

Real-time Accurate Optical Flow-based Motion Sensor

Zhaoyi Wei, Dah-Jye Lee, Brent E. Nelson, and James K. Archibald
Dept. of Electrical and Computer Engineering, Brigham Young University
agouwin@et.byu.edu djlee@ee.byu.edu nelson@ee.byu.edu jka@ee.byu.edu

Abstract

An accurate real-time motion sensor implemented in an FPGA is introduced in this paper. This sensor applies an optical flow algorithm based on ridge regression to solve the collinear problem existing in traditional least squares methods. It additionally applies extensive temporal smoothing of the image sequence derivatives to improve the accuracy of its optical flow estimates. Implemented on a customized embedded FPGA platform, it is capable of processing 60 320×240 images or 15 640×480 images per second. By evaluating its accuracy on synthetic sequences, it is shown here that the proposed design achieves very high accuracy compared to other known hardware-based designs.

1. Introduction

Compact embedded real-time optical flow sensors can be used for many computer vision tasks such as small unmanned autonomous vehicle navigation, vehicle collision detection, 3D reconstruction, and many other applications which require standalone real-time motion estimation. Because of the high computational cost of optical flow algorithms, traditional general purpose processors typically cannot meet the size, power and real-time requirements. While GPUs or specialized processors [1] can achieve very high calculation rates, their power consumption is not acceptable for most unmanned vehicle applications. In recent years, a number of FPGA-based solutions have been proposed to resolve this issue [2]-[7].

Published FPGA-based solutions achieve much higher processing speed than software-based solutions, but provide much less accurate results. Using the Yosemite sequence as an example, a good software-based algorithm can achieve an angular error of around 1° [8]. In comparison, the angular errors associated with FPGA-based designs are typically more than 10°

[4, 7]. There are two main reasons for this performance difference. First, most optical flow algorithms were developed for software implementations and are simply not a good match for implementation using custom hardware pipeline structures. Second, optical flow algorithms require strong smoothing to suppress noise in order to extract accurate motion information. The limited hardware resources available on most FPGA platforms do not allow proper smoothing compared to software implementations.

Achieving the proper tradeoff between speed, efficiency and accuracy is critical to all hardware designs. Stronger smoothing is needed to suppress noise but it increases latency, utilizes more hardware resources and usually lowers processing speed. In this paper, two efforts are made to optimize speed and accuracy. These include the following:

a) Ridge regression is applied to solve the collinear problem in the traditional least squares method at the cost of a small bias in the estimate.

b) A unique hardware structure is proposed in order to incorporate temporal smoothing which substantially increases the accuracy of the solution.

To the best of our knowledge, this optical flow sensor achieves the best accuracy to date on the Yosemite sequence compared to other published hardware-based designs.

2. Algorithm description

The brightness constancy assumption can be written as $g(x+\Delta x, y+\Delta y, t+\Delta t) = g(x, y, t)$ where x and y are the spatial components, t is the temporal component. An equation regarding derivatives g_x , g_y , and g_t and velocity components v_x and v_y was derived in [9] as

$$g_x v_x + g_y v_y + g_t - \varepsilon = 0 \quad (1)$$

$$\Rightarrow g_t = g_x \cdot (-v_x) + g_y \cdot (-v_y) + \varepsilon$$

where $v_x = \Delta x / \Delta t$, $v_y = \Delta y / \Delta t$, and ε is the error accounting for the higher order terms and noise. Each pixel in the image has one set of observation $\{g_{ii}, g_{xi}, g_{yi}\}$. 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 $\mathbf{g}_i = \mathbf{g}_x \cdot (-v_x) + \mathbf{g}_y \cdot (-v_y) + \boldsymbol{\varepsilon}$ (2) where $\mathbf{g}_i = \{g_{i1}, g_{i2}, \dots, g_{im}\}^T$, $\mathbf{g}_x = \{g_{x1}, g_{x2}, \dots, g_{xm}\}^T$, $\mathbf{g}_y = \{g_{y1}, g_{y2}, \dots, g_{ym}\}^T$, $\boldsymbol{\varepsilon} = \{\varepsilon_1, \varepsilon_2, \dots, \varepsilon_n\}^T$. It is assumed that $E(\varepsilon_i) = 0$ and the variance is σ^2 i.e. $\varepsilon_i \sim (0, \sigma^2)$. Denoting $\mathbf{Y}^{n \times 1} = \mathbf{g}_i$, $\mathbf{X}^{n \times 2} = (\mathbf{g}_x, \mathbf{g}_y)$, $\boldsymbol{\theta}^{2 \times 1} = -(v_x, v_y)^T$, the equation regarding the observation $\{g_{xi}, g_{yi}\}$ and the parameter $\boldsymbol{\theta}$ can be written as $\mathbf{Y} = \mathbf{X}\boldsymbol{\theta} + \boldsymbol{\varepsilon}$ (3).

A normal least squares solution of $\boldsymbol{\theta}$ in equation (3) is $\hat{\boldsymbol{\theta}}_{LS} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y}$. (4) As shown in [10], $E(\hat{\boldsymbol{\theta}}_{LS}) = \boldsymbol{\theta}$ and its covariance matrix is $\text{Cov}(\hat{\boldsymbol{\theta}}_{LS}) = \sigma^2 (\mathbf{X}^T \mathbf{X})^{-1}$. If \mathbf{g}_x and \mathbf{g}_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 paper, a ridge estimator [10]-[12] as formulated in (5) is proposed to address this.

$$\hat{\boldsymbol{\theta}}_{RE} = (\mathbf{X}^T \mathbf{X} + k \mathbf{I}_p)^{-1} \mathbf{X}^T \mathbf{Y} \quad (5)$$

In (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 [10] that the expectation and covariance matrices of $\hat{\boldsymbol{\theta}}_{RE}$ are

$$E(\hat{\boldsymbol{\theta}}_{RE}) = \boldsymbol{\theta} - k(\mathbf{X}^T \mathbf{X} + k \mathbf{I}_p)^{-1} \boldsymbol{\theta} \quad (6)$$

$$\text{Cov}(\hat{\boldsymbol{\theta}}_{RE}) = \sigma^2 \mathbf{X}^T \mathbf{X} (\mathbf{X}^T \mathbf{X} + k \mathbf{I}_p)^{-2} \quad (7)$$

Although a ridge estimator is biased, i.e. $E(\hat{\boldsymbol{\theta}}_{RE}) \neq \boldsymbol{\theta}$ as shown in (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 [10]. As to the selection of k , an HKB estimator [13] shown as

$$k = p \hat{\sigma}^2 / \hat{\boldsymbol{\theta}}_N^T \hat{\boldsymbol{\theta}}_N \quad (8)$$

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

$$\hat{\sigma}^2 = \frac{(\mathbf{Y} - \mathbf{X}\hat{\boldsymbol{\theta}}_N)^T (\mathbf{Y} - \mathbf{X}\hat{\boldsymbol{\theta}}_N)}{n - p} \quad (9)$$

There exist other methods to estimate the scalar e.g. an iterative HK estimator [14] and LW estimator [15] etc. The HKB estimator is chosen for its efficiency and

non-iterative property. After obtaining k , the optical flow is estimated using (5).

In a real implementation, an $n \times n$ weighting matrix \mathbf{W} is used to assign weights to each set of observation based on their distance to the central pixel. Equations (5) and (9) are rewritten as

$$\hat{\boldsymbol{\theta}}_{RE} = (\mathbf{X}^T \mathbf{W} \mathbf{X} + k \mathbf{I}_p)^{-1} \mathbf{X}^T \mathbf{W} \mathbf{Y}, \quad (10)$$

$$\hat{\sigma}^2 = \frac{(\mathbf{Y} - \mathbf{X}\hat{\boldsymbol{\theta}}_N)^T \mathbf{W} (\mathbf{Y} - \mathbf{X}\hat{\boldsymbol{\theta}}_N)}{n - p}. \quad (11)$$

To suppress noise, the derivatives g_x , g_y , and g are spatio-temporally smoothed, respectively before they are used in (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.

3. Sensor system design

The proposed compact optical flow sensor is an embedded vision system including video capture, processing, transferring and other functionality. Various modules are connected to the buses as shown in Fig. 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 (DERivatives calculation), OFC (Optical Flow Calculation), SDRAM, camera and USB modules. Lower speed modules such as the UART, interrupt controller, and GPIO are connected to the OPB bus. The PLB and OPB buses are interconnected through a bridge.

Data flow in the system is directed by software running on built-in PowerPC processors. The dataflow in the system is as follows. The DER module is triggered when a new image frame is captured by the camera and stored in the SDRAM. After the DER module processing of that frame is complete, the intermediate results of the DER module are stored in both SRAM and SDRAM. The OFC module then processes the intermediate results and stores the resulting motion vectors in SDRAM. The motion vectors can then be transferred to the host PC through the USB interface. A graphical user interface has been developed to observe and store the video and status variables transferred from the sensor.

The DER module diagram is shown in Fig. 2. It calculates the derivative frames g_x , g_y , g_t from raw images. To obtain a suitable tradeoff between accuracy and efficiency, mask $\mathbf{D} = (1, -8, 0, 8, -1)/12$ is used to calculate the derivatives. Note that the calculation of g_t is a temporal calculation - calculating it requires frame(t), frame($t-1$), frame($t-3$), and frame($t-4$) whereas calculating g_x and g_y is a spatial calculation

requiring only $\text{frame}(t-2)$. The results of computing these derivatives are stored both in SRAM and SDRAM. The reason for doing this is that the derivative frames (the intermediate results) do not fit completely within the either FIFO between the DER and OFC modules or the SRAM and thus SDRAM must be used to hold them.

The OFC module diagram is shown in Fig. 3. It reads the derivative frames from the SRAM and SDRAM into the hardware pipeline and then temporally and spatially smoothes them. These smoothed derivative frames are then combined to build regression model components. These components are spatially smoothed and then fed into the scalar estimator in (8) to calculate scalar k . Optical flow can then be calculated with the scalar k and the smoothed regression model components in (11). The optical flow vector is buffered to be lined up with the pixel right below on the next row in order to calculate k . The optical flow vector is then spatially smoothed to obtain the final optical flow vector.

In the OFC module, the size of the temporal smoothing window is three, which means three sets of derivative frames need to be stored. Besides temporal smoothing, there are three spatial smoothing units as well: a derivative frame spatial smoothing unit, a regression model components spatial smoothing unit, and an optical flow spatial smoothing unit. Components of these smoothing masks are in the shape of a 2D Gaussian function and their mask sizes are 5×5 , 3×3 , and 7×7 respectively. The configurations of these masks were evaluated carefully in software before implementation to achieve an optimal tradeoff between accuracy and efficiency.

To accommodate temporal smoothing, the hardware pipeline is divided into two parts (DER, OFC) because the derivative frames are too big to be stored in the pipeline itself. A drawback of this scheme 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.

4. Experiment results

The design was implemented on the Helios FPGA platform [16]. This platform is equipped with a Virtex-4 FX60 FPGA which has two PowerPC processors in addition to configurable logic resources. The system runs at 100 MHz. In the system, the camera core is set to output 640×480 images at 15 fps or 320×240 images at 60 fps depending on the application. For debugging and evaluating purposes, a bit-level accurate MATLAB simulation code was programmed to match the hardware circuit.

Fig. 4 shows the effect of ridge regression tested on the lower part of the Yosemite sequence. Fig. 4(a) shows the result using the least squares method. Fig. 4(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).

Table 1 shows the accuracies on the Yosemite sequence under different settings. The accuracies are measured in average angular error. 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. A_3 is the accuracy without applying temporal smoothing. A_4 is the accuracy using least squares and without temporal smoothing. Fig.(5) shows the test results on the Yosemite, flower garden, SRI tree, and BYU corridor sequences. The proposed algorithm works very well on these sequences as well.

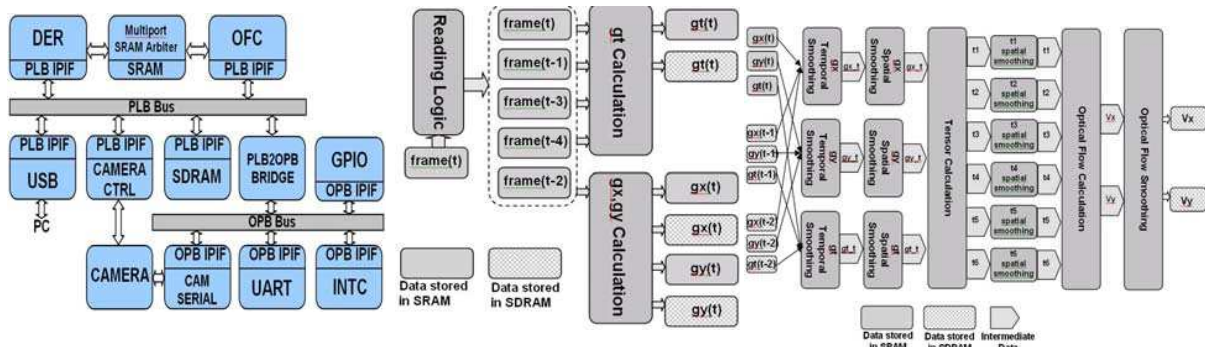


Figure 1 System diagram

Figure 2 DER module

Figure 3 OFC module

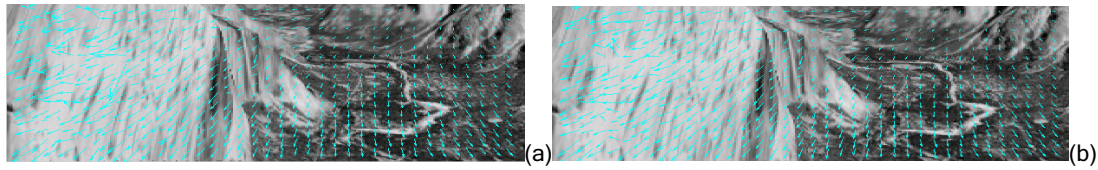


Figure 4 Performance improvement using ridge regression.

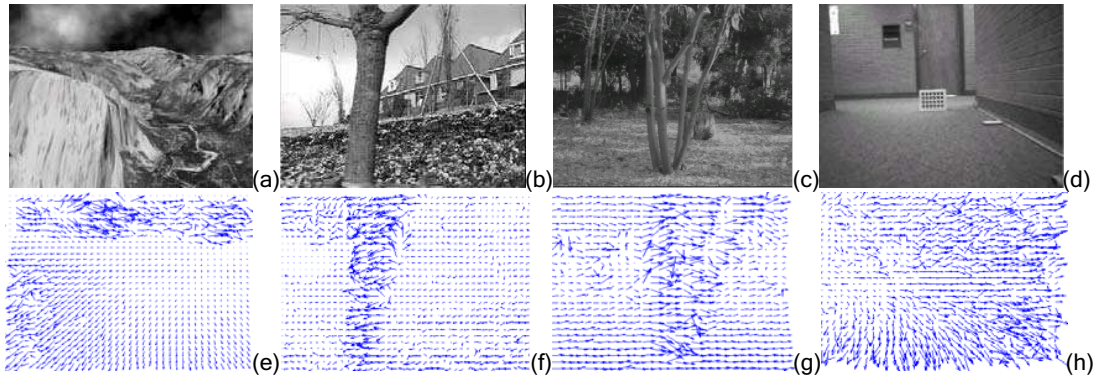


Figure 5 Experiment results

Table 1 Accuracies comparison

A_1	A_2	A_3	A_4
6.8°	7.2°	10.9°	11.4°

5. Conclusions and future work

An accurate optical flow algorithm implemented in an embedded motion sensor is discussed in this paper. 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. These efforts substantially improve the accuracy.

Future work will include using real-time optical flow for small unmanned vehicle navigation tasks such as obstacle avoidance and self-motion estimation. Motion information will be fed into the control loop to help make control decisions.

References

- [1] M. Correia, A. Campilho. Real-time implementation of an optical flow algorithm, *Proc. ICPR*, 4:247-250, 2002.
- [2] A. Zuloaga, J. L. Martin, J. Ezquerro. Hardware architecture for optical flow estimation in real time. *Proc. ICIP*, 3:972-976, 1998.
- [3] J. L. Martin, A. Zuloaga, C. Cuadrado, J. Lázaro, U. Bidarte. Hardware implementation of optical flow constraint equation using FPGAs. *Computer Vision and Image Understanding*, 98:462-490, 2005.
- [4] J. Díaz, E. Ros, F. Pelayo, E. M. Ortigosa, S. Mota. FPGA-based real-time optical-flow system. *IEEE Trans.*

Circuits and Systems for Video Technology, 16(2): 274-279, February 2006.

- [5] P. C. Arribas, F. M. H. Maciá. FPGA implementation of camus correlation optical flow algorithm for Real Time Images. *14th Int. Conf. Vision Interface*, 32-38, 2001.
- [6] H. Niituma, T. Maruyama. High speed computation of the optical flow. *Lecture notes in Computer Science*, 3617:287-295, 2005.
- [7] Z.Y. Wei, D.J. Lee, B. Nelson, M. Martineau. A Fast and Accurate Tensor-based Optical Flow Algorithm Implemented in FPGA. *IEEE WACV*, p18, Nov. 2007.
- [8] G. Farneback, "Very high accuracy velocity estimation using orientation tensors, parametric motion, and simultaneous segmentation of the motion field", *Proc. ICCV*, vol. 1, pp. 77-80, 2001.
- [9] B. Horn, B. Schunck, "Determining optical flow", *Artificial Intelligence*, vol. 17, pp. 185-203, 1981.
- [10] Jürgen Groß. *Linear Regression*. Lecture Notes in Statistics 175. Springer-Verlag, Berlin, Heidelberg, 2003.
- [11] A. Hoerl, R. Kennard. Ridge regression. Biased estimation for nonorthogonal problems, *Technometrics*, 12(1):55-67, 1970.
- [12] D. Comaniciu. Nonparametric Information Fusion for Motion Estimation. *Proc. CVPR*, 1:59-66, June, 2003.
- [13] A. Hoerl, R. Kennard, K. Baldwin. Ridge regression: some simulations. *Communications in Statistics, Theory and Methods*, 4:105-123, 1975.
- [14] A. Hoerl, R. Kennard. Ridge regression iterative estimation of the biasing parameters. *Communication in Statistics, Theory and Methods*, 5:77-88, 1976.
- [15] J. Lawless, P. Wang. A simulation study of ridge and other regression estimators. *Communications in Statistics, Theory and Method*, 5:307-323, 1976.
- [16] <http://www.ece.byu.edu/roboticvision/helios/>