International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 4, Number 1, pages 174–188, January 2014

Implementations of the Hough Transform on the Embedded Multicore Processors

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

Received: July 27, 2013
Revised: October 26, 2013
Accepted: November 29, 2013
Communicated by Akihiro Fujiwara

Abstract

Embedded multicore processors represented by FPGAs and GPUs have lately attracted considerable attention for their potential computation ability and power consumption. 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. Meanwhile, recent GPUs can be used for general purpose computation. Users can develop parallel programs running on GPUs using programming architecture called CUDA provided by NVIDIA. The main contribution of this paper is to present two implementations of the Hough transform on the FPGA and the GPU. The first idea of the implementations is an efficient usage of DSP slices and block RAMs for FPGAs, and the shared memory for GPUs. The second idea is to partition the voting space in the Hough transform and the voting operation is performed in parallel. The implementation results show that the Hough transform for a 512×512 image with 33232 edge points can be done in 135.75μs and 637.88μs on the FPGA and the GPU, respectively. On the other hand, a conventional CPU implementation runs in 37.10ms. Thus, both implementations achieve a sufficient speed-up.

Keywords: Image processing, Line detection, Hough transform, FPGA, GPU

1 Introduction

Multicore processors are widely used in many application domains such as general purpose computing, digital signal processing, and image processing. In multicore processors, especially embedded multicore processors represented by Field Programmable Arrays (FPGAs) and Graphics Processing Units (GPUs) have lately attracted considerable attention for their potential computation ability and power consumption [2, 5, 31].

An 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 [35]. 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 configured as 36Kbit dual port block RAMs, FIFOs, or two 18Kbit dual port RAMs. In our architecture, it is used as a 1K×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 [1, 6, 14, 16, 17, 18, 19, 29].

On the other hand, recent GPUs, which have many processing units, can be used for general purpose parallel computation. To utilize the powerful computing ability, GPUs are widely used. CUDA (Common Unified Device Architecture) [25] is the architecture for general purpose parallel computation on GPUs. Before the appearance of CUDA, GPUs could only be programmed through graphics API. However, CUDA comes with a software environment that allows developers to use C-like high-level programming language. Using CUDA, we can develop parallel algorithms to be implemented in GPUs. Therefore, many studies have been devoted to implement parallel algorithm using CUDA [7, 8, 13, 20, 24, 28, 32].

Hough transform is a technique to find shapes in images [15]. 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.

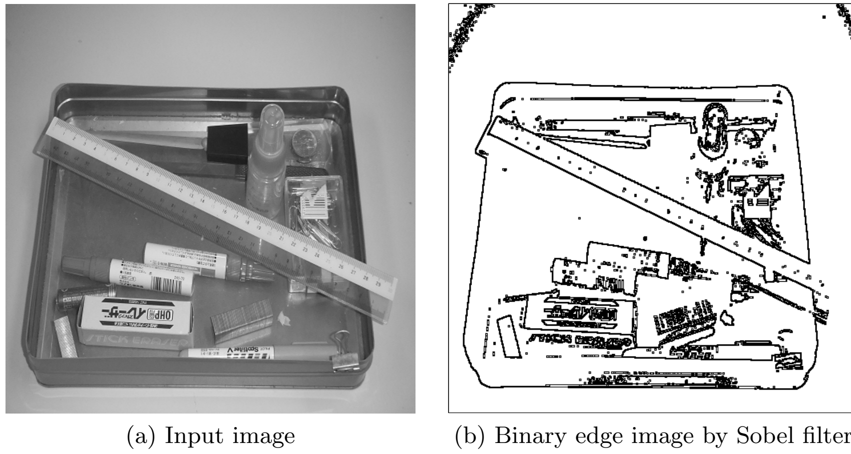
The Hough transform can be used to extract straight lines in a binary image [11]. The idea of this method is to exploit the duality between points of a line and parameters of that line. A point in the image is represented by a curve in the parameter space and lines of collinear points intersect in the 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-dimentional 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)), its 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 2. 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)).

The main contribution of this paper is to present two implementations of the Hough transform on the FPGA and the GPU. Generally, to achieve high performance on the FPGA we have to design in the detailed circuit level considering the features of FPGAs. On the other hand, in the GPU implementation, we can use C-like high-level programming language. We propose two implementations of the Hough transform on the above two devices and their performances are compared. The first idea of the implementations is an efficient usage of DSP slices and block RAMs for FPGAs, and the shared memory for GPUs. The second idea is to partition the voting space in the Hough transform and the voting operation is performed in parallel. We describe the ideas of our FPGA and GPU implementations, as follows.

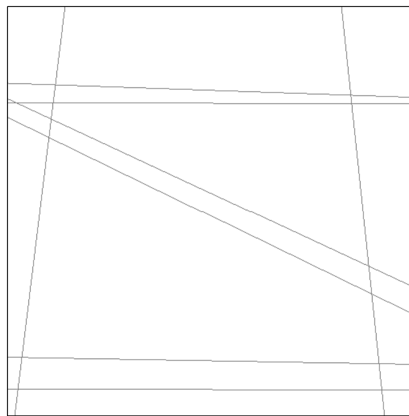
Our new FPGA architecture for the Hough transform fully utilizes embedded DSP slices and block RAMs. Our new idea includes:

Voting Space Partitioning:

Polar coordinate voting space (θ, ρ) is partitioned and arranged into block RAMs. This enables



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



(c) Line detection using the Hough transform

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

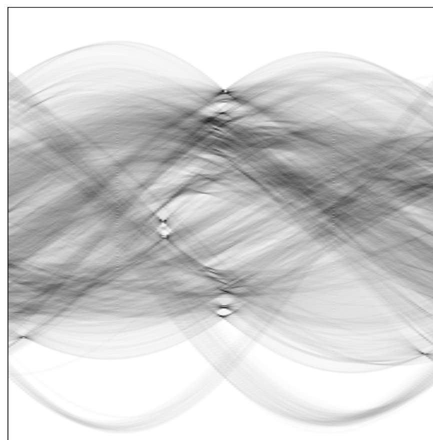


Figure 2: Hough parameter space

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.

Efficient Usage of DSP slices:
DSP slices are used to compute $x \cos\theta$ and $y \sin\theta$ in parallel for each edge pixel (x, y). We compute $x \cos\theta$ and $y \sin\theta$ 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 $\cos\theta$ and $\sin\theta$ by pre-loading them in the DSP slices.

Fully Pipelined Architecture:
We take into account a layout of DSP slices and block RAMs in Virtex-6 FPGA architecture, and design our Hough transform architecture as a fully pipelined one. For example, in the Virtex-6 FPGA XC6VLX240T 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 2 columns to compute $x \cos\theta$ and $y \sin\theta$ each, and uses a pipeline technique to maximize the clock frequency.

Using these ideas, our new architecture for the Hough transform uses 178 DSP48E1 slices and 180 block RAMs with 18Kbits that work in parallel. One of the most important key techniques for accelerating computation using FPGAs is an efficient usage of DSP slices and block RAMs. Nevertheless, as far as we know, there is no previously published work that fully utilizes DSP slices and block RAMs for the Hough transform. Roughly speaking, a conventional sequential implementation performs $180m$ voting operations for m edge points. Our architecture performs voting operations in parallel, and outputs identified straight lines in $m + 97$ clock cycles. Since $180m$ voting operations are performed using 178 DSP48E1 slices, the lower bound of the computing time is m clock cycles. Hence our implementation is close to optimal. We have implemented our new architecture on a Virtex-6 family FPGA XC6VLX240T-1. The circuit runs in 245.519MHz and outputs identified straight lines in $m + 97$ cycles. For example, Figure 1 includes 33232 edge points. Therefore, the circuit can perform the Hough transform in $135.75\mu s$.

In our new GPU implementation, we also partition the voting space and voting operation is performed in parallel. Additionally, we have considered shared memory bank conflict that is one of the important programming issues of the GPU system. We have implemented our new GPU implementation in the NVIDIA GeForce GTX680. For Figure 1, our GPU implementation can perform the Hough transform in $637.88\mu s$. According to the above results, the FPGA implementation can run about 4 times faster than the GPU implementation. However, the GPU implementation attains a speed-up factor of more than 68 over the sequential implementation on the CPU. That is, both our FPGA and GPU implementations achieve a sufficient speed-up.

Many hardware algorithms for FPGA implementation of the Hough transform for lines have been proposed in past. As far as we know, however, there is no published hardware algorithm using embedded DSP slices or multipliers in the FPGA. In the existing researches, instead of circuits of multiplication with DSP slices or multipliers, they introduced incremental Hough transform [4, 10, 30], CORDIC [9, 22], and hybrid-log arithmetic [23] to the computation of the Hough transform. Since most of recent FPGAs produced by principal vendors equip embedded DSP slices [3, 33, 34], one of the most important key techniques for accelerating computation using FPGAs is an efficient usage of DSP slices and block RAMs.

Meanwhile, GPU implementations for the Hough transform have also been proposed. In [12], an initial GPU implementation written by OpenGL is proposed. In this implementation, the computation of the Hough transform is transformed to matrix calculation that can be performed on the GPU. Jošth *et al* proposed a GPU implementation of the Hough transform by CUDA [21]. In this implementation as well as our GPU implementation, the voting space is partitioned into small spaces and they are arranged to the shared memory. However, the shared memory bank conflict has not been considered.

This paper is organized as follows. Section 2 introduces the Hough transform algorithms for lines. We show the FPGA architecture for the Hough transform in Section 3. In Section 4, we briefly explain modern GPU architecture and describe our GPU implementation for the Hough transform. Section 5 shows the experimental results. Finally, Section 6 concludes the paper.

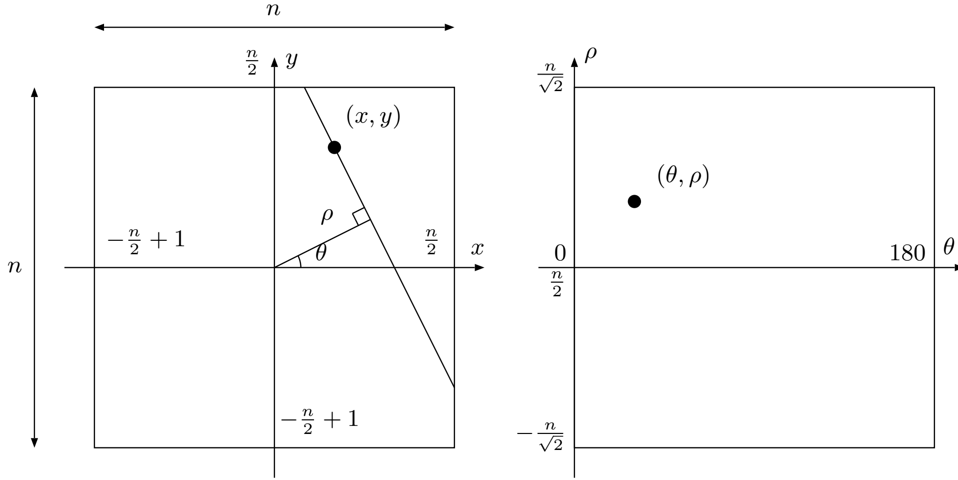


Figure 3: Two dimensional Spaces xy and $\theta\rho$ used in the Hough transform

2 Hough Transform

The main purpose of this section is to review the 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 3. Hence, both coordinates x and y take integers in the range $[-\frac{n}{2}+1, \frac{n}{2}]$.

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:

$$\rho \quad = \quad x\cos\theta + y\sin\theta \quad (0 \leq \theta < 180) \tag{1}$$

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 3. For such a line, 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 for every pixel in the xy-space. Let $(x_0, y_0), (x_1, y_1), \ldots, (x_{k-1}, y_{k-1})$ be the k pixels in xy-space. The Hough transform is spelled out as follows:

[Straight Forward Hough Transform]
for $i \leftarrow 0$ to $k-1$
    for $\theta \leftarrow 0$ to 179
        begin
            $\rho \leftarrow x_k \cos\theta + y_k \sin\theta$
            $v[\theta][\rho] \leftarrow v[\theta][\rho] + 1$
        end
for $\theta \leftarrow 0$ to 179 do in parallel
    for $\rho \leftarrow -\frac{n}{\sqrt{2}}$ to $\frac{n}{\sqrt{2}}$ do in parallel
        output (θ, ρ) if $v[\theta][\rho] \geq$ *threshold*

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_k, y_k), the values of $x_k \cos\theta$ and $y_k \sin\theta$ are computed for $\theta = 0, 1, \ldots, 179$. If $v[\theta][\rho]$ is storing a large value, many points in the k 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

$$\begin{aligned} \rho &= x_k \cos(180-\theta) + y_k \sin(180-\theta) \\ &= -x_k \cos(\theta) + y_k \sin(\theta). \end{aligned} \tag{2}$$

Using Formula (2), 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 $i \leftarrow 0$ to $k-1$ do
  begin
    for $\theta \leftarrow 0$ to 89 do
      begin
        $\rho \leftarrow x_k \cos\theta + y_k \sin\theta$
        $v[\theta][\rho] \leftarrow v[\theta][\rho] + 1$
        output (θ, ρ) if $v[\theta][\rho] = \mathit{threshold}$
      end
    for $\theta \leftarrow 1$ to 90 do
      begin
        $\rho \leftarrow -x\cos(\theta) + y\sin(\theta)$
        $v[180-\theta][\rho] \leftarrow v[180-\theta][\rho] + 1$
        output (θ, ρ) if $v[\theta][\rho] = \mathit{threshold}$
      end
  end

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

3 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 FPGA. We use Xilinx Virtex-6 Family FPGA XC6VLX240T-1 as the target device [37].

3.1 Structure of our architecture for the Hough transform

Figure 4 illustrates our architecture for the Hough transform. We use 178 DSP slices $X_1, X_2, \ldots X_{89}$ and $Y_1, Y_2, \ldots, Y_{89}$. For each θ $(0 \leq \theta \leq 90)$ X_θ and Y_θ compute $x_k \cos\theta$ and $y_k \cos\theta$ for given x_k and y_k, respectively. Since $x_k \cos 0 = x_k$, $x_k \cos 90 = 0$, $y_k \sin 0 = 0$, and $y_k \cos 90 = y_k$, DSP slices X_0, X_{90}, Y_0, and Y_{90} are not necessary. Using an adder and a subtractor for each pair X_θ and Y_θ, $\rho_\theta = x_k \cos\theta + y_k \cos\theta$ and $\rho_{180-\theta} = -x_k \cos\theta + y_k \cos\theta$ 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 $v[\theta][\rho]$.

To minimize the delay between registers, DSP slices $X_1, \ldots, X_{90}$ are connected in a pipeline fashion as illustrated in Figure 4. Each X_θ has a register to store the value of x_k. In every clock cycle, the value is transferred from X_θ to $X_{\theta+1}$. Similarly, DSP slices $Y_0, Y_1, \ldots, Y_{90}$ are connected in a pipeline fashion.

Figure 5 illustrates two DSP slices X_θ and Y_θ with an adder and subtractor to compute ρ. In X_θ, the value of x_k 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_k and $\cos\theta$ is computed in a multiplier of the DSP slice X_θ. Similarly, the value of $\sin\theta$ used in Y_θ is a fixed value and the product of y_k and $\sin\theta$ is computed in a multiplier of the DSP slice Y_θ.

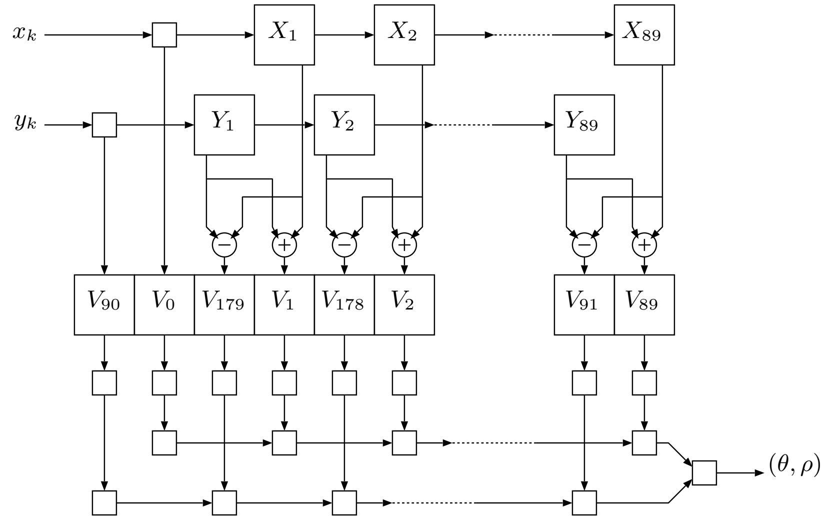


Figure 4: The outline of our FPGA architecture for the Hough transform

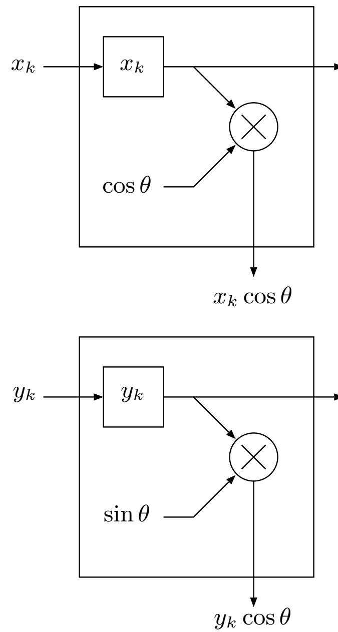


Figure 5: Two DSP slices X_θ and Y_θ with an adder and subtractor to compute ρ

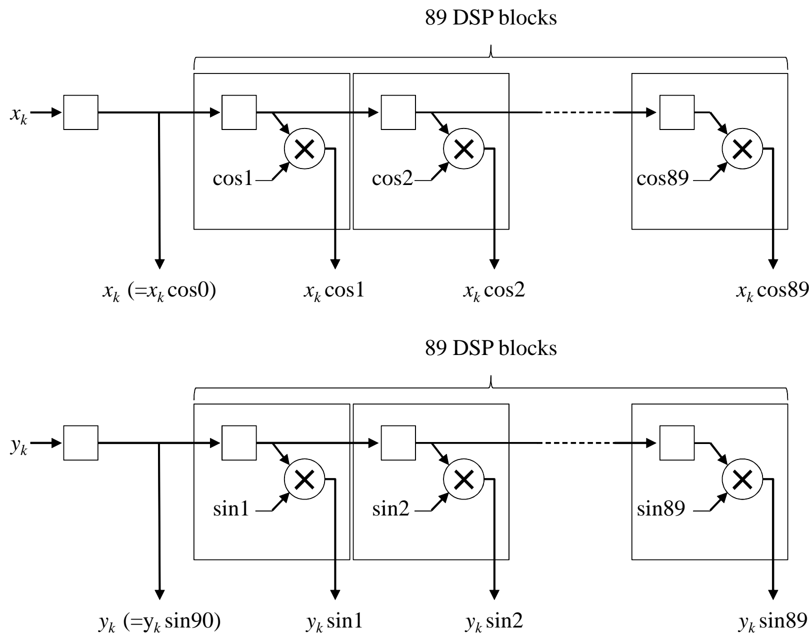


Figure 6: Pipeline architecture to compute $x_k \cos\theta$ and $y_k \sin\theta$ with DSP slices

In the Virtex-6 FPGA XC6VLX240T, that is our target device, has 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 2 columns to compute $x_k \cos\theta$ and $y_k \sin\theta$ each, and uses a pipeline technique to maximize the clock frequency (Figure 6).

Figure 7 illustrates the architecture of V_θ using a block RAM. A block RAM in the FPGA is dual port architecture. Xilinx Virtex-6 Family has 18Kbit dual-port block RAMs, which have two sets of ports operated independently. Two sets of ports are:

Port Set A *ADDRA* (ADDRess A), *DOA* (Data Output A), *DIA* (Data Input A), and

Port Set B *ADDRB* (ADDRess B), *DOB* (Data Output B), *DIB* (Data Input B).

Let $M[i]$ denote a data of address i of the block RAM. In read operation of Port Set A, $M[ADDRA]$ is output from *DOA* after the rising clock edge. In write operation of Port Set A, the data given to *DIA* is written in $M[ADDRA]$ at the rising clock edge. Read/write operations of Port Set B are the same as Port Set A. Port Set A and Port Set B work independently. In the block RAMs in the target device of this work, read/write operations can be configured as either RF (Read First) mode or WF (Write First) mode. In the RF mode, if reading and writing operations are performed to the same address, reading operation is performed before the reading operation. Hence the reading data is the data before writing data. On the other hand, in the WF mode, since the writing performed before the reading, the reading data is the updated data. However, when a dual port is used, there is a restriction that if read and write operations to the same address are performed for each port, the setting of block RAMs must be RF [36].

We use the block RAM to store the values of $v[\theta][\rho]$ $(-\frac{n}{\sqrt{2}} < \rho \leq \frac{n}{\sqrt{2}})$. 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

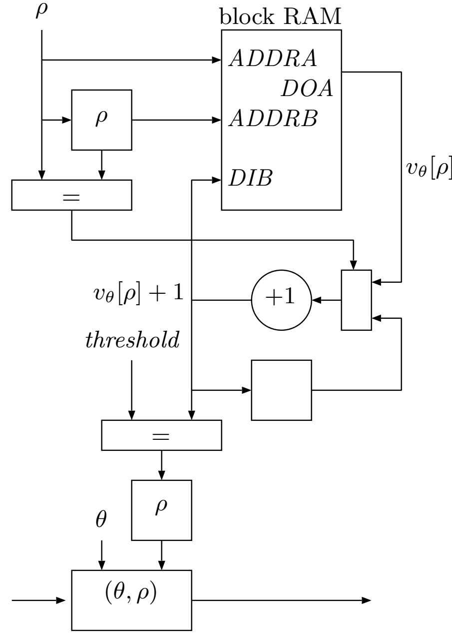


Figure 7: A block RAM V_θ to store $v[\theta][\rho]$

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.

In the same time, a comparator is used to determine if $v_\theta[\rho] + 1 = threshold$. If so, the value of ρ is written in a register. After that, a pair (θ, ρ) is written into a next register. The pair (θ, ρ) represents a probable line. It moves toward the output of the circuit using series of shift registers one by one shown in Figure 4. In order to reduce the number of clock cycles necessary to move data to the output, we use two series of shift registers. One is used for output data of $V_0, \ldots, V_{89}$. The other is used for output data of $V_{90}, \ldots, V_{179}$. Therefore, the number of clock cycles necessary to move data to the output is reduced to at most 90 clock cycles.

3.2 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 our work, 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 pairs of coordinates x_k and y_k are 10bit two's complement integer each. Also, 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.

4 Our GPU implementation for the Hough transform

This section describes our GPU Implementation for the Hough transform. We use GeForce GTX 680 as the target GPU and CUDA as the development environment by NVIDIA.

4.1 Compute Unified Device Architecture (CUDA)

Graphics Processing Units (GPUs) can achieve a high computational throughput due to their large number of processing cores and different memory spaces. All the processing cores are organized into several streaming multi-core processors as shown in Figure 8. For fully utilizing all the processing cores of a GPU, numerous threads are required. Compute Unified Device Architecture (CUDA) [25] organizes these threads into a large *grid* of *thread blocks*. Each thread block contains a number of threads which can be executed on an assigned streaming multi-core processor. Threads of a thread block are organized into several *warps* and each warp contains 32 threads. At a time, only a warp of a thread block can be executed by the assigned streaming multi-core processor concurrently.

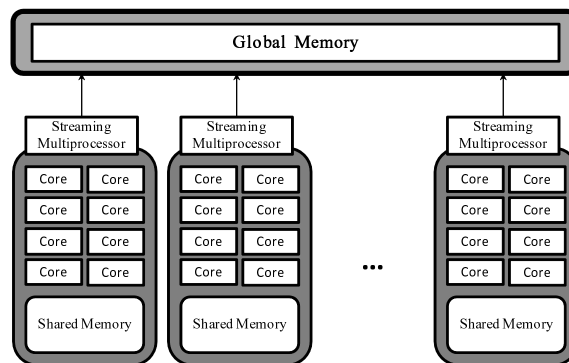


Figure 8: CUDA hardware architecture

CUDA uses two types of memories in the NVIDIA GPUs: *the global memory* and *the shared memory* [27]. The global memory is implemented as an off-chip DRAM of the GPU, and has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The efficient usage of the global memory and the shared memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the bank conflict* of the shared memory access [24, 26]. Figure 9 illustrates the CUDA hardware architecture. The shared memory is divided into 32 equally-sized modules of 32-bit width, called banks. It means that, in the shared memory, the successive 32-bit words are assigned to successive banks. To achieve maximum throughput, concurrent threads of a thread block should access different banks, otherwise, bank conflicts will occur.

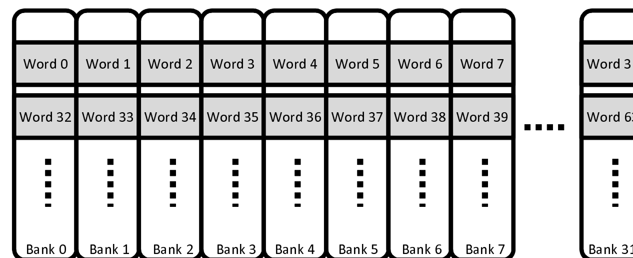


Figure 9: Structure of the shared memory

4.2 GPU implementation

The main idea of our GPU implementation it also to use the voting space partitioning shown in the above. Another idea is to avoid bank conflict on the shared memory. In the followings, the details of our GPU implementation are shown.

The voting space (θ, ρ) of the Hough transform is partitioned for each θ and the voting procedure for each partition is performed in parallel. In our GPU implementation, a thread block is assigned to each θ and threads in the thread block concurrently vote for input edge points. Partitioning the voting space for each θ, each voting space can be stored in the shared memory. Therefore, the voting procedure is performed to the voting space in the shared memory. In our GPU implementation, the voting space is partitioned into 180 spaces and 180 thread blocks vote in parallel.

In each thread block, the values of the trigonometric functions $\cos\theta$ and $\sin\theta$ are initially computed since the values are common for every thread in a thread block. After that, threads read coordinates of input points stored in the global memory. Using the values of $\cos\theta$ and $\sin\theta$, Eq. 1 for each point is computed and vote to the voting space in parallel. In the parallel voting procedure, some threads may vote to the same ρ simultaneously. To avoid it, we use the atomic add operation supported by CUDA [27].

Additionally, to reduce the bank conflict of the shared memory, the voting is distributed to the different banks. We use 32 voting spaces of the same size and each space is assigned to a bank. Every thread in a warp votes to the different banks without the bank conflict. After voting, to merge the results of the voting in the different banks the sums of the values in the voting spaces are computed. In the sum computation, if it is performed in parallel straightforwardly, the bank conflict occurs in the read operation (Figure 10(a)). Therefore, to avoid the bank conflict in each warp, the order of each sum computation is shifted thread by thread shown in Figure 10(b). As a result, the parallel voting can be performed without the bank conflict. After that, the results of the voting are stored back to the global memory.

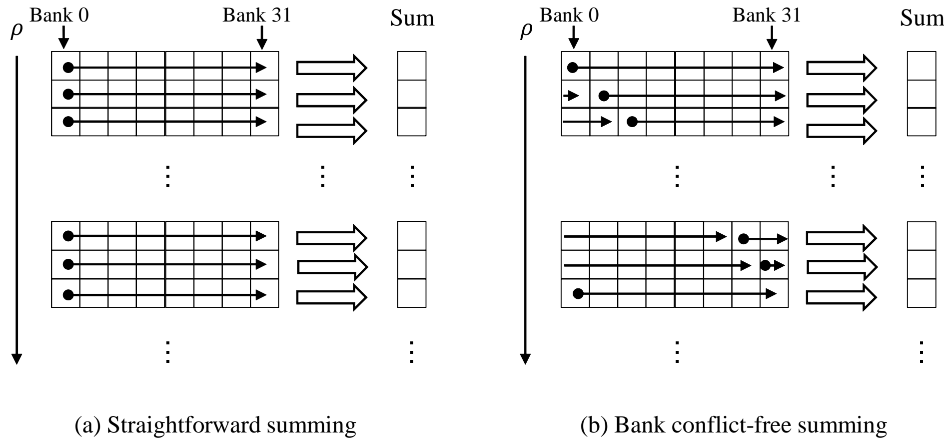


Figure 10: Summing the voting spaces in the different banks on the shared memory

5 Experimental Results

We have implemented and evaluated our proposed methods of the Hough transform on the FPGA and the GPU. For the purpose of estimating the speed up of our implementations, we have also implemented a conventional software approach of the Hough transform using GNU C. We have used Intel Xeon X7460 running in 2.66GHz and 128GB memory to run the sequential algorithm for the Hough transform. For the image shown in Figure 1(b) that includes 33232 edge points, the software implementation can perform the Hough transform in 37.10ms. If the input image is worst case in

terms of the computing time, that is, if all the points of an image of size $512 \times 512 (= 262144)$ are edge points, it takes $359.27ms$ to complete to output the results.

In the evaluation of our FPGA implementation, we have used the Xilinx Virtex-6 FPGA XC6VLX240T-1. Table 1 shows the experimental results using Xilinx ISE 13.1. In the implementation, to reduce the delay of the circuit, some pipeline registers are inserted into between circuit elements. It takes 3 clock cycles to compute the values of ρ for given x_k and y_k. Also, 4 clock cycles are necessary to output a pair (θ, ρ) that represents a probable line. Moreover, the number of clock cycles necessary to move data to the output is reduced to at most 90 clock cycles. Therefore, this circuit can output identified straight lines represented by (θ, ρ) in $m + 97$ cycles, i.e., $\frac{m+97}{245.519}\mu s$. For example, Figure 1(b) includes 33232 edge points. Therefore, the circuit can perform the Hough transform in $135.75\mu s$. Also, if all the points of an image of size $512 \times 512 (= 262144)$ are edge points, it takes $1068.11\mu s$ to complete to output the results. Of course, it is not possible that all points are edge points, however, this fact guarantees that our Hough transform implementation for any 512×512 image terminates in less than $1068.11\mu s$. Therefore, our FPGA implementation attains a speed-up factor of more than 300 over the sequential implementation on the CPU.

Table 1: Performance evaluation of the proposed architecture for the Hough transform

DSP48E1 slices (out of 768)	178 (23.1%)
18Kbit block RAMs (out of 832)	180 (21.6%)
Slices (out of 301440)	14493 (4.81%)
Clock frequency [MHz]	245.519

There are a number of literatures reported to implement the Hough transform for lines using the FPGA shown in Section 1. Performances such as device, logic blocks, DSP slices, frequency and throughput are compared in Table 2. 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.

Table 2: Comparison with related works for the Hough transform using FPGAs

	Karabernou [22]	Deng [9]
Device	XC4010EPC84	XC4010XL
Logic blocks	205 CLBs	333 CLBs
DSP slices	—	—
Frequency	23.166MHz	40MHz
Throughput	10.368Mpixel/s	0.623Mpixel/s

	Lee [23]	This work
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

On the other hand, in the evaluation of our GPU implementation, we have used an NVIDIA GeForce GTX680 with 1536 processing cores (8 Streaming Multiprocessors which have 192 processing cores each) running in 1.006GHz and 4GB memory. We select the numbers of thread blocks and threads in each block are 180 and 1024, respectively. For Figure 1(b), our GPU implementation can perform the Hough transform in $637.79\mu s$. Also, if all the points of an image of size $512 \times 512 (= 262144)$ are edge points, it takes $4348.83\mu s$ to complete to output the results.

According to the above results, the FPGA implementation can run about 4 times faster than the

GPU implementation. However, the GPU implementation attains a speed-up factor of more than 68 over the sequential implementation on the CPU. That is, both our FPGA and GPU implementations achieve a sufficient speed-up. Regarding the power consumption, it is not easy to measure and compare the power consumption of the FPGA, GPU, and the CPU. However, Thomas et al. show the comparison of them about the performance and the power consumption for random number generation [31]. The problem is different and utilized devices are slightly older, but it seems to be a good indicator to consider the power consumption. According to the paper, the power consumption of the GPU and CPU is almost the same and six times more than that of the FPGA. Therefore, the performance per power of the FPGA is also much better than that of the GPU and CPU.

6 Conclusions

We have proposed two implementations of the Hough transform that identifies straight lines on the FPGA and the GPU and their performances have been compared. The first idea of the implementations is an efficient usage of DSP slices and block RAMs for FPGAs, and the shared memory for GPUs. The second idea is to partition the voting space in the Hough transform and the voting operation is performed in parallel. The implementation results show that the Hough transform for a 512×512 image with 33232 edge points can be done in $135.75\mu s$ and $637.88\mu s$ on the FPGA and the GPU, respectively. On the other hand, a conventional CPU implementation runs in $37.10ms$. Thus, both implementations achieve a sufficient speed-up.

References

[1] Yuki Ago, Yasuaki Ito, and Koji Nakano. An FPGA implementation for neural networks with the FDFM processor core approach. *International Journal of Parallel, Emergent and Distributed Systems*, 28(4):308–320, 2013.

[2] Syed Zahid Ahmed, Gilles Sassatelli, Lionel Torres, and Laurent Rougé. Survey of new trends in industry for programmable hardware: FPGAs, MPPAs, MPSoCs, structured ASICs, eFPGAs and new wave of innovation in FPGAs. In *Proc. of the 2010 International Conference on Field Programmable Logic and Applications*, pages 291–297. IEEE, 2010.

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

[4] 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*, pages 176–179, 2008.

[5] Geoffrey Blake, Ronald G Dreslinski, and Trevor Mudge. A survey of multicore processors. *Signal Processing Magazine, IEEE*, 26(6):26–37, 2009.

[6] J. L. Bordim, Y. Ito, and K. Nakano. Accelerating the CKY parsing using FPGAs. *IEICE Transactions on Information and Systems*, E86-D(5):803–810, May 2003.

[7] Daniel Castaño-Díez, Dominik Moser, Andreas Schoenegger, Sabine Pruggnaller, and Achilleas S. Frangakis. Performance evaluation of image processing algorithms on the GPU. *Journal of Structual Biology*, 164:153–160, 2008.

[8] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, and Kevin Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[9] Dixon D. S. Deng and Hossam ElGindy. High-speed parameterisable Hough transform using reconfigurable hardware. In *Proc. of the Pan-Sydeny area workshop on Visual information processing*, volume 11, pages 51–57, 2001.

[10] O. Djekoune and K. Achour. Incremental Hough transform: an improved algorithm for digital device implementation. *Real-Time Imaging*, 10(6):351–363, 2004.

[11] Richard O. Duda and Peter E. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

[12] James Fung, Steve Mann, and Chris Aimone. OpenVIDIA: parallel GPU computer vision. In *Proceedings of the 13th annual ACM international conference on Multimedia*, pages 849–852, 2005.

[13] Juan Gómez-Luna, José María González-Linares, José Ignacio Benavides, Emilio L. Zapata, and Nicolás Guil Mata. Load balancing versus occupancy maximization on graphics processing units: The generalized hough transform as a case study. *International Journal of High Performance Computing Applications*, 25(2):205–222, 2011.

[14] M.C. Herbordt, T. VanCourt, Yongfeng Gu, B. Sukhwani, A. Conti, J. Model, and D. DiSabello. Achieving high performance with FPGA-based computing. *Computer*, 40(3):50–57, 2007.

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

[16] Yasuaki Ito and Koji Nakano. 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*, pages 1373–1386, 2008.

[17] Yasuaki Ito and Koji Nakano. Low-latency connected component labeling using an FPGA. *International Journal on Foundations of Computer Science*, pages 405–426, 2010.

[18] Yasuaki Ito and Koji Nakano. Efficient exhaustive verification of the Collatz conjecture using DSP blocks of Xilinx FPGAs. *International Journal of Networking and Computing*, 1(1):49–62, 2011.

[19] Yasuaki Ito, Koji Nakano, and Song Bo. The parallel FDFM processor core approach for CRT-based RSA decryption. *International Journal of Networking and Computing*, 2(1):56–78, 2012.

[20] Yasuaki Ito, Kouhei Ogawa, and Koji Nakano. Fast ellipse detection algorithm using Hough transform on the GPU. In *Proc. of International Workshop on Challenges on Massively Parallel Processors (CMPP)*, pages 313–319, December 2011.

[21] Radovan Jošth, Markéta Dubská, Adam Herout, and Jiří Havel. Real-time line detection using accelerated high-resolution Hough transform. In *Proceedings of the 17th Scandinavian conference on Image analysis*, pages 784–793, 2011.

[22] Si Mahmoud Karabernou and Fayçal Terranti. Real-time FPGA implementation of Hough transform using gradient and CORDIC algorithm. *Image and Vision Computing*, 23(11):1009–1017, 2005.

[23] Peter Lee and Alexiadis Evagelos. An implementation of a multiplierless Hough transform on an FPGA platform using hybrid-log arithmetic. In *Proc. of Real-Time Image Processing 2008*, volume 6811, pages 68110G–1, 2008.

[24] Duhu Man, Kenji Uda, Hironobu Ueyama, Yasuaki Ito, and Koji Nakano. Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs. *International Journal of Networking and Computing*, 1(2):260–276, 2011.

[25] NVIDIA Corp. CUDA ZONE. http://developer.nvidia.com/category/zone/cuda-zone.

[26] NVIDIA Corp. *CUDA C Best Practice Guide Version 5.0*, 2012.

[27] NVIDIA Corp. *NVIDIA CUDA C Programming Guide Version 5.0*, 2012.

[28] Kohei Ogawa, Yasuaki Ito, and Koji Nakano. Efficient Canny edge detection using a GPU. In *Proceedings of International Workshop on Advances in Networking and Computing*, pages 279–280, 2010.

[29] Prasanna Sundararajan. High performance computing using FPGAs. *Xilinx White Paper: FPGAs*, WP375 (v1.0):1–15, 2010.

[30] Samir Tagzout, Karim Achour, and Oualid Djekoune. Hough transform algorithm for FPGA implementation. *Signal Processing*, 81(6):1295–1301, 2001.

[31] David B. Thomas, Lee Howes, and Wayne Luk. A comparison of CPUs, GPUs, FPGAs, and massively parallel processor arrays for random number generation. In *Proc. of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 63–72, 2009.

[32] Akihiro Uchida, Yasuaki Ito, and Koji Nakano. An efficient GPU implementation of ant colony optimization for the traveling salesman problem. In *Proc. of International Conference on Networking and Computing*, pages 94–102, Dec. 2012.

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

[34] Xilinx Inc. *Virtex-5 FPGA User Guide(v5.2)*, 2009.

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

[36] Xilinx Inc. *Virtex-6 FPGA Memory Resources User Guide (v1.6)*, 2011.

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