Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2008, Article ID 636145, 8 pages
doi:10.1155/2008/636145

Research Article

FPGA-Based Embedded Motion Estimation Sensor

Zhaoyi Wei, Dah-Jye Lee, Brent E. Nelson, James K. Archibald, and Barrett B. Edwards

Electrical and Computer Engineering Department, Brigham Young University, Provo, UT 84602, USA

Correspondence should be addressed to Dah-Jye Lee, djlee@ee.byu.edu

Received 27 March 2008; Accepted 24 June 2008

Recommended by Fernando Pardo

Accurate real-time motion estimation is very critical to many computer vision tasks. However, because of its computational power and processing speed requirements, it is rarely used for real-time applications, especially for micro unmanned vehicles. In our previous work, a FPGA system was built to process optical flow vectors of 64 frames of 640 × 480 image per second. Compared to software-based algorithms, this system achieved much higher frame rate but marginal accuracy. In this paper, a more accurate optical flow algorithm is proposed. Temporal smoothing is incorporated in the hardware structure which significantly improves the algorithm accuracy. To accommodate temporal smoothing, the hardware structure is composed of two parts: the derivative (DER) module produces intermediate results and the optical flow computation (OFC) module calculates 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. This new design has been implemented on a compact, low power, high performance hardware platform for micro UV applications. It is able to process 15 frames of 640 × 480 image per second and with much improved accuracy. Higher frame rate can be achieved with further optimization and additional memory space.

1. Introduction

Optical flow aims to measure motion field from the apparent motion of the brightness pattern. Optical flow is one of the most important descriptions for an image sequence and is widely used in 3D vision tasks such as motion estimation, structure from motion (SfM), time-to-impact, and so forth. An accurate hardware-friendly optical flow algorithm is proposed and its hardware design is presented in this paper. This design can be used for unmanned air vehicle (UAV) navigation tasks, moving object detection, and other applications which require fast and accurate motion estimation.

The basic assumption of optical flow algorithms is the brightness constancy constraint, which assumes that image brightness changes are due only to motion. In other words, if motion between frames is small, other effects (such as changes in lighting conditions) causing brightness changes can be neglected. However, one limitation of the use of optical flow is its computational requirement. The processing time of existing optical flow algorithms is usually on the order of seconds or longer per frame. This long processing time thus prevents optical flow algorithms from being used for most real-time applications such as autonomous navigation for unmanned vehicles.

In recent years, a number of different schemes have been proposed to implement optical flow algorithms for real-time applications. The basic idea behind them is to use pipeline or parallel processing architectures to speed up computations. For example, graphics processing unit (GPU) devices have recently been used for optical flow implementation with good results [1–3]. Alternatively, Correia and Campilho [4] proposed a design which can process the Yosemite sequence in 47.8 milliseconds using a pipeline image processor. Compared to GPUs or processors, FPGAs are more suitable for small UAV operation because of their compact size and low power consumption. FPGAs have been used to calculate optical flow [5–13] in the past few years because of their configuration flexibility and high data processing speed. For example an iterative algorithm proposed by Horn and Schunck [14] was implemented in an FPGA in [5–7]. The classical Lucas and Kanade approach [15] was also implemented in [7, 8] and provided a tradeoff between accuracy and processing efficiency.

To improve accuracy and provide high-performance solutions, many different 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 [16–19]. The 3D tensor provides a powerful and closed form representation of the local brightness structure. In our previous work, a tensor-based optical flow algorithm was implemented on an FPGA to process 64 frames of 640×480 image per second.

Despite the fact that hardware-based optical flow design can achieve much higher processing rates than software, hardware-based designs are usually less accurate. Using the design in [12] as an example, its accuracy measured in angular error is 12.7° on the Yosemite sequence while state-of-art optical flow algorithms using software and with sufficient memory space can achieve accuracy close to 1.0° on the same sequence. Two main reasons affecting accuracy are the following.

(1) Algorithm limitation: many software-based algorithms use iteration and apply various optimization techniques to find optimal values. However, hardware is best for algorithms which can be pipelined and processed in parallel. Therefore, only a very limited subset of the available software-based algorithms is a good fit for hardware. Sacrificing accuracy for processing speed is inevitable for hardware implementation.

(2) Hardware resource limitations: for optical flow algorithms, smoothing is a very important operation for suppressing noise and extracting accurate motion information. However, current smoothing algorithms for software are not suitable for hardware implementation where a tradeoff between accuracy and feasibility needs to be made. Moreover, software-based algorithms mostly use floating point computations. When implementing software optical calculations in hardware, data truncation, rounding, and saturation operations lower the algorithm's accuracy.

Our goal in this paper is to improve the accuracy of previous hardware optical flow designs under the prerequisite of speed and feasibility. The design described herein uses temporal smoothing in addition to spatial smoothing in the calculation to improve the accuracy and stability of the algorithm. To accommodate temporal smoothing (which is not trivial in hardware), a new hardware organization is used. The hardware pipeline is decomposed into two modules. The first module calculates derivative frames for each image frame. The second module calculates optical flow from the derivative frames. From the experimental analysis done, it can be seen that this design achieves an error rate of 6.7° on the Yosemite sequence which doubles the accuracy of previous work [12]. The temporal smoothing used prevents the creation of a fully pipelined design. Further, it requires additional memory bandwidth, leading to a processing rate of 15 frames per second for 640×480 images. Higher frame rates could be achieved with further optimizations and additional memory space. However, this frame rate is sufficient for many unmanned vehicle applications that require higher accuracy.

The proposed algorithm is implemented on a newly developed standalone hardware platform. Images captured by the on-board CMOS camera are processed by hardware and can be transferred to PC for debugging and evaluation purposes.

This paper is organized as follows. In Section 2, the algorithm is formulated and our modifications are introduced. In Section 3, the hardware structure and tradeoffs made in the design are discussed. In Section 4, the hardware platform is introduced. Performance analysis of the proposed design on synthetic and real sequences is also shown. Conclusions and future work are discussed in Section 5.

2. Optical Flow Algorithm

2.1. Algorithm Description

An image sequence $g(\mathbf{X})$ can be treated as volume data where $\mathbf{X} = (x, y, t)^T$, x and y are the spatial components, and t is the temporal component. According to the brightness constancy constraint, object movement in the spatiotemporal domain will generate brightness patterns with certain orientations. The 3D tensor is a compact representation of local orientation and there are different types of 3D tensors based on different formulations [20, 21]. The gradient tensor is used in this design because it is easier to implement with pipelines in hardware.

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

$$\mathbf{O} = \nabla\overline{g}(\mathbf{x})\nabla\overline{g}(\mathbf{x})^T = \begin{pmatrix} o_1 & o_4 & o_5 \\ o_4 & o_2 & o_6 \\ o_5 & o_6 & o_3 \end{pmatrix}, \quad (1)$$

where

$$\nabla\overline{g}(\mathbf{x}) = \sum_i w_i \nabla g(\mathbf{x}_i) = \left(\overline{g}_x(\mathbf{x}) \;\; \overline{g}_y(\mathbf{x}) \;\; \overline{g}_t(\mathbf{x})\right)^T, \quad (2)$$

and w_i are weights for averaging the gradients. The gradient component is calculated using a simple mask shown in (3) which is the same as in [12], which was chosen as a tradeoff between accuracy and efficiency,

$$\mathbf{d} = \begin{pmatrix} 1 & -8 & 0 & 8 & -1 \end{pmatrix}. \quad (3)$$

Gradient tensor $\mathbf{T}$ is a 3×3 positive semidefinite matrix that is constructed by weighting $\mathbf{O}$ in a small neighborhood as

$$\mathbf{T} = \sum_i c_i \mathbf{O}_i = \begin{pmatrix} \overline{t}_1 & \overline{t}_4 & \overline{t}_5 \\ \overline{t}_4 & \overline{t}_2 & \overline{t}_6 \\ \overline{t}_5 & \overline{t}_6 & \overline{t}_3 \end{pmatrix}. \quad (4)$$

Optical flow $(v_x, v_y)^T$ is measured in pixels per frame and can be extended to a 3D spatiotemporal vector, $\mathbf{V} = (v_x, v_y, 1)^T$. For an object with only translational movement and without noise in the neighborhood, $\mathbf{V}^T\mathbf{T}\mathbf{V} = \mathbf{0}$. In the presence of noise and rotation, $\mathbf{V}^T\mathbf{T}\mathbf{V}$ will not be zero.

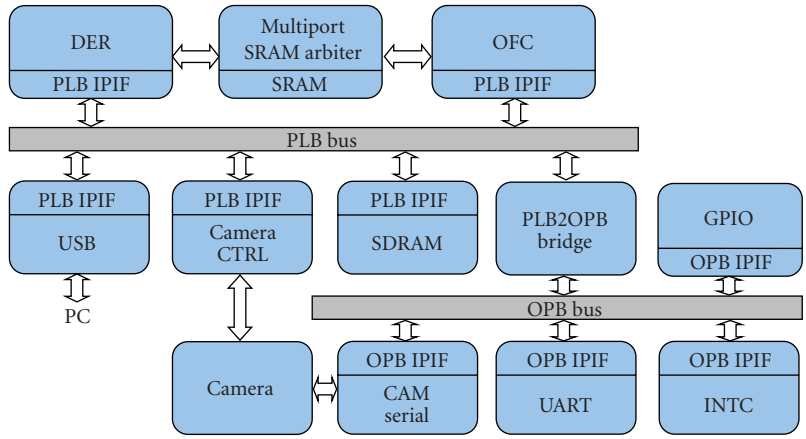


Figure 1: System hardware diagram.

Instead, $\mathbf{V}$ can be determined by minimizing $\mathbf{V}^T\mathbf{T}\mathbf{V}$. A cost function can be defined for this purpose as

$$e(\mathbf{V}) = \mathbf{V}^T\mathbf{T}\mathbf{V}. \tag{5}$$

The velocity vector $\mathbf{V}$ is the 3D spatiotemporal vector which minimizes the cost function at each pixel. The initial optical flow can be solved as

$$\begin{aligned} v_x &= \frac{(\bar{t}_6\bar{t}_4 - \bar{t}_5\bar{t}_2)}{(\bar{t}_1\bar{t}_2 - \bar{t}_4^2)}, \\ v_y &= \frac{(\bar{t}_5\bar{t}_4 - \bar{t}_6\bar{t}_1)}{(\bar{t}_1\bar{t}_2 - \bar{t}_4^2)}. \end{aligned} \tag{6}$$

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

$$\bar{\mathbf{v}}_i = \begin{pmatrix} \bar{v}_x \\ \bar{v}_y \end{pmatrix} = \sum_i m_i \begin{pmatrix} v_{xi} \\ v_{yi} \end{pmatrix}. \tag{7}$$

The algorithm proposed in this paper is similar to the one in [17] which assumes a constant motion model. Affine motion model is often used to incorporate tensors in a small neighborhood [17] 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.

2.2. Smoothing Masks

As mentioned in Section 2.1, smoothing is very important to the algorithm performance. Smoothing is performed through three smoothing masks w_i, c_i, and m_i as shown in (2), (4), and (7) where w_i is for gradient averaging, c_i is for tensor construction, and m_i is for velocity smoothing. The first mask is a spatiotemporal smoothing mask, and the other two are spatial smoothing masks.

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., 19×19 or larger) are often used. In hardware, smaller masks must be used because of resource limitations (e.g., 7×7 or smaller). As for mask shape, a square mask is usually used for the sake of simplicity and efficiency. Spatial and temporal smoothings can use different size masks to improve performance. While spatial smoothing can be readily performed in either software or hardware, spatiotemporal smoothing is much more challenging in hardware than in than software. 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. In this design, the size of the first smoothing mask w_i in (2) is $5 \times 5 \times 3$ of which 3 is the temporal smoothing size (3 frames). The number of frames used for temporal smoothing is largely determined by hardware resources and processing speed requirement. The sizes of the other two spatial smoothing masks c_i, and m_i are 3×3 and 7×7, respectively. 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 [12]. 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.

3. Hardware Structure

The diagram of the hardware platform using a Xilinx Virtex-4 FX series FPGA for implementing our optical flow algorithm discussed in Section 2 is shown in Figure 1. Most hardware components are either connected to the processor local bus (PLB) or on-chip peripheral bus (OPB). The PLB

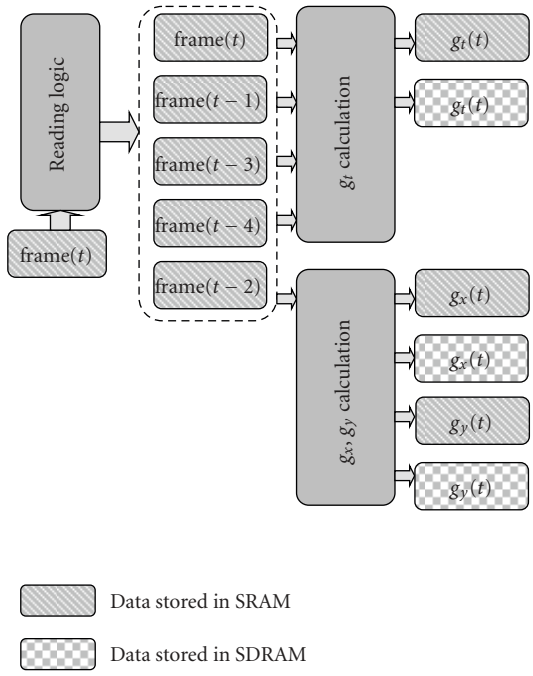


Figure 2: DER module diagram.

is used for connecting hardware components that require high bandwidth such as camera and USB ports, and so forth. The OPB is used for connecting relatively slower units such as UART, interrupt controller, and so on. PLB and OPB are connected through PLB2OPB bridge. Two main components of this design: derivative (DER) module and optical flow computation (OFC) module are connected to the PLB bus through a bus interface. The DER and OFC modules share the SRAM through a multiport SRAM arbiter connected to the SRAM to obtain optimal accessing bandwidth. The slower SDRAM is accessed through the PLB bus.

The main difference between this design and our previous design [10] is that it incorporates temporal smoothing in the pipeline. In this design, three sets of derivative frames, g_x, g_y, and g_t shown in (2) are temporally smoothed.

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 frames on-chip using the hardware resources that are available. Therefore, the optical flow calculation pipeline has to be divided into two steps. The first part (called DER module) generates derivative frames and the second part (called OFC module) handles the rest of calculations. 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 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 g_x, g_y, and g_t using (3). g_x and g_y are calculated from the current incoming frame(t) and g_t is calculated from frame($t-4$), frame($t-3$), frame($t-2$), frame($t-1$), and the current incoming frame. 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 1, if the hardware platform used had sufficient SRAM (currently only 4 Mb), then all 9 derivative frames (3 sets of g_x, g_y, and g_t) could be stored in the SRAM and take the advantage of a high-speed multiport memory interface.

Figure 3 shows the dataflow of the OFC module. 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 ($g_x(t)$, $g_y(t)$, and $g_t(t)$) and the derivative frames already stored in the SDRAM ($g_x(t-1)$, $g_y(t-1)$, and $g_t(t-1)$) and ($g_x(t-2)$, $g_y(t-2)$, and $g_t(t-2)$) are first read into the pipeline for temporal smoothing. Because the size of the temporal smoothing mask is 3, derivative frames at time t, $t-1$, and $t-2$ are averaged to obtain the smoothed derivative frames for the current frame at time t (gx_t, gy_t, and gt_t, in Figure 3). These frames are then spatially smoothed. The spatially smoothed derivative frames ($\overline{gx\_t}$, $\overline{gy\_t}$, and $\overline{gt\_t}$, in Figure 3) are then used to construct the tensors as shown in (4). Six tensor components t_1, t_2, t_3, t_4, t_5, and t_6 are obtained after this step. These six components are spatially smoothed using a 3×3 smoothing mask. Then, v_x and v_y can be calculated from these smoothed components as shown in (6). Two motion components v_x and v_y are spatially smoothed with m_i to get the final optical flow vectors as shown in (7).

Figure 4 shows the system software diagram. There are three types of frames existing 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. A frame table entry (FTE) 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.

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

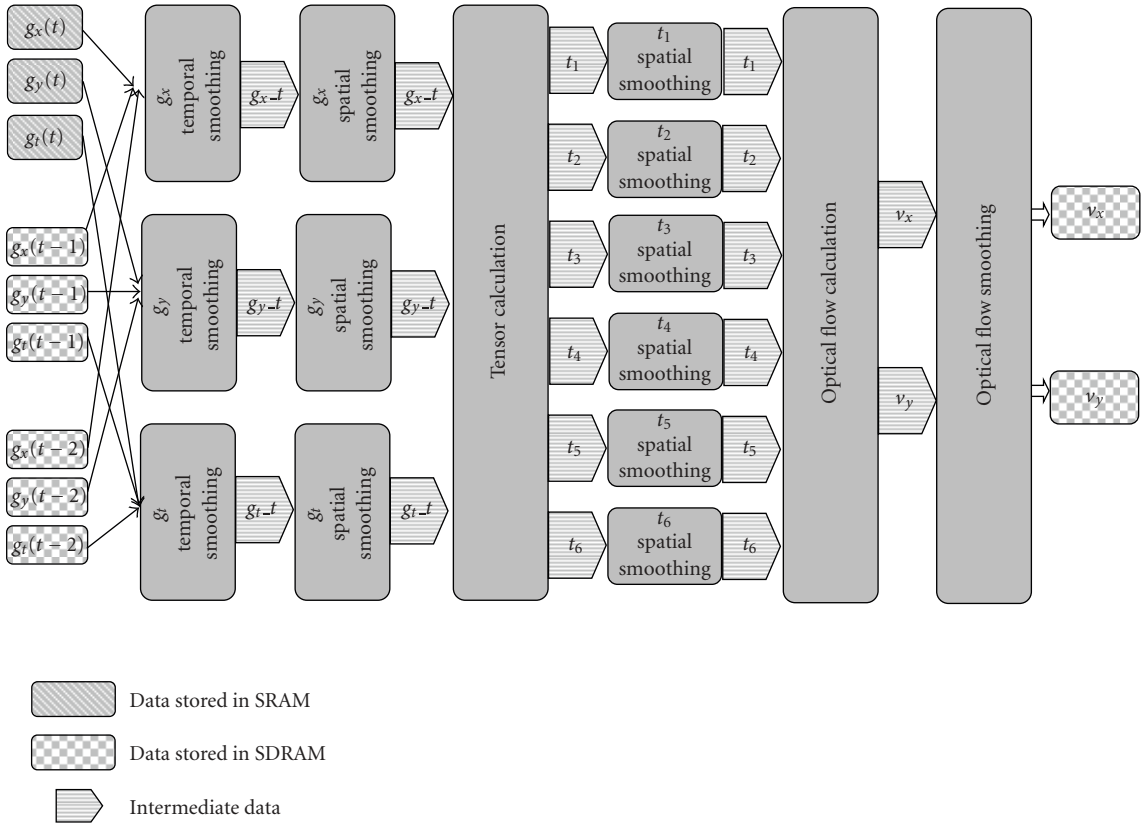


Figure 3: OFC module diagram.

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. Results

4.1. Hardware Platform

This design was implemented on the BYU Helios Robotic Vision board [22, 23]. This embedded vision sensor board was developed in the Robotic Vision Laboratory at Brigham Young University for real-time visual computing. The Helios board, shown in Figure 5(a), is compatible with Xilinx Virtex-4 FX series FPGAs, up to the FX60. The Virtex-4 FX60 has 25 280 slices and two built-in 400 MHz powerPC processors, allowing for both custom hardware and software development. The Autonomous Vehicle Toolkit (AVT) daughterboard (Figure 5(c)) which is capable of supporting up to two CMOS cameras (Figure 5(b)) 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 640×480 8-bit color image data per second in this design.

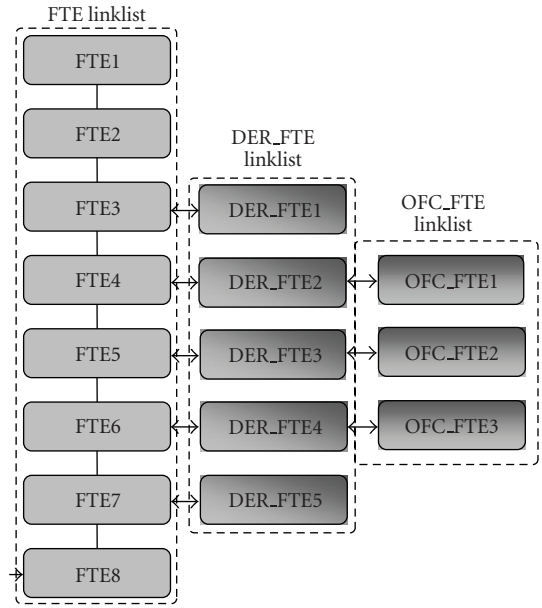
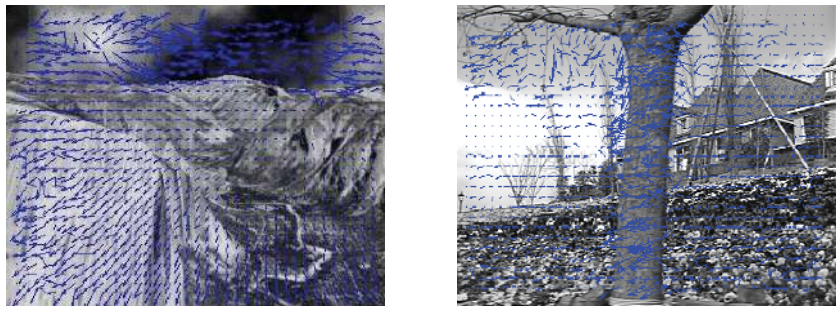


Figure 4: Software diagram.

For computer vision tasks, high-speed memory is critical. There are currently 32 Mb SDRAM and 4 Mb SRAM on the Helios platform. The SRAM is faster and easier to access than the SDRAM. In this design, we tried to store as much intermediate data as possible (e.g., image frames and derivative frames) in the SRAM. The rest of the data was stored in the SDRAM and accessed through the PLB Bus. There are other hardware resources and features on the Helios that are useful for other real-time vision tasks but which were not used in this design. This hardware platform

(a) (b) (c)

Figure 5: Hardware components: (a) Helios FPGA board, (b) CMOS imager, and (c) AVT daughter board.



(a) (b)

Figure 6: Result of synthetic sequences: (a) Yosemite, and (b) Flower garden.

is suitable for small UGV or UAV which requires low power, real-time computation capability. Detailed information can be found in [22, 23].

The whole system executes at a clock speed of 100 MHz 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 computations used 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 to increased processing speed. Our estimation is that higher frame rate can be achieved with further optimization work and additional memory space (specifically on-board SRAM). These processing speed improvements are discussed in Section 5. 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 21481 slices (84% of the total 25280 slices on a Virtex-4 FX60 FPGA). The DER module used 2196 slices (8% of the total) and the OFC module used 6324 slices (25% of the total). The remainder was used for the camera core, I/O, and other interface circuitry. A graphical user interface (GUI) was developed to transfer the status of the Helios board and the real-time video to PC for display through the USB interface.

4.2. Synthetic Sequences

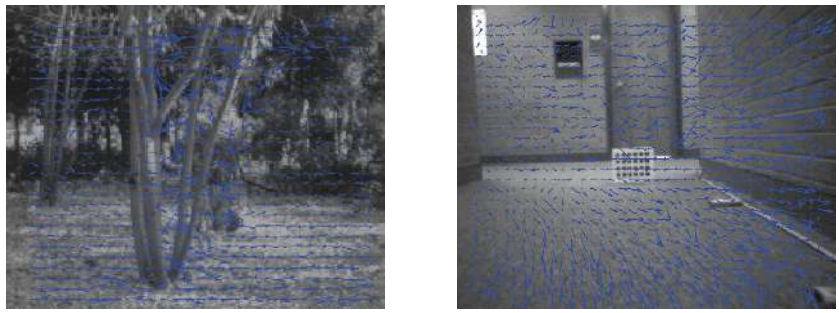
This design was tested on two synthetic sequences to show its effectiveness. One of these sequences (Yosemite) has an established ground truth and is commonly used for benchmarking. A bit level simulation coded in MATLAB was programmed to evaluate the algorithm's accuracy. The error is measured in angular error which is widely used for evaluating optical flow algorithm.

Figure 6(a) shows one frame of the Yosemite sequence with the calculated optical flow vectors superimposed for visual feedback. The angular error of the Yosemite sequence was 6.7°. To the best of our knowledge, this is the most accurate result generated to date using hardware, and has half the error rate of our previous design [12] which achieved an error rate of 12.7°. There are two major reasons for this improvement. The first is that a better hardware platform (the Helios platform) was used and supported the implementation of a more complex algorithm on the hardware. The second reason is the incorporation of temporal smoothing in the algorithm. Without temporal smoothing, the accuracy would deteriorate to approximately 10.5°. This experiment demonstrates that temporal smoothing has a significant impact on the algorithm performance. Using temporal smoothing is an acceptable tradeoff of processing speed for accuracy for applications that require higher accuracy.

Figure 6(b) shows the result of the other synthetic sequence (flower garden). While the proposed algorithm did not perform well in regions without much texture, this is a shortcoming of all optical flow and motion estimation algorithms. In regions with noticeable texture, the algorithm performed very well.

4.3. Real Sequences

Two real sequences were also used to test the performance of the proposed algorithm. Figure 7(a) shows the result of the



(a) (b)

Figure 7: Result of real sequences: (a) SRI trees and (b) corridor.

(a) (b)

Figure 8: Real-time result from hardware: (a) original frame and (b) coded optical flow.

SRI tree sequence, another standard benchmark sequence for testing optical flow algorithms based on captured image data. It can be seen that the proposed algorithm performed very well on this sequence. Optical flow vectors were generally very smooth except for some few noisy vectors along the motion boundary. This was caused by the violation of the assumption that the motion should be constant in a local neighborhood which is another shortcoming of all optical flow and motion estimation algorithms.

Figure 7(b) shows the result of the corridor sequence, an image sequence consisting of around 500 frames captured in our lab. As can be seen, there are noisy motion vectors in the background area which has less texture than other regions. In this case, even stronger smoothing would help suppress the noise and "infer" the velocity from the local neighborhoods.

The data width transferred between the Helios board and GUI was 8 bits per pixel. To transfer the optical flow vector values, we defined the data as a 4.4 format signed fixed point number which means it had 1 sign bit, 3 integer bits, and 4 fraction bits. Therefore, small optical flow vectors are coded with gray intensity values. Large positive optical flow vectors (object moves down or to the right) are coded with bright intensity values. Similarly, large negative optical flow vectors (object moves up or to the left) are coded with dark intensity values. This coding mechanism was for the display of the motion field in real time for debugging purpose.

Figure 8(a) shows the screenshot of an original video frame that was captured using our hardware platform, transferred to the PC through the USB interface, and displayed in the GUI. The palm moved downwards from the top of the screen at a steady speed. Figure 8(b) shows the motion field (mostly in y direction). Since the palm moving downwards, the resulting optical flow vectors are coded with low intensity as explained previously. The light gray intensity values in the background represent very small motion that was caused by the flickering of the fluorescent lights in the lab and the digitization noise from the camera.

5. Conclusion

High computation and low power consumption requirements make it difficult to use general purpose processors or GPU to implement optical flow algorithms for real-time small unmanned vehicle applications. A hardware accelerated design of a tensor-based optical flow algorithm has been developed and implemented in an FPGA platform. Two synthetic sequences and two real sequences were used to test its effectiveness. The accuracy on the Yosemite sequence was shown to be substantially better than our previous design and other hardware implementation found in the literature.

The accuracy improvement in this design is due to two reasons.

(1) A better hardware platform is used in this design to be able to compute a more complex algorithm.

(2) Temporal smoothing is incorporated into the calculation. Temporal smoothing increases the temporal coherency, provides stronger smoothing, and improves performance.

To accommodate temporal smoothing, the pipelined hardware structure is divided into two parts: DER and OFC modules. Accordingly, the amount of image data that must be stored and transferred increases dramatically. Currently, this design is running slower (15 fps of 640×480 image) than our previous design. The limiting factor of the processing speed as well as the accuracy is the PLB bus bandwidth usage. This limiting factor can be alleviated by reducing the image size or increasing the SRAM size. Processing speed could reach 60 fps if the image size is reduced to 320×240. 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.

Our future work includes improving the hardware structure so that the design uses less bus bandwidth and thereby is able to achieve a higher frame rate. Another goal is to develop navigation algorithms for obstacle avoidance and implement the whole design on a UGV. Once this goal has been achieved, we will look into the hallway or canyon navigation applications for UAV.

Acknowledgment

This work was supported in part by David and Deborah Huber.

References

[1] Y. Mizukami and K. Tadamura, "Optical flow computation on compute unified device architecture," in *Proceedings of the 14th International Conference on Image Analysis and Processing (ICIAP '07)*, pp. 179–184, Modena, Italy, September 2007.

[2] R. Strzodka and C. Garbe, "Real-time motion estimation and visualization on graphics cards," in *Proceedings of the 15th IEEE Visualization Conference (VIS '04)*, pp. 545–552, Austin, Tex, USA, October 2004.

[3] J. Chase, B. Nelson, J. Bodily, Z. Wei, and D.-J. Lee, "FPGA and GPU architectures for real-time optical flow calculations," in *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines*, Palo Alto, Calif, USA, April 2008.

[4] M. V. Correia and A. C. Campilho, "Real-time implementation of an optical flow algorithm," in *Proceedings of the 16th International Conference on Pattern Recognition (ICPR '02)*, vol. 4, pp. 247–250, Quebec City, Canada, August 2002.

[5] A. Zuloaga, J. L. Martín, and J. Ezquerra, "Hardware architectural for optical flow estimation in real time," in *Proceedings of the IEEE International Conference on Image Processing (ICIP '98)*, vol. 3, pp. 972–976, Chicago, Ill, USA, October 1998.

[6] J. L. Martín, A. Zuloaga, C. Cuadrado, J. Lázaro, and U. Bidarte, "Hardware implementation of optical flow constraint equation using FPGAs," *Computer Vision and Image Understanding*, vol. 98, no. 3, pp. 462–490, 2005.

[7] J. C. Sosa, J. A. Boluda, F. Pardo, and R. Gómez-Fabela, "Change-driven data flow image processing architecture for optical flow computation," *Journal of Real-Time Image Processing*, vol. 2, no. 4, pp. 259–270, 2007.

[8] J. Díaz, E. Ros, S. Mota, and R. Rodriguez-Gomez, "FPGA-based architecture for motion sequence extraction," *International Journal of Electronics*, vol. 94, no. 5, pp. 435–450, 2007.

[9] J. Díaz, E. Ros, F. Pelayo, E. M. Ortigosa, and S. Mota, "FPGA-based real-time optical-flow system," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 16, no. 2, pp. 274–279, 2006.

[10] P. C. Arribas and F. M. H. Maciá, "FPGA implementation of camus correlation optical flow algorithm for real time images," in *Proceedings of the 14th International Conference on Vision Interface (VI '01)*, pp. 32–38, Ottawa, Canada, June 2001.

[11] H. Niitsuma and T. Maruyama, "High speed computation of the optical flow," in *Proceedings of the 13th International Conference on Image Analysis and Processing (ICIAP '05)*, vol. 3617 of *Lecture Notes in Computer Science*, pp. 287–295, Cagliari, Italy, September 2005.

[12] Z. Wei, D.-J. Lee, B. Nelson, and M. Martineau, "A fast and accurate tensor-based optical flow algorithm implemented in FPGA," in *Proceedings of the IEEE Workshop on Applications of Computer Vision (WACV '07)*, p. 18, Austin, Tex, USA, February 2007.

[13] Z. Wei, D.-J. Lee, and B. E. Nelson, "FPGA-based real-time optical flow algorithm design and implementation," *Journal of Multimedia*, vol. 2, no. 5, pp. 38–45, 2007.

[14] B. K. P. Horn and B. G. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, no. 1–3, pp. 185–203, 1981.

[15] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proceedings of the DARPA Image Understanding Workshop*, pp. 121–130, Washington, DC, USA, April 1981.

[16] G. Farnebäck, "Very high accuracy velocity estimation using orientation tensors, parametric motion, and simultaneous segmentation of the motion field," in *Proceedings of the 8th IEEE International Conference on Computer Vision (ICCV '01)*, vol. 1, pp. 171–177, Vancouver, Canada, July 2001.

[17] G. Farnebäck, "Fast and accurate motion estimation using orientation tensors andparametric motion models," in *Proceedings of the 15th IEEE International Conference on Pattern Recognition (ICPR '00)*, vol. 1, pp. 135–139, Barcelona, Spain, September 2000.

[18] B. Jähne, H. Haussecker, H. Scharr, H. Spies, D. Schmundt, and U. Schur, "Study of dynamical processes with tensor-based spatiotemporal image processing techniques," in *Proceedings of the 5th European Conference on Computer Vision (ECCV '98)*, vol. 2, pp. 322–336, Freiburg, Germany, June 1998.

[19] G. Farnebäck, "Orientation estimation based on weighted projection onto quadratic polynomials," in *Proceedings of the Conference on Vision, Modeling, and Visualization*, pp. 89–96, Saarbrücken, Germany, November 2000.

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

[21] H. Haussecker and H. Spies, *Handbook of Computer Vision and Application. Vol 2*, chapter 13, Academic Press, New York, NY, USA, 1999.

[22] Robotic Vision Lab, Brigham Young University, http://www.ee.byu.edu/roboticvision/helios.

[23] W. S. Fife and J. K. Archibald, "Reconfigurable on-board vision processing for small autonomous vehicles," *EURASIP Journal of Embedded Systems*, vol. 2007, Article ID 80141, 14 pages, 2007.