High Speed Computation of the Optical Flow

Hiroaki Niitsuma and Tsutomu Maruyama

Systems and Information Engineering, University of Tsukuba,
1-1-1 Ten-ou-dai Tsukuba Ibarakim, 305-8573, Japan
niitsuma@darwin.esys.tsukuba.ac.jp

Abstract. In this paper, we describe a compact system for high speed computation of the optical flow. This system consists of one off-the-shelf PCI board with one Field Programmable Gate Array (FPGA) chip, and its host computer. With this system, we can generate dense vector maps at (1) 840 frames per second (fps) in small size (320×240) images, and (2) 30 fps in standard size (640×480) images by configuring different circuits on the FPGA chip. In the two circuits, vectors for all pixels in the images are obtained by the area-based matching (windows of 7×7 pixels are compared with 121 and 441 windows in the target image respectively). The circuits implemented on the FPGA do not require any special hardware resources, and can be implemented on many off-the-shelf FPGA boards shipped from many vendors. This system can also be used for the stereo vision by slightly modifying the circuits, and achieve the same performance.

1 Introduction

Compact vision systems are very important for autonomous vehicles. Field Programmable Gate Arrays (FPGAs) are ideal devices for the compact systems, because any kinds of circuits can be realized on FPGAs by just downloading configuration data to FPGAs from external memories or host computers (loading time is 10 to 100 msec in general). Depending on situations, autonomous vehicles may try to reconstruct the 3-D geometry to understand its circumstances, to find out moving objects to move safely, and to find out marker objects to check its position. FPGAs can support all these functions by reconfiguration.

In this paper, we describe a compact system for the optical flow which consists of an off-the-shelf PCI board with one FPGA chip and its host computer (to download configuration data and display the results). In our system, the most similar parts to small windows ($w \times w$ pixels) in one image are looked up in the next image taken by the same camera to obtain the optical flow. In these comparison of small windows, the SAD (Sum of Absolute Difference) algorithm is used because of its simplicity. The amount of the computation in the optical flow is, however, still very large because of the two dimensional search to find out objects moving to all directions, and high speed computation is not easy even on the latest LSIs owing to the limited memory bandwidth.

We implemented two kinds of circuits on the FPGA chip. In the first implementation, intermediate results in the computation along x and y axes are

F. Roli and S. Vitulano (Eds.): ICIAP 2005, LNCS 3617, pp. 287–295, 2005.
© Springer-Verlag Berlin Heidelberg 2005

stored on the chip and reused w times (but part of them are recalculated in order to minimize the amount of data which have to be stored on the chip) in order to achieve highest performance on small size (320×240) images. In the second implementation, intermediate results along x axis are stored, but operations along y axis are re-executed w times in order to minimize the circuit size while maintaining video-rate processing on standard size images (640×480).

This system can also be used for the stereo vision by slightly modifying the circuits, and it becomes possible to detect moving objects in images taken by moving cameras by combining the stereo vision with the optical flow.

2 The Optical Flow

In an image taken by a camera, each pixel corresponds to the intensity value obtained by the projection of an object in 3-D space onto the image plane. When the object or the camera moves, its corresponding projection also changes position in the image plane. Optical flow is a vector field that shows the direction and magnitude of these intensity changes from one image to the other. In the optical flow, the corresponding point to a given point in an image is searched in the next image taken by the same camera. Area-based (or correlation-based) algorithms match small windows centered at a given pixel to find corresponding points between the two images. They yield dense maps, but fail within occluded areas (occlusions are caused by the movement of the camera). Feature-based algorithms match local cues (e.g., edges, lines, corners) and can provide robust, but sparse maps which require interpolation. In hardware systems, area-based algorithms are widely used, because the operations required in those algorithms are very regular and simple.

The most common pixel-based matching algorithm is squared intensity differences (SSD) and absolute intensity differences (SAD). We used the SAD (Sum of Absolute Difference) algorithm because it is the simplest, and its result is almost same as other algorithms in the stereo vision[1]. In the SAD algorithm for the optical flow, ξ and η which minimize the following equation are searched.

$$SAD(x,y,\xi,\eta) = \sum_{i=-w/2}^{w/2} \sum_{j=-w/2}^{w/2} |I_0(x+i, y+j) - I_1(x+i+\xi, y+j+\eta)|$$

In this equation, I_0 and I_1 are images in $time = t$ and $time = t + \Delta t$ respectively, and $w \times w$ is the size of the window centered at a given pixel (its position is (x,y)). The range of ξ and η decides the size of area where the corresponding point to (x,y) is searched.

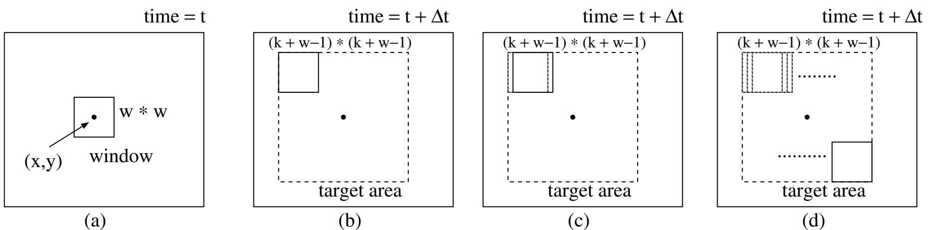

Fig. 1. Area-Based Matching in the Optical Flow

In Figure 1, a small window centered at (x, y) (Figure 1(a)) is compared with all windows in its target area centered at (x, y) (Figure 1(b)(c)(d)). When the size of the target area is $(k + w - 1) \times (k + w - 1)$, there are $k \times k$ windows in the target area, and $k \times k$ SADs (Sum of Absolute Differences) are calculated. Then, the window which gives the minimum SAD is chosen, and its center point (x', y') is considered as the corresponding point to (x, y). In this comparison, every pixel in the window in $time = t$ is compared with $k \times k$ pixels in the target area (the range of ξ and η is $-k/2$ *to* $k/2$). By this two-dimensional search, we can obtain one vector from (x, y) to (x', y').

3 Previous Works

Many approaches to reduce the computational complexity of the optical flow have been proposed[2][3], but in those algorithms, computations of areas which seem to be unnecessary for detecting moving objects are not executed, and users need to think of trade-offs between accuracy and efficiency.

In order to accelerate its performance by hardware, many systems have been proposed to date[4][5][6][7]. In those systems, in order to achieve real-time processing, sizes of images are limited or only sparse vector fields are generated. Their performances are, however, still slower than video-rate in the standard size images.

4 Computation Methods of the Optical Flow on the FPGA

Suppose that the size of the images is $X \times Y$, and N pair of images are processed in one second. Then, in order to find vectors for all pixels in the images, we have to calculate $X \times Y \times N$ vectors in one second. This means that only 108 nano seconds is allowed to find out one vector, when the image size is 640×480 and N is 30. Furthermore, $k \times k$ SADs have to be calculated for one vector, and $w \times w$ ADs (Absolute Differences) are necessary for calculating each SAD. This requirement means that we need to reuse intermediate results generated during the calculation of a window for the calculations of other windows. In the following, we first show a technique to realize the maximum performance, then we discuss another technique to compare a window with more number of windows in the target area at the video-rate.

4.1 A Technique to Realize the Maximum Performance

In order to achieve maximum performance on hardware, all operations have to be processed in parallel and in pipeline. Therefore, suppose that all operations described in this subsection are executed in parallel and in pipeline.

In Figure 2(a), suppose that we have calculated $k \times k$ SADs ($k \times k \times w \times w$ ADs (Absolute Differences) have been calculated) and chosen the minimum of them to obtain one vector (computations of only two SADs are shown to simplify the figure). During these computations, no operations on same data are executed.

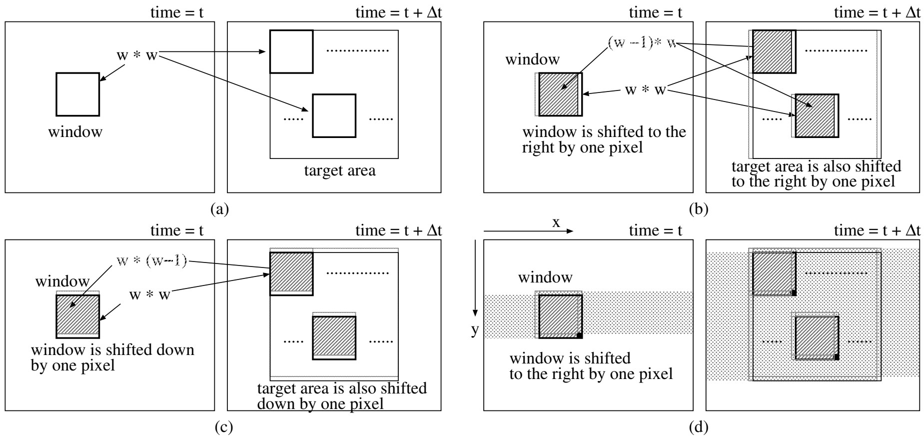

Fig. 2. Reuse of the Intermediate Results

Then, the window is shifted to the right by one pixel to obtain next vector (Figure 2(b)). At this point of time, pixels in a rectangle with slanting lines in the shifted window ($time = t$) are already compared with pixels in rectangles with slanting lines in its target area ($time = t + \Delta t$) during the computation of the previous vector. Therefore, by storing $k \times k \times (w-1) \times w$ ADs (Absolute Differences) calculated in Figure 2(a), the number of new ADs to obtain the new vector can be reduced to $k \times k \times w$. When the window is shifted down by one pixel as shown in Figure 2(c), pixels in a rectangle with slanting lines in the shifted window are already compared with pixels in rectangles with slanting lines in its target area. In this case, we can also reduce the number of AD operations to $k \times k \times w$ by storing and reusing the $k \times k \times w \times (w-1)$ ADs.

In Figure 2(d), suppose that the image size is $X \times Y$ and the window is shifted to the right (along x axis) first, and when the window reaches to the right-end of the image, the window is moved to the left-end again and shifted down by one pixel. In this case, when the window is shifted to the right by one pixel, ADs for $w \times w - 1$ pixels in the shifted window (all pixels in the window except for one pixel shown by a black dot) are already calculated ($k \times k \times (w \times w - 1)$ ADs are already calculated) during the computation of previous vectors. Therefore, by storing $k \times k \times (w-1) \times X$ ADs (which correspond to the gray area in Figure 2(d)), we can calculate $k \times k$ SADs which are necessary to obtain a new vector by only calculating $k \times k$ new ADs (ADs between the pixel shown by the black dot and $k \times k$ pixels in the target area). In this computation, we need to access $k \times k \times (w \times w - 1)$ ADs (which are already calculated and stored) in parallel in order to achieve maximum performance.

Figure 3 shows an implementation technique to make the parallel access possible (the upper half of the figure shows the two images which are compared, and the lower half of the figure shows an array of SAD units, and the inside of a SAD unit). In Figure 3, suppose that the vector for a pixel (light gray square in the window) was just obtained, and the window is shifted to the right to find the

vector for the next pixel (dark gray square). Then, the window is compared with $k \times k$ windows in its target area ($k \times k$ SADs are calculated), and the minimum SAD is searched. In order to achieve maximum performance, $k \times k$ SAD units are prepared and the $k \times k$ SADs are calculated in parallel (In Figure 3, only two units are shown to simplify the figure). In Figure 3, $A_{i,j}$ are ADs (Absolute Differences) which are already calculated during the computation of previous vectors. In Figure 3, a new SAD is calculated using $A_{i,j}$ as follows.

1. w-1 ADs ($A_{i,6}(i = 2, 5)$ (squares with sparse slanting lines)) are read out from memory M_A.
2. A new AD for the black square (which becomes $A_{6,6}$) is calculated (pixel data of the black square (I_4) is broadcasted to all SAD Units on the array).
3. These w ADs are held on w shift registers in the SAD unit. Each shift register can hold w ADs (w is 5 in Figure 3). Thus, $w \times w$ ADs are on the shift registers in total. The ADs on the shift registers are shifted when a new SAD (consequently a new vector) is obtained.
4. These $w \times w$ ADs on the shift registers are summed up to calculate a new SAD.
5. Among w ADs which are shifted out from the shift registers, w-1 ADs are written back to M_A ($A_{i,1}(i = 3, 6)$ (squares with dense slanting lines)). Thus, each AD is summed up w times while it is on the shift register, and is stored and read out from the memory w-1 times, which means each AD is used for calculating $w \times w$ SADs.

By repeating the procedure above with $k \times k$ SAD units which run in parallel and in pipeline, we can continue to obtain a new vector in every clock cycle.

In this implementation, the width of memory M_A must be w-1 words. Therefore, the total number of memory banks required in this implementation becomes $k \times k \times (w - 1)$, and these memory banks must be accessed in parallel. This means that these memory banks have to be located on the FPGA (because the input/output performance of LSIs (including FPGAs) is very limited). However, the number and width of internal memory banks of the latest LSIs are not enough under the practical w and k.

In order to reduce the number of memory banks, the procedure described above is modified as follows.

1. Only the sum of the w-1 ADs is stored in the memory (suppose that $\sum_{i=2}^{5} A_{i,6}$ is in the memory).
2. The sum is read out, and added with $A_{6,6}$ ($\sum_{i=2}^{6} A_{i,6}$ can be obtained).
3. w sums on a shift register are summed up to calculate new SAD ($\sum_{i=2}^{6} \sum_{j=2}^{6} A_{i,j}$).
4. At the same time, $A_{2,6}$ is calculated again, and subtracted from $\sum_{i=2}^{6} A_{i,6}$.
5. The result ($\sum_{i=3}^{6} A_{i,6}$) is stored in the memory to obtain vectors on the next row.

With this technique, we can reduce the number of memory banks to $k \times k$ from $k \times k \times (w - 1)$, though we need double SAD units. The total hardware resources required by this technique are $k \times k \times 2$ SAD units, $k \times k$ memory banks and a unit to choose the minimum among $k \times k$ SADs in pipeline.

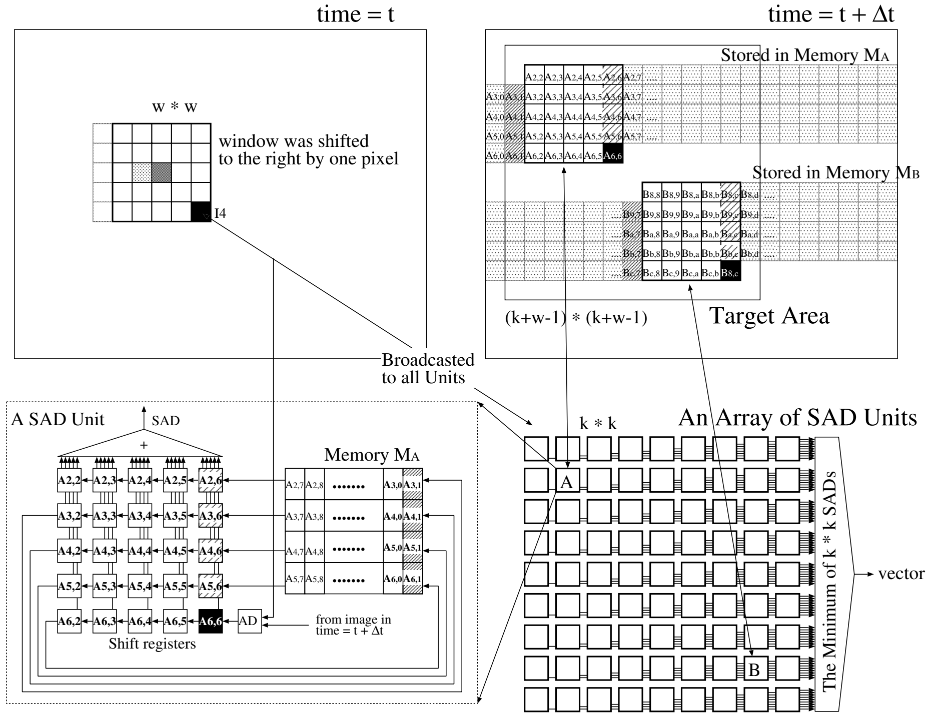

Fig. 3. An Implementation Technique to Achieve Maximum Performance

4.2 Video-Rate Processing

We can reduce the circuit size by recalculating some of ADs instead of storing all of them. This approach makes it possible to enlarge the size of the target area.

By recalculating sums of ADs which were given by the memory banks ($\sum_{i=2}^{5} A_{i,6}$ in Figure 3), we can calculate SADs without memory banks. In Figure 4, I_0 is broadcasted to $k \times k$ SAD units first, and $k \times k$ ADs for I_0 ($|A_{2,6} - I_0|$ in Figure 3) are calculated in the $k \times k$ SAD units in parallel. In the same way, ADs for $I_j (j = 1, 4)$ are calculated sequentially. These calculations take w clock cycles in total. These ADs are, then, summed up, and held on the shift registers. The sums held on the shift registers are used w times to calculate w SADs and discarded after shifted w times.

Though this implementation requires w clock cycles to generate one vector, we can calculate vectors with $k \times k$ SAD units, no memory banks and a unit to choose the minimum among $k \times k$ SADs in pipeline. Furthermore, the size of the unit to choose the minimum SAD can be reduced to almost $1/w$, because many parts of the unit can be shared by w SAD units (each SAD unit generates one SAD in every w clock cycles).

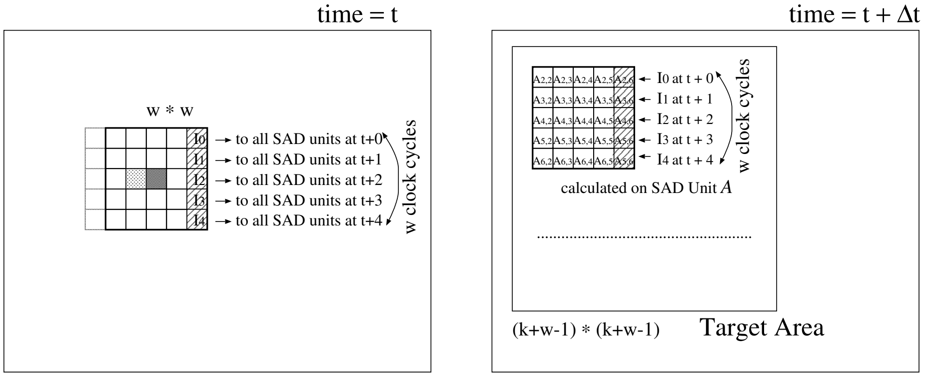

Fig. 4. An Implementation Method by Recalculation

The requirement for the video-rate processing is to obtain one vector in 108 nano seconds. Therefore, if we can build a circuit which runs faster than $108/w$ nano seconds, we can realize video-rate processing by this implementation method. The typical w used for the area-based matching is 7. Therefore, our goal is to build a circuit which runs faster than 65MHz.

5 Performance

Two kinds of circuits were implemented on PCI board (ADM-XRC-II by Alpha Data [9]) with one FPGA (Xilinx XC2V6000 [10]). Both circuits run at 66 MHz. Table 1 shows the hardware usage and the performance of the two circuits. The maximum performance of our camera is 30 frames per second. Therefore, we could not demonstrate higher frame rates than that on our system, but we confirmed that our circuits can process a pair of images at the speeds which are shown in Table 1.

The value of k in the circuit-1 (the circuit for the maximum performance) is much smaller than the circuit-2 (the video-rate circuit), because $k \times k \times 2$ SAD units are required in the circuit-1, and any part of the unit for choosing the minimum SAD can not be shared. 7 memory banks in the circuit-1 and 16 memory banks in the circuit-2 are used as cache memory to store a part of images in order to reduce data access to external memory banks on the FPGA board.

The computation time of the both circuits is promotional to the image size. The performance of the circuit-2 (the video-rate circuit) decreases proportional to w because same operations are executed w times in this implementation technique, while the performance of the circuit-1 (the circuit for the maximum performance) is constant. The size of the both circuits is almost proportional to $k \times k$, and w.

Figure 5 shows an example of the output by the circuit-2 (only a part of vectors are shown in the figure, because the image size is 640×480 and dense

Table 1. Performance of the Circuits

	k	usage of hardware resources		frame per second	
		memory banks	logic blocks	640 × 480	320 × 240
circuit-1 (max. performance)	11	128	71%	210	840
circuit-2 (video-rate)	21	16	84%	30	120

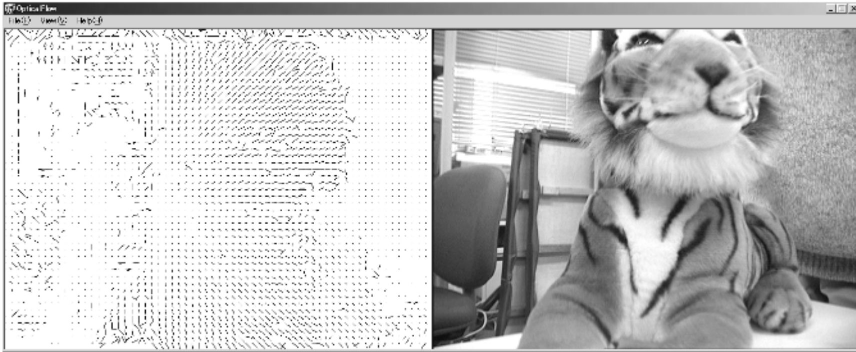

Fig. 5. The Output by the Circuit

vector map is generated by the circuit). In Figure 5, a stuffed toy (tiger) is moving to the right, but some noises are found on areas with only small changes in the contexts. We need to add some circuits to suppress these noises.

6 Conclusions

In this paper, we described a compact system for high speed computation of the optical flow. The system was implemented on an off-the-shelf PCI board with one FPGA. With this system, we could generate dense vector maps at (1) 840 frames per second (fps) in small size (320 × 240) images, and (2) 30 fps in standard size (640 × 480) images by configuring different circuits on the FPGA chip. In the two circuits, vectors for all pixels in the images are obtained by the area-based matching (windows of 7 × 7 pixels are compared with 121 and 441 windows in the target image respectively).

We are now improving the system to work with (1) the stereo vision to detect moving objects in the images taken by moving cameras, and (2) edge detections to clearly distinguish borders of the moving objects.

References

1. T.Kanade, "Development of a video-rate stereo machine", IUW, pp. 549-557. 1994.
2. Camus, T.A., "Real-Time Quantized Optical Flow", Workshop on Computer Architectures for Machine Perception 1995
3. Zelek, J.S., "Bayesian Real-Time Optical Flow", Vision Interface 2002,
4. Liu, H., Hong, T.H., Herman, M., Camus, T.A., Chellappa, R., "Accuracy vs. Efficiency Trade-Offs in Optical Flow Algorithms", Computer Vision and Image Understanding, 72(3), 1998, pp. 271-286
5. P.C. Arribas, F.M.H. Macia, "FPGA Implementation of Camus Correlation Optical Flow Algorithm", Vision Interface 2001
6. M. Fleury, A.F. Clark and A.C.Downton, "Evaluating optical-flow algorithms on a parallel machine", Image and Vision Computing, 19(3), 2001, pp. 131-143.
7. Correia, M.V., Campilho, A.C., "Real-time implementation of an optical flow algorithm", International Conference on Pattern Recognition 2002, pp. 247-250.
8. H. Niitsuma and T. Maruyama, "Real-time Detection of Moving Objects", 14th International Conference on Field-Programmable Logic and Applications, 2004.
9. http://www.alpha-data.com
10. http://www.xilinx.com