Change-Driven Image Processing Architecture with Adaptive Threshold for Optical-Flow Computation

Julio C. Sosa, Rocío Gómez-Fabela, José A. Boluda and Fernando Pardo
Departamento de Informática, Escuela Técnica Superior de Ingeniería
Universidad de Valencia
Av. Vicente Andres Estelles s/n, 46.100 Burjassot, Valencia, Spain
email: [jucesosa, rogodefa]@alumni.uv.es, [Jose.A.Boluda, Fernando.Pardo]@uv.es

Abstract

Optical flow computation has been extensively used for object motion estimation in image sequences. However, the results obtained by most optical flow techniques are as accurate as computationally intensive due to the large amount of data involved. A new strategy for image sequence processing has been developed; pixels of the image sequence that significantly change fire the execution of the operations related to the image processing algorithm. The data reduction achieved with this strategy allows a significant optical flow computation speed-up. Furthermore, FPGAs allow the implementation of a custom data-flow architecture specially suited for this strategy. The foundations of the change-driven image processing are presented, as well as the hardware custom implementation in an EP20K1000C FPGA showing the achieved performance.

1. Introduction

Apparent object movement estimation from a sequence of images is a fundamental task in the field of computer vision. However, motion estimation is a high processing power demanding application that may become a critical bottleneck when real-time constrains are required.

The optical-flow computation consists on the estimation of the apparent 2D movement field in the image sequence. In this way, each pixel has an associated velocity vector. This technique can be combined with several segmentation techniques in order to improve its accuracy or implement object tracking. There are many strategies for optical-flow computation in the literature [1]. Among these methods, the gradient-based and the correlation-based approaches are the two most widely used techniques.

Most of the recent advances in optical-flow computation have focused their improvements on achieving a high accuracy. Examples of these research fields can be found in [2],[3],[4],[5]. Moreover, there are few works that pay attention to the computation speed aspects, like the achievement of a faster speed with real time constrains. Additionally, novel theoretical analysis of motion and optical flow estimation encourage the use of custom electronic devices [6].

The classical approach for image sequence analysis usually involves full image processing. In the optical flow computation the spatial and temporal derivatives are calculated for all pixels on all images, despite the fact that images could have suffered minor changes from one frame to the next.

It is possible to reduce the processing time realizing that images usually change little from frame to frame, especially if the acquisition time is short.

This paper presents a new architecture for speeding-up the optical-flow computation. This method is based on pixel change instead of full image processing and it shows a good speed-up. The system has been developed for being implemented on an Altera development board, which includes an EP20K100 FPGA and 32 Mbytes of SDRAM. The Overall system was designed in VHDL and it has been simulated with Modelsim.

2. Theoretic Analisys

In this section, the original problem for optical-flow computation according to Horn & Schunck's [7] algorithm is reviewed, and the novel proposed technique for optical flow computation is introduced.

2.1 Optical Flow Computation

The brightness of a pixel at point (x,y) in an image plane at time t is denoted by $E(x,y,t)$. Let $E(x,y,t)$ and $E(x,y,t+1)$ be two successive images of a video sequence.

The basic assumptions underlying all motion estimation methods are: the sampled pixel brightness remains constant in time and all apparent variations of the brightness, throughout a video sequence, are due to spatial displacement of the pixels, which again are caused by motion of objects in the video sequence. The brightness conservation assumption is described as:

$$E_{of} = E_x u + E_y v + E_t = 0, \quad (1)$$

This equation is called the optical flow constraint, where E_x, E_y and E_t are the bright changes of a point in the image for horizontal, vertical and temporal directions. The unknowns u and v, denote the horizontal and the vertical component of the motion vector at each pixel position.

Equation (1) has two unknowns (u,v), therefore a second constraint must be proposed in order to solve this ill-posed problem. The smoothness constrain can be formulated for minimizing the sum of squares of the Laplacian of the motion vector field u, v.

$$\nabla^2 u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}, \quad (2a)$$

$$\nabla^2 v = \frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}, \quad (2b)$$

Where the average of the values are defined as:

$$\hat{u}_{i,j,k} = \frac{1}{6}\left\{u_{i-1,j,k} + u_{i,j+1,k} + u_{i+1,j,k} + u_{i,j-1,k}\right\}$$

$$+ \frac{1}{12}\left\{u_{i-1,j-1,k} + u_{i-1,j+1,k} + u_{i+1,j+1,k} + u_{i+1,j-1,k}\right\}, \quad (3a)$$

$$\hat{v}_{i,j,k} = \frac{1}{6}\left\{v_{i-1,j,k} + v_{i,j+1,k} + v_{i+1,j,k} + v_{i,j-1,k}\right\}$$

$$+ \frac{1}{12}\left\{v_{i-1,j-1,k} + v_{i-1,j+1,k} + v_{i+1,j+1,k} + v_{i+1,j-1,k}\right\}, \quad (3b)$$

Now there are two equations for each point in the image.

$$u = \hat{u} - \frac{E_x(E_x\hat{u} + E_y\hat{v} + E_t)}{(\alpha^2 + {E_x}^2 + {E_y}^2)} = \hat{u} - E_x\frac{Num}{Den}, \quad (4a)$$

$$v = \hat{v} - \frac{E_y(E_x\hat{u} + E_y\hat{v} + E_t)}{(\alpha^2 + {E_x}^2 + {E_y}^2)} = \hat{v} - E_y\frac{Num}{Den}, \quad (4b)$$

where $\hat{u}$, $\hat{v}$ are the Laplacian of velocities u, v and α is an adjust parameter. These equations have an iterative form because u and v are function of $\hat{u}$ and $\hat{v}$.

2.2. Implementation Considerations

It is necessary to obtain the spatial and temporal gradient values for computing optical-flow. To do this, at least two consecutive images of the sequence are required. In the other hand, equations (4a) and (4b) show an iterative dependence, thus the result depends of the previous calculation. The iterative process is repeated a fixed number of times. When the iterative process is concluded, the result is presented to the system output. Afterward, a new pair of images is processed.

A custom architecture of this type was proposed by Arribas & Monasterio [8]. They processed 50x50 pixel, images at 19 fps using only three iteration cycles per each pair of images. Initially, the default result is assumed as a zero value for all optic flow field vectors.

In a recent work, Martin et al. [6] show a new optical flow computing technique. In this work, the iterations are made between a group of successive images in an image sequence, not only between a pair of two consecutive images. In other words, a single iteration per image pair is performed and the results are used as input for a second iteration, but now with a new pair of images. This idea is based on the assumption that changes between two consecutive images are negligible and the obtained result are useful for an iterative process. Martin et al. present results after processing 64 images, but with only 1 iteration per each pair of images.

This paper uses the same principles of the two works described before in [6] and [8]: image changes between two consecutive images are minor and the image intensity changes appear due to a local pixel movement. In this way, if there is not any intensity change in any pixel of a consecutive images pair then there is no movement, so these images will not be processed. This is the reason that justifies the existence in this architecture of an element that locates only the pixels that changed. Additionally, a memory must be used for storing the pixels that changed between two consecutives images. In this way, the iteration cycles are made with the memorized pixels and thus full image processing is avoided. The temporal redundancy at the images sequence is reduced using this strategy.

This condition can be expressed as follows: when the pixel intensity level difference between two consecutive images is less than a threshold, then this pixel will not be processed. This restriction can be written with the equation:

$$ch = \begin{cases} 1 & mag > th \\ 0 & mag \leq th, \end{cases} \quad (5)$$

where *mag* is the gray level difference, *th* is the threshold, an integer value always small. Detected change *ch* will be 1 if there is a change in the pixel or 0 if not. With small values for the threshold many pixels will be detected as moving pixels and there will not be a greater time reduction. In the other hand, with upper values for the threshold a major speed-up will be achieved, but with a loss of accuracy.

2.3. Change-Driven Image Processing

The change-driven policy has been applied before to another image processing algorithm [9]. The change-driven image processing theoretical speed-up was estimated by software before the hardware implementation. The formal optical flow and change-driven optical flow algorithms have been implemented in C++.

The algorithm is significantly different, and subsequently, the computing resources needed (and therefore the hardware required) are reduced when change-driven processing is applied. A LUT is necessary to compare two consecutive images to detect which pixels have changed on the images. Only pixels that have changed (above threshold) fire the corresponding processing instructions.

The number of clock cycles needed to implement the original optical flow algorithm can be expressed as:

$$cycles = 3\,(M \times N) + 2K\,(M \times N), \quad (6)$$

where $M \times N$ is the image size and K is the number of iterations. The optical flow modified algorithm computation cost of is represented as:

$$cycles = (M \times N) + 3\,(M \times N)\,\rho + 2K\rho\,(M \times N), \quad (6a)$$

where ρ is defined by:

$$\rho = \frac{1\text{- } No.\ pixels\ that\ not\ change}{(M \times N)} \quad (7)$$

There will be few pixels that change if the threshold is high. The experiments performed, with constant brightness, recommend a threshold less than five for obtaining useful results in the optical flow computation. Moreover, if the changes between two consecutive images are small (that's true especially if the acquisition time is short and there are not many moving objects) there will be a very small percentage of pixels to be processed.

3. Threshold Selection for change-detection

The threshold selection is a very important step. The focus in this section is to determine what threshold value is the best. In [10] it has been presented an example (a reduced version) of the change-driven image processing policy applied to the optical flow computation. It was shown that the system effectiveness depends on the percentage of static pixels whose variation intensity is below the threshold. In real scenes captured by camera, there will be inevitable noise that randomly can change pixel intensity values. As a result, this noise will give a number of false positives that will degrade system accuracy and performance. However, adapting the threshold dynamically significantly reduces the problem.

It can be concluded that the threshold is a critical value in change-driven processing algorithms. Too low value may include spurious changes, while a too high value will erase significant scene changes. There are many thresholding algorithms published in the literature [11], [12]. Nevertheless, selection of an appropriate algorithm is not an easy problem since each algorithm makes different assumptions about the image content or environment dependencies.

Our approach takes the gray level average of a pixel set. The pixel set must be distributed uniformly in each image. The gray level average is computed by:

$$\hat{a} = \sum_{i}^{m} \sum_{j}^{n} \frac{p(i,j)}{m \times n}, \quad (8)$$

where $m \times n$ denote the grid dimension, i and j are the pixel coordinates (they are not consecutive pixels, i,j =1k, 2k, … k≥1, if k=1 all the image is averaged). Therefore, the difference between two consecutive averages (of two consecutive images), will indicate the appropriate threshold.

The principle is simple. If there are not changes between two consecutive images, that is, the brightness is constant and there is not movement, then the gray level average must be the same for each image.

However, if there is not movement but there is illumination or brightness changes, as it is shown at Fig. 1, then the gray average is different between two consecutive images. This difference will determine the threshold value utilized to implement the change-driven image processing. It is necessary the experimentation with different image sequences in order to determine the total amount of pixels needed.

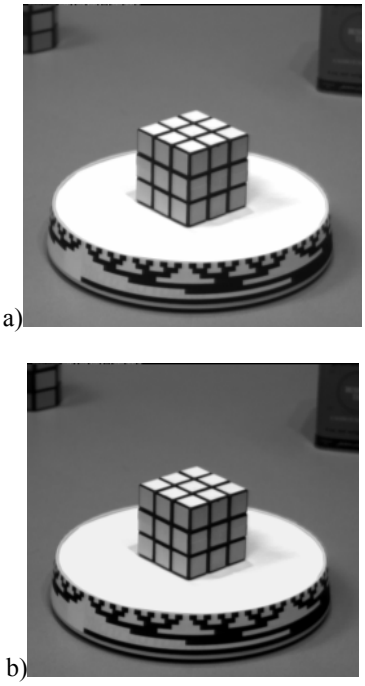

Figure 1. There is illumination variation but not movement.

4. Architecture

The iterative form for optical flow computation can be implemented by software on a general purpose microprocessor. However, in order to process image sequences in real time it is interesting to use Programmable Devices, ASIC, analog integrated circuit VLSI [13], cluster of processors [14] or special processors [15], though these last solutions are not practical or they are very expensive. FPGAs have been chosen to allow a cheaper an easy prototype development.

The system design was divided into three parts. First the processing modules included into FPGA were developed in VHDL. Afterwards the system was simulated including the development board SDRAM modules [16]. Finally, a full simulation was performed including the pci_mt32 MegaCore function. The Quartus II software and ModelSim-Altera tools were used.

The FPGA architecture has been divided into four modules: gradient/LUT module, velocities module, Laplacian module and iteration-fifo control module.

The optical flow algorithm requires a large FIFO memory that can be implemented in the FPGA. The EP20K1000C has up to 327.680 RAM bits. A block

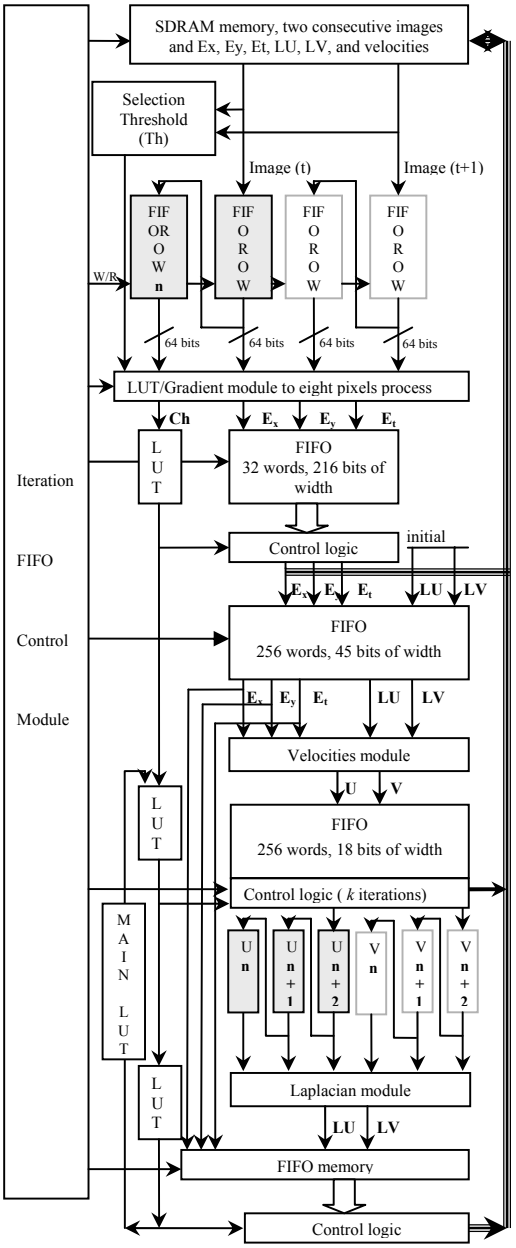

Figure 2. Architecture for optical-flow computation (u,v) using the iterative equation set and a LUT.

schema of the proposed hardware architecture is shown in Fig. 2.

It must be noted that the calculations made with the hardware modules are implemented using integer arithmetic, since it requires less resources than a floating point approach. The divisions have been normalized to the power of two and implemented as a right shift. The idea is to optimize the use of resources and to increase the speed of the system.

4.1. Threshold selection Module

The threshold selection module computes the average gray level of a pixel set. The pixel amount is a power of two. Thus it is not necessary implementing a division, which is a computational expensive operation. The pixel set was fixed as a grid of 8 × 8 pixels distributed uniformly along the image. That allocation gives a total of 64 pixels per image. This module has a very simple architecture. One fast-carry adder and one right shifter have been employed.

4.2. LUT/Gradient Module

The gradient module computes the horizontal, vertical and temporal image gradients. Several pixels from current and previous images are necessary. Only four rows, two of each image, are required initially. Those rows are kept in FIFO memories. Subsequently, a new row of each image must be introduced (I1 row n+1 and I2 row n+1) for finishing all image rows.

Eight pixels are processed in parallel at the LUT/Gradient module being the LUT computed for the next stage. The operations are implemented with additions, subtractions and two right shifters for the division by 4. The result, expressed in 2's complement, is obtained in one clock cycle.

Gradient module outputs are stored in the FIFO memories. Only those pixels that have suffered a change in its gray value will be taken into account. These pixels will be stored at the SDRAM memory until they are processed at the velocities module.

4.3. Velocities Module

It is necessary to divide the process into stages for implementing the equations, (4a) and (4b) in a pipeline. That operation is broken into five smaller operations. The latency in this module is 8 clock cycles, at 33 Mhz.

A VHDL standard multiplying function has been used with minor modification for calculating these terms. First *Num* and *Den* are calculated. In addition, four temporal registers for E_x, E_y, $\hat{u}$ and $\hat{v}$ terms are necessary. Although the *Num/Den* term is common for both equations, it is necessary to note that all calculations are made with integer arithmetic. Initially the products between ($Num \cdot E_x$) and ($Num \cdot E_y$) are implemented. Three temporal registers for *Den*, $\hat{u}$ and $\hat{v}$ terms are again necessary. The divisions with *Den* as divider are performed when the products are obtained. Finally with a last subtraction the process is accomplished and the velocities ***u*** and ***v*** are obtained.

A VHDL standard multiplying function has been used with minor modification for calculating these terms. First *Num* and *Den* are calculated. In addition, four temporal registers for E_x, E_y, $\hat{u}$ and $\hat{v}$ terms are necessary. Although the *Num/Den* term is common for both equations, it is necessary note that all calculations are made with integer arithmetic. Initially the products between ($Num * E_x$) and ($Num * E_y$) are implemented. Three temporal registers for *Den*, $\hat{u}$ and $\hat{v}$ terms are again necessaries. The divisions with *Den* as divider are performed when the products are obtained. Finally with a last subtraction the process is accomplished and the velocities ***u*** and ***v*** are obtained.

4.4. Laplacian Module

Once all LUT pixels have been processed, but with the number of iterations no completed, a feedback loop of the Laplacian convolution to the velocity module result is applied. The convolution smoothes the optical flow computed. Nine previous pixels of three neighbour rows are needed for performing this action. In fact equations (3a) and (3b) are implemented at this module. There are two divisions in these equations that are not powers of two (6 and 12). In order to simplify the logic required and to speed-up the computation time, the divisions are implemented in two stages. First an integer division by 3 is implemented an afterwards a right shifting (of one or two bits) is performed. Outputs from the Laplacian module are stored in a FIFO memory and afterwards in SDRAM memory.

This process is repeated a fixed number of times. When the iterative process concludes, the result is presented at the system output.

4.5. Iteration-FIFO Control Module

This module activates the memory input sequence and regulates the number of iteration cycles. Additionally, it addresses the pixel to process and the place where it is stored. It is necessary to take care of two different clocks, one for write/read to/from the SDRAM memory at 100 Mhz, and the other at 33 Mhz used for the modules.

5. Results

Several tests have been performed for validating the design. Different tools were used depending on the part of the implemented system being considered.

First the processing modules included into FPGA were tested on Quartus II software. Afterwards the system was simulated including the SDRAM modules

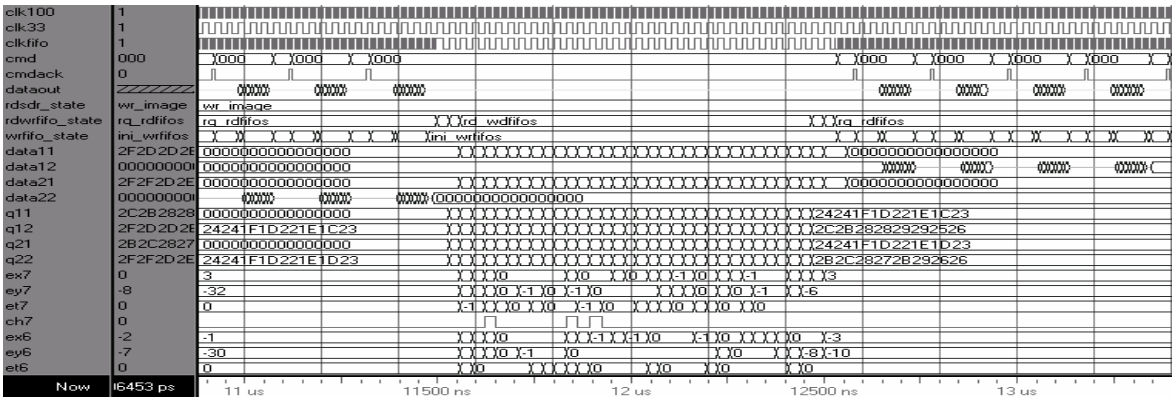

Figure 3. System simulation on ModelSim-Altera.

and the pci_mt32 MegaCore function. In this stage synthetic images were used as well as real image sequences.

5.1. Modules and Overall System Simulation

Several system module simulations are presented at this section. Specifically the timing analysis of the velocities module and Laplacian module. Quartus II has been used and the EP20K1000CF672C7 has been selected.

A latency of 240 ns between data input E_x, E_y, E_t, LU, LV, and the result output U and V is shown. It is necessary to remark that the output U and V must be scaled to the final velocities by a constant factor, power of two.

The LU and LV require a register for a pipeline implementation. Those elements are necessary in the last stage of the velocities module. The values were stored on LU8 and LV8 for the final process.

Quartus II software can not simulate the operations of a system which includes external devices, such as, SDRAM memory. Therefore ModelSim-Altera has been used to test all the system. The simulation has been performed for verifying that the circuit works correctly as shown in Fig. 3.

It is necessary to create a top-level testbench file that instantiates the PCI testbench elements and the IP functional simulation model of the PCI MegaCore functions, connecting all the signals.

The master transactor simulates the master behavior on the PCI bus. It serves as an initiator of PCI transactions. The master transactor has three sections: procedures, initialization and user commands. The bus monitor displays PCI transactions and information messages in the simulator's console window when an event occurs on the PCI bus. The arbiter simulates the PCI bus arbiter. The pull_up simulates the PCI signals pull up functionality, such as the AD, cben, frame, and other signals. The clock_gen generates the PCI clock for the testbench. The pci_top module is a MegaCore function that complies with the requirements specified in the PCI SIG. The top_o_flow is the module that includes all the system to optical flow computation. The sdram_model is a simulation model of the SDRAM MT48LC4M16A2TG-7E.

5.2. Result Validation

At this section the optical flow computation performed by the hardware system is compared to a floating-point software implementation. Additionally, a study was made for analyzing the accuracy of the model proposed with respect to the original Horn & Schunck model (without the change-driven policy) always using 10 iterations.

The algorithm and images sequences are available from (ftp://ftp.csd.uwo.ca/pub/v3ision/) and from http://www.cs.otago.ac.nz/research/vision.

The comparison between the floating point and the integer arithmetic algorithm used in this paper shows that the same average relative error in both cases is obtained. As an example, Yosemite sequence has 0,9º of average relative error and the diverging tree has 1,71º. The worst case found with these image test benches is the sinusoid sequence that has 9,74º of average relative error.

Fig. 4 shows the optical flow computation when there is no movement but there are brightness changes. Fig. 4a) has a constant threshold (th=2), and Fig. 4b) has a variable threshold which takes into account illumination changes. Fig. 5 shows the same experiment but with critical illumination and brightness changes conditions.

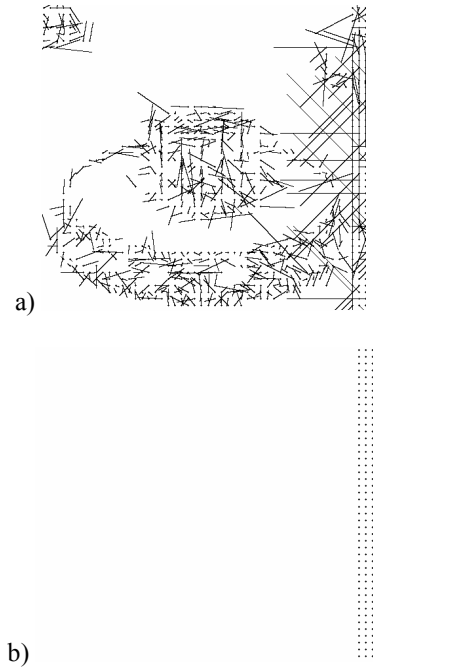

Figure 4. Optical flow of rubic_data sequence without movement. a) Change driven processing (th=2), b) Change driven processing *th* adaptive. (10 iterations).

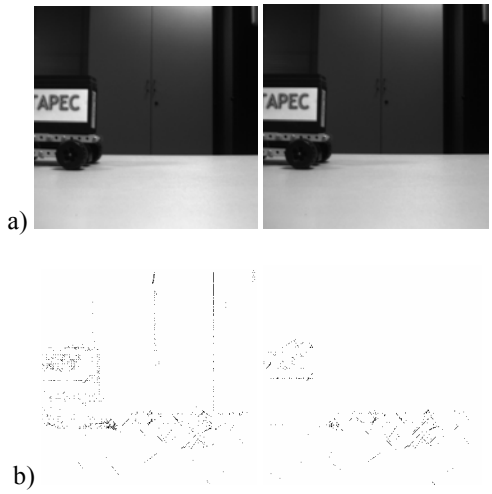

Figure 5. Optical flow of tapec_data sequence. 5a) There are movements in the two consecutives images. 5b) (left to right) Change driven processing (th=2), Change driven processing *th* adaptative. (10 iterations).

Fig. 6 shows that there are few differences between the obtained result of a change-driven image processing and the original algorithm processing at the rubic_data sequence. Moreover, Fig. 7 shows that

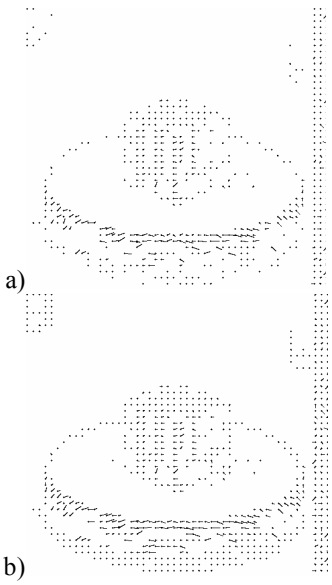

Figure 6. Optical flow of rubic_data sequence. a) Change driven processing (th=1), b) Original algorithm. (10 iterations).

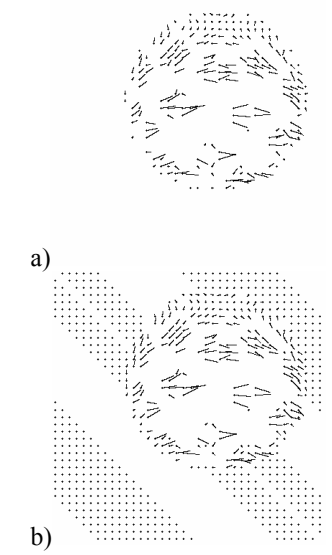

Figure 7. Optical flow of sphere sequence. a)Change driven processing (th=1), b)Original algorithm (10 iterations).

change-driven optical flow implementation has some noise immunity.

The most important result is that the processing time is dramatically reduced. Fig. 8 shows the speed-up comparison between the original model (100% temporal cost) and the proposed model for different

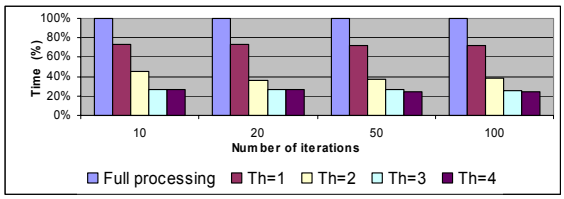

Figure 8. Speed up comparison.

number of firing thresholds and iterations. The sequence used was a rubic_data.

6. Conclusions

The change-driven image processing strategy presented at this work allows the implementation of a new architecture for speeding-up the optical-flow computation. This method is based on pixel change instead of full image processing and it shows a good speed-up. It has been tested using classical optical flow test sequences and full-custom sequences. The average error is similar to other full-processing implementations, but this algorithm is faster. The number of fps that can be processed depends of the image changes. In this way, it is not possible to predict a fixed number of images processed. The results obtained with the optical flow image test bench have determined that the system is able to process images of 256x256 pixels at 30 fps.

7. Acknowledgments

This work has been supported by the project UV-AE-20060242 of the University of Valencia. Julio C. Sosa is a scholarships student COFAA-IPN.

8. References

[1] J. L. Barron, D. J. Fleet, and S. S. Beauchemin, "Performance of optical flow techniques" *International Journal of Computer Vision*, vol. 12 no. 1, pp. 43-77, 1994.

[2] Yeon-Ho Kim, A. M. Martinez, and A.C. Kak, "Robust motion estimation under varying illumination" *Image and Vision Computing*, vol. 23, pp. 365-375, 2005.

[3] C. H. Teng, S. H. Lai, and Y. S. Chen, "Accurate optical flow computation under non-uniform brightness varations" *Computer Vision and Image Understanding*, vol. 97, 315-346, 2005.

[4] T. Brox, A. Bruhn, N. Papenberg, and J. Weickert, "High Accuracy Optical Flow Estimation Based on a Theory for Warping" *Proc. 8th European Conference on Computer Vision, Springer LNCS 3024,* vol. 4, pp. 25–36, May. 2004.

[5] S. H. Lim, J. G. Apostolopoulos, and A. E. Gamal, "Optical Flow Estimation Using Temporally Oversampled Video" *IEEE Transactions on Image Processing*, vol. 14, no. 7, pp. 890–903, July 2005.

[6] J. L. Martin, A. Zuloaga, C. Cuadrado, J. Lazrao, and U. Bidarte, "Hardware implementation of optical flow constraint equation using FPGAs" *Computer Vision and Image Understanding*, vol. 98, pp. 462-490, 2005.

[7] B. K. P. Horn and B. G. Schunck, "Determining Optical Flow" *Artificial Intelligence*, vol. 17, pp. 185–203, 1981.

[8] P. C. Arribas and F. Monasterio, "FPGA implementation of the Horn&Schunck Optical Flow Algorithm for Motion detection in real time Images" *Proceeding XIII Design of circuits and integrated systems conference*, pp. 616-621, 1998.

[9] F. Pardo, J.A. Boluda, X. Benavent, J. Domingo, and J. C. Sosa. "Circle detection and tracking speed-up based on change-driven image processing". ICGST International Conference on Graphics, Vision and Image Processing. GVIP'05. pp. 131-136. Cairo, Egypt, December 2005.

[10] Julio C. Sosa, R. Goméz, J. A. Boluda, and F. Pardo, "FPGA Implementation of a Change-driven Image Architecture for Optical Flow Computation", 16th International Conference on Field Programmable Logic and Aplications, FPL06. Accepted. To be presented in august 2006.

[11] Rosin, Paul L., and Ioannidis E., " Evaluation of global image thresholding for change detection" Pattern Recognition Letters. Vol. 24, p.p. 2345-2356, January 2003.

[12] Sezing, M., and Sankur, B., "Survey over image thresholding techniques and quantitative performance evaluation" Journal of Electronic Imaging. Vol. 13, Issue 1, pp. 146-168, Jan. 2004.

[13] A. A. Stocker and R. J. Douglas, "Analog Integrated 2D Optical Flow Sensor" IEEE, *International Symposium on Circuits Ans System,* vol. 3, pp. 9-12, Vancouver Canada, May. 2004.

[14] A. G. Dopico, M. V. Correia, J. A. Santos, and L. M. Nunes, "Distributed Computation of Optical Flow" International Conference on Computational Science, pp. 380-387, 2004.

[15] M. V. Correia, and A. Campilho, " A pipelined Real-Time Optical Flow Algorithm" *Analog Integrated Circuits and Signal Processing*, Springer, vol. 46, no. 2, p.p. 121-138, Junuary 2004.

[16] Micron Technology, Inc., "SDRAM memory simulation model: MT48LC4M16A2TG-7E" http://www.micron.com.