An Efficient Implementation of the Hough Transform using DSP slices and block RAMs on the FPGA

Xin Zhou, Yasuaki Ito, and Koji Nakano
Department of Information Engineering
Hiroshima University
Kagamiyama 1-4-1, Higashi Hiroshima, 739-8527 Japan

***Abstract*—Since FPGA chips maintain relatively low price and its programmable features, it is widely used in those fields which need to update architecture or functions frequently such as communication and education areas. Especially, in mobile devices that recently require the ability to perform computation such as real-time image processing, FPGAs are promising devices. Recent FPGAs have hundreds of embedded DSP slices and block RAMs. For example, Xilinx Virtex-6 Family FPGAs have a DSP48E1 slice, which is a configurable logic block equipped with fast multipliers, adders, pipeline registers, and so on. They also have a dual-port memory with 18Kbits as a block RAM. Therefore, one of the most important key techniques for accelerating computation using such FPGAs is an efficient usage of DSP slices and block RAMs. The main contribution of this paper is to present a new FPGA architecture for the Hough transform for all the pixel data input in raster scan order. The architecture uses 90 DSP48E1 slices and 181 block RAMs with 18Kbits that work in parallel. The experimental results show that this implementation runs in 247.525MHz and given a binary image of size $n \times n$, our circuit can perform in $n^2 + \sqrt{2}n + 379$ clock cycles.**

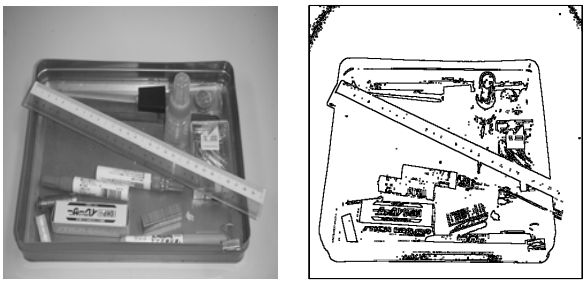
***Keywords*-Image processing, Line detection, Hough transform, FPGA, Embedded DSP slices, Embedded block RAMs**

I. INTRODUCTION

A Field Programmable Array (FPGA) is a programmable logic device designed to be configured by the customer or designer by hardware description language after manufacturing. The most common FPGA architecture consists of an array of logic blocks, I/O pads, block RAMs and routing channels. Furthermore, recent FPGAs have embedded DSP slices that make a higher performance and a broader application. The Xilinx Virtex-6 series FPGAs have DSP48E1 slices that are equipped with a multiplier, adders, logic operators, etc [1]. More specifically, the DSP48E1 slice has a two-input multiplier followed by multiplexers and a three input adder/subtractor/accumulator. The DSP48E1 multiplier can perform multiplication of an 18bit and a 25bit two's complement numbers and produces one 48bit two's complement production. Programmable pipelining of input operands, intermediate products, and accumulator outputs enhances throughput and improves frequency. The DSP48E1 also has pipeline registers between operators to reduce the delay. The block RAM in the Virtex-6 FPGA is an embedded memory supporting synchronized read and write operations. In the Virtex-6 FPGA, it can be configured as 36Kbit dual port block RAMs, FIFOs, or two 18Kbit dual port RAMs. In our architecture, it is used as a 1K$\times$18bit dual port RAM. Since FPGA chips maintain relatively low price and its programmable features, it is widely used in those fields which need to update architecture or functions frequently such as communication and education areas. They are widely used in consumer and industrial products for accelerating processor intensive algorithms [2]–[9].

Hough transform is a technique to find shapes in images [10]. In particular, it has been utilized to extract lines, circles, ellipses and arbitrary shapes. The Hough transform defines a mapping from an image into a parameter space represented by an accumulate array. The parameter space is defined by parameterizing detected shapes. Based on each edge point of the image, the mapping adds a vote to corresponding elements in the accumulate array. The elements that are increased represent associated parameters based on detected shapes. Therefore, the elements that are voted intensively correspond to the parameters of shapes in the image space.

The Hough transform can be used to extract straight lines in a binary image [11]. The idea of this method is to exploit the duality between points of a line and parameters of that line. A point in the image is represented by a curve in the parameter space and lines of collinear points intersect in the parameter space at one point. These intersections are counted in an array of accumulators that quantizes the parameter space appropriately. In the followings, we call this counting to the accumulators *voting*. More specifically, for each edge point (x, y) in a 2-dimensional image, the voting is performed along a curve $\rho = x\cos\theta + y\sin\theta$ ($0 \leq \theta < 180$). Possible lines can be detected by searching points that are voted intensively. Figure 1 shows an example of straight line detection using the Hough transform. For an input image (Figure 1(a)), the binary edge image (Figure 1(b)) is obtained by the edge detector such as Sobel filter. The result of voting to the parameter space is shown in Figure 1(d). In this figure, darker points show points that are voted intensively, that is, represent probable lines. According to the result of voting, the principal lines are detected (Figure 1(c)).



(a) Input image (b) Binary edge image by Sobel filter

(c) Line detection using the Hough transform (d) Hough parameter space

Figure 1. Example of straight line detection using the Hough transform

In our previous work [12], we proposed an FPGA implementation of the Hough transform with DSP slices and block RAMs. We used 178 DSP slices and 180 block RAMs and they work in the fully pipelined architecture. The implementation need to accept the coordinates of edge points as input. However, since pixel data of input images from digital video cameras are generally input in raster scan order, the requirement might not to be versatile. Also, since, after voting, identified straight lines are obtained just by thresholding, similar to lines in the input image but incorrect lines are also detected.

In this paper, we improve the FPGA implementation of our architecture. One of the main different points of this paper from our previous work is that our improved architecture processes pixel data given in raster scan order, and outputs the identified straight lines. Therefore, the voting time is a fixed clock cycles corresponding to the size of the image. Also, compared to our previous implementation, the number of used DSP slices becomes approximately half. Our new idea includes: **(i) Voting Space Partitioning:** Polar coordinate voting space (θ, ρ) is partitioned and arranged into block RAMs. This enables us to perform voting operations in parallel. Also, the function of dual-port of block RAMs are fully used to accumulate the voting value instantly. **(ii) Efficient Usage of DSP slices:** DSP slices are used $x\cos\theta$ and $y\sin\theta$ in parallel for each edge pixel (x, y). We compute $x\cos\theta$ for θ such that $0 \leq \theta < 90$ instead of computing them for θ such that $0 \leq \theta < 180$. Also, we avoid the computation of the values of $\cos\theta$ by pre-loading them in the DSP slices. In addition, since pixel data are input in raster scan order, we use the fact that the value of y in a certain row is not change. When pixels in a certain row y are processed, we pre-compute $(y+1)\sin\theta$ for θ such that $0 \leq \theta < 90$ in the next row. According to the above, compared with our previous implementation, the number of utilized DSP slices is reduced to approximately half. **(iii) Fully Pipelined Architecture:** We design our new Hough transform architecture as a fully pipelined one using the Virtex-6 FPGA XC6VLX240T that has 768 DSP48E1 slices arranged in 8 columns of 96 adjacent DSP48E1 slices. Neighboring DSP48E1 slices are connected directly through pipeline registers. Our Hough transform architecture uses only 1 DSP slice to compute $y\sin\theta$, and uses 1 columns to compute $x\cos\theta$. **(iv) More precise line detection:** In our previous work, the straight lines are output such that the number of votes exceeds a certain threshold value. However, the output includes many mistaken straight lines due to the discretization error in voting. In this paper, therefore, after voting process, to obtain more precise straight lines, we apply the 3×3 maximum filter for the voted results.

Many hardware algorithms for FPGA implementation of the Hough transform for lines have been proposed in past. In the existing researches, they introduced incremental Hough transform [13]–[15], CORDIC [16], [17], and hybrid-log arithmetic [18] to the computation of Hough transform. Since most of recent FPGAs produced by principal vendors equip embedded DSP slices [19]–[21], one of the most important key techniques for accelerating computation using FPGAs is an efficient usage of DSP slices and block RAMs.

II. Hough Transform

The main purpose of this section is to review Hough transform algorithms for straight lines. Suppose that we have an image of size $n \times n$. We assume that $n \times n$ pixels are arranged in two dimensional xy-space such that the origin is in the center of the image as illustrated in Figure 2. Hence, both coordinates x and y take integers in the range

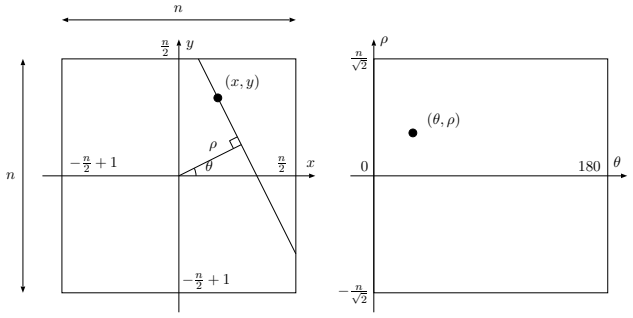


Figure 2. Two dimensional Spaces xy and $\theta\rho$ used in the Hough transform

$[-\frac{n}{2}+1, \frac{n}{2}]$. A pixel (x, y) $(-\frac{n}{2}+1 \leq x, y \leq \frac{n}{2})$ in the

xy-space is converted to a curve in the $\theta\rho$-space by the following formula: $\rho = x\cos\theta + y\sin\theta \quad (0 \leq \theta < 180)$. Clearly, the double inequality $-\frac{n}{\sqrt{2}} < \rho \leq \frac{n}{\sqrt{2}}$ is satisfied. The values of θ and ρ can also be obtained geometrically. Suppose that we draw a line going through the origin with angle θ as illustrated in Figure 2. For such lines, we can draw the orthogonal line going through a pixel (x, y). The value of ρ corresponds to the distance to the line. In other words, a point (θ, ρ) of $\theta\rho$-space corresponds to a line of xy-space.

The key idea of the Hough transform is to vote in $\theta\rho$-space only for every edge pixel in the xy-space. Let (x, y) be the pixel in xy-space, and let $p[x][y]$ be the value of the pixel such that $p[x][y] = 1$ if a pixel (x, y) is an edge pixel and $p[x][y] = 0$ if a pixel (x, y) is a non-edge pixel. In this paper we process the image in raster scan order, the Hough transform is spelled out as follows:

[Straight Forward Hough Transform]

```
for y ← −n/2 + 1 to n/2
    for x ← −n/2 + 1 to n/2
        if p[x][y] = 1
            for θ ← 0 to 179
                begin
                    ρ ← x cos θ + y sin θ
                    v[θ][ρ] ← v[θ][ρ] + 1
                end
```

For simplicity, we assume that the value of ρ is automatically rounded to an integer. In the Straight Forward Hough Transform, for each point (x, y), the values of $x\cos\theta$ and $y\sin\theta$ are computed for $\theta = 0, 1, \ldots, 179$. If $v[\theta][\rho]$ is storing a large value, many edge pixels in the input pixels lie in the line in xy-space corresponds to a point (θ, ρ) in $\theta\rho$-space.

We will show that, it is sufficient to compute these values for $\theta = 0, 1, \ldots, 90$. From the addition theorem of trigonometric functions, we have

$$\begin{aligned} \rho &= x\cos(180-\theta) + y\sin(180-\theta) \\ &= -x\cos(\theta) + y\sin(\theta). \end{aligned} \quad (1)$$

Using Formula (1), the Hough transform can also be done by partitioning the range $[0, 179]$ of θ into two ranges $[0, 89]$ and $[90, 179]$. Also, we avoid going through array v for finding elements larger than a threshold. Thus, our new Hough transform, called the Circuit-oriented Hough Transform is be spelled out as follows:

[Circuit-oriented Hough Transform]

```
for y ← −n/2 + 1 to n/2
    for x ← −n/2 + 1 to n/2
        if p[x][y] = 1
            begin
                for θ ← 0 to 89 do
                    begin
                        ρ ← x cos θ + y sin θ
                        v[θ][ρ] ← v[θ][ρ] + 1
                    end
                for θ ← 1 to 90 do
                    begin
                        ρ ← −x cos(θ) + y sin(θ)
                        v[180 − θ][ρ] ← v[180 − θ][ρ] + 1
                    end
            end
```

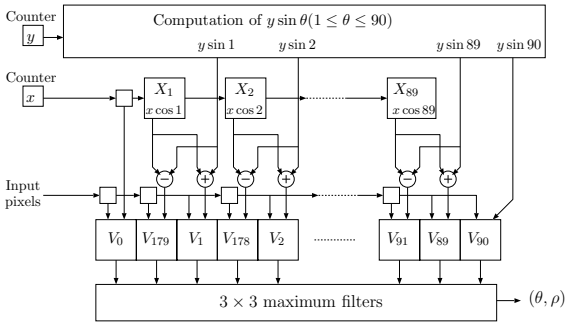


Figure 3. The outline of our FPGA architecture for the Hough transform

In the following section, we show an efficient implementation of the Circuit-oriented Hough Transform.

III. Our FPGA architecture for the Hough transform

This section describes our FPGA architecture for the Hough transform using DSP slices and block RAMs in Xilinx Virtex-6 Family FPGA XC6VLX240T-1 as the target device [22].

A. Structure of our architecture for the Hough transform

Figure 3 illustrates the outline of our FPGA architecture for the Hough transform. Whenever each input pixel is given, the two counters for x and y increment appropriately. We use 89 DSP slices $X_1, X_2, \ldots X_{89}$. For each θ $(0 \leq \theta \leq 90)$, X_θ computes $x\cos\theta$. Since $x\cos 0 = x$ and $x\cos 90 = 0$, DSP slices X_0 and X_{90} are not necessary. Also, we use a module to compute $y\sin\theta$ $(1 \leq \theta \leq 90)$. Using an adder and a subtractor for $x\cos\theta$ and $y\sin\theta$, $\rho_\theta = x\cos\theta + y\sin\theta$ and $\rho_{180-\theta} = -x\cos\theta + y\sin\theta$ are computed. We also use 180 block RAMs $V_0, V_1, \ldots, V_{179}$ to store the voting value. Address ρ of each V_θ $(0 \leq \theta \leq 179)$ is used to store the value of $v[\theta][\rho]$. After voting, to obtain identified straight lines, we use 3×3 maximum filters. These filters simultaneously work row by row.

To minimize the delay between registers, DSP slices $X_1, \ldots, X_{90}$ are connected in a pipeline fashion as illustrated in Figure 3. Each X_θ has a register to store the value of x. In every clock cycle, the value is transferred from X_θ to $X_{\theta+1}$.

Figure 4 illustrates a DSP slice X_θ. In X_θ, the value of x is loaded in an internal register. Also, the value of $\cos\theta$ is pre-computed. Note that the value of $\cos\theta$ used in X_θ is a fixed value. The product of x and $\cos\theta$ is computed in a multiplier of the DSP slice X_θ.

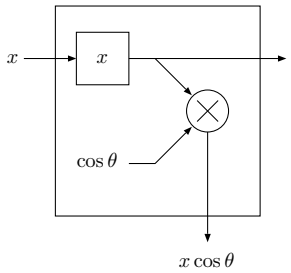


Figure 4. A DSP slice X_θ to compute $x\cos\theta$

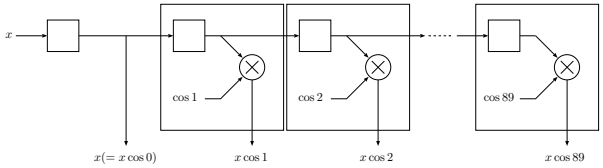


Figure 5. Pipeline architecture to compute $x\cos\theta$ with DSP slices

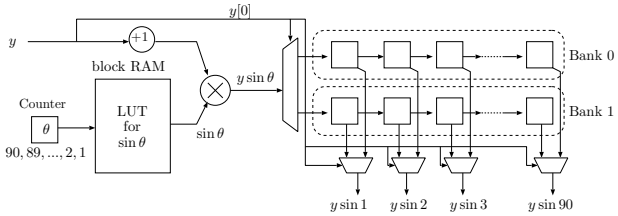


Figure 6. Architecture of computing $y\sin\theta$ with one DSP slice

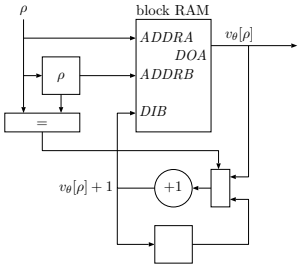


Figure 7. A block RAM V_θ to store $v[\theta][\rho]$

In the Virtex-6 FPGA XC6VLX240T, that is our target device, DSP48E1 slices are arranged in 8 columns of 96 adjacent DSP48E1 slices. Neighboring DSP48E1 slices are connected directly through pipeline registers. Our Hough transform architecture uses 1 column to compute $x\cos\theta$, and uses a pipeline technique to maximize the clock frequency (Figure 5).

Also, to compute $y\sin\theta$ $(1 \leq \theta \leq 90)$ we use the fact that the value of y in a certain row is not change since pixel data are input in raster scan order. Therefore, when pixels in a certain row y are processed, we pre-compute $(y+1)\sin\theta$ for θ such that $0 \leq \theta < 90$ in the next row and store them into the registers. In the next row $y+1$, the computed values of $(y+1)\sin\theta$ are used. Figure 6 illustrates our architecture to compute $y\sin\theta$. We use a look-up-table using a block RAM to compute $\sin\theta$. and a DSP slice to compute a product of y and $\sin\theta$. Also, we utilize two series of registers, called banks. One is used to pre-compute the values of $y\sin\theta$ for the next row. The other is used to output the already computed $y\sin\theta$ for the current processing row. To compute the values of $\sin\theta$ we successively generate the value of $\theta = 90, 89, 88, \ldots, 2, 1$ by a counter. By inputting them to the look-up-table, the values of $\sin\theta$ are obtained. Using the DSP slice, the products of $y\sin\theta$ are computed. Note that the values of $y\sin\theta$ is for the next row. Therefore, the value of y is incremented in advance. The obtained values are successively input to a bank. In each bank, registers are cascaded shown in the figure. The values shift one by one until all the values are input to the bank. When pixels in a row are finished, the banks are switched.

Let $v_\theta[i]$ denote the data of address i in block RAM V_θ. Since ρ is given to it ADDRA, $v_\theta[\rho]$ is output from *DOA* after the rising clock edge as illustrated in Figure 7. After that, $v_\theta[\rho]+1$ is computed and it is given to *DOB*. Since ρ is given to *ADDB*, $v_\theta[\rho]+1$ is written in $v_\theta[\rho]$. In other words, $v_\theta[\rho] \leftarrow v_\theta[\rho]+1$ is performed. At that time, according to the restriction stated in the above, since the same value of ρ may be input continuously, the setting of block RAMs must be RF. Namely, when the same value of ρ is input continuously, the former voted value is not read from the block RAM. To avoid this situation, we use an additional register to store the latest voted value and if the same value of ρ is input continuously, the stored value is used instead of the value read from the block RAM. Note that the above voting process is performed when the input value is an edge pixel. Namely, when the it is a non-edge pixel, the voting process is not performed.

In the following, when all the voting operations are completed, we utilize 3×3 maximum filters to output the final correct identified straight lines. The maximum filter is defined as the maximum of all pixels within a local region of an image. In here, for each value in the voting space, this filter copies the largest value from a 3×3 region to it. In the voting process, the vote concentrates to each point (θ, ρ) corresponding to a line in the original image. However, it also concentrates to around the point. If identified straight lines are determined by thresholding for voted values, many lines which are not in the original image may be obtained.

Figure 8 illustrates our architecture to perform a 3×3 maximum filter to the voted results. Since the voted values in the same ρ can be obtained from $V_0, V_1, \ldots, V_{179}$, this

architecture works row by row in a pipeline fashion. To perform a 3×3 maximum filter to each value in a certain row, it is concurrently read from $V_0, V_1, \ldots, V_{179}$. After that, using comparators, local maxima of each 3 neighboring votes in the row are obtained. These local maxima are input shift registers. After local maxima in the 3 rows are computed, local maxima of each 3×3 votes are obtained by computing maxima from corresponding 3 values. If the maximum equals to the original value of the center in the corresponding 3×3 votes, its (θ, ρ) that represents a probable line is input to the shift registers and output through the registers.

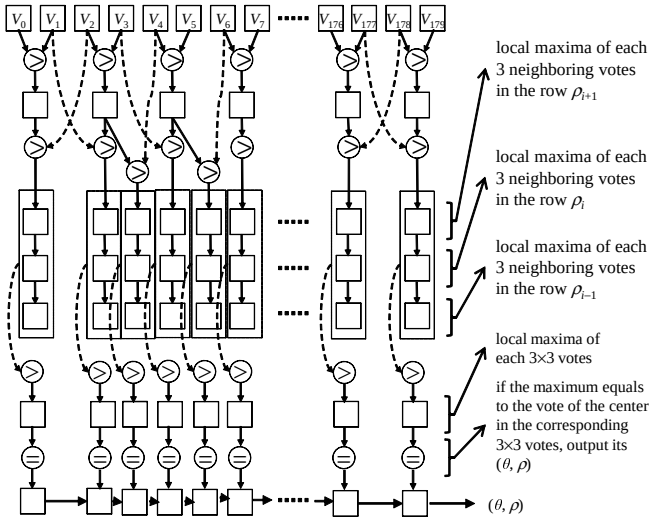


Figure 8. Pipeline architecture of 3×3 maximum filters

B. Data representation

The choice of data precision is guided by the implementation cost in terms of area, simplicity of design, speed and power consumption. Higher precision will lead to less quantization error in the final implementation. On the other hand, lower precision will produce more compaction and faster designs with less power consumption. A trade-off choice needs to be made depending on the given application and available FPGA resources.

In this paper, the data format of inputs are values (0 or 1) of all pixels in the image, these values are input in raster scan order. The coordinates (x, y) which are necessary to compute ρ are generated by the counter as shown in Figure 3. In order to minimize chip space and computation time, short fixed point representation of numbers are used. Considering the structure of DSP slices and block RAMs, we choose the data presentation in our implementation, as follows. The data format of inputs that are values of pixels $p[x][y]$ are 1bit binary number. The data format of $\cos\theta$ and $\sin\theta$ is 16bit fixed point number, which consists of 1bit sign, 1bit integer and 14bit fraction based on two's complement. On the other hand, the data format of ρ is 10bit two's complement integer. The data format of the voted value is 18bit integer. Namely, the number of the vote is at most $2^{18} - 1$. Since the range of the value of θ is 0 to 180, the data format of θ is 8bit integer.

IV. Experimental Results

We have implemented the proposed architecture for Hough transform and evaluated it on the Xilinx Virtex-6 FPGA XC6VLX240T-1. Table I shows the experimental results using Xilinx ISE 13.4. In the implementation, to reduce the delay of the circuit, some pipeline registers are inserted into between circuit elements. To compute $y \sin\theta$ for $(1 \leq \theta \leq 90)$ in the first row, i.e., $y = -\frac{n}{2} + 1$, in advance, 94 clock cycles are necessary. It takes 3 clock cycles to compute the values of ρ for given x and the precomputed $y \sin\theta$. Also, 4 clock cycles are necessary to vote to the Hough space. Moreover, to perform vote for each V_θ, the number of clock cycles necessary to move data from the leftmost register to the rightmost register in Figure 3 is 91. Since all of the points in the binary image are input into our architecture, the voting operations are performed for an $n \times n$ image in $n^2 + 192$ clock cycles, i.e., $\frac{n^2+192}{247.525}\mu s$. After voting, $\sqrt{2}n + 187$ clock cycles, i.e., $\frac{\sqrt{2}n+187}{247.525}\mu s$ are necessary to output identified straight lines with 3×3 maximum filters. Therefore, in total, $n^2 + \sqrt{2}n + 379$ clock cycles, i.e., $\frac{n^2+\sqrt{2}n+379}{247.525}\mu s$ are necessary to perform the Hough transform for an $n \times n$ image. If the size of an input image is 512×512, our circuit performs in $1.065ms$.

Table I
Performance evaluation of the proposed architecture for Hough transform

DSP48E1 slices (out of 768)	90 (11.1%)
18Kbit block RAMs (out of 832)	181 (21.7%)
Slices (out of 301440)	40487 (13%)
Clock frequency [MHz]	247.525

For the purpose of estimating the speed up of our FPGA implementation, we have also implemented a conventional software approach of Hough transform using GNU C. We have used Intel Xeon X7460 running in 2.66GHz and 128GB memory to run the sequential algorithm for Hough transform. For the image shown in Figure 1(b) of size 512×512, the software implementation performs the Hough transform in $41.408ms$. On the other hand, our circuit performs in $1.065ms$. Therefore, our FPGA implementation attains a speed-up factor of more than 38 over the sequential implementation on the CPU.

There are a number of literatures reported to implement Hough transform for lines using the FPGA shown in Section I. Performances such as device, logic blocks, DSP slices, frequency and throughput are compared in Table II. It is difficult to directly compare to other works because utilized

FPGAs and supported size of images differ. Considering the throughput, however, it is clear that the performance of our FPGA implementation is better than that of other works. In addition, although the new architecture takes more time than our previous work to perform Hough transform, the number of DSP slices are less than our previous work, and the result is filtered.

Table II
COMPARISON WITH RELATED WORKS FOR HOUGH TRANSFORM

	Karabernou [16]	Deng [17]
Device	XC4010EPC84	XC4010XL
Logic blocks	205 CLBs	333 CLBs
DSP slices	—	—
Frequency	23.166MHz	40MHz
Throughput	10.368Mpixel/s	0.623Mpixel/s
	Lee [18]	previous work [12]
Device	Virtex 4	XC6VLX240T-1
Logic blocks	314 CLBs	14493 Slices
DSP slices	—	178 DSP48E1s
Frequency	132MHz	245.519MHz
Throughput	32.768Mpixel/s	245.428Mpixel/s
	This work	
Device	XC6VLX240T-1	
Logic blocks	40487 Slices	
DSP slices	90 DSP48E1s	
Frequency	247.525MHz	
Throughput	246.219Mpixel/s	

V. CONCLUSIONS

We have presented a new architecture of the Hough transform for the straight lines using DSP slices and block RAMs in the Virtex-6 Family FPGA. Partitioning the parameter space to vote, the 180 voting operations are performed in parallel with 91 DSP48E1s and 181 18Kbit block RAMs. We have implemented our architecture on the Virtex-6 Family FPGA XC6VLX240T-1. The experimental results show that this implementation runs in 247.525MHz and given a binary image of size $n \times n$, our circuit can perform in $n^2 + \sqrt{2}n + 379$ clock cycles, i.e., $\frac{n^2+\sqrt{2}n+379}{247.525}\mu s$.

REFERENCES

[1] Xilinx Inc., *Virtex-6 FPGA DSP48E1 Slice User Guide (v1.3)*, 2011.

[2] J. L. Bordim, Y. Ito, and K. Nakano, "Accelerating the CKY parsing using FPGAs," *IEICE Transactions on Information and Systems*, vol. E86-D, no. 5, pp. 803–810, May 2003.

[3] ——, "Instance-specific solutions to accelerate the CKY parsing for large context-free grammars," *International Journal on Foundations of Computer Science*, pp. 403–416, 2004.

[4] Y. Ito and K. Nakano, "Low-latency connected component labeling using an FPGA," *International Journal on Foundations of Computer Science*, pp. 405–426, 2010.

[5] ——, "A new FM screening method to generate cluster-dot binary images using the local exhaustive search with FPGA acceleration," *International Journal on Foundations of Computer Science*, pp. 1373–1386, 2008.

[6] ——, "Efficient exhaustive verification of the Collatz conjecture using DSP blocks of Xilinx FPGAs," *International Journal of Networking and Computing*, vol. 1, no. 1, pp. 49–62, 2011.

[7] Y. Ito, K. Nakano, and S. Bo, "The parallel FDFM processor core approach for CRT-based RSA decryption," *International Journal of Networking and Computing*, vol. 2, no. 1, pp. 56–78, 2012.

[8] K. Nakano and E. Takamichi, "An image retrieval system using FPGAs," *IEICE Transactions on Information and Systems*, vol. E86-D, no. 5, pp. 811–818, May 2003.

[9] K. Nakano and Y. Yamagishi, "Hardware n choose k counters with applications to the partial exhaustive search," *IEICE Trans. on Information & Systems*, 2005.

[10] P. V. C. Hough, "Method and means for recognizing complex patterns," U.S. Patent 3,069,654, 1962.

[11] R. O. Duda and P. E. Hart, "Use of the Hough transformation to detect lines and curves in pictures," *Communications of the ACM*, vol. 15, no. 1, pp. 11–15, 1972.

[12] X. Zhou, N. Tomagou, Y. Ito, and K. Nakano, "Efficient Hough transform on the FPGA using DSP slices and block RAMs," in *Proc. of Workshop on Advances in Parallel and Distributed Computational Models*, 2013, pp. 771–778.

[13] S. Tagzout, K. Achour, and O. Djekoune, "Hough transform algorithm for FPGA implementation," *Signal Processing*, vol. 81, no. 6, pp. 1295–1301, 2001.

[14] H. Bessalah, S. Seddiki, F. Alim, and M. Bencherif, "On line mode incremental Hough transform implementation on Xilinx fpga's," in *Proc. of the 8th conference on Signal, Speech and image processing*, 2008, pp. 176–179.

[15] O. Djekoune and K. Achour, "Incremental Hough transform: an improved algorithm for digital device implementation," *Real-Time Imaging*, vol. 10, no. 6, pp. 351–363, 2004.

[16] S. M. Karabernou and F. Terranti, "Real-time FPGA implementation of Hough transform using gradient and CORDIC algorithm," *Image and Vision Computing*, vol. 23, no. 11, pp. 1009–1017, 2005.

[17] D. D. S. Deng and H. ElGindy, "High-speed parameterisable Hough transform using reconfigurable hardware," in *Proc. of the Pan-Sydeny area workshop on Visual information processing*, vol. 11, 2001, pp. 51–57.

[18] P. Lee and A. Evagelos, "An implementation of a multiplier-less Hough transform on an FPGA platform using hybrid-log arithmetic," in *Proc. of Real-Time Image Processing 2008*, vol. 6811, 2008, pp. 68 110G–1.

[19] Xilinx Inc., *Virtex-4 FPGA User Guide(v2.6)*, 2008.

[20] ——, *Virtex-5 FPGA User Guide(v5.2)*, 2009.

[21] Altera Corp., *Stratix V Device Handbook*, 2012.

[22] Xilinx Inc., *Virtex-6 Family Overview(v2.4)*, 2012.