmentation. Kühne et al. [83] derived a coherence measure based on eigenvalues. This measure was integrated into an active contour model for segmentation of moving objects. In [43], a corner measure was applied to adaptively adjust the Gaussian window function to improve tensor accuracy. All of the above work first calculated the eigenvalues of the structure tensor and then devised measures using certain combinations of these eigenvalues. The main challenge of using these measures in hardware is that eigenvalues are difficult to obtain. In this section, the following cost function [84] is used to indicate the certainty

$$c_t(\mathbf{T}) \quad = \quad \frac{t_3 - \mathbf{t}^T \bar{\mathbf{T}}^{-1} \mathbf{t}}{trace(\mathbf{T})} \text{where} \quad \mathbf{T} = \begin{pmatrix} \bar{\mathbf{T}} & \mathbf{t} \\ \mathbf{t}^T & t_3 \end{pmatrix}. \tag{3.9}$$

$\bar{\mathbf{T}}$ is a 2×2 symmetric matrix and its inverse matrix can be easily computed. c_t is the minimum value of $\mathbf{v}^T\mathbf{T}\mathbf{v}$ [84] normalized by the trace of the tensor and indicates the variation along direction $\mathbf{v}$. Therefore, the smaller c_t is, the more reliable the tensor will be and vice versa.

Cost function c_t is not directly related to w_i in Equation (3.2). It is difficult to decouple c_i and w_i and to adjust w_i by evaluating the cost function. Cost function c_t in Equation (3.9) directly indicates the performance of the weights c_i in (3.3) if w_i is assumed to be fixed. Our algorithm adjusts c_i while using fixed w_i. The support of

w_i is chosen to be relatively small to prevent *over smoothing*. The reason for applying two small weighting masks instead of one big mask is that it is more economical to implement such a scheme in hardware.

The c_i are given by the Gaussian function. The shape of the Gaussian function can be characterized by its standard deviation. The underline problem can be stated as *finding the optimal standard deviation for c_i such that the cost function $c_t(\mathbf{T})$ is minimized*.

Liu et al. [43] proposed an adaptive standard deviation updating algorithm. Starting from a small value, standard deviation σ was increased by $\Delta\sigma$ if the confidence measure did not reach the threshold. The support size was then adjusted according to the standard deviation σ. This process was repeated until the confidence measure reached the desired value or a set number of iterations were reached.

In this section, an efficient and accurate algorithm for updating the standard deviation is proposed. It is different from the adaptive algorithm in [43] in four major aspects:

1. It uses a different confidence measure;
2. It has a fixed support size;
3. It uses a better initial value;
4. It uses different searching strategy.

Using a different confidence measure has been discussed above. The support size is fixed because changing the support size usually requires changes in the hardware structure as opposed to simply changing weights.

During the weighting process, the mask is moved sequentially along a certain direction, say from left to right and then from top to bottom. Two adjacent pixels have most of the masked region overlapped. It can be expected that the brightness patterns of the two masked regions are very similar. Therefore, instead of using one initial value for every pixel, the standard deviation $\sigma_f(x-1, y)$ resolved from the previous pixel on the same row is used to initialize the standard deviation $\sigma_1(x, y)$ of

the current pixel. The standard deviation is set to zero for the first pixel of each row. There are two advantages of this method:

1. It dramatically decreases the number of iterations;
2. It increases the dynamic range of the standard deviation and can handle signals in a wider range.

The pseudocode of the algorithm used for searching and for building the tensor for pixel (x, y) is shown in Algorithm 3.1. The threshold for cost function is t_c and the iteration limit is t_{iter}, $\sigma_i(x, y)$ is the standard deviation for pixel (x, y) at the ith iteration.

Algorithm 3.1: Tensor calculation algorithm

```
begin
1   Initialize σ_1(x, y);
2   Compute tensor T_1 using σ_1(x, y) ;
3   Compute c_t as in Equation (3.9) ;
4   if c_t is smaller than t_c then
5       Set the final standard deviation σ_f(x, y) = σ_1(x, y);
        return T_1 ;
6   Judge which mode will be taken ;
7   for i=1 to t_iter do
8       if it is increasing mode then
9           σ_{i+1}(x, y) = σ_i(x, y) + Δσ ;
        else
10          σ_{i+1}(x, y) = σ_i(x, y) − Δσ ;
11      Compute tensor T_{i+1} using σ_{i+1}(x, y) ;
12      Compute c_t as in Equation (3.9) ;
13      if c_t is smaller than t_c then
            σ_f(x, y) = σ_{i+1}(x, y) ;
            return T_{i+1} ;
        else if c_t is increasing then
            σ_f(x, y) = σ_i(x, y) ;
            return T_i
    σ_f(x, y) = σ_{i+1}(x, y) ;
end
```

By using this scheme, for each iteration, a better initial value is given, the distance to the ideal standard deviation is shortened, and the number of iterations is decreased.

The tensor can be computed using Equation (3.3) or the method introduced below. There are two modes for updating the standard deviation; increasing mode and decreasing mode. One way to decide which mode should be taken is to compute tensors and cost functions using both modes and choose the one with smaller cost. The iteration will be terminated when one of the following three conditions is met:

1. The cost decreases below a preset threshold;
2. A maximum iteration limit is reached;
3. The cost increases compared to the last iteration.

If the weighting process in Equation (3.3) is directly implemented in hardware, for a $(2n+1)\times(2n+1)$ mask, all $(2n+1)^2$ data in the mask need to be stored in hardware registers at each iteration for computing the tensor in the next iteration. With an increase of mask size, the number of hardware registers will increase quadratically.

A method to implement the weighting process efficiently is proposed here. First, the weighting mask is divided into concentric rings centered at the center of the mask. The weights on each ring are similar because their distances to the center of the mask are roughly the same. Therefore, the parameters $c_{j,k}$ and outer products $\mathbf{O}_{j,k}$ of the averaged gradient on the j^{th} ring $\mathbf{R}_j$ can be replaced by $\bar{c}_j$ and $\bar{\mathbf{O}}_j$ shown as

$$\mathbf{T} = \sum_i c_i \mathbf{O}_i = \sum_j \sum_k c_{j,k} \mathbf{O}_{j,k} \approx \sum_j (\bar{c}_j \sum_k \mathbf{O}_{j,k}) = \sum_j m_j \bar{c}_j \bar{\mathbf{O}}_j \tag{3.10}$$

where m_j is the number of the weights on ring $\mathbf{R}_j$.

A $(2n+1)\times(2n+1)$ weighting mask can be divided into $2n$ rings. If $c(x,y)$ is denoted as the parameter at (x,y) on the mask and S as the summation of the absolute values of the x and y coordinates (which will have a range of $[0,2n]$), the

division of the rings can be formulated as follows:

$$\mathbf{R}_1 = \{c(x,y) : S(x,y) = 0\}, \tag{3.11}$$

$$\mathbf{R}_j = \{c(x,y) : S(x,y) = j-1, x = 0 \text{ or } y = 0\} \cup \tag{3.12}$$
$$\{c(x,y) : S(x,y) = j, x \neq 0, y \neq 0\},$$
$$j = 2,3,\ldots,2n; x,y \in [-n,n]\,; x,y \in \mathbf{Z},$$

$$\mathbf{D}_j = \{(x,y) : c(x,y) \in \mathbf{R}_j\}, j = 1,2,\ldots,2n. \tag{3.13}$$

$\bar{c}_j$ is calculated as

$$\bar{c}_j(\sigma) = \frac{1}{\sqrt{2\pi}\sigma}\epsilon^{-\frac{\bar{d}^2}{2\sigma^2}}, \text{ where } \bar{d}^2 = \frac{1}{m_j}\sum_{(x,y)\in\mathbf{D}_j}(x^2+y^2). \tag{3.14}$$

The distribution of rings of a 7×7 mask is shown in Figure 3.9 as an example. Values of S are shown in each grid of the image. Locations that belong to the same ring are connected by a dashed line in the figure. When the standard deviation is changed from σ_1 to σ_2, a new tensor can be calculated as

$$\mathbf{T} = \sum_j \frac{\bar{c}_j(\sigma_2)}{\bar{c}_j(\sigma_1)} m_j \bar{\mathbf{O}}_j. \tag{3.15}$$

As a result of using this simplified tensor computation, the required number of hardware registers increases linearly instead of quadratically with increasing mask size. In this implementation, spatial smoothing instead of spatiotemporal smoothing is used due to the difficulty of implementing temporal smoothing in hardware. Nevertheless, this computation method works for temporal smoothing as well. Although the accuracy of the ring approximation decreases as the mask size increases, acceptable accuracy is obtained with the current settings as shown in the experiment. Using a large mask size is not necessary.

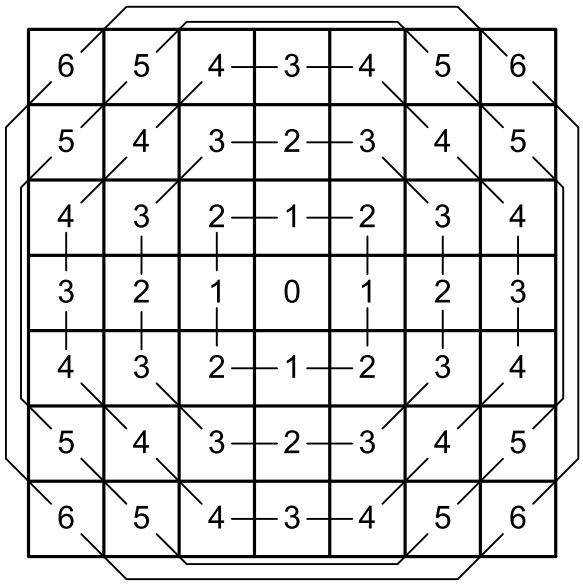


Figure 3.9: The distribution of rings of a 7 by 7 mask

3.5.3 Experiment

The proposed implementation was simulated in MATLAB to evaluate its performance. The algorithm was tested on the Yosemite sequence as well as the Flower Garden sequence.

The results for the Yosemite sequence are shown in Table 3.2. The first weighting mask in the proposed algorithm is a 5×5 mask whose parameters are ones and $\Delta\sigma = 0.5$, $t_c = 0.001$, and $t_{\text{iter}} = 10$. Two limits $\sigma_{\max} = 8$, $\sigma_{\min} = 1$ are set for the standard deviation σ to ensure it is within a reasonable range.

Table 3.2: Performances of the adaptive algorithm under different configurations.

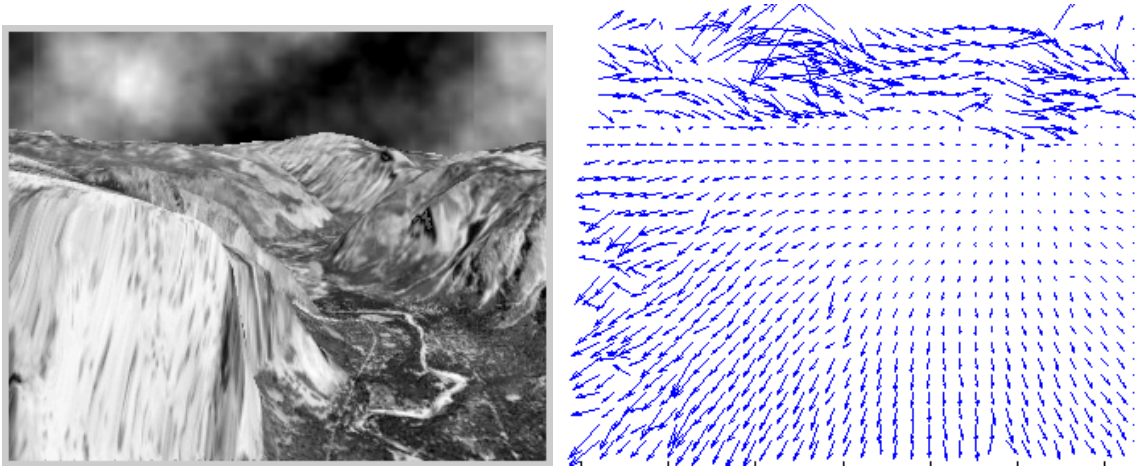
n	ACUR[1]	ACUR[2]	ACUR[3]	ACUR[4]	AVG ITER[1]	AVG ITER[2]
2	11.73°	11.73°	11.89°	22.34°	2.31	4.12
3	9.12°	9.12°	9.13°	22.69°	2.08	4.04
4	7.65°	7.65°	7.56°	22.93°	1.85	3.97
5	6.75°	6.77°	6.66°	23.22°	1.66	3.90

The accuracy of these results is measured in angular error as shown in columns 2-5, where n is the half size of the second mask (column 1). ACUR^1 is the accuracy of the adaptive algorithm, and ACUR^2 is the accuracy of the same algorithm without the simplified tensor computation. ACUR^3 is the highest accuracy obtained for the algorithm introduced in Section 3.2 by tweaking different standard deviations manually, and ACUR^4 is the lowest accuracy of this algorithm. AVG ITER^1 represents the average iterations per pixel using the proposed algorithm. AVG ITER^2 represents the average iterations per pixel using the searching scheme in [43].

This improvement uses a simplified tensor computation method which reduces the number of hardware registers required for implementation. The accuracy is compared with (ACUR^1) and without simplified tensor computation method (ACUR^2); we conclude that the proposed simplification process does not affect the accuracy. This new algorithm ACUR^3 and ACUR^4 are obtained by setting the first weighting mask in algorithm in Section 3.2, the same as that in the adaptive algorithm and increasing the standard deviation for the second weighting process from 0.5 to 8 at an interval of 0.5. ACUR^1 is very close to the highest accuracy (ACUR^3) and much better than the lowest (ACUR^4). This demonstrates the effectiveness of the new weights searching strategy. The average number of iterations required for the proposed searching strategy is compared to those in [43] using AVG ITER^1 and AVG ITER^2, respectively. The 8th frame of the Yosemite sequence and the 10th frame of the Flower Garden sequence and their optical flow fields using the adaptive algorithm are shown in Fig. 3 (and others settings are the same as the above).

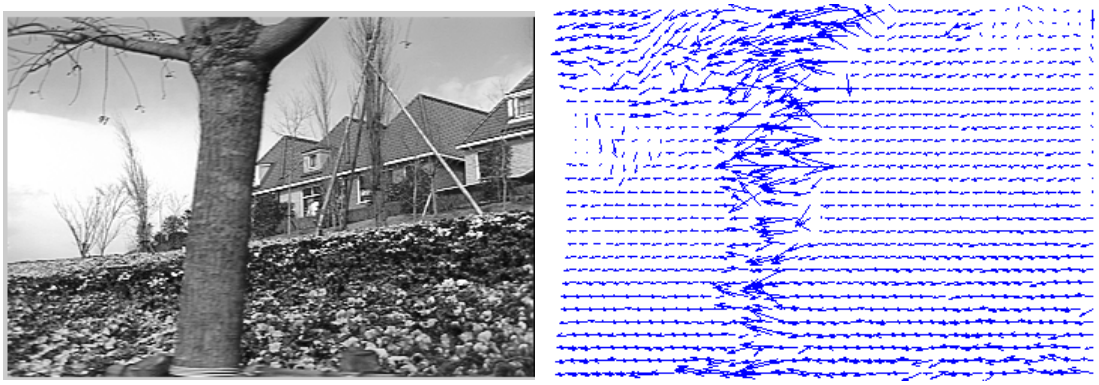
3.6 Summary

Before hardware implementation, the expected design performance in terms of accuracy and resource utilization was carefully evaluated in software. The tradeoff study was conducted through tuning several important parameters such as the kernel sizes of the weighting processes and the derivative operator. Software simulation results were also used for hardware performance verification.



(a) 8th frame of the Yosemite sequence (b) Optical flow field

Figure 3.10: The Yosemite sequence and the measured optical flow field using the adaptive optical flow algorithm



(a) 10th frame of Flower Garden sequence (b) Optical flow field

Figure 3.11: The Flower Garden sequence and the measured optical flow field using the adaptive optical flow algorithm

This new algorithm was tested on both synthetic and real image sequences. This design worked well on synthetic sequences but was not satisfactory on the real sequence. Revisions were required to improve the system design which, in turn, require more hardware resources. This part of research explored the possibility of future improvement with more hardware resources and built a good foundation for a better algorithm reported in Chapter 4.

Chapter 4

Improved Optical Flow Algorithm

4.1 Algorithm Overview

In the previous chapter, a FPGA system was built to process optical flow vectors of 64 frames of 640-by-480 image per second. Compared to software-based algorithms, this system achieved much higher frame rate but marginal accuracy. In this chapter, a more accurate optical flow algorithm is proposed to substantially improve the accruacy over the previous design.

A few major efforts having been made in this new algorithm. *First*, temporal smoothing is incorporated in the hardware structure which is proved to significantly improve the algorithm accuracy based on the simulation results. To accommodate temporal smoothing, the hardware structure is divided into two parts. The DERivative (DER) module produces intermediate results which are then processed by the Optical Flow Computation (OFC) module to obtain the final optical flow vectors. Software running on a built-in processor on the FPGA chip is used in the design to direct the data flow and manage hardware components. *Second*, a ridge estimator is developed to cope with the collinear issue which is a more severe problem for hardware-based design because of the resource limitations. The ridge estimator can be efficiently implemented using current hardware architectures. *Third*, more spatial smoothing stages are incorporated into the hardware pipeline to improve the system performance.

All these aspects are possible because of the availability of the new FPGA platform. It has been implemented on a compact, low power, high performance hardware platform for micro UV applications. The current hardware platform is able to process 15 frames of 640-by-480 image per second but with significant improvement

in accuracy over the previous work. Higher frame rate can be achieved with further optimization work and additional memory space. This new design is tested with synthetic sequences and real sequences to show its effectiveness.

4.2 Algorithm Formulation

Based on the BCC as in (1.1), an equation regarding derivatives g_x, g_y, and g_t and velocity components v_x and v_y was derived in [8] as

$$\begin{aligned} I_x v_x + I_y v_y + I_t + \varepsilon = 0 \\ \Rightarrow -I_t = I_x v_x + I_y v_y + \varepsilon, \end{aligned} \tag{4.1}$$

where $v_x = \frac{\Delta x}{\Delta t}$, $v_y = \frac{\Delta y}{\Delta t}$, and ε is the error accounting for the higher order terms and noise. Each pixel in the image has one set of observation g_{ti}, g_{xi}, g_{yi}. In a small neighborhood of n pixels, it is assumed that they all have the same velocity v_x and v_y. Then the n sets of observations for these n pixels can be expressed as

$$\mathbf{I}_t = \mathbf{I}_x(-v_x) + \mathbf{I}_y(-v_y) + \epsilon, \tag{4.2}$$

where $\mathbf{I}_t = (I_{t1}, I_{t2}, \cdots, I_{tn})^T$, $\mathbf{I}_x = (I_{x1}, I_{x2}, \cdots, I_{xn})^T$, $\mathbf{I}_y = (I_{y1}, I_{y2}, \cdots, I_{yn})^T$, $\epsilon = (\varepsilon_1, \varepsilon_2, \cdots, \varepsilon_n)^T$.

It is assumed that the expectation of ε_j satisfies $E(\varepsilon_j) = 0$ and the variance is σ^2 i.e. $\varepsilon \sim (0, \sigma^2)$. Denoting $\mathbf{Y}^{n\times 1} = \mathbf{I}_t$, $\mathbf{X}^{n\times 2} = (\mathbf{I}_x, \mathbf{I}_y)$, $\boldsymbol{\theta} = -(v_x, v_y)^T$, the equation regarding the observation (I_{ti}, I_{xi}, I_{yi}) and the parameter $\boldsymbol{\theta}$ can be written as

$$\mathbf{Y} = \mathbf{X}\boldsymbol{\theta} + \epsilon. \tag{4.3}$$

A normal least squares solution of $\boldsymbol{\theta}$ in Equation (4.3) is

$$\hat{\boldsymbol{\theta}}_{LS} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}. \tag{4.4}$$

As shown in [81], $E(\hat{\boldsymbol{\theta}}_{LS}) = \boldsymbol{\theta}$ and its covariance matrix is $\text{Cov}(\hat{\boldsymbol{\theta}}_{LS}) = \sigma^2(\mathbf{X}^T\mathbf{X})^{-1}$. If $\mathbf{I}_x$ and $\mathbf{I}_y$ exhibit near linear dependency, i.e. one vector is nearly a scale of the other; small amounts of noise in the observation will cause relatively large changes in the inversion $(\mathbf{X}^T\mathbf{X})^{-1}$, and produce very large and inaccurate motion vectors. For hardware-based algorithms, because of resource limitations, the vector length n is usually much smaller than in software-based algorithms. This, in turn, increases the possibility of a collinear $(\mathbf{X}^T\mathbf{X})$ matrix. The resulting abnormal motion vectors will have a negative impact on neighboring motion vectors in the subsequent smoothing process.

One simple solution is to simply restrict the magnitude of each motion vector, but this is not an optimal solution. In this dissertation, a ridge estimator [81, 85, 86] as formulated in (4.5) is proposed to address this.

$$\hat{\boldsymbol{\theta}}_{RE} = (\mathbf{X}^T\mathbf{X} + k\mathbf{I}_p)^{-1}\mathbf{X}^T\mathbf{Y}. \tag{4.5}$$

In (4.5), $\mathbf{I}_p$ is a unit matrix of the same size as $\mathbf{X}^T\mathbf{X}$ where p equals 2 in this case. k is a weighting scalar for $\mathbf{I}_p$. It is shown in [81] that the expectation and covariance matrices of $\hat{\boldsymbol{\theta}}_{RE}$ are

$$E(\hat{\boldsymbol{\theta}}_{RE}) = \boldsymbol{\theta} - k(\mathbf{X}^T\mathbf{X} + k\mathbf{I}_p)^{-1}\boldsymbol{\theta}, \tag{4.6}$$

$$\text{Cov}(\hat{\boldsymbol{\theta}}_{RE}) = \sigma^2\mathbf{X}^T\mathbf{X}(\mathbf{X}^T\mathbf{X} + k\mathbf{I}_p)^{-2}. \tag{4.7}$$

Although a ridge estimator is biased, i.e. $E(\hat{\boldsymbol{\theta}}_{RE}) \neq \boldsymbol{\theta}$ as shown in (4.6), it is better than a least squares estimator if evaluated based on risk instead of observed loss. Risk is defined as the expectation of loss which is independent of the observed $\mathbf{Y}$. More details are given in [81].

As to the selection of k, an HKB estimator [87] shown as

$$k = \frac{p\hat{\sigma}^2}{\hat{\boldsymbol{\theta}}_N^T\hat{\boldsymbol{\theta}}_N} \tag{4.8}$$

was chosen. In the content, $\hat{\boldsymbol{\theta}}_N$ is the estimate right above the current pixel and it is preset to $(1,1)^T$ on the first row. The error variance is estimated as

$$\hat{\sigma}^2 = \frac{(\mathbf{Y} - \mathbf{X}\hat{\boldsymbol{\theta}}_N)^T(\mathbf{Y} - \mathbf{X}\hat{\boldsymbol{\theta}}_N)}{n-p}. \tag{4.9}$$

There exist other methods to estimate the scalar e.g. an iterative HK estimator [88], an LW estimator [89], etc. The HKB estimator is chosen for its efficiency and non-iterative property. After obtaining k, the optical flow is estimated using Equation (4.5).

In a real implementation, an n-by-n weighting matrix $\mathbf{W}$ is used to assign weights to each set of observation based on their distance to the central pixel. Equations (4.5) and (4.9) are rewritten as

$$\hat{\boldsymbol{\theta}}_{RE} = (\mathbf{X}^T\mathbf{W}\mathbf{X} + k\mathbf{I}_p)^{-1}\mathbf{X}^T\mathbf{W}\mathbf{Y}, \tag{4.10}$$

$$\hat{\sigma}^2 = \frac{(\mathbf{Y} - \mathbf{X}\hat{\boldsymbol{\theta}}_N)^T\mathbf{W}(\mathbf{Y} - \mathbf{X}\hat{\boldsymbol{\theta}}_N)}{n-p}. \tag{4.11}$$

To suppress noise, the derivatives I_x, I_y, and I_t are spatiotemporally smoothed, respectively before they are used in (4.2). Motion vectors are also spatially smoothed to obtain a smooth motion field. The proposed algorithm uses only simple arithmetic operations which are ideal for hardware implementation.

To evaluate the quality of a motion vector, a cost function can also be calculated as

$$\begin{aligned} c_t(\mathbf{T}) &= \frac{m - \mathbf{t}^T\mathbf{Q}^{-1}\mathbf{t}}{trace(\mathbf{T})}, \\ \text{where} \quad \mathbf{T} = \begin{pmatrix} \mathbf{Q} & \mathbf{t} \\ \mathbf{t}^T & m \end{pmatrix} &= \begin{pmatrix} \mathbf{X}^T\mathbf{W}\mathbf{X} + k\mathbf{I}_p & \mathbf{X}^T\mathbf{W}\mathbf{Y} \\ (\mathbf{X}^T\mathbf{W}\mathbf{Y})^T & \mathbf{Y}^T\mathbf{W}\mathbf{Y} + k \end{pmatrix} \\ &= \begin{pmatrix} t_1 + k & t_4 & t_5 \\ t_4 & t_2 + k & t_6 \\ t_5 & t_6 & t_3 + k \end{pmatrix}. \end{aligned} \tag{4.12}$$

The initial optical flow vector is smoothed in a local neighborhood to suppress noise further. The final optical flow vector is formulated as

$$\bar{\mathbf{v}} = \begin{pmatrix} \bar{v}_x \\ \bar{v}_y \end{pmatrix} = \sum_i m_i \begin{pmatrix} \bar{v}_{xi} \\ \bar{v}_{yi} \end{pmatrix}. \tag{4.13}$$

The algorithm assumes a constant motion model. An affine motion model is often used to incorporate tensors in a small neighborhood [45] where pixels in a neighborhood are assumed to belong to the same motion model. To conserve hardware resources, the constant model is used in this design. The constant model performs almost as well as affine motion model when operating in a small neighborhood.

4.3 System Design and Optimization

The proposed compact optical flow sensor is an embedded vision system including video capture, processing, transferring and other functionalities. Various modules are connected to the buses as shown in Figure 4.1. There are two types of buses in the system: PLB bus and OPB bus. The PLB bus connects high speed modules such as DER module (DERivatives calculation), OFC module (Optical Flow Calculation), SDRAM, camera and USB modules. Lower speed modules such as UART, interrupt controller, and GPIO are connected to the OPB bus. The PLB and OPB buses are interconnected through a bridge.

A main difference between this design and the design in Chapter 3 is it incorporates temporal smoothing in the pipeline. Temporal smoothing is significantly more complicated than spatial smoothing because it involves multiple image frames. However, temporal smoothing is important for estimating motion field consistency over a short period of time. Incorporating temporal smoothing substantially improves algorithm performance as can be seen in the experimental analysis section. The number of frames used for temporal smoothing is largely determined by hardware resources and processing speed requirement. With temporal smoothing, multiple sets (3 in this design) of derivative frames must be stored as they are calculated and then reloaded during the smoothing process. It is impossible to store multiple sets of derivative

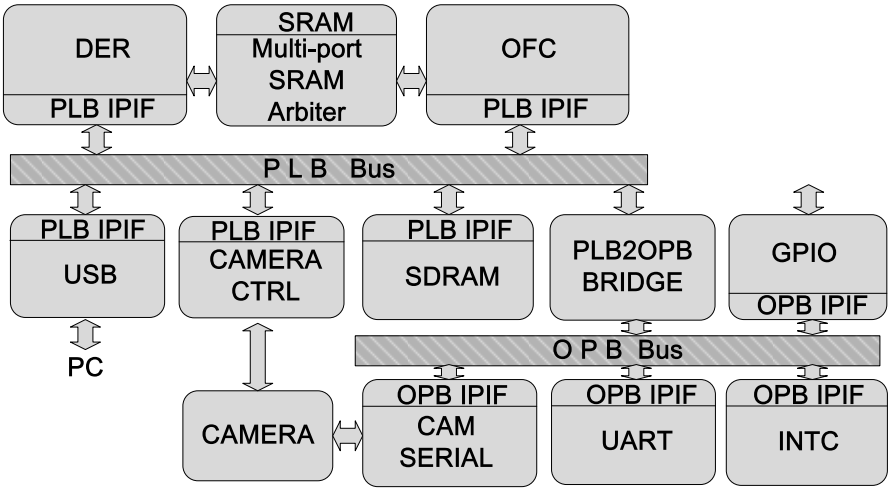


Figure 4.1: Ridge regression-based system diagram

frames on-chip using the hardware resources inside the FPGA. A drawback of the temporal smoothing is the increase in the memory throughput required and the resulting decrease in processing speed. However, as long as these modules are able to keep up with the camera frame rate, temporal smoothing can substantially improve the accuracy without impacting real-time performance.

To accommodate temporal smoothing, the hardware pipeline is divided into two parts (DER, OFC). The DER module generates derivative frames and the OFC module handles the rest of calculations. Results from the DER module are stored in SRAM and SDRAM. The DER and OFC modules share the high-speed SRAM through a multi-port SRAM arbiter. The OFC module stores the resulting motion vectors in SDRAM. The intermediate or final results can be transferred to the host PC through the USB interface. A graphical user interface has been developed to observe and store the video and display status variables transferred from the sensor. These two hardware modules must be managed to synchronize their computation tasks and handle exceptions such as dropped frames. Software running on the built-in on-chip PowerPC processors is used for this management task.

Figure 4.2 shows the diagram of the DER module. Every cycle when a new image frame(t) is captured directly into the SDRAM through the PLB bus, reading

logic reads the captured image from the SDRAM into a pipeline and stores it in the SRAM. Whenever there are five consecutive image frames stored in the SRAM, they are all read out for computing the derivative frames I_x, I_y and I_t. I_x and I_y are calculated from frame(t-2) and I_t is calculated from frame(t-4), frame(t-3), frame(t-1) and the current incoming frame(t) using the mask shown in (3.7). The resulting derivative frames are stored in the SRAM as well as the SDRAM for future usage. The duplicate copy stored in the SDRAM is needed for temporal smoothing for future frames. This storing and retrieving of derivative frames from SDRAM consumes PLB bus bandwidth and hence slows down the processing speed. As shown in Figure 4.1 and Figure 4.2, if the hardware platform used had sufficient SRAM (currently only 4 Mb), then all 9 derivative frames (3 sets of I_x, I_y and I_t) could be stored in the SRAM and take the advantage of a high-speed multi-port memory interface.

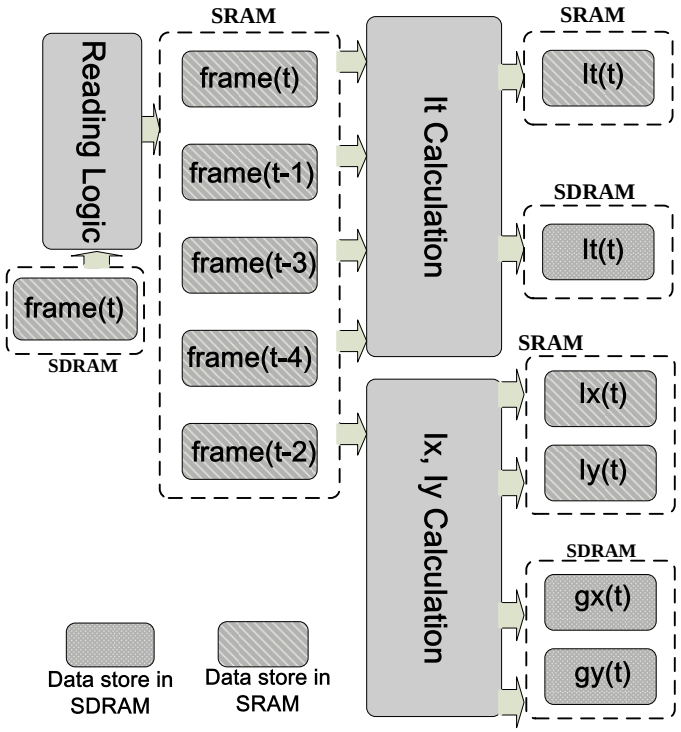


Figure 4.2: DER module diagram

The OFC module diagram is shown in Figure 4.3. Once a new set of derivative frames is calculated, software will trigger the OFC module to start the calculation of optical flow. Derivative frames for the current frame in the SRAM ($I_x(t)$, $I_y(t)$ and $I_t(t)$) and the derivative frames already stored in the SDRAM ($I_x(t-1)$, $I_y(t-1)$ and $I_t(t-1)$) and ($I_x(t-2)$, $I_y(t-2)$ and $I_t(t-2)$) are first read into the pipeline for temporal smoothing. Because the size of the temporal smoothing mask is 3, derivative frames at time t, $t-1$, $t-2$ are temporally smoothed and then spatially smoothed to obtain the smoothed derivative frames for the current frame at time t (Ix_t, Iy_t, and It_t in Figure 4.3).

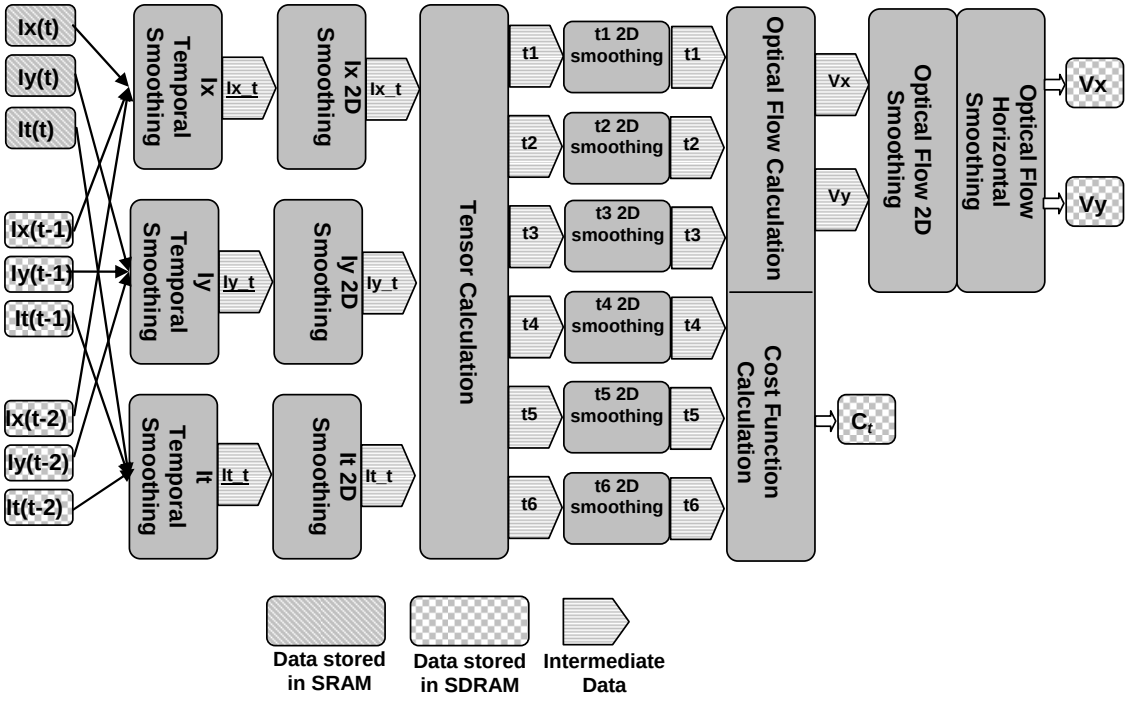


Figure 4.3: OFC module diagram

In the OFC module, besides temporal smoothing, there are four spatial smoothing units: derivative frames 2D smoothing, regression model components 2D smoothing, and optical flow 2D smoothing and optical flow horizontal smoothing. To reach

an optimal tradeoff between algorithm performance and hardware resources, the configurations of these masks were evaluated carefully by simulation before implementation. Smoothing mask parameters were determined by three factors: mask shape, mask size, and mask kernel components. In software, large smoothing masks (e.g. 19-by-19 or larger) are often used. In hardware, smaller masks must be used because of resource limitations (e.g. 7-by-7 or smaller). In this design, the size of the first smoothing mask is 5-by-5. The sizes of the other two spatial smoothing masks are 3-by-3 and 7-by-7, respectively. As for mask shape, a square mask is usually used for the sake of simplicity and efficiency. Parameters of all the smoothing masks are in a shape of Gaussian function. To save hardware resources, a 2D Gaussian mask is decomposed into two 1D Gaussian masks which are cascaded and convolved along x and y directions separately which is the same as the scheme in Chapter 3. Different settings of these masks are simulated by software at bit-level accuracy and evaluated on synthetic sequence with ground truth to obtain an optimal combination in practice.

The optical flow horizontal smoothing is added to better filter out the noise in the motion field. It could be configured to be either 7-by-1, 9-by-1, or 11-by-1 depending on the noise level. The reason to use a horizontal smoothing unit is to save the hardware resource and achieve better smoothing effect. Weighting parameters for the horizontal smoothing unit are all one. The cost function is calculated at the same time with the optical flow calculation.

These smoothed derivatives frames are combined to build ridge regression model components. These components are spatially smoothed and then fed into the scalar estimator in (4.8) and (4.11) to calculate scalar k. Then, $\bar{\mathbf{v}}$ can be calculated from these smoothed components and k as shown in Equations (4.12), (4.10) and (4.13) and Figure 4.3.

Figure 4 shows the system software diagram. There are three types of frames in the system:

1. Image frames captured by the camera;

2. Derivative frames calculated by the DER module;

3. Optical flow fields calculated by the OFC module.

The DER module uses the raw images as input and the OFC module uses the output from the DER module (derivative frames) as the input. Three linked lists are used to store these frames and maintain their temporal correspondence. An FTE (Frame Table Entry) is used to store image frames, a DER_FTE is used to store derivative frames and an OFC_FTE is used to store optical flow frames. As shown in Figure 4, there are 5 corresponding pairs of FTE and DER_FTE and 3 pairs of DER_FTE and OFC_FTE.

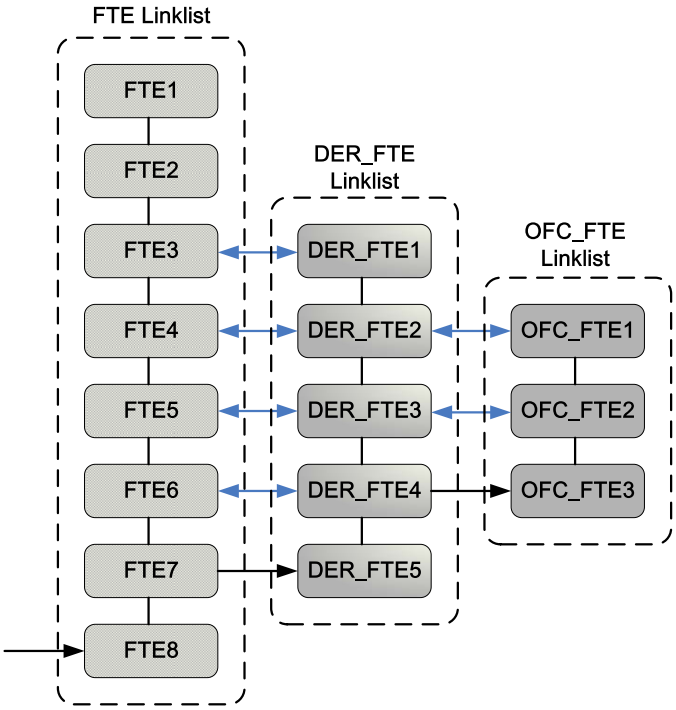


Figure 4.4: System software diagram

It is noted that these modules execute asynchronously. When a new raw image is captured (FTE7 in this case), the camera core invokes an interrupt. This interrupt

is sensed by the FTE interrupt handler in software and a trigger signal is generated and sent to the DER module to initiate a derivative computation. When a new set of derivative frames is calculated (DER_FTE4 in this case), the DER module invokes an interrupt. This interrupt is sensed by the DER_FTE interrupt handler in software and a trigger signal is generated and sent to the OFC module to initiate an optical flow computation.

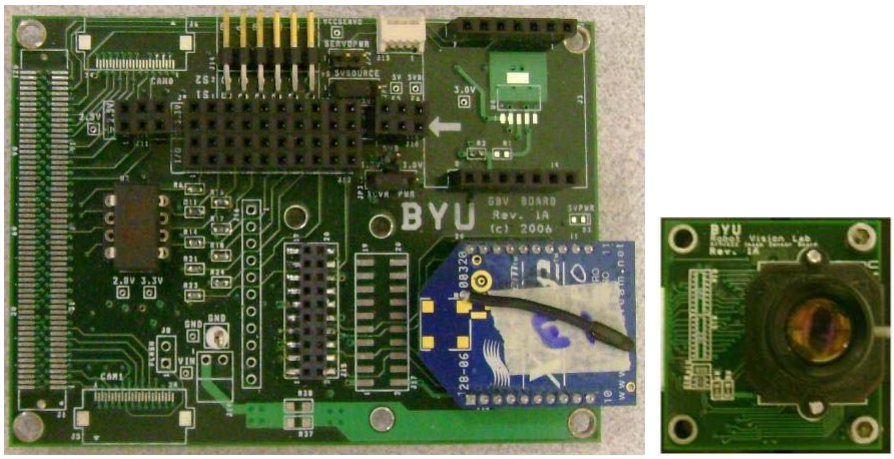
4.4 Experimental Results

This design was implemented on the BYU Helios Robotic Vision board as introduced in Chapter 2. The Helios board was installed with an Autonomous Vehicle Toolkit (AVT) daughterboard (Figure 4.5(b)) which is capable of supporting up to two CMOS cameras (Figure 4.5(a)) and that was designed to enhance the functionalities of the Helios board. The CMOS camera (capable of acquiring 60 frames per second) was set to capture 30 frames of 640-by-480 8-bit color image data per second in this design.

The whole system executes at a clock speed of 100MHz and the system is using one clock domain. At this frequency, it is able to calculate 15 frames of optical flow field per second. All the computation uses fixed point representation to save hardware resources. Temporal smoothing requires much more data transfer than spatial smoothing, and the PLB bandwidth is the limiting factor in increasing processing speed. Our estimation is that higher frame rates can be achieved with further optimization work and additional memory space (specifically on-board SRAM). These processing speed improvements are discussed in the next section. From the performance analysis in the following sections, it can be seen that with temporal smoothing the accuracy of the estimated optical flow is improved substantially on a synthetic sequence that is commonly used for benchmarking. The whole design utilized 22,506 slices (89% of the total 25,280 slices on a Virtex-4 FX60 FPGA). The DER module used 2,196 slices (8% of the total) and the OFC module used 6,946 slices (27.5% of the total). The remainder was used for the camera core, I/O, and other interface circuitry. A graphical user interface (GUI) was developed by other member in the lab

to transfer the status of the Helios board and the real-time video to PC for display through the USB interface.

The design was simulated in MATLAB. For debugging and evaluating purposes, a bit-level accurate MATLAB simulation code was programmed to match the actual hardware circuits. In this paper, our focus is on evaluating the effectiveness of the new smoothing units and cost function calculation module.

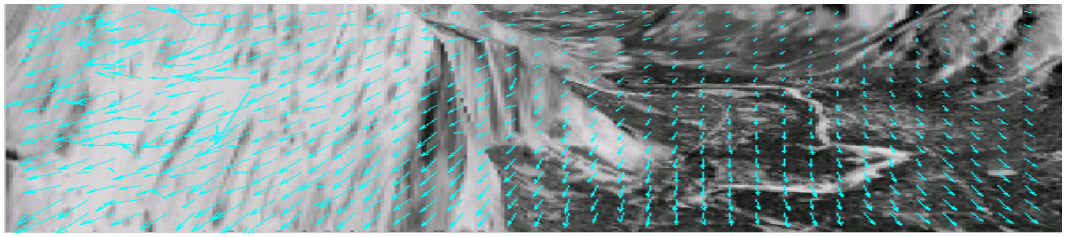


(a) AVT daughter board (b) CMOS imager

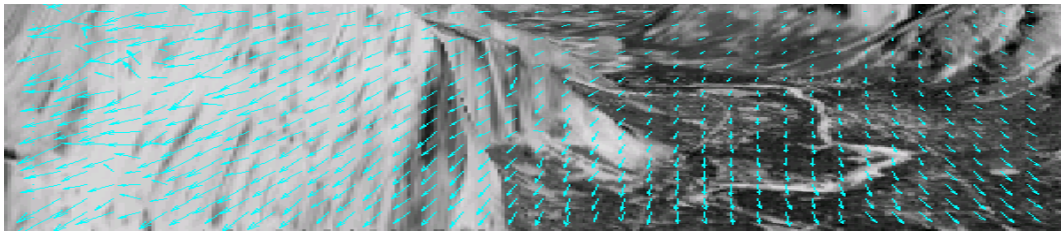
Figure 4.5: Hardware components

4.4.1 Synthetic Sequence

Figure 4.6 shows the effect of ridge regression tested on the bottom region of the Yosemite sequence. Figure 4.6(a) shows the result using the least squares method. Figure 4.6(b) shows the result using the proposed ridge regression method. It can be seen that these two algorithms perform comparably on the right half of image which has abundant textures. On the left half which does not have many distinguishable textures, the ridge regression based method generates smoother results compared to the least squares method (fewer spurious vectors).



(a) Before ridge regression



(b) After ridge regression

Figure 4.6: Performance improvement using ridge regression

Table 4.1 shows the accuracies on the Yosemite sequence under different settings. The accuracies are measured in average angular error. The velocity horizontal smoothing is not applied in A_1-A_4. A_1 is the accuracy of the proposed design using ridge regression and temporal smoothing. A_2 is the accuracy using the least squares method with temporal smoothing. A_3 is the accuracy using ridge regression without applying temporal smoothing. A_4 is the accuracy using least squares and without temporal smoothing. A_5 includes every features introduced in this chapter (ridge regression, temporal smoothing and velocity horizontal smoothing).

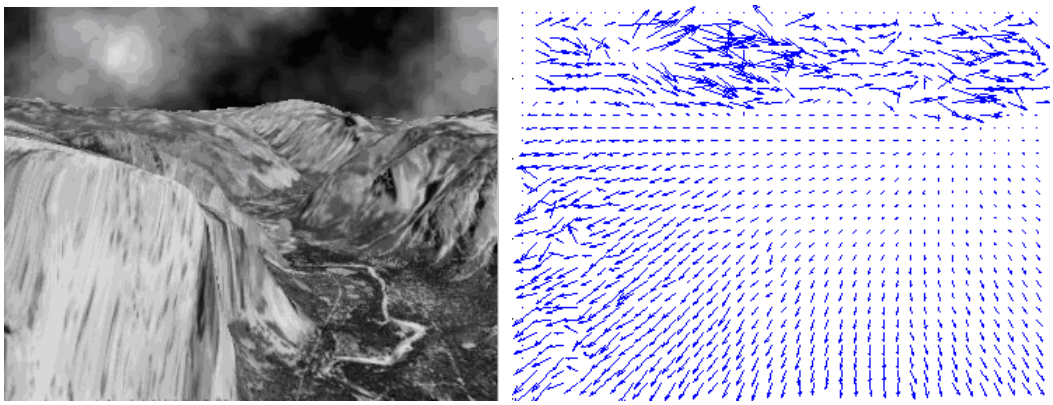
Table 4.1: Accuracy comparison of different configurations

A_1	A_2	A_3	A_4	A_5
6.8°	7.2°	10.9°	11.4°	4.8°

From the table, it can be concluded that each individual feature contributes to the performance improvement by a specific amount. Therefore, it is a progressive

improvement. Also, adding a feature costs extra hardware resources and it requires tradeoff analysis to decide which feature is best under limited resources.

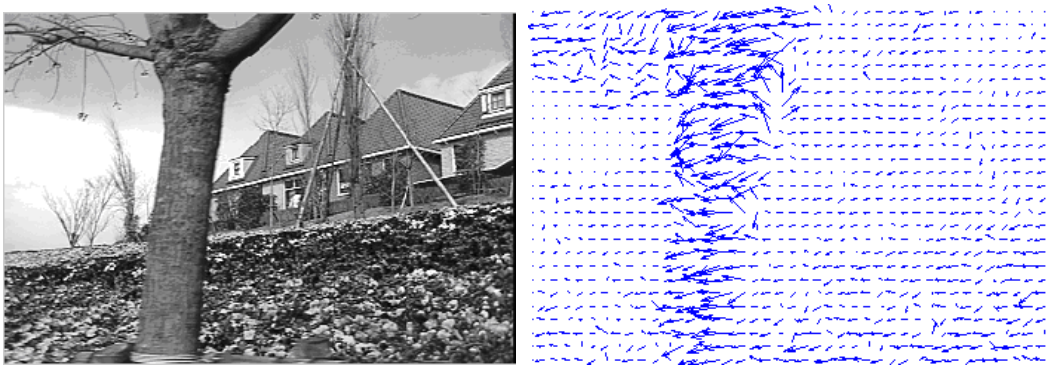
Figure 4.7 shows the raw image and the motion estimation result of the Yosemite sequence.



(a) Raw image (b) Calculated motion field

Figure 4.7: Result on the Yosemite sequence

Figure 4.8 shows the raw image and the motion estimation result of the Flower Garden sequence.

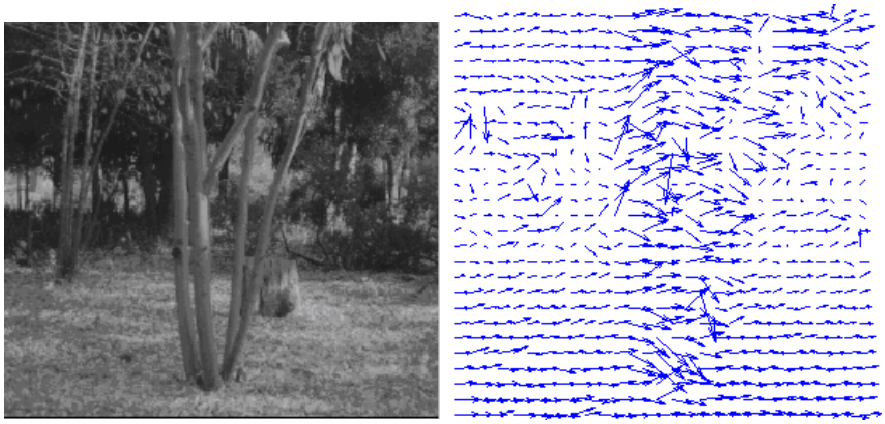


(a) Raw image (b) Calculated motion field

Figure 4.8: Result on the Flower Garden sequence

4.4.2 Real Sequence

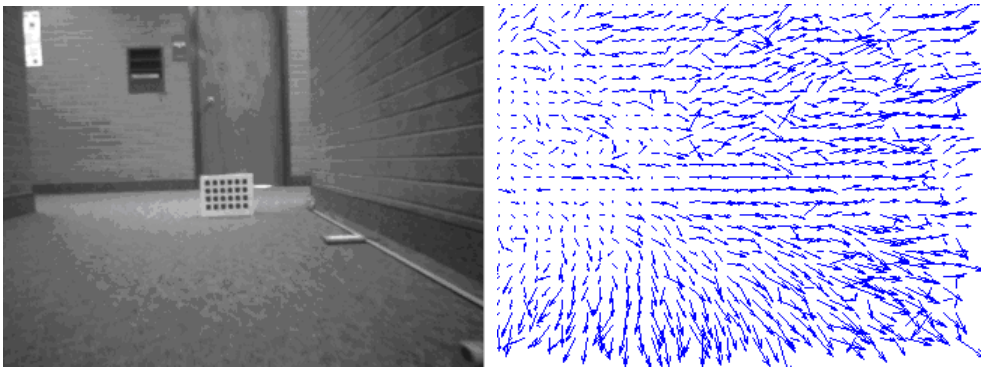
Figure 4.9 shows the raw image and the motion estimation result of the SRI Tree sequence.



(a) Raw image (b) Calculated motion field

Figure 4.9: Result on the SRI Tree sequence

Figure 4.10 shows the raw image and the motion estimation result of the BYU Corridor sequence.



(a) Raw image (b) Calculated motion field

Figure 4.10: Result on the BYU Corridor sequence

4.5 Summary

An accurate optical flow algorithm implemented on Helios platform is discussed in this chapter. To minimize the problem of deficient smoothing in hardware-base designs, ridge regression is applied to avoid abnormal motion vectors. A new hardware structure is devised to incorporate temporal smoothing. More spatial smoothing units are imposed to filter out the noise. These efforts significantly improve the accuracy of this design compared to its previous version.

Besides utilizing more resources, the cost of improving the accuracy is the increasing design complexity and lower system throughput. To accommodate temporal smoothing, the pipelined hardware structure is divided into two parts: DER and OFC modules. Extra software and hardware codes are developed to design the interface, synchronize different modules and direct the data flow. All these aspects require more time to implement, simulate and verify, therefore substantially increase the project development time. Because the pipeline is divided into two parts, the amount of image data that must be stored and transferred increases dramatically. Currently, this design is running slower (15 fps of 640-by-480 images) than the previous design. The limiting factor of the processing speed is the PLB bus bandwidth. There reason is, as a bus centric system, different modules are connected to the bus to receive or transfer data. When the DER and OFC modules communicate through the PLB bus and data throughput is approaching the limit of bus bandwidth, the pipelines behind it will run empty for a large amount of time. This limiting factor can be alleviated by increasing the SRAM size. In this design, SRAM is working as a specialized high speed buffer between the DER and OFC modules. Increasing the SRAM size could avoid the storing and retrieving of derivative frames from the slower SDRAM through the PLB bus. Also, increasing the SRAM size would allow the use of a larger temporal smoothing kernel which would significantly improve the optical flow computation accuracy.

Chapter 5

Obstacle Detection Using Optical Flow

5.1 Motivation

A robot relies on various sensors for obstacle detection, localization (keeping track of the position), mapping (learning the environment) and path planning. These sensors include vision-based sensor, GPS, active beacon sensors (e.g. LIDAR and sonar), odometric sensors, and inertial sensors. Every sensor has its own applicability, advantages and disadvantages. For example, GPS is passive and good for global localization while it is not a good choice for local localization. Vision-based sensors are passive, the closest to human perception and involve 2D projections of real 3D information. Robot navigation is one classical application for computer vision [90]. A vision-based sensor is dependant on the knowledge of the environment and requires more computation power to interpret a scene. Depending on the application, the tasks of vision-based sensor include obstacle detection/avoidance, map building, localization, scene understanding, 3D reconstruction, landmark detection, etc. Since NAVLAB was developed in the 1980s [91], many works have been done on applying computer vision to robot navigation. But because of the complexity of the human vision system and the limited computing power of the available schemes, vision-based robot navigation still remains a big challenge.

In this dissertation, the focus on the application example is not to address the navigation problems at the system level. Instead, the goal is to develop a real-time vision-based sensor for a specific task, i.e., obstacle avoidance. Obstacle avoidance is one of the most critical tasks for robot navigation. It is usually composed of two parts: obstacle detection and avoidance. In this dissertation, the focus is on the obstacle detection problem. The goal of obstacle detection is to locate possible col-

liding objects in the robot's pathway in order to avoid collision. It works with path planning to perform the navigation task. There are three types of sensors used for obstacle detection tasks [92]: Sonar, LIDAR and vision-based sensor. LIDAR [93] and sonar work by sending out active beams or beacon and measuring the delay of the reflecting signals from the obstacle. LIDAR and sonar are accurate in measuring distance but they are active and basically one dimensional in nature. Compared to other methods, vision-based obstacle detection sensor is passive and could provide more information about the environment. Vision-based obstacle detection could be roughly divided into several categories [94]: edge-detection, certainty grid, potential field, virtual force field, and optical flow based methods. Optical flow based methods are best suited for hardware pipelined structure in FPGAs. Compared to the existing optical flow methods which were designed for general-purpose processors, the focus in this work is to develop a sensor which takes the advantages of the latest FPGA technology and uses algorithms optimized for pipelined hardware structure so that specific functionalities can be efficiently implemented on a small, lower power hardware platform. This sensor can be installed onto an integrated system to perform a more general task.

As discussed, different vision-based navigation algorithms will work under different and specific circumstances. It is necessary to define the specifications of the application before developing the algorithm. In this dissertation, the focus is on small ground vehicles. For ground vehicles, it can be assumed that the robot is moving on the ground plane while it is not necessarily true for aerial vehicles. In many cases, useful constraints could be drawn from this ground vehicle specific application. For ground vehicle navigation, according to the difference in methodology, vision-based robot navigation can be divided into two categories [90]: indoor [95, 96] and outdoor navigations [91, 97]. One basic difference between indoor and outdoor navigation is that indoor navigation often has prior knowledge (map) about the environment while for outdoor navigation, that kind of prior knowledge is harder to obtain. However, both of them could find applications for an obstacle detection sensor to detect obstacles in the pathway.

Using optical flow for obstacle detection has been an active research topic since the 1980s. In [26], an algorithm was proposed to use divergence in the motion field to locate the obstacle. This method did not take the ground floor into account and it used the derivative of the motion field which is sensitive to noise. In [94, 98, 99], Vector Field Histogram and an improvement were proposed to create a certainty grid map of the surrounding environment of the robot. In [100, 101], Enkelmann et al. calculated optical flow and analyzed the motion field in a transputer network to speed up the computation. Obstacles are found by comparing the expected motion field without obstacles and the calculated motion field. In practice, one major problem of using optical flow is that motion field is noisy and contains too many outliers. Zhu et al. [102] used a mean shift algorithm to fuse the motion vectors and they obtained reliable detection results of passing vehicles. However, the mean shift algorithm is time consuming and not suitable for a small obstacle detection sensor. In [103], a vision-based sensor, LIDAR and sonar are combined to achieve a synergy. Sonar and LIDAR can measure distance more accurately than vision-based sensor, but contain less information about the environment. In [104], Focus of Expansion (FOE) is estimated first based on the principle orientation of flow vectors. After that, a depth could be calculated to locate the obstacle. This method didn't use ground plane as in [26].

From the literature review above, it was determined that these algorithms listed above were not suitable for the embedded platform because of their high computation requirement. On the other hand, a standalone vision-based obstacle detection sensor has great potential for robot navigation applications. If an application can be specified and a suitable algorithm can be simplified based on valid assumptions for each specific application, it is possible to develop an embedded vision-based obstacle detection sensor with the available and limited hardware resources. In this dissertation, obstacle detection task for small unmanned ground vehicles is the main focus. For this application, one assumption can be made:

Assumption 1 *Vehicle moves on a planar ground plane. This assumption is true for vehicle under most circumstances and it limits the degrees of freedom of the vehicle moving on this plane.*

Further discussion based on this assumption is shown in the following sections.

5.2 Algorithm Formulation

5.2.1 Motion Model Deduction

Optical flow is a 2D projection of 3D motion in the world on the image plane. The camera frame model depicts the spatial relationship between one point in the world and the camera. Suppose at time t, a point $\mathbf{P} = (X, Y, Z)^T$ in the camera frame is projected onto $\mathbf{p} = (x, y, f)^T$ on the image frame where f is the focal length as shown in Figure 5.1. Three projection models: perspective projection, weak perspective projection, and pinhole camera model can be used to model projection from the camera frame to image frame. In this work, a perspective projection model is used because it is the closest to the physical model of the three. In the perspective projection model,

$$\mathbf{p} = f\frac{\mathbf{P}}{Z}. \tag{5.1}$$

At time t', assuming a point $\mathbf{P}$ moves to $\mathbf{P'} = (X', Y', Z')^T$ which is $\mathbf{p'} = (x', y', f)^T$ on the image frame. The motion vector in the 3D world is $\mathbf{V} = (\mathbf{P} - \mathbf{P'})^T = (X - X', Y - Y', Z - Z')^T$. On the image plane, the 2D projected motion vector (optical flow) can be represented as $\mathbf{v} = (\mathbf{p} - \mathbf{p'})^T = (x - x', y - y', 0)^T$. Because the third component of $\mathbf{v}$ (the focal length $\mathbf{f}$) is cancelled out, $\mathbf{v} = (\mathbf{p} - \mathbf{p'})^T = (x - x', y - y')^T$ is used instead.

As in [105], rigid motion for one point in the camera frame could be formulated as

$$\mathbf{V} = -\mathbf{T} - \omega \times \mathbf{P}, \tag{5.2}$$

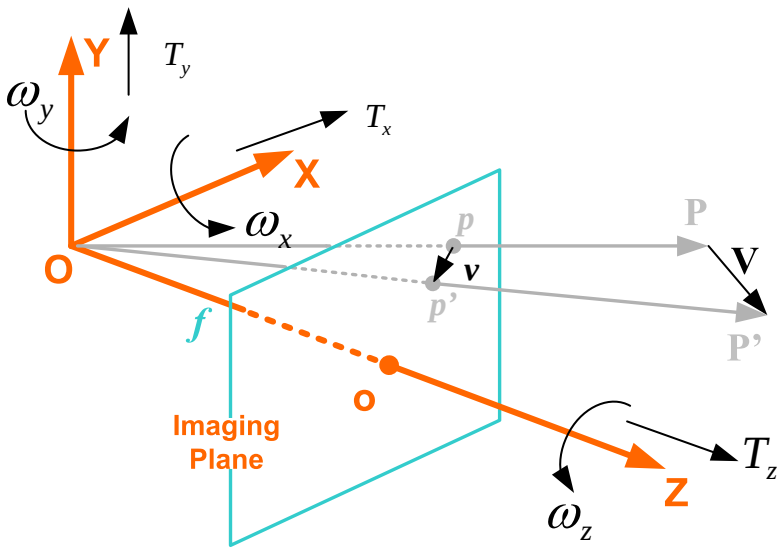


Figure 5.1: Camera projection model

where $\mathbf{T} = (T_x, T_y, T_z)^T$ is the translational component and $\omega = (\omega_x, \omega_y, \omega_z)^T$ is the rotational component. As in [105], each component of $\mathbf{V}$ can be represented as

$$\begin{aligned} V_x &= -T_x - \omega_y Z + \omega_z Y, \\ V_y &= -T_y - \omega_z X + \omega_x Z, \\ V_z &= -T_z - \omega_x Y + \omega_y X. \end{aligned} \tag{5.3}$$

To convert the motion in the camera frame to optical flow which is the projected motion on the image plane, as in [105] the derivative of (5.1) can be calculated as

$$\mathbf{v} = \frac{d\mathbf{p}}{dt} = f\frac{Z\mathbf{V} - V_z\mathbf{P}}{Z^2}. \tag{5.4}$$

By combining (5.3) and (5.4), components of $\mathbf{v}$ are derived as

$$\begin{pmatrix} v_x \\ v_y \end{pmatrix} = \frac{\omega_x}{f}\begin{pmatrix} xy \\ y^2 + f^2 \end{pmatrix} + \frac{\omega_y}{f}\begin{pmatrix} -x^2 - f^2 \\ -xy \end{pmatrix} + \omega_z \begin{pmatrix} y \\ -x \end{pmatrix} + \frac{1}{Z}\begin{pmatrix} T_z x - T_x f \\ T_z y - T_y f \end{pmatrix}. \tag{5.5}$$

As shown in (5.3) and (5.5), there are six motion parameters representing the rigid motion of one point in the camera frame. Retrieving all these six motion parameters from the two components, i.e., v_x and v_y of optical flow vector is an ill-conditioned problem. It would be even harder to classify based on these motion parameters. Therefore, to improve this situation, motion patterns are often restricted to a certain number of degrees of freedom for specific applications. For ground vehicle applications, it is usually assumed that the vehicle travels on a planar surface. In [106], three motion parameters, i.e. forward translation T_z, yaw ω_y and pitch ω_x were assumed for a forward-looking camera (ground plane is parallel with plane **XZ** as in Figure 5.1). In [107], two sets of motion parameters were discussed. One set was (ω_y, T_x, T_z) for forward looking camera. Ke et al. [107] argued that T_x is not negligible. And in the same paper [107], Ke et al. used a virtual downward-looking camera (ground plane is parallel with plane (XY) so that the motion parameters were selected as (ω_z, T_x, T_y).

The choices of motion parameters above were for ego-motion estimation tasks which required high accuracy in parameter estimations. For obstacle detection tasks, the accuracy requirement on motion parameters could be lower. After several experiments, it was determined that two parameters, i.e.(ω_y, T_z), as shown in Figure 5.2 could work well for obstacle detection for a forward-looking camera mounted on a ground vehicle traveling on a planar surface. More importantly, with this two-parameter setting, the obstacle detection algorithm only requires simple linear algebra which can be efficiently implemented in hardware for real-time performance. (5.5) can be reorganized as

$$\begin{pmatrix} v_x \\ v_y \end{pmatrix} = \frac{\omega_y}{f} \begin{pmatrix} -x^2 - f^2 \\ -xy \end{pmatrix} + \frac{1}{Z} \begin{pmatrix} T_z x \\ T_z y \end{pmatrix}. \tag{5.6}$$

This equation is composed of two parts, rotation and translation. The rotational component is associated with ω_y and the translational component is associated with T_z. As seen in (5.6), the rotational component does not carry any depth information. The translational motion is the one containing the depth information. The translational component must be decoupled from the rotational component first so that the

de-rotated motion field can be analyzed to determine the obstacle's distance to the camera. With the description above, obstacle detection problem can be defined as: Given a sequence of optical flow fields, estimate the two motion parameters yaw ω_y and forward translation T_z according to the planar surface assumption and indentify pixels with inconsistent motion pattern as obstacles. This algorithm attempts to estimate the parameters one at a time and can be decomposed into three steps: de-rotation (estimate ω_y), de-translation (estimate T_z) and post-processing (identify inconsistent points). Each step will be introduced individually as follows.

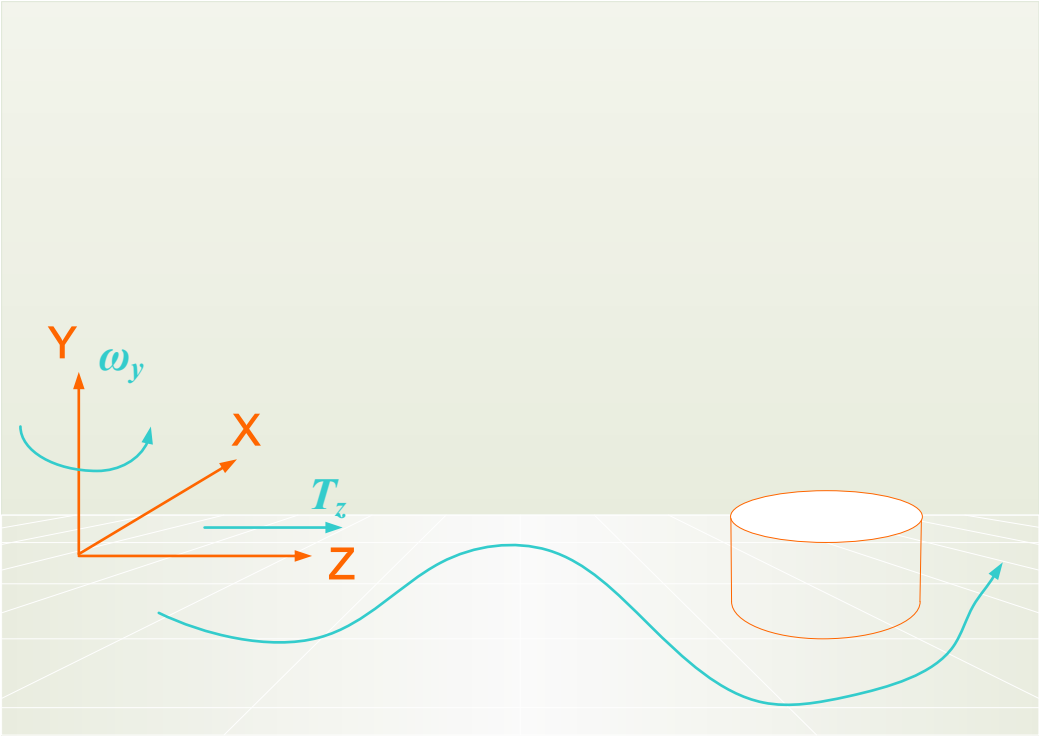


Figure 5.2: Obstacle avoidance illustration

Before proceeding, two more assumptions must be made.

Assumption 2 *Ground plane occupies a dominant region of the image. With this assumption, it can be assumed that the ground motion parameters can be extracted from the optical flow field that contains obstacle pixels with inconsistent motion.*

Assumption 3 *The **XZ** plane of the camera frame is parallel to the ground plane. This assumption is valid if the camera is mounted on the vehicle correctly.*

5.2.2 De-rotation

The motion field is studied first without any obstacle. With Assumption 1 and 3, it is shown below that on the image plane, depth is related to the y coordinate and is independent of the x coordinate. Because the ground plane is in parallel with the **XZ** plane, in camera frame the ground plane is formulated as

$$G : Y = Y_0. \tag{5.7}$$

A line l_0 on the ground plane with the depth $\mathbf{Z_0}$ is

$$L : \begin{cases} Y = Y_0 \\ Z = Z_0. \end{cases} \tag{5.8}$$

and points on this line satisfy $\mathbf{P} = (X, Y_0, Z_0)^T$. According to projection equation (5.1), mapping of $\mathbf{P}$ on the image plane is

$$\mathbf{p} = f\frac{\mathbf{P}}{Z_0} = (f\frac{X}{Z_0}, f\frac{Y_0}{Z_0}, f)^T. \tag{5.9}$$

Equation (5.9) depicts that line L with depth Z_0 on the ground plane is mapped to $l : y = f\frac{Y_0}{Z_0}$ on the image plane if Assumptions 1 and 3 are both satisfied. In other words, the depth of line L can be inferred solely based on its y coordinate on the image plane and its depth Z_0 is independent of x.

With the above observation, by extracting the v_y component in (5.6), the partial derivative of v_y can be derived as

$$\begin{aligned} \frac{\partial v_y}{\partial x} &= -\frac{\omega_y}{f}y + \partial(\frac{T_z y}{Z})/\partial x \\ &= -\frac{\omega_y}{f}y = \beta, \end{aligned} \tag{5.10}$$

where β is the slope ratio for v_y. β is fixed for each row, if depth Z is not a function of x. This is true if Assumptions 1-3 are satisfied. Denote $(v_{x_{i,j}}, v_{y_{i,j}})$ as the optical flow at pixel (i, j), one β_j can be calculated for each row while ω_{y_j} should be the same for all rows. Least Squares (LS) estimation is used to correctly estimate β_j. For row j,

$$v_{y_{i,j}} = \beta_j x_i + \alpha_j, \tag{5.11}$$

where $\alpha_j = \frac{T_z}{Z_j} y_j$ is the translational component which is constant for each row. Without loss of generality, it is assumed that $i = 1, 2, ..., 2n+1$ and $x_i = i - n - 1 \in [-n, n]$ and the LS solution for β_j is

$$\beta_j = \frac{(2n+1)\sum x_i v_{y_{i,j}} - \sum x_i \sum v_{y_{i,j}}}{(2n+1)\sum x_i^2 - (\sum x_i)^2}. \tag{5.12}$$

It is obvious that $\sum_i x_i = 0$ and Equation (5.12) can be simplified as

$$\beta_j = \frac{\sum x_i v_{y_{i,j}}}{\sum x_i^2}. \tag{5.13}$$

Besides β_j, a confidence index c_j can be calculated for each row which measures how well this linear model matches the local signal,

$$c_j = \frac{(\sum x_i v_{y_{i,j}})^2}{\sum x_i^2 \sum v_{y_{i,j}}^2}. \tag{5.14}$$

When most pixels in one row belong to the ground plane, c_j will be close to 1. When an obstacle is present, because its depth is different from the ground plane, the linear model in (5.11) will not hold and c_j will be smaller. c_j is a good indicator of the presence of an obstacle in one row.

5.2.3 De-translation

After the rotation parameter ω_y is obtained, the rotational component can be removed from v_y. From Equation (5.9) and (5.11), the resulting de-rotated v_y

component

$$v^R_{y_{i,j}} = \alpha_j = \frac{T_z}{Z_j} y_j = f \frac{Y_0}{Z_j^2} T_z \tag{5.15}$$

is fixed for each row as well. Again, the true value of the translation parameter T_z is not needed for obstacle detection. Instead, the goal is to identify pixels with depth values different from other pixels on each row. These pixels are considered to be part of the obstacle.

To simplify the algorithm and make it fit for hardware implementation, the mean of the de-rotated v_y components is used as the translational component and the de-translated v_y component is derived as

$$v^D_{y_{i,j}} = v^R_{y_{i,j}} - \bar{v}^R_{y_j} = v^R_{y_{i,j}} - \frac{\sum_{k=j-m}^{k=j+m-1} \sum_{i=-n}^{i=n-1} (v^R_{y_{i,k}})}{4mn}. \tag{5.16}$$

After de-translation, if one pixel belongs to the ground plane, its de-translated motion component $v^D_{y_{i,j}}$ should be very close to 0. In comparison, a pixel on the obstacle should have a larger de-translated motion component. The post-processing step in the next section shows how obstacles can be located based on the de-translated motion component value.

5.2.4 Post-processing

Ideally, after the motion field is de-rotated and de-translated, the residual motion components will be zero for ground plane pixels and non-zero for obstacle pixels. This is illustrated in Figure 5.3. G is a visible point on the ground plane if there is no obstacle. O is the point on the obstacle which blocks point G. Based on Equations (5.6) and (5.15), the projected v_y motion component difference which can

be measured from the optical flow estimation is formulated as

$$\begin{aligned} v_y^{\Delta} = v_y^G - v_y^O &= (\frac{1}{Z^G} - \frac{1}{Z^O})T_z y \\ &= fT_z(\frac{Y^G}{Z^{G^2}} - \frac{Y^O}{Z^{O^2}}) \\ &= cfT_z(\frac{1}{Z^G} - \frac{1}{Z^O}), \end{aligned} \tag{5.17}$$

where $c = \frac{Y^G}{Z^G} = \frac{Y_O}{Z^O}$ is the slope ratio for the line passing through point G and O. In (5.17), Z^G is the depth for point G and Z^O the depth for point O. As in (5.17), the motion difference is proportional to T_z (the translational motion of the camera) and c (the line slope ratio), and the depth difference item. From (5.17), it can be concluded as follows. If $T_z = 0$ (no translational motion), the obstacle will not be detectable which agrees with the basic idea of using a single camera for depth estimation. Also, assuming other parameters are fixed, the closer the obstacle is to the camera (smaller Z^O), the bigger absolute value of (5.17) will be.

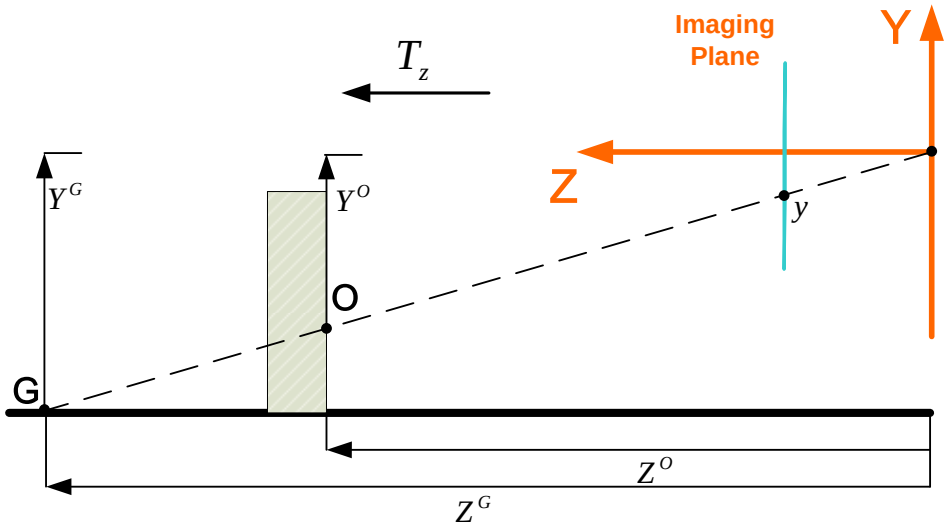


Figure 5.3: Depth difference diagram

In practice, a threshold δ is used to convert the motion component difference image into a binary image indicating the initial obstacle detection result as shown in (5.18). Detecting obstacles in this manner has two advantages. First, the operations can be efficiently implemented in hardware. Second, it avoids estimating the obstacle depth Z^O which is not a trivial task in many circumstances.

$$b_{init}(x,y) = \begin{cases} 1, & \text{if } \left|v_y^{\Delta}(x,y)\right| \geq \delta; \\ 0, & \text{if } \left|v_y^{\Delta}(x,y)\right| < \delta. \end{cases} \tag{5.18}$$

5.3 Hardware Structure

The hardware diagram of the obstacle detection module is shown in Figure 5.4. In this diagram, the obstacle detection module is divided into three sub-modules: de-rotation, de-translation and post-processing. These sub-modules are fully pipelined and concatenated in the pipeline. Detail explanation of each sub-module will be presented in the following subsections.

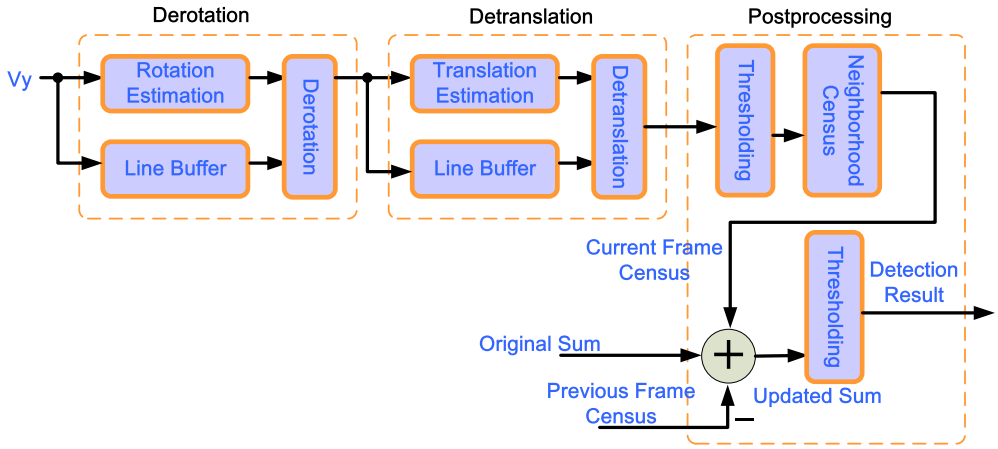


Figure 5.4: Diagram of the obstacle detection module

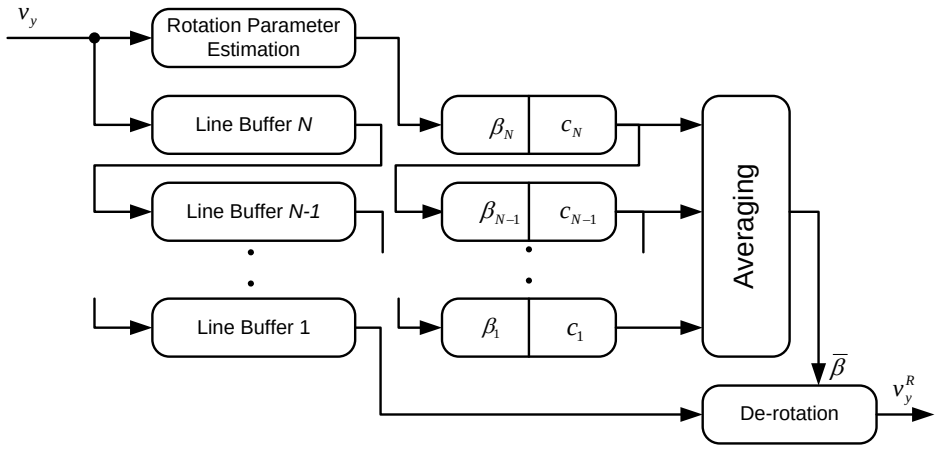


Figure 5.5: Flowchart of de-rotation module

5.3.1 De-rotation Hardware Structure

To obtain an accurate rotation estimation, rotation parameters across multiple rows are averaged to obtain an averaged parameter. The confidence index in (5.14) is used as weights in averaging. The averaged slope ratio $\bar{\beta}$ is expressed as

$$\bar{\beta} = \frac{\sum_{j=1}^{N} \beta_j c_j}{\sum_{j=1}^{N} c_j}. \tag{5.19}$$

Diagram of de-rotation hardware is shown in Figure 5.5. Pixelwise motion estimation v_y is the input and it is fed into rotation parameter estimation and the line buffer at the same time. Depending on the number of lines (N) to be averaged, the same number of line buffers are concatenated before v_y is de-rotated. At the same time, the slope ratio β_j and confidence index c_j for each row are registered and shifted in synchronization with the line buffer. Once the N sets of β_j and c_j are ready, they are averaged as shown in (5.19) to obtain $\bar{\beta}$. Selection of N is a tradeoff between accuracy and processing speed. If N is small, e.g., 2, $\bar{\beta}$ will be sensitive to noise or even corrupted when there are many obstacle pixels in the row. This will cause a bias in the translational component α as shown in (5.11) and this bias will be carried into subsequent calculations. The bias is worse for the left-most and right-most regions

in the image because x_i has much bigger magnitude along image boundary as shown in (5.11). If N is large, e.g., 20, it will require hardware resources for line buffering and averaging β_j across 20 rows. Also, when N is too big, β_j will be over-smoothed because of the difference of y as shown in (5.10). In this work, N is chosen to be 8. With $\bar{\beta}$ estimation, v_y can be de-rotated to obtain v_y^R.

5.3.2 De-translation Hardware Structure

Similar to the de-rotation module, de-translated motion is calculated in (5.16). The averaged translational motion is estimated by averaging the de-rotated motion component across a certain number of rows. There is also a tradeoff which is similar to the de-rotation setting. In this work, the row number is set to be 8 to achieve a balance between accuracy and speed.

5.3.3 Post-processing Hardware Structure

The main purpose of post-processing is to filter out the false positives. After v_y motion component is de-rotated and de-translated, as shown in Figure 5.4, it is binarized by applying a threshold to obtain the initial detection result as shown in (5.18). The binary image b_{init} is smoothed in spatial and temporal domain separately. The assumption behind spatial and temporal smoothing is that the obstacle is coherent both spatially and temporally. By counting the initial detection results in the spatiotemporal domain, random false positives can be detected. The spatial smoothing calculates the number of initial deleted pixels in a local neighborhood. The temporal smoothing updates the number of overall initial detected pixels in the temporal domain. To efficiently calculate the sum of the detected pixels, the following equation is applied

$$S_c = S_l + S_p - S_f, \tag{5.20}$$

where S_c is the sum of current frame, S_l is the spatial sum of the current frame, S_p is the sum of the previous frame and S_f is the spatial sum of the first frame in the temporal volume.

5.4 Experimental Results

The developed algorithm was implemented in VHDL and simulated in ModelSim to test its performance. The system was built based on the optical flow module as discussed in Chapter 4. The obstacle detection module was appended to the optical flow module to directly read out the v_y component of the motion vector. This section includes the following subsections: setup, software simulation results and hardware simulation results. In the setup section, settings of the testing environment are discussed. In the software simulation section, the simulation results using MATLAB are presented. In the hardware simulation section, results obtained from the hardware test bench are given.

5.4.1 Experiment Setup

There are two stages in the algorithm development and hardware design. The first stage is simulation in software and the second stage is hardware implementation. In software simulation, algorithm is tested to achieve the best performance using the optimal algorithm framework. It does not consider any constraints for hardware implementation. For example, it uses floating point computations and optimized codes. In the second stage, the algorithm developed in the first stage is converted into hardware. Because of hardware resource limitations, the algorithm has to be revised to be fit on the hardware, e.g., fixed point computations and simplified algorithm. These kinds of revision significantly affect algorithm performance. Extra care is taken to minimize their effects. In this section, both software and hardware simulation results are given for comparison to explain the effects of realizing software-based algorithms onto hardware.

Software simulation was coded in MATLAB. The video was captured by a SONY SD CCD camcorder. The computing platform was a Lenovo T61 laptop with a 2.4GHz Intel Core2 Duo T8300 CPU and 2Gb memory. The software version of the algorithm uses floating point operations and contains complicated computations to obtain better motion estimations which benefit spatiotemporal smoothing in post-processing. Important parameters such as the spatial and temporal smoothing mask

sizes are the same for both software and hardware. Another difference is that the video captured by a camcorder had better picture quality than the CMOS camera installed on our test vehicles. Also, the camcorder has a wider range of focal length which is desirable in practical application.

Hardware resources used by the obstacle detection module is listed in Table 5.1. The obstacle avoidance module does not consume many resources by itself. As shown in Table 5.2, when this module is integrated into the whole system, it substantially increases the overall resources utilization. This is because the FPGA has already used a large amount of hardware resources before the integration. After the obstacle avoidance module is integrated, the situation is worse because it takes extra resources to incorporate the new module which deteriorates the situation further.

Table 5.1: Resources utilization for obstacle detection module.

Hardware resources	Number	Percentage
Slices	1679	6.7%
Flip Flops	2069	4.1%
4-input LUTs	2934	5.8%
DSP48s	11	8.6%

Table 5.2: Overall resources utilization before and after integrating the module.

	Number	Percentage	Number	Percentage
Hardware resources	Before integration		After integration	
Occupied Slices	22,483	88%	25,278	99%
Flip Flops	24,070	48%	30,368	60%
4-input LUTs	31,866	63%	45,240	89%
DSP48s	37	28%	48	37%

Due to hardware resource limitations, the whole design could not be fit on the current hardware platform. To simulate the performance, the obstacle detection module had to be run offline and it was not directly connected to the hardware system.

The input to the obstacle detection module is the v_y component of the motion vector. To get the simulation result, the optical flow calculation ran first to obtain the motion field for image frames captured by the camera. The v_y motion component was then transferred to the GUI through the USB interface and saved as a video file. The video file was uncompressed to assure the data were lossless. Video file was then parsed into frames and stored as text files. Each saved v_y frame was loaded into the obstacle detection module through the test bench. The test bench also controlled the output logics and saved the obstacle detection results.

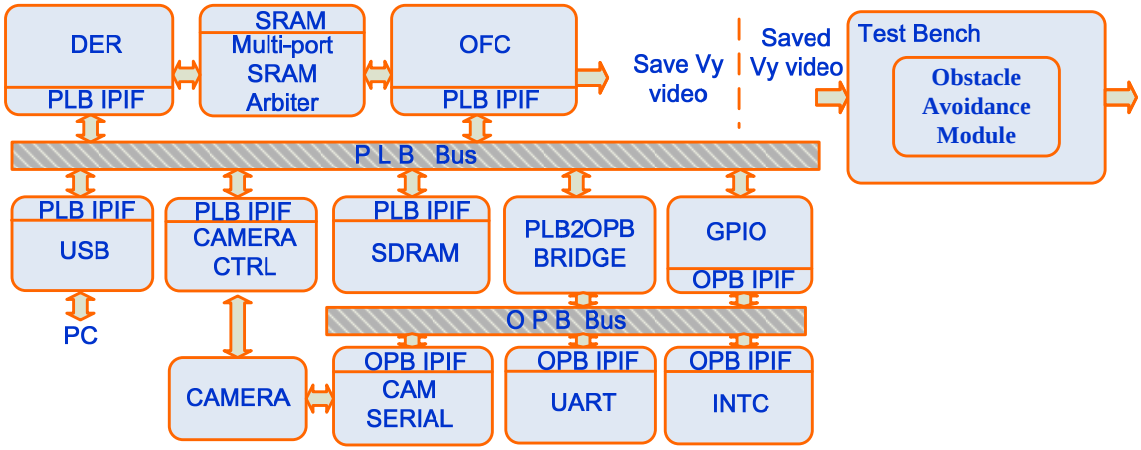


Figure 5.6: Test bench setup

5.4.2 Software Simulation

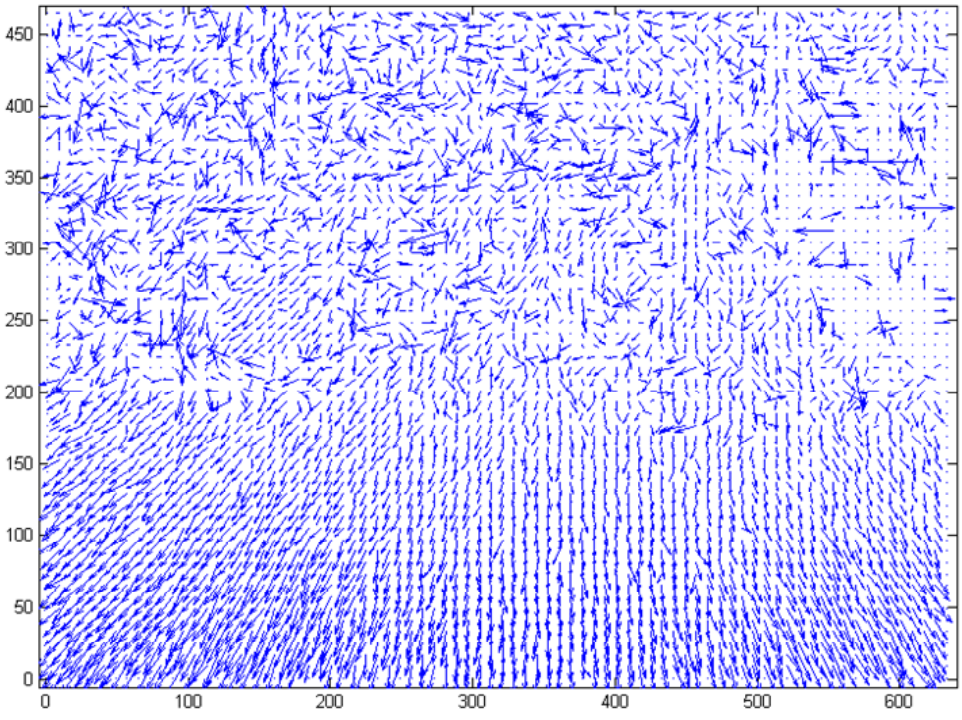
In this section, a sequence of testing images is presented to illustrate the obstacle detection algorithm. These images include the original image and the intermediate results obtained from the different stages in the algorithm. Figure 5.7(a) shows one frame of the original video sequence. The video was taken by mounting a camcorder on the truck and manually moving the truck. As shown in Figure 5.7(a), there are several obstacles in the scene. In the front, there are two boxes located on the left and the right separately. In the back, there are two trash cans, a refrigerator and

a wall behind them. The truck was manually driven toward the wall and the video was taken as it was moved. Figure 5.7(b) shows the estimated motion field from the video sequence featured by Figure 5.7(a). The motion estimation algorithm is the improved algorithm discussed in Chapter 4. It can be observed that the motion field of the ground plane is smooth because the carpet carries abundant texture while the motion field of the refrigerator and the wall is noisy due to the absence of texture.

In Figure 5.8(a), the v_y component of the optical flow field (Figure 5.7(b)) is normalized and displayed as a pseudo-color image in order to display the motion field distribution better. In this image, the lower the color temperature, the smaller value is at that pixel and vice versa. As shown in the figure, values at the lower-left corner are smaller compared to those at the lower-right corner. The difference between these two lower regions represents the existence of a rotational motion field as shown in Equations (5.5) and (5.10). In Figure 5.8(b), the de-rotated v_y component is also normalized and displayed. Compared to Figure 5.8(a), it can be seen that the rotational component is diminished after the de-rotation operation. In software-based design, de-rotation is not performed if the index c_j (5.14) on row j is below a certain threshold. If c_j is below a certain threshold, the v_y component in that row cannot be approximately well with a linear model. This may be caused by the obstacle points or noisy motion estimations. As a result, β_j in Equation (5.13) cannot be extracted reliably from the motion component in row j. After the de-rotation opertation, the motion component is subtracted by an offset which is considered as the translational motion. The translational motion is estimated by averaging the de-rotated motion field in the ground area which has high c_j values. By subtracting an offset, the normalized image looks exactly the same as Figure 5.8(b). The normalized de-translated image is not displayed here since it is identical to Figure 5.8(b).

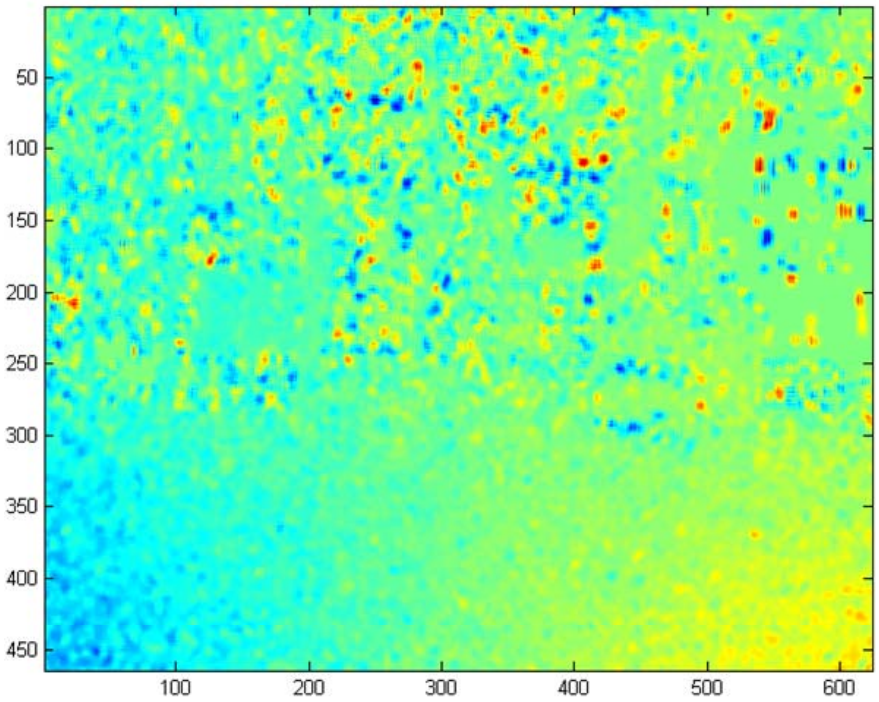
After the motion component is de-rotated and de-translated, a threshold is applied to binarize the resulting image. Figure 5.9(a) shows the binary image by imposing a threshold of 0.5. Obstacle pixels are marked in black. By applying the spatiotemporal smoothing, false positives are filtered out at the cost of removing some true positives which is tolerable if the removal of true positives is only a small amount.

(a) One frame of testing video

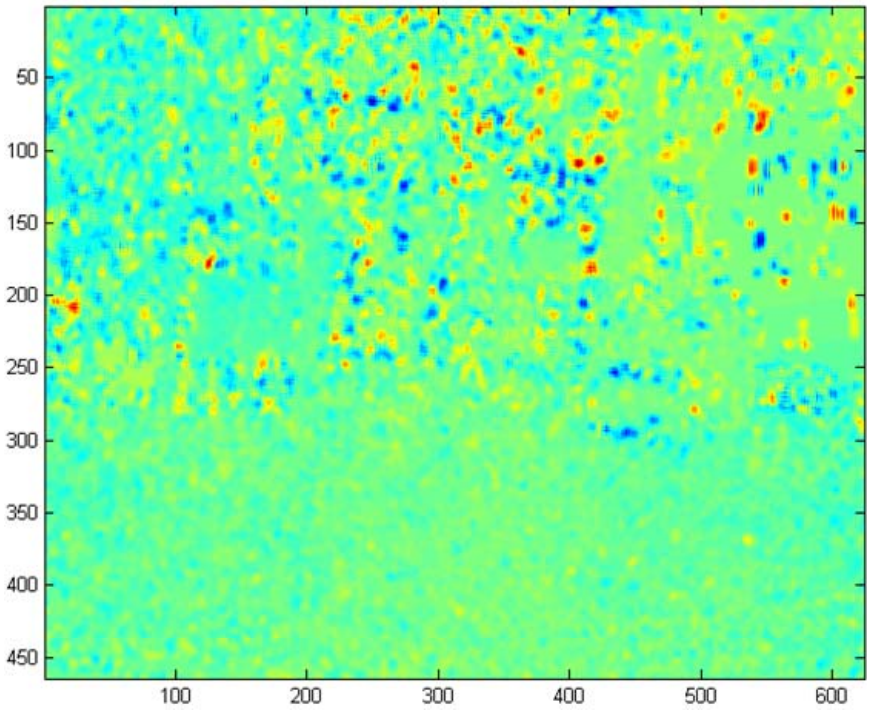


(b) Corresponding motion field of Figure 5.7(a)

Figure 5.7: One video frame and its corresponding motion field



(a) v_y component of motion field as in Figure 5.7(b)



(b) De-rotated motion field v_y^R from Figure 5.8(a)

Figure 5.8: v_y component before and after derotation calculation

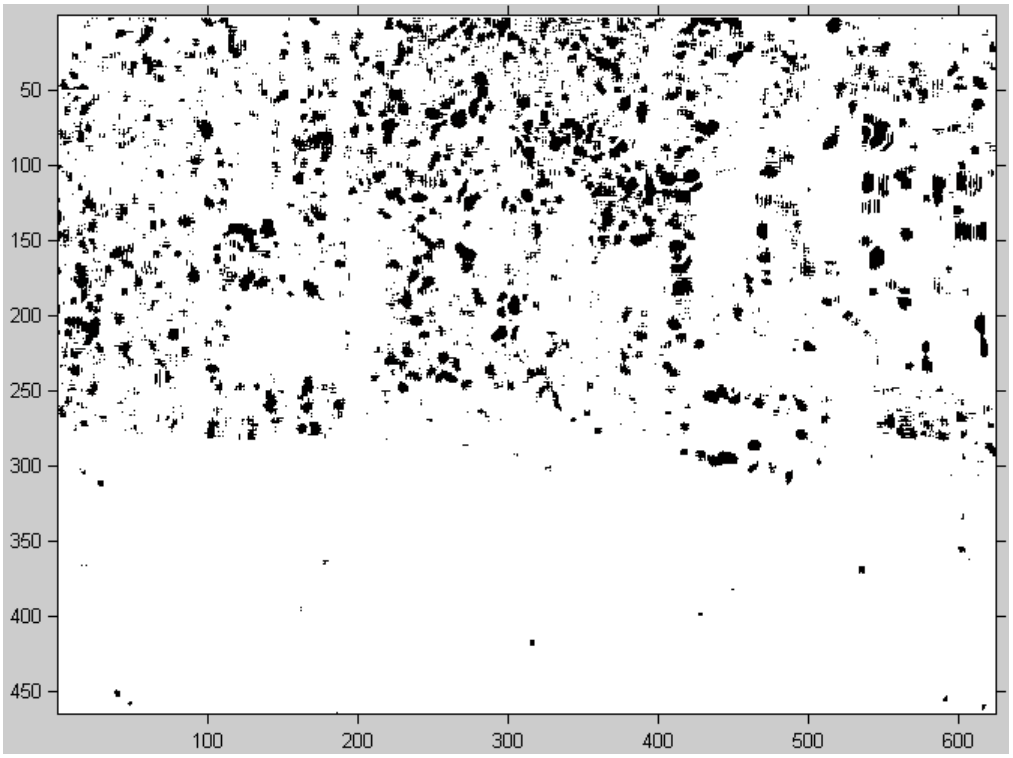
The final obstacle detection result is shown in Figure 5.9(b). As shown in the figure, some false positives in the lower region of the image are removed because of their discontinuity in the spatiotemporal volume. Figure 5.10 shows the raw image overlaid with the final detection result. To better demonstrate the result, the detected obstacle pixels are marked in brighter color. It can be observed that the detection result accurately spots the obstacles and the result shows very low rate of false positive. Although not the whole obstacle objects are indentified, the result can be further improved by post-processing across consecutive frames.

5.4.3 Hardware Simulation

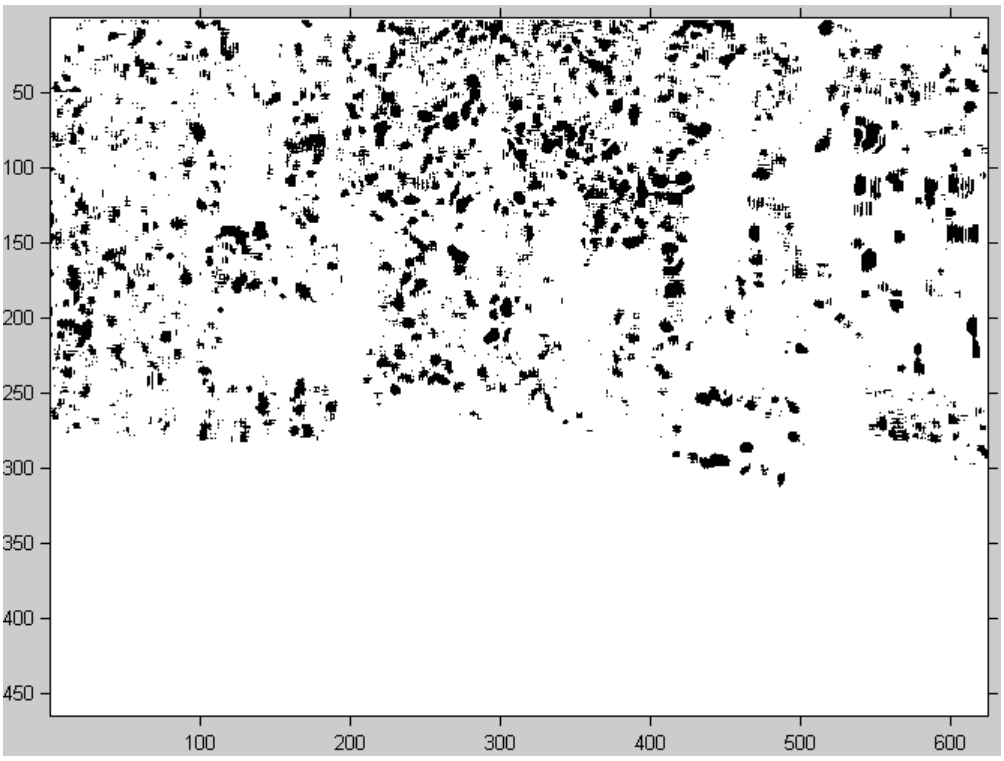
Because of resource limitations, the obstacle detection module could not be integrated into the same FPGA with the optical flow estimation module. The simulation setup is shown in Figure 5.6. In the hardware simulation, a video sequence was captured by a CCD camcorder and fed into the motion estimation module to obtain the optical flow field. The produced motion field was then converted to the format understood by the testbench. Obstacle detection results are obtained by running the simulation model and saving the results from the testbench. In Figure 5.11, the results of the obstacle detection hardware simulation and software simulation results are compared.

From the comparison of results, it is observed that the hardware simulation result contains more false negatives and false positives than the software simulation result. Some reasons are concluded on the performance degeneration:

- In the hardware design, because of the limited resource, the possible number of rows used for averaging the rotation parameter in Equation (5.13) is very limited. In comparison, linear fitting is used in the rotation parameter estimation which allows fitting the rotation parameter for one row from its neighboring rows. This is helpful when the rotation parameter for a specific row cannot be accurately estimated because of obstacles or other causes.



(a) Initial obstacle detection result



(b) Final obstacle detection result after spatial-temporal smoothing

Figure 5.9: Binary masks indicating the initial and final obstacle detection result

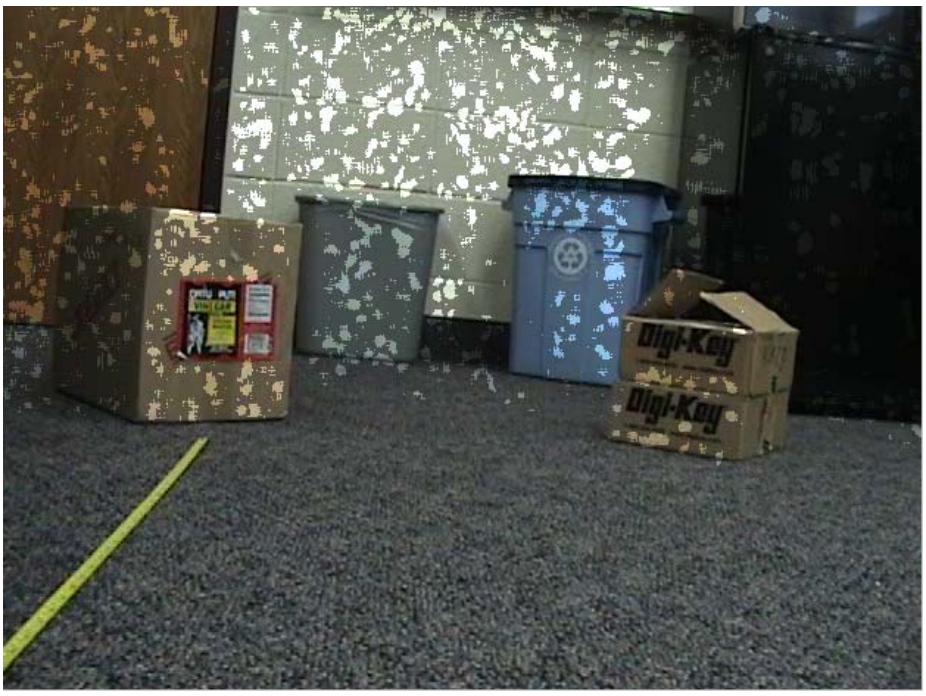


Figure 5.10: Overlaying the original image with the detection result

- The fixed-point based calculation in the hardware lowers the accuracy of the motion parameters estimation, e.g. the rotation parameter. The small difference in accuracy will be amplified at the left and right side of the image by multiplying by x coordinate as in Equation (5.11). This effect is visible in the lower left region of Figure 5.11(a).

The purpose of the hardware design in this dissertation is to prove the feasibility of implementing the proposed obstacle detection algorithm on the FPGAs. Further research can be conducted on optimizing the hardware design to achieve better accuracy.

5.5 Conclusions

This chapter presents an example application of the optical flow sensor for small UGV navigation tasks. One important constraint on UGV applications is the computation power which is restricted by the available computation platforms. The

(a) Hardware simulation results

(b) Software simulation results

Figure 5.11: Comparison between obstacle detection hardware simulation and software simulation results

traditional computationally expensive algorithms are not suitable for UGV applications. Using the developed real-time optical flow sensor which is implemented on a low power, compact size FPGA platform, an efficient algorithm has been developed to detect obstacles in the pathway by analyzing the motion field.

With the assumptions about the camera geometrical model, the driving terrain, and the UGV motion pattern to simplify the problem, only the v_y component of the optical flow vector is needed to locate the obstacles. The obstacle detection algorithm consists of three main steps. First, the motion field is de-rotated to obtain a motion field without rotational motion. To estimate the parameter of the rotational motion, the classical Least Squares method is used to analyze the image of the v_y component row by row. Second, the de-rotated motion field is then de-translated to remove the effects of translational motion. The parameter of translational motion is estimated by weighting the de-rotated motion field across multiple rows. Third, by applying a threshold, the processed motion field is converted to a binary image which is used as the initial obstacle detection result. To suppress noise in the motion estimations, the initial detection result is post-processed to obtain the final result by assuming the spatial and temporal continuity of the obstacles. The final detection result is saved in the memory as a temporal sequence. An old resulting frame is shifted out from the temporal sequence as the incoming new frame becomes available and the sequence can be used for future detection.

Architecture of the targeted hardware platform (FPGA) is taken into account during the algorithm development process so that the algorithm could be efficiently implemented. One representation is that the algorithm is developed to use the v_y component only. This simplifies the hardware logics and lowers the memory bandwidth requirement. Also, the developed algorithm uses simple arithmetic operations to save hardware resources.

The obstacle detection algorithm has been developed in the VHDL language. An effort was made to put this design on the same FPGA which contains the optical flow module. However, the available FPGA (Xilinx FX-60) does not have enough capacity to include both modules. To verify the performance of this design, both

software and hardware simulations have been performed. Results have shown that this algorithm is able to detect obstacles on real testing video sequences.

Chapter 6

Conclusions and Future Work

This dissertation presents a new and accurate optical flow algorithm that is suitable for hardware implementation. The development of a standalone vision sensor using this optical flow algorithm is discussed. An application of using this embedded vision sensor for obstacle detection for a small UGV is demonstrated. The research work is motivated by the fact that foreseeable rapid growth of intelligent vehicles is on the horizon. In 2001, the U.S. Congress mandated that by 2015, one third of military vehicles should be unmanned, and that number will go up to two thirds by 2025. Besides military applications, unmanned vehicles have found their applications in civilian areas such as hazardous area exploration, remote surveillance and other missions which are dangerous for human onboard operators.

The prerequisites for navigating an unmanned vehicle are different types of sensors which can perceive the vehicle's status and environment such as speed and position. To an intelligent vehicle, these sensors are just like eyes and ears to humans. Without them, the vehicle would just be blind and deaf. For driving, among all other information sources, visual signals are by far the most important for human drivers. Correspondingly, a vision sensor which can process and interpret visual signals will be of great practical value. With current advances in computer vision research, it is extremely difficult to design a sensor which can imitate the human vision system or even come close to this goal. Instead, one practical way is to abstract important visual features and apply image processing or computer vision algorithms to extract one or more features.

In this dissertation, motion (optical flow) sensors and their applications are the main research focuses. As discussed in Chapter 1, motion is an important visual

cue for many creatures including human beings. For human drivers, motion is used for time-to-impact estimation, egomotion calculation, obstacle detection, etc. Despite all the advances in technology, motion estimation sensor development remains a challenging task. Motion estimation algorithms operate in the spatiotemporal domain which has a large amount of data and, therefore, requires extensive computation power to process. In addition, certain requirements on size and power requirements are often imposed on motion sensors for practical uses. These constraints prevent traditional CPUs-based motion estimation algorithms from being incorporated into embedded motion sensors.

In this dissertation, an FPGA was used as the platform for motion estimation. Due to the significant differences between FPGA and CPU architectures, traditional software-based algorithms are not fit for FPGA architectures. Based on this observation, two algorithms and their corresponding architectures were developed in order to achieve accurate and real-time performance. To validate these algorithms, software simulation codes were carefully designed to evaluate the performance before implementation. After software simulations, these two algorithms were implemented on two FPGAs platforms. Real results have shown the effectiveness of these algorithms. With a valid optical flow sensor, research on applying optical flow to obstacle detection was further investigated. Similar to the optical flow algorithm development, an obstacle detection algorithm was designed for implementation on an FPGA platform. Tradeoffs were considered throughout the design to balance accuracy and efficiency. Due to the limitations on the available FPGA resources, the obstacle detection algorithm could not be integrated into a single FPGA with the optical flow module. In order to verify the effectiveness of this algorithm, software and hardware simulation results were given.

6.1 Contributions

The contributions of this research work are summarized as follows.

First, a lightweight tensor-based algorithm was developed. This algorithm adopted the concept of a tensor to represent local 3D motion orientation in spa-

tiotemporal volume. Tensors have been used by many algorithms in the literature and proved to be a robust, accurate and powerful way to represent motion orientation. However, after extensive studies, it was determined that all existing tensor-based algorithms were not suitable for FPGA architecture. With optimization, a tensor-based optical flow algorithm and its fully pipelined architecture were developed. Key parameters determining the algorithm performance were indentified. At the same time, rough resource utilization under different parameter settings was quantized in order to evaluate quantitatively the tradeoff between performance and efficiency. Experimental results revealed high accuracy and resource efficiency of this design.

Second, based on the first algorithm, a robust optical flow algorithm using a ridge estimator was developed. The motivation for improving the first algorithm was the availability of a better FPGA platform which could accommodate more complicated algorithms. Due to hardware resource limitations, the first algorithm was simplified to a level that the collinear problem became a serious issue and produced erroneous estimations. These false values propagated to the neighboring estimations through the smoothing process. The ridge estimator was developed to suppress erroneous estimations. Also, from analyses and simulations, it was found that incorporating temporal smoothing could significantly improve accuracy. New hardware architecture was developed to include temporal smoothing in hardware implementation. This new hardware architecture had two parts. One was used to calculate the gradient vectors and stored the results in the memory. The other part read the results from the first part and calculated optical flow values. This new algorithm achieved a remarkable improvement on accuracy. The tradeoff was that the new architecture was not fully pipelined and had lower throughput compared to the first design. However, the system was not able to run at maximum throughput anyway due to the bus bandwidth bottleneck of the current hardware platform. Therefore, higher accuracy with adequate throughput is more valuable in practice.

Third, as an application example, an obstacle detection algorithm was developed to detect obstacles in the pathway. Similar algorithms in the literature usually assumed a general motion pattern, and to retrieve the motion parameters, required

complex computation steps and long processing time. In order to make it feasible to realize a standalone obstacle detection sensor, an efficient and hardware-friendly algorithm was developed. Instead of assuming a general motion model, a simplified but valid motion model was used. With this motion model, only one component in the optical flow vector was needed, and the obstacle detection task was decomposed into three steps. The first step was to de-rotate the motion field. The second step was to de-translate the motion field based on results from the first step. The third step was to post-process the resulting motion field and obtain a robust detection result. All of these steps were implemented efficiently into the hardware pipelined structure.

The overall methodology in this research was to start from simple but feasible solutions. After these solutions had been realized and tested, improved algorithms could then be developed to take the advantages of a better computation platform that became available after the development of the first and simpler version algorithm.

The *fourth* contribution worth mentioning is that the developed optical flow algorithms have been used by other researchers as a typical class of computationally extensive tasks to compare the performance of different hardware parallelism architectures. These architectures include FPGAs, GPUs and MPPAs. Very interesting and insightful conclusions have been drawn from these comparisons, and will benefit not only the computer vision society but also other research areas such as high performance computing and hardware parallel architecture.

6.2 Future Work

There are several areas in which this work can be further improved: more tests on the optical flow module, multi-FPGA platform architecture research, integration of vision sensors in UGV control loop and improvement of the obstacle detection model. Improving these areas will have a positive impact on the applicability of the obstacle detection sensor, so that it can be installed onto the small UGVs or other ground vehicles to monitor road conditions in real time. This proposed future work will significantly improve the sensor quality.

In this dissertation, the Yosemite sequence is the only testing sequence with ground truth. The Yosemite sequence has been widely used in the literature and it is the most common benchmark to compare the performance of one optical flow algorithm to others. With the development of optical flow algorithms in recent years, new evaluation methodology and sequences have been designed [108] to provide a more comprehensive and objective testing method. Therefore, the proposed optical flow algorithms in this dissertation should be tested using the new method to better understand their advantages and limitations.

When more complicated and computationally intensive algorithms need to be implemented, the current single FPGA platform may not be able to offer enough capacity. A multi-FPGA platform is a possible solution to this problem. When the capacity of a single FPGA is exceeded using current technology, a multi-FPGA solution can offer more computational power than merely using higher end FPGAs. In a multi-FPGA design, hardware pipelines can be divided into separate partitions which can then be realized on different FPGAs. Data flow and control signals are communicated between FPGAs through normal or high speed I/Os. Data flow control and synchronization pose new challenges in algorithm development, FPGA implementation, simulation and verification. Another possibility is to design a heterogeneous system which includes different architectures so that different architectures can process different segments of the algorithm depending on different architecture properties. One simple example is using DSPs with FPGAs in which case FPGAs are usually better at repetitive fixed-point operations while DSPs are more suitable for general and floating-point operations.

With a more powerful computation platform, it would be possible to integrate the obstacle detection, obstacle avoidance, and control modules onto the same hardware platform with the optical flow module. However, the obstacle avoidance algorithm for path planning and the control module are not the focuses of this work and are not included in this dissertation. The goal of the obstacle avoidance module would be to map the obstacle pixels on the 2D image to 3D coordinates and determine

an optimal path for the vehicle. The control module would then compute the servo control signals according to inputs from the obstacle avoidance module.

As discussed previously, the motion model used in the obstacle detection algorithm might be violated in some terrains or driving conditions. With more computational power, more general motion models can be developed to accommodate for more general environments. Also, it is possible to use the optical flow sensor for Unmanned Aerial Vehicle (UAV) applications which have the most general motion patterns and require a higher processing speed.

6.3 Summary

Computer vision systems with real-time performance are attracting researchers in the computer vision research communities with the bloom of different available architectures. This is a great opportunity but also a big challenge for researchers to explore in this new multidisciplinary research area which requires knowledge in computer vision, different computing architectures, electrical engineering, etc. In this dissertation, classical optical flow algorithms were researched in the new field of parallelism computation. Efforts of optimizing algorithms and designing parallel hardware architectures were made to realize the real-time optical flow sensor. An obstacle detection algorithm and its hardware architecture were also designed to demonstrate the practicality of a real-time vision sensor. Beyond the specific area in computer vision research, this dissertation is an actual example of the design process of realizing computationally expensive algorithms on a parallel hardware structure. Also, it can be used as a platform to compare the performance of different parallel architectures on a specific category of algorithms.

Bibliography

[1] S. Wuerger, R. Shapley, and N. Rubin, ""On the visually perceived direction of motion" by Hans Wallach: 60 years later," *Perception*, vol. 25, pp. 1317–1367, 1996. 1

[2] C. Harris and M. Stephens, "A combined corner and edge detector," in *The Fourth Alvey Vision Conference*, 1988, pp. 147–151. 2

[3] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision (DARPA)," in *Proceedings of the 1981 DARPA Image Understanding Workshop*, April 1981, pp. 121–130. 3, 9, 24, 26

[4] C. Tomasi and T. Kanade, "Detection and tracking of point features," Tech. Report CMU-CS-91-132, Carnegie Mellon University, Tech. Rep., April 1991. 3

[5] J. Shi and C. Tomasi, "Good features to track," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1994, pp. 593 – 600. 3

[6] Y.-W. Huang, C.-Y. Chen, C.-H. Tsai, C.-F. Shen, and L.-G. Chen, "Survey on block matching motion estimation algorithms and architectures with new results," *The Journal of VLSI Signal Processing*, vol. 42, no. 3, pp. 297–320, March 2006. 3

[7] Z. Chen, J. Xu, Y. He, and J. Zheng, "Fast integer-pel and fractional-pel motion estimation for H.264/AVC," *Journal of Visual Communication and Image Representation*, vol. 17, no. 2, pp. 264–290, April 2006. 3

[8] B. K. Horn and B. G. Schunck, "Determing optical flow," *Artifical Intellignece*, vol. 17, pp. 185–203, 1981. 3, 7, 8, 9, 24, 25, 56

[9] G. Johansson, "Visual perception of biological motion and a model for its analysis," *Perception & Psychophysics*, vol. 14, pp. 201–211, 1973. 4

[10] S. Zeki, "Parallelism and functional specialization in human visual cortex," in *Cold Sprint Harbor Symposia on Quantitative Biology*, vol. 55, 1990, pp. 651–661. 4

[11] A. Puri, X. Chen, and A. Luthra, "Video coding using the H.264/MPEG-4 AVC compression standard," *Signal Processing: Image Communication*, vol. 19, no. 9, pp. 793 – 849, 2004. 6

[12] I. E. Richardson, *Video Codec Design: Developing Image and Video Compression Systems*. New York, NY, USA: John Wiley & Sons, Inc., 2002. 6

[13] C. Poynton, *Digital Video and HDTV Algorithms and Interfaces*. San Francisco, CA, USA: Morgan Kaufmann, 2003. 6

[14] J. K. Aggarwal and Q. Cai, "Human motion analysis: A review," *Computer Vision and Image Understanding*, vol. 73, pp. 90–102, 1999. 6

[15] A. Jaimes and N. Sebe, "Multimodal human-computer interaction: A survey," *Comput. Vis. Image Underst.*, vol. 108, no. 1-2, pp. 116–134, 2007. 6

[16] T. B. Moeslund, A. Hilton, and V. Krüger, "A survey of advances in vision-based human motion capture and analysis," *Computer Vision and Image Understanding*, vol. 104, no. 2, pp. 90–126, 2006. 6

[17] M. Yamamoto and K. Koshikawa, "Human motion analysis based on a robot arm model," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1991, pp. 664–665. 6

[18] I. A. Kakadiaris, I. A. Kakadiaris, D. Metaxas, R. Bajcsy, and R. Bajcsy, "Active part-decomposition, shape and motion estimation of articulated objects: A physics-based approach," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1994, pp. 980–984. 6

[19] A. Zisserman, A. W. Fitzgibbon, and G. Cross, "VHS to VRML: 3d graphical models from video sequences," in *ICMCS, Vol. 1*, 1999, pp. 51–57. 7

[20] D. Nister, F. Kahl, and H. Stewenius, "Structure from motion with missing data is np-hard," in *International Conference on Computer Vision*, 2007, pp. 1–7. 7

[21] C. T. Frank Dellaert, Steven Seitz and S. Thrun, "Structure from motion without correspondence," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, June 2000. 7

[22] S. Z. Emily Baird, Mandyam V. Srinivasan and A. Cowling, "Visual control of flight speed in honeybees," *Journal of Experimental Biology*, vol. 208, pp. 3895–3905, 2005. 7

[23] O. Shakernia, R. Vidal, C. S. Sharp, Y. Ma, and S. Sastry, "Multiple view motion estimation and control for landing an unmanned aerial vehicle," in *Proceedings of IEEE International Conference on Robotics and Automation*, 2002, pp. 2793–2798. 7

[24] A. Lazanas and J.-C. Latombe, "Landmark-based robot navigation," in *Algorithmica*, 1992, pp. 816–822. 7

[25] T. Y. Tian, C. Tomasi, and D. J. Heeger, "Comparison of approaches to egomotion computation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1996, pp. 315–320. 7

[26] R. Nelson and J. Aloimonos, "Obstacle avoidance using flow field divergence," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 11, no. 10, 1989. 7, 73

[27] S. S. Beauchemin and J. L. Barron, "The computation of optical flow," *ACM Comput. Surv.*, vol. 27, no. 3, pp. 433–466, 1995. 8

[28] B. Jähne, *Handbook of Computer Vision and Applications: Volume 2: From Images to Features*, H. Haussecker and P. Geissler, Eds. Orlando, FL, USA: Academic Press, Inc., 1999. 8, 9, 12, 13, 14, 46

[29] ——, *Spatio-Temporal Image Processing: Theory and Scientific Applications*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1993. 9, 36

[30] H. Nagel, "Displacement vectors derived from second-order intensity variations in image sequences," *Computer Vision, Graphics, and Image Processing (GVGIP)*, vol. 21, no. 1, pp. 85–117, January 1983. 9

[31] S. Uras, F. Girosi, A. Verri, and V. Torre, "A computational approach to motion perception," *Biological cybernetics*, vol. 60, pp. 79–87, 1988. 9

[32] D. J. Fleet and A. D. Jepson, "Computation of component image velocity from local phase information," *International Journal of Computer Vision*, vol. 5, no. 1, pp. 77–104, 1990. 10

[33] H. Bårman, L. Haglund, H. Knutsson, and G. H. Granlund, "Estimation of velocity, acceleration and disparity in time sequences," in *IEEE Workshop on Visual Motion*. Princeton, NJ, USA: IEEE Computer Society Press, October 1991, pp. 44–51. 10

[34] E. Adelson and J. Bergen, "The extraction of spatio-temporal energy in human and machine vision," in *Workshop on Motion: Representation and Analysis*, 1986, pp. 151–155. 10

[35] D. Heeger, "Optical flow using spatiotemporal filters," *International Journal of Computer Vision*, vol. 1, no. 4, pp. 279–302), January 1988. 10

[36] D. J. Fleet and K. Langley, "Recursive filters for optical flow," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 17, no. 1, pp. 61–67, 1995. 10

[37] P. Anandan, "A computational framework and an algorithm for the measurement of visual motion," *International Journal of Computer Vision*, vol. 2, no. 3, pp. 283–310, January 1989. 12

[38] B. Jähne, *Digital Image Processing: Concepts, Algorithms, and Scientific Applications*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1997. 12

[39] A. Singh, *Optic Flow Computation: A Unified Perspective*. IEEE Press, 1990. 12

[40] J. Bigün and G. H. Granlund, "Optimal orientation detection of linear symmetry," in *Proceedings of the IEEE First International Conference on Computer Vision*, London, Great Britain, June 1987, pp. 433–438. 13, 14

[41] H. Knutsson, "Representing local structure using tensors," in *The 6th Scandinavian Conference on Image Analysis*, Oulu, Finland, June 1989, pp. 244–251, report LiTH–ISY–I–1019, Computer Vision Laboratory, Linköping University, Sweden, 1989. 14, 32

[42] J. Bigün, "Local symmetry features in image processing," Ph.D. dissertation, Linköping University, Sweden, 1988, dissertation No 179, ISBN 91-7870-334-4. 14

[43] H. Liu, R. Chellappa, and A. Rosenfeld, "Accurate dense optical flow estimation using adaptive structure tensors and a parametric model," *IEEE Transactions on Image Processing*, vol. 12, pp. 1170–1180, Oct. 2003. 14, 32, 46, 47, 52

[44] G. Farnebäck, "Spatial domain methods for orientation and velocity estimation," Dept. EE, Linköping University, SE-581 83 Linköping, Sweden, Lic. Thesis LiU-Tek-Lic-1999:13, March 1999, thesis No. 755, ISBN 91-7219-441-3. 14, 32

[45] ——, "Fast and accurate motion estimation using orientation tensors and parametric motion models," in *Proceedings of 15th International Conference on Pattern Recognition*, Barcelona, Spain, September 2000, pp. 135–139. 14, 32, 34, 59

[46] B. Johansson and G. Farnebäck, "A theoretical comparison of different orientation tensors," in *Proceedings SSAB02 Symposium on Image Analysis*. Lund: SSAB, March 2002, pp. 69–73. 14

[47] P. Pirsch, N. Demassieux, and W. Gehrke, "VLSI architectures for video compression: A survey," *Proceedings of the IEEE*, vol. 83, no. 2, pp. 220–246, February 1995. 17

[48] A. Dasu and S. Panchanathan, "A survey of media processing approaches," *IEEE Transaction on Circuits and Systems for Video Technology*, vol. 12, no. 8, pp. 633–645, 2002. 17

[49] P. Tseng, Y. Chang, Y. Huang, H. Fang, C. Huang, and L. Chen, "Advances in hardware architectures for image and video coding: A survey," *Proceedings of the IEEE*, vol. 93, no. 1, pp. 184–197, January 2005. 17

[50] R. Hartenstein, "A decade of reconfigurable computing: a visionary retrospective," in *DATE '01: Proceedings of the conference on Design, automation and test in Europe*. Piscataway, NJ, USA: IEEE Press, 2001, pp. 642–649. 18

[51] T. Ramdas and G. Egan, "A survey of FPGAs for acceleration of high performance computing and their application to computational molecular biology," in *Proceedings of IEEE TENCON*. IEEE Press, 2005, pp. 1–6. 19

[52] W. J. MacLean, "An evaluation of the suitability of FPGAs for embedded vision systems," in *Proceedings of the 2005 IEEE Conference on Computer Vision and Pattern Recognition*. Washington, DC, USA: IEEE Computer Society, 2005, p. 131. 19

[53] K. Roy-Neogi and C. Sechen, "Multiple FPGA partitioning with performance optimization," in *FPGA '95: Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM, 1995, pp. 146–152. 20

[54] Altera, "An 549: Managing designs with multiple FPGAs," www.altera.com/literature/an/an549.pdf. 20

[55] M. Gong and Y.-H. Yang, "Near real-time reliable stereo matching using programmable graphics hardware," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 924–931. 21

[56] N. Cornelis and L. Van Gool, "Real-time connectivity constrained depth map computation using programmable graphics hardware," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 1099–1104. 21

[57] NVIDIA, "NVIDIA CUDA programming guide version 2.1," December 2008. 22

[58] J. Bodily, "An optical flow comparison study," ECE Dept., Brigham Young University, Provo, UT, M.S. Thesis, March 2009. 22, 25, 26

[59] D.-T. Lin and C.-Y. Yang, "H.264/AVC video encoder realization and acceleration on TI DM642 DSP," in *PSIVT '09: Proceedings of the 3rd Pacific Rim Symposium on Advances in Image and Video Technology*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 910–920. 23

[60] H. Kubosawa, H. Takahashi, S. Ando, Y. Asada, A. Asato, A. Suga, M. Kimura, N. Higaki, H. Miyake, T. Sato, H. Anbutsu, T. Tsuda, T. Yoshimura, I. Amano, M. Kai, and S. Mitarai, "A 1.2-w, 2.16-gops/720-mflops embedded superscalar microprocessorfor multimedia applications," *IEEE Journal of Solid-State Circuits*, vol. 33, pp. 1640–1648, 1998. 23

[61] M. Butts, "Synchronization through communication in a massively parallel processor array," *IEEE Micro*, vol. 27, no. 5, pp. 32–40, 2007. 23

[62] M. Butts, A. M. Jones, and P. Wasson, "A structural object programming model, architecture, chip and tools for reconfigurable computing," in *FCCM '07: Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 55–64. 23

[63] L. Bonetto, "Accelerating high-performance embedded applications using a massively parallel processing array," Ambric Inc.," Ambric Article for Video Imaging/DesignLine, 2008. 23

[64] B. Hutchings, B. Nelson, S. West, and R. Curtis, "Optical flow on the Ambric Massively Parallel Processor Array (MPPA)," in *FCCM '09: Proceedings of the 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*. Washington, DC, USA: IEEE Computer Society, 2009. 23, 26

[65] A. Zuloaga, J. Martin, and J. Ezquerra, "Hardware architecture for optical flow estimation in real time," in *1998 International Conference on Image Processing, 1998. ICIP 98. Proceedings*, 1998, pp. 972–976. 24

[66] P. C. Arribas and F. M. H. Maciá, "FPGA implementation of camus correlation optical flow algorithm for real time images," in *14th International Conference on Vision Interface*, 2001, pp. 32–38. 24

[67] J. L. Martín, A. Zuloaga, C. Cuadrado, J. Láizaro, and U. Bidarte, "Hardware implementation of optical flow constraint equation using FPGAs," *Comput. Vis. Image Underst.*, vol. 98, no. 3, pp. 462–490, 2005. 24

[68] H. Niitsuma and T. Maruyama, "High speed computation of the optical flow," in *2005 International Conference on Image Analysis and Processing*, 2005, pp. 287–295. 24

[69] J. Diáz, E. Ros, F. Pelayo, E. Ortigosa, and S. Mota, "FPGA-based real-time optical-flow system," *IEEE Transaction on Circuits and Systems for Video Technology*, vol. 16, no. 2, pp. 274–279, February 2006. 24, 32, 41

[70] G. Botella, M. Rodríguez, A. García, and E. Ros, "Neuromorphic configurable architecture for robust motion estimation," *International Journal of Reconfigurable Computing*, vol. 2008. 24

[71] R. Strzodka, "Real-time motion estimation and visualization on graphics cards," in *Proceedings of IEEE Visualization 2004*, 2004, pp. 545–552. 25

[72] Y. Mizukami and K. Tadamura, "Optical flow computation on compute unified device architecture," in *Image Analysis and Processing, 2007. ICIAP 2007. 14th International Conference on*, 2007, pp. 179–184. 25

[73] K. Pauwels and M. Van Hulle, "Realtime phase-based optical flow on the GPU," 2008, pp. 1–8. 25

[74] J. Chase, B. Nelson, J. Bodily, Z. Wei, and D.-J. Lee, "Real-time optical flow calculations on FPGA and GPU architectures: A comparison study," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, pp. 173–182, 2008. 25

[75] M. V. Correia and A. C. Campilho, "Real-time implementation of an optical flow algorithm," in *ICPR '02: Proceedings of the 16 th International Conference on Pattern Recognition (ICPR'02) Volume 4*. Washington, DC, USA: IEEE Computer Society, 2002, p. 40247. 26

[76] A. Dopico, M. Correia, J. Santos, and L. Nunes, "Parallel computation of optical flow," in *International Conference on Image Analysis and Recognition*, 2004, pp. II: 397–404. 26

[77] Xilinx, "Xilinx university program Virtex-II pro development system hardware reference manual," March 2005. 27, 38

[78] Digilent, "Digilent XUP support documents," March 2005, http://www.digilentinc.com/Products/Detail.cfm?Prod=XUPV2P. 27, 38

[79] W. S. Fife and J. K. Archibald, "Reconfigurable on-board vision processing for small autonomous vehicles," *EURASIP J. Embedded Syst.*, vol. 2007, no. 1, pp. 33–33, 2007. 28

[80] B. Jähne, H. W. Haussecker, H. Scharr, H. Spies, D. Schmundt, and U. Schurr, "Study of dynamical processes with tensor-based spatiotemporal image processing techniques," in *ECCV '98: Proceedings of the 5th European Conference on Computer Vision-Volume II*. London, UK: Springer-Verlag, 1998, pp. 322–336. 32

[81] J. Groß, *Linear Regression*. Berlin Heidelberg: Springer-Verlag, 2003. 44, 57

[82] M. Middendorf and H.-H. Nagel, "Estimation and interpretation of discontinuities in optical flow fields," *IEEE International Conference on Computer Vision*, vol. 1, p. 178, 2001. 46

[83] G. Kuhne, J. Weickert, O. Schuster, and S. Richter, "A tensor-driven active contour model for moving object segmentation," in *IEEE International Conference on Image Processing*, 2001, pp. II: 73–76. 46

[84] G. Farnebäck, "Very high accuracy velocity estimation using orientation tensors, parametric motion, and simultaneous segmentation of the motion field," in *Proceedings of the Eighth IEEE International Conference on Computer Vision*, vol. I, Vancouver, Canada, July 2001, pp. 171–177. 46

[85] A. E. Hoerl and R. W. Kennard, "Ridge Regression: Biased Estimation for Nonorthogonal Problems," *Technometrics*, vol. 12, no. 1, pp. 55–67, 1970. 57

[86] D. Comaniciu, "Nonparametric information fusion for motion estimation," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, vol. 1, 2003. 57

[87] A. E. Hoerl, R. W. Kannard, and K. F. Baldwin, "Ridge regression:some simulations," *Communications in Statistics - Theory and Methods*, vol. 4, pp. 105–123, 1975. 57

[88] A. E. Hoerl and R. W. Kannard, "Ridge regression iterative estimation of the biasing parameters," *Communications in Statistics - Theory and Methods*, vol. 5, pp. 77–88, 1976. 58

[89] J. F. Lawless and P. Wang, "A simulation study of ridge and other regression estimators," *Communications in Statistics - Theory and Methods*, vol. 5, pp. 307–323, 1976. 58

[90] G. DeSouza and A. Kak, "Vision for mobile robot navigation: a survey," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 24, no. 2, 2002. 71, 72

[91] C. Thorpe, M. Hebert, T. Kanade, and S. Shafer, "Vision and navigation for the Carnegie-Mellon Navlab," *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 10, no. 3, 1988. 71, 72

[92] S. Modi, P. Chandak, V. S. Murty, and E. L. Hall, "A comparison of three obstacle avoidance methods for a mobile robot," Center for Robotics Research, University of Cincinati, Tech. Rep., 2001. 72

[93] A. Najmi, A. Mahrane, D. Estève, and G. Vialaret, "A scanning LIDAR system for obstacle avoidance in automotive field," in *Proceedings of the 3rd IEEE Conference on Control Application*, August 1994, pp. 379–384. 72

[94] J. Borenstein and Y. Koren, "The vector field histogram - fast obstacle avoidance for mobile robots," *IEEE Trans. on Robotics and Automation*, vol. 7, no. 3, 1991. 72, 73

[95] A. Kosaka and A. Kak, "Fast vision-guided mobile robot navigation using model-based reasoning and prediction of uncertainties," in *Proceedings of International Conference on Intelligent Robots and Systems*, Raleigh, NC, July 1992, pp. 2177–2186. 72

[96] M. Meng and A. Kak, "NEURO-NAV: a neural network based architecture for vision-guided mobile robot navigation using non-metrical models of the environment," in *Proceedings of International Conference on Robotics and Automation*, Atlanta, GA, May 1993, pp. 750–757. 72

[97] "DARPA Urban Challenge," http://www.darpa.mil/grandchallenge/index.asp. 72

[98] I. Ulrich and J. Borenstein, "VFH+: Reliable obstacle avoidance for fast mobile robots," in *Proceedings of the 2000 IEEE Conference on Robotics and Automation*, Leuven, Belgium, May 1998. 73

[99] ——, "VFH*: Local obstacle avoidance with look-ahead verfication," in *Proceedings of the 2000 IEEE Conference on Robotics and Automation*, San Francisco, CA, April 2000. 73

[100] W. Enkelmann, "Obstacle detection by evaluation of optical flow fields from image sequences," in *Proceedings of First European Conference on Computer Vision*, Antibes, France, April 1990, pp. 134–138. 73

[101] W. Enkelmann, V. Gengenbach, W. Küger, S. Rössle, and W. Tölle, "Obstacle detection by real-time optical flow," in *IEEE Proceedings of Intelligent Vehicles '94 Symposium*, October 1994, pp. 97–102. 73

[102] Y. Zhu, D. Comaniciu, M. Pellkofer, and T. Koehler, "Passing vehicle detection from dynamic background using robust information fusion," in *7th IEEE Conference on Intelligent Transportation Systems*, October 1994, pp. 97–102. 73

[103] T. Low and G. Wyeth, "Obstacle detection using optical flow," in *Proceedings of the 2005 Australasian Conference on Robotics and Automation*, Sydney, Australia, December 2005. 73

[104] K. Souhila and A. Karim, "Optical flow based robot obstacle avoidance," *IEEE Journal of Advanced Robotic Systems*, vol. 4, no. 1, 2007. 73

[105] E. Trucco and A. Verri, *Introductory Technique for 3-D Computer Vision.* Upper Saddle River, NJ, USA: Prentice Hall, 1998. 74, 75

[106] G. P. Stein, O. Mano, and A. Shashua, "A robust method for computing vehicle ego-motion," in *Proceedings of the IEEE Intelligent Vehicles Symposium, 2000*, Dearborn, MI, October 2000. 76

[107] Q. Ke and T. Kanade, "Transforming camera geometry to a virtual downward-looking camera: robust ego-motion estimation and ground-layer detection," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, June 2003. 76

[108] S. Baker, S. Roth, D. Scharstein, M. Black, J. Lewis, and R. Szeliski, "A database and evaluation methodology for optical flow," in *International Conference on Computer Vision*, 2007, pp. 1–8. 101