A Multiplier-Less Implementation of the Canny Edge Detector on FPGA and Microcontroller

Hung Kwan Fung and Kin Hong Wong

***Abstract*—The Canny edge detector plays an important role in pre-processing images for many virtual reality systems. In this paper, a multiplier-less implementation of the Canny edge detector is proposed, which enables the system to be built at a lower cost. It is a heterogeneous system using both Field Programmable Gate Array (FPGA) and microcontroller. It is well known that the Canny edge detector involves a number of local operators such as image smoothing and gradient computation which are complex and time consuming. So, it is not efficient to be implemented on low end processors. In this work, the time consuming tasks in the local operators are offloaded to the FPGA so that real time processing can be achieved. Our experiments show that the proposed system can be implemented with less than 1000 lookup tables which are suitable for low end FPGA products while the precision of the edge detection result is comparable to that implemented by software running on a personal computer.**

***Index Terms*—FPGA, canny edge detection, smart camera, feature extraction.**

I. Introduction

Computer vision is essential for virtual reality systems and applications. And recently, there is a trend of using computer vision in mobile devices such as cellphones, game console and PADs. Computer vision techniques allow machines to understand and interact with the world based on information extracted from the images. Typical useful information in computer vision includes the edge and the corner features.

For edge detection methods, the Canny edge detector, first or second derivative operator methods are popular choices. As for corner feature detection, the SUSAN corner detector [1] and Harris corner detectors [2] are effective tools. These algorithms are local operators that involve operations of nearly all pixels in an image. Therefore, the running time of these kinds of operators are directly proportional to the size of the image and the number of the neighboring pixels required to be examined. This property prohibits the use of these methods on a small microcontroller because of the limited processing power and memory size available. Therefore, unless special hardware chips (for example, Digital Signal Processor (DSP) and Single Instruction Multiple Destination (SIMD) processor) are used, it is difficult to achieve a high throughput computer vision system at low cost. Many people are interested in finding hardware solutions for computer vision algorithms. For example, it is possible to use FPGAs for building feature detectors to enable them to be more efficient in processing and power consumption.

In the paper [3] the whole Harris detector is implemented using purely FPGAs. The speed is fast but the function may not be flexible enough to cater for different applications. That means you need to redesign the whole system for a specific task and specification. To make the system more flexible, some projects have demonstrated that the computationally intensive computer vision algorithms can be running at high frame rate on heterogeneous systems with an ordinary microcontroller and an FPGA. There are other examples such as [4], Benedetti and Perona designed and implemented a real-time 2D feature detection algorithm on FPGA. In the many projects the feature detection system is based on the work of feature detectors such as that in [5]-[7] and the KLT tracker [8]. However all these depend on image gradient information in both horizontal (x) and vertical (y) directions. To compute these image gradients, the whole image is convolved with a kernel and it is therefore very time consuming.

There are also recent projects using FPGAs for computer vision, examples are [9], [10] and [11]. However, they are using high speed FPGAs for the vision tasks, the performances are good but the costs are relatively higher. Leeser, Miller and Yu [12] designed a smart camera setup based on FPGAs and demonstrated two very different applications of this setup: medical image processing and fluid dynamics computation. Part of the computation is shared by the FPGA and the result is sent to a computer for further processing. The first application requires comparing eight 11 ×11 templates to be operated over the whole image. While cross-correlation of two areas (40×40 and 32×32) over two images of size 1008×1016 is needed in the second application. In both applications, they have speedups of over 20 times compared to a pure software implementation. There is a also hybrid approach such as the one by Tippetts *et al*. [13], it implemented a Harris Feature detector and priority queue using an FPGA. By using a micro-processor to run a RANSAC [14] algorithm on top of the result from the FPGA, they successfully implemented an on-board vision solution in determining movement of small unmanned air vehicles that should be small in size, weight and power consumption. This work shows that it is possible to use the FPGA approach in mobile devices.

In this paper, we proposed a smart camera that finds the edge pixels by using the Canny edge detector [15] using an off-the-shelf microcontroller and an FPGA.

Manuscript received August 16, 2016; revised December 14, 2016. This work is supported by a direct grant (Project Code: 4055045) from the Faculty of Engineering of The Chinese University of Hong Kong.

Hung Kwan Fung and Kin Hong Wong are with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, HSH Engineering Building, CUHK, Shatin, Hong Kong (e-mail: edwardfung123@gmail.com, khwong@cse.cuhk.edu.hk).

DOI: 10.7763/IJCTE.2017.V9.1133

This system is intended to be used in applications like making a mobile virtual reality system that is required finding the correct display area for the projector etc. The FPGA acts as a special processor designed to handle the image processing tasks while the microcontroller is used to complete the other content to the correct position in a camera projector system. In such a system it is essential to have a high throughput image processor in order to achieve smooth and real-time projections.

The organization of the paper is as follows. In Section II, the theory of our method will be discussed. Section III is about the details of the implementation. Experimental results are discussed in Section IV and we conclude the whole paper in Section V.

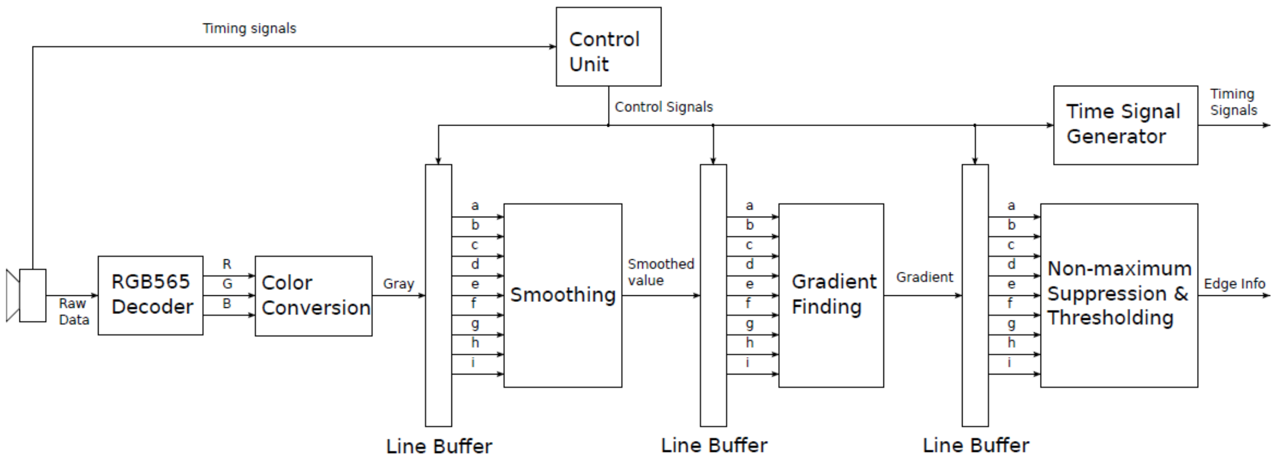


Fig. 1. Overview of our proposed system.

TABLE I: ACTIVATION OF EACH MODULE

	i-th row																							
Module name	0	1	2	3	4	5	6	7	8	9	10	…	h-2	h-1	h	h+1	h+2	h+3	h+4	h+5	h+6	h+7	h+8	h+9
Color Conversion	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	…	Y	Y										
Smooth				Y	Y	Y	Y	Y	Y	Y	Y	…	Y	Y	Y	Y	Y							
Gradient							Y	Y	Y	Y	Y	…	Y	Y	Y	Y	Y	Y	Y	Y				
Thinning										Y	Y	…	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	

II. THEORY AND DESIGN

A. The Canny Edge Detector

The Canny edge detector is still the most widely used algorithm to extract edges in a scene since its introduction in 1986. The Canny edge detector is preferred because by its accuracy and efficiency. The detector takes a gray level image and two thresholds as the input. By adjusting the values of the two thresholds, the user can control the number of edge pixels or the strength of the edges found. The edge detector takes the gray level image as the input and it will be processed by multiple stages sequentially, such as (i) noise reduction by smoothing; (ii) image gradient; (iii) non-maximum suppression and thresholding; and (iv) edge tracing. The output is a binary image representing the edges.

B. System Design

Our design is intended to offload the heavy image processing work from the microcontroller to the FPGA hardware. Our smart camera is a combination of an ordinary camera, a microcontroller and a FPGA module. The overview of system is shown in Fig. 1. In the diagram, the camera provides a stream of data to the processor. The processor extracts edge pixels from the data by using the Canny edge detector. However, since the number of iterations required depends on the image input and cannot be determined beforehand, therefore, only stage (i) to (iii) mentioned in the last section are suitable for FPGA processing. While stage (iv) is handled by software running on a microcontroller in our system.

Since the image processing techniques require neighboring pixel values of the target pixel, it is necessary to delay the output of one stage so that the value can be used in the next stage. For example, to find the gradient of pixel P at (x, y), 8 neighboring pixels are required, where 6 pixels are from the $y - 1$ row and the $y + 1$ row, 2 pixels are from the left and right pixel. To eliminate the need of storing the whole image before any processing to take place, line buffers are added between stages. Each line buffer is a circular buffer which can store up to 4 rows of pixels. Each row is stored in independent RAM chip so that three of them provide the data for the next stage while the one left stores the data from the previous stage. With the line buffers, each stage can run in parallel and return the edge information to the microcontroller as soon as possible. Thus, the delay is minimized.

In Fig. 1 the camera interface has two parts: timing signal and data signal. Since the data stream is a raster scan from the camera, the stream can be considered as a 1D array. To convert this 1D array to 2D array which is more suitable for most image processing techniques, the 1D array has to be divided into different rows. The timing signal tells (i) when the data for this pixel is ready; (ii) when the row has ended; and (iii) when the data stream has ended. The data signals are the raw pixel intensities from different channels. The signal values obtained directly from the camera are encoded, therefore decoding is required to extract the values of individual channels. Although the modules for each stage are running in parallel, the activation times are different as some of the data are delayed. A control unit is used to generate control signals to each module. The control signals are derived from the timing signal of the camera. Table I shows the activation of different modules when the i-th row is sending from the camera. The module is active if there is a 'Y' in the cell. Otherwise, the module is deactivated. As shown in Table I, h is the height of the image, and we can see that the first pixel of the result image is ready after reading 9 rows of pixels from the camera. Since the result is delayed, the original timing signal from the camera is not usable and cannot be sent to the microcontroller directly without modification. To synchronize the system a timing signal generator is used to generate the correct timing signals to the microcontroller to coordinate the system. It is achieved by sending a delayed version of the timing signal of the camera to other parts to achieve synchronization.

C. Image Acquisition

There are various kinds of image encoding standards available in the industry. In this paper, we assume the pixels in the data stream are encoded in the RGB565 format. Each pixel of the multichannel image I_{RGB} is 16 bits long and the first 5 bits are the value of the red channel. The following 6 bits are the value of the green channel. The last 5 bits are the value of the blue channel. The resolution of the red channel and blue channel is 1-bit lower than the green channel. Table II shows the number of bits, resolution and range of each channel.

To process the image, the pixels have to be decoded in order the find the value of each channel. Because the ranges of the three channels are not the same, the values of red channel and blue channel have to scale up (by multiplying two) before any processing with the green channel.

In order to decode the pixel, the processor has to process 3 masking operations and shift the masked results accordingly. The temporary results for red and green channel have to be shifted to the right by 10 bits and 5 bits respectively. For the blue channel, the temporary result has to be shifted to left by 1. It would be computationally expensive to use the microcontroller to decode the image. But by using the FPGA the masking and bit shifting operation can be processed instantly by just wiring the signal to the correct places. Zeroes are padded in the front so that a pixel is represented by 3 bytes.

TABLE II: SUMMARY OF EACH CHANNEL

Channel	No. of bits	Resolution	Range
Red	5	32	0-31
Green	6	64	0-63
Blue	5	32	0-31

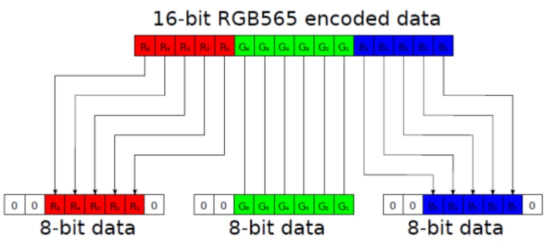


Fig. 2. The bit arrangement of the 16-bit RGB pixel data.

In Fig. 2, we show how a RGB565 encoded pixel data can be decoded by the FPGA. It is achieved by rewiring the signals through masking and shifting.

D. RGB-to-Gray Conversion Module

After decoding, the RGB image is ready to be converted into a grayscale format. RGB conversion is a weighted average operation of the three channels. The conventional grayscale value P_{gray} is calculated by using the formulation introduced in [16], the formula is shown here as follows.

$$P_{\text{gray}} = 0.3 * P_R + 0.59 * P_G + 0.11 * P_B \tag{1}$$

However, for simplicity in design and avoiding floating point calculation, we use equation (2) to convert a RGB pixel to a grayscale pixel, because all multiplications in equation 2 can be implemented by bit shifting.

$$P_{\text{gray}} = 0.25 * P_R + 0.5 * P_G + 0.25 * P_B \tag{2}$$

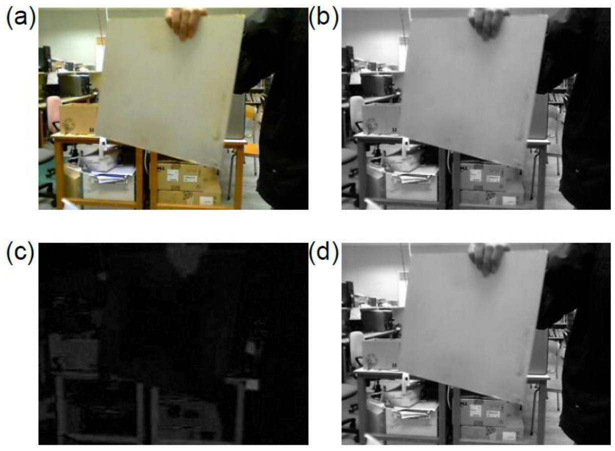


Fig. 3. Conversions of the original image to different formats.

The visual effect of the conversion by applying equation (2) 2 is more or less the same as the result of applying the conventional equation (1). Fig. 3 shows the conversion result of the two formulae. The result of conversion is stored in the line buffer for the next stage of operation. Since the resolution of the green channel is 6-bit, the converted grayscale value is also in 6-bit resolution. In Fig. 3, the image (a) is the original RGB image, (b) and (d) are the grayscale conversion results by applying the conventional equation (1) and our simplified equation (2) respectively. The image in (c) shows the absolute difference of (b) and (d). It is scaled up by a factor of 5 to make the difference more noticeable.

E. Image Smoothing Module

The first stage of the Canny edge detector is noise reduction by image smoothing. It is achieved by convolving a Gaussian kernel with the image. The Gaussian kernel is an approximation of the 2D Gaussian distribution function. The constants in the kernel are floating point numbers smaller than 1. But it is not practical to use floating point numbers since their operations are slower than integer arithmetic.

Furthermore, we found that the result of using floating point arithmetic does not yield significantly better result than using integer arithmetic. Therefore we use integer arithmetic here as follows. First, the constants of the kernel are scaled up by an integer so that all the constants are greater than 1. Then, the kernel values are rounded to the nearest integers. And, the image is convolved with the new integer kernel. Finally, the result of the convolution is divided by the scale factor.

However, the direct implementation of this integer arithmetic does not favor FPGA implementation because of two reasons. The first reason is that it requires a multiplication unit for each constant in the kernel. A multiplication unit will consume many logic blocks. The second reason is that division has to take several clock cycles to complete and it also uses a lot of logic blocks to make one division unit. To make it more suitable for the FPGA implementation, we specially designed a kernel where the convolution operation will only need shifting and adding to complete. The proposed kernel is:

$$K_{smooth} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

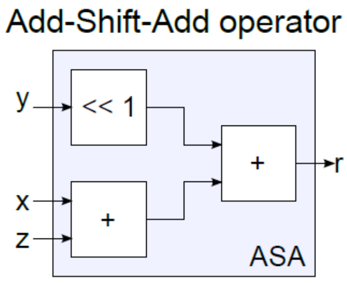


Fig. 4. Block diagram of the Add-Shift-Add module.

The proposed kernel is an approximation of the Gaussian Kernel with sigma equal to 0.76. Notice that, to compute the convolution of this kernel with a 3 × 3 pixel patch *P*, four "Add-Shift-Add" (ASA in short) operations are required. In Fig. 4, the inputs and output relation of the ASA operator is shown and >> is bitwise right shift and << is bitwise left shift. Consider the following:

$$P = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}, hence$$

$$P * K_{smooth}$$
$$= \frac{1}{16}\left((a + c + 2b) + (g + i + 2h) + 2(d + f + 2e)\right)$$
$$= \frac{1}{16}\left(ASA(ASA(a,b,c), ASA(d,e,f), ASA(g,h,i))\right)$$

where ASA(x, y, z) = $x + 2y + z$. The multiplication of 2 in the ASA operator can be realized by shifting the input to the left by 1 bit. The ASA operator is simple in architecture because it is multiplier-less and has only two adders and a shifter. This ASA operator can be reused in the gradient operators which will be discussed later. Since the scale factor of the Gaussian filter is 16, the implementation is simple. It just performs an arithmetic right shift by 4 bits on the result of the final ASA module. Fig. 4 shows the block diagram of the ASA operator. The architecture of the ASA module is simple which consists of two adders and a shift-left-by-1 module. The implementation of the proposed Gaussian filter is shown in Fig. 5. It uses four ASA modules and an arithmetic right shift module. The right shift module will shift the result of the last ASA module by 4 bits.

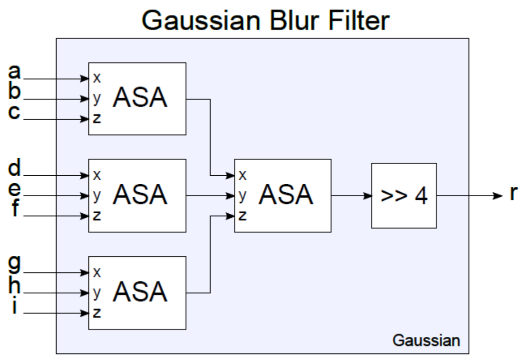


Fig. 5. Block diagram of the Gaussian filter module.

F. Image Gradient Module

The Sobel gradient operator is used to find the image gradient. The Sobel gradient operator returns the horizontal gradient G_x and vertical gradient G_y of an image *I*. They are defined as:

$$G_x = \frac{1}{4}\begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * I, G_y = \frac{1}{4}\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ -1 & +2 & +1 \end{bmatrix} * I$$

We can reuse the Add-Shift-Add module introduced previously to implement the kernels. Similar to the Gaussian filter, the Sobel operator can be implemented with adder and shifter. In Fig. 6, it shows our implementations of the Sobel operator in the vertical and horizontal directions. In the system, the two operators are having a similar architecture the only difference is the inputs they take. It is noted that, in the architectures, the right shift module must be arithmetic right shift since the result from the subtraction module can be negative.

The result of the horizontal gradient and vertical gradient calculation is 7 bits long but not the same as the 6-bit long input. It is because the gradient could be a negative value and one extra bit is required to represent the negative values. Fig. 7 shows the block diagram of the Sobel operator. In the Figure, it shows the absolute value module takes a 7 bits binary number b using the two's complement format, b0 is the least significant bit while b6 is the most significant bit. The label 5:0 and 6:1 indicate the numbers of bits and how they are connected. For example, 6:1 means from bit 6 to bit 1 are connected to inputs, and bit 0 is not connected. The magnitude G and the angle θ of the gradient is calculated as:

$$G = |G_x| + |G_y|$$
$$\theta = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Normally, the magnitude *G* is calculated using the Euclidean norm which involves squaring the values and then taking square root.

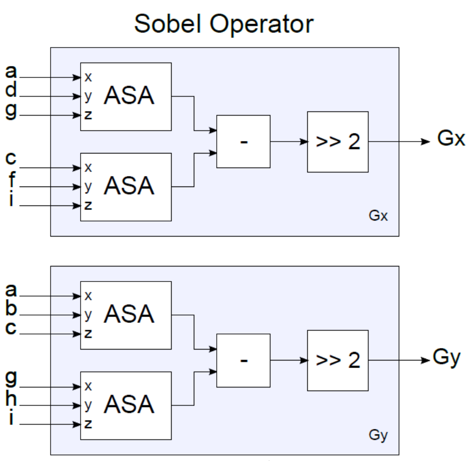


Fig. 6. Block diagram of Sobel operator.

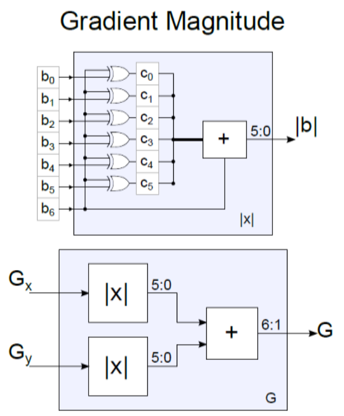


Fig. 7. Block diagram of gradient magnitude module.

In practice, to increase the processing speed, the sum of the absolute values of the horizontal gradient and vertical gradient is used and it gives accurate enough result. This operation is also suitable to be implemented by a FPGA.

The value of θ in Equation 5 is rounded to the four angles to representing horizontal line (0^o), vertical line (90^o) and the two diagonals (45^o and 135^o). The trigonometric function $\tan^{-1}(\ )$ is involved and it is implemented using table lookup. By using table lookup, the calculation of (G_y/G_x) is not needed. Since the function is symmetric, it is possible to consider the first quadrant only. We first determine whether the line is horizontal or vertical. If the line is neither horizontal nor vertical, then it must be a diagonal line. To check if it is a 45^o-diagonal line or a 135^o-diagonal line, we only have to examine the signs of G_x and G_y. If both of the signs are the same, the line is a 45^o-diagonal line.

TABLE III. GRADIENT ORIENTATION LOOKUP TABLE. THE COLUMN AND THE ROW ARE THE ABSOLUTE VALUE OF G_X AND G_Y RESPECTIVELY. THE INCLINED ANGLE OF THE LINE (0^O, 45^O, 90^O, 135^O) IS MAPPED TO 0, 1, 2 AND 3 RESPECTIVELY

		$\lvert G_x\rvert$					
		0	1	2	3	4	5
$\lvert G_y\rvert$	0	0	0	0	0	0	0
	1	2	1	1	0	0	0
	2	2	1	1	1	1	0
	3	2	2	1	1	1	1
	4	2	2	1	1	1	1
	5	2	2	2	1	1	1

Otherwise, it is a 135^o-diagonal line. Since the first quadrant is considered, the absolute values of G_x and G_y are used in the table lookup. A portion of the lookup table is shown in Table III as it is too messy to show all entries. The reader should able to deduce the remaining values.

The number of bits required to store G_x and G_y is 7 bits. However, the absolute value of them is not greater than 63. Therefore, 6 bits are used to store the absolute value of G_x or G_y. The magnitude of the gradient is the sum of them, and therefore 7 bits are needed to be stored. The number of bits required for orientation is 2 bits. So the total required is bits. However, since most memory unit available nowadays requires 8-bit word size, we discard the least significant bit of the magnitude. By using such arrangement, we can encode the magnitude and orientation in an 8-bit data and store them in the line buffer.

G. Non-maximum Suppression and Hysteresis Thresholding

To thin the line into one-pixel wide, non-maximum suppression or NMS is applied to the gradient magnitude image. To determine whether the pixel is on the edge or not, the gradient magnitude G of this pixel is compared with the other adjacent pixels along the gradient orientation O. If the gradient magnitude of this pixel is less than the other two pixels $G1$ and $G2$, then this pixel is not an edge. Otherwise, the pixel is compared to the high threshold T_{high} and low threshold T_{low}. If $G < T_{low}$, the pixel is not an edge pixel. If $T_{\text{low}} < G < T_{\text{high}}$, it is a weak edge. Otherwise it is a strong edge. To summarize, an edge pixel must fulfill the following conditions simultaneously: (1) Greater than the adjacent pixel along the gradient orientation and; (2) Greater than the low threshold. Fig. 8 shows the modules that process the non-maximum suppression stage in one clock cycle. In the diagram, we use two bits to encode the gradient orientation result O. Strong edge, weak edge and non-edge is encoded into 11(b), 01(b) and 00(b), respectively.

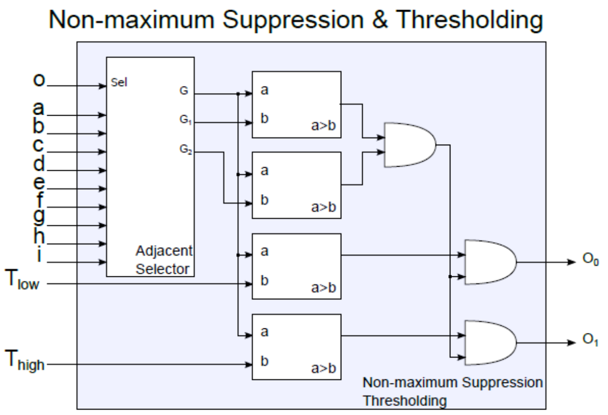

Non-maximum Suppression & Thresholding
o
a
b
c
d
e
f
g
h
i
Sel
G
G1
G2
Adjacent Selector
Tlow
Thigh
a
b
a>b
O0
O1
Non-maximum Suppression Thresholding

Fig. 8. Non-maximum suppression and hysteresis thresholding module.

Four comparison statements are required to determine whether the point is a strong edge, weak edge or not on edge. They should be executed sequentially in software when implemented using a single CPU system. However, for our FPGA approach they can be handled in parallel, since they are independent with each other. In Fig. 8, the component relies on the Adjacent Selector to select suitable values of *G*, *G1* and *G2* from the gradient magnitudes of the neighbor pixels according to the value of the gradient orientation O. This step can be achieved by using a lookup table.

The architecture is shown in Fig. 8. In the diagram, the adjacent selector picks the correct value of G, $G1$ or $G2$ according to the gradient orientation O. The comparator outputs logic 1 if the input a is greater than b. Otherwise, the output will be 0. The result is encoded by using 3 2-input AND gates.

III. Implementation

Our proposed smart camera is implemented using the VHSIC hardware description language (VHDL) on a Xilinx Spartan-3E XC3S250E device. Three sets of line buffers are used for different stages. We assumed the data stored in the line buffers is one byte long and the size of the image is 320 × 240. Since each line buffer stores 4 rows of data, the number of memory cells required for one line buffer is 1280 bytes. Therefore, the total number of bytes for the line buffers is 3840 bytes. The data of 3 rows in the line buffer should be accessed in parallel and used as the inputs of the image processing module.

The number of Lookup Tables (LUTs) is another important resource in FPGA since all combinational logic is implemented using LUTs. The number of LUTs of RGB-to-gray convertor, Gaussian blur module, Gradient module and non-maximum suppression and thresholding module are 9 LUTs, 64 LUTS, 265 LUTS and 50 LUTS respectively. The number of LUTs for the image processing modules is 388. We estimate that the total number of LUTs required for the whole system including the control units, line buffers and timing signal generator is the 776. As the XC3S250E has 4896 LUTs, only around 16% of LUTs is needed. This result gives enough room for implementing more sophisticated color space converters and Gaussian kernels for the smoothing module. As the number of LUT required in this design is low, it is possible to implement another detector into one FPGA. For example, the Harris feature detector can reuse the result of the gradient module and outputs the result together with the Canny edge detector.

IV. Experiment

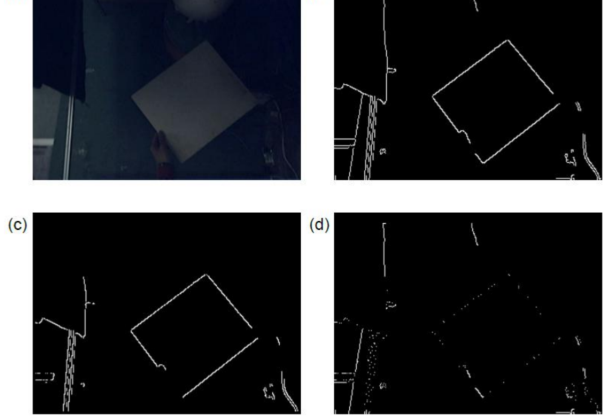


Fig. 9. Comparison of the edge detector results of the proposed hardware approach and OpenCV software implementation. (a) is the original image. (b) and (c) are the edge image from OpenCV and the proposed system respectively. (d) shows the difference between the two edge images.

To evaluate the result of our proposed hardware Canny edge detector on FPGA, we also implemented a pure software edge detector based on the OpenCV [17] library for comparison. Fig. 9 shows the results of our proposed system and the result of using the software library. The results are comparable to most significant edges are found in the images. The discrepancy in the two edge images, as shown in part (d) of Fig. 9, are caused by different RGB-to-gray conversion methods and Gaussian kernels used in the smoothing process. For more information, please visit our project web page at https://www.cse.cuhk.edu.hk/~khwong/www2/conference/2016/IWPR2016/IWPR2016.html.

V. Conclusion

A hardware implementation of the Canny edge detector is proposed in this paper. It is multiplier-less and is suitable for low cost devices. In this work, the Canny edge detector is divided into different stages and the system can process the data stream directly. The detail of each module is discussed in this paper. They are implemented using VHDL and synthesized using the target device Xilinx-Spartan-3E. It shows that the number of LUTs required is low, so there is a possibility that additional functions such the Harris feature detector can be added to the device in future. The result of our hardware approach is compared to that generated by a pure software method using OpenCV. Both results are similar and visually acceptable. It is believed that our method is suitable for many mobile virtual reality applications.

References

[1] S. M. Smith and J. M. Brady, "Susana new approach to low level image processing," *International Journal of Computer Vision*, vol. 23, no. 1, pp. 45–78, 1997.

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

[3] T. L. Chao and K. H. Wong, "An efficient fpga implementation of the Harris corner feature detector," in *Proc. 2015 14th IAPR International Conference on Machine Vision Applications (MVA)*, 2015, pp. 89–93.

[4] A. Benedetti and P. Perona, "Real-time 2-d feature detection on a reconfigurable computer," in *Proc. the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1998, p. 586.

[5] M. Krystian and C. Schmid, "An affine invariant interest point detector," *Computer Vision—ECCV 2002*, Springer Berlin Heidelberg, pp. 128-142, 2002.

[6] J. Shi and C. Tomasi, "Good features to track," in *Proc. 1994 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 1994, pp. 593–600.

[7] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," pp. 674–679, 1981.

[8] C. Tomasi and T. Kanade, "Detection and tracking of point features," *International Journal of Computer Vision, Tech. Rep.*, 1991.

[9] D. Honegger, H. Oleynikova, and M. Pollefeys, "Real-time and low latency embedded computer vision hardware based on a combination of fpga and mobile cpu," in *Proc. 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2014, pp. 4930–4935.

[10] Q. Xu, S. Varadarajan, C. Chakrabarti, and L. J. Karam, "A distributed canny edge detector: algorithm and fpga implementation," *IEEE Transactions on Image Processing*, vol. 23, no. 7, pp. 2944–2960, 2014.

[11] P. R. Possa, S. A. Mahmoudi, N. Harb, C. Valderrama, and P. Manneback, "A multi-resolution fpga-based architecture for real-time edge and corner detection," *IEEE Transactions on Computers*, vol. 63, no. 10, pp. 2376–2388, 2014.

[12] M. Leeser, S. Miller, and H. Yu, "Smart camera based on reconfigurable hardware enables diverse real-time applications," in *Proc. the 12th Annual IEEE Symposium on Field-Programmable*

Custom Computing Machines, Washington, DC, USA, 2004, pp. 147–155.

[13] B. Tippetts, S. Fowers, K. Lillywhite, D.-J. Lee, and J. Archibald, "Fpga implementation of a feature detection and tracking algorithm for realtime applications," in *Proc. the 3rd International Conferenceon Advances in Visual Computing*, Volume Part I, , 2007, pp. 682–691.

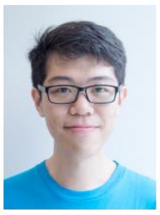
[14] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, pp. 381–395, June 1981.

[15] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, Nov. 1986.

[16] W. K. Pratt, *Digital Image Processing*, New York, NY, USA: John Wiley & Sons, Inc., 1978.

[17] OPENCV, Intel. Open Source Computer Vision Library. [Online]. Available: http://www.intel.com/technology/computing/opencv/index.html

Kin Hong Wong obtained his Ph.D. from the University of Cambridge, UK and is now an Associate Professor of the Computer Science and Engineering Department of the Chinese University of Hong Kong. His research interests include computer vision, signal processing and virtual reality.



Hung Kwan Fung is now at the Department of Computer Science and Engineering, The Chinese University of Hong Kong, HSH Engineering Building, CUHK, Shatin, Hong Kong, he has obtained M.Phil. and B.Eng. degrees in computer engineering.