Constant-Time Hough Transform On A 3D Reconfigurable Mesh Using Fewer Processors

Yi Pan

Department of Computer Science
University of Dayton, Dayton, OH 45469-2160

Abstract. The Hough transform has many applications in image processing and computer vision, including line detection, shape recognition and range alignment for moving imaging objects. Many constant-time algorithms for computing the Hough transform have been proposed on reconfigurable meshes [1, 5, 6, 7, 9, 10]. Among them, the ones described in [1, 10] are the most efficient. For a problem with an $N \times N$ image and an $n \times n$ parameter space, the algorithm in [1] runs in a constant time on a 3D $nN \times N \times N$ reconfigurable mesh, and the algorithm in [10] runs in a constant time on a 3D $n^2 \times N \times N$ reconfigurable mesh. In this paper, a more efficient Hough transform algorithm on a 3D reconfigurable mesh is proposed. For the same problem, our algorithm runs in constant time on a 3D $n \log^2 N \times N \times N$ reconfigurable mesh.

1 Introduction

The Hough transform of binary images is an important problem in image processing and computer vision and has many applications such as line detection, shape recognition and range alignment for moving imaging objects. It is a special case of the Radon transform which deals with gray-level images. The Radon transform of a gray-level image is a set of projections of the image taken from different angles. Specifically, the image is integrated along line contours defined by the equation:

$$\{(x, y) : x \cos(\theta) + y \sin(\theta) = \rho\}, \tag{1}$$

where θ is the angle of the line with respect to positive x-axis and ρ is the (signed) distance of the line from the origin.

The computation of the Radon and Hough transforms on a sequential computer can be described as follows. We use an $n \times n$ array to store the counts which are initialized to zero. For each of the black pixels in an $N \times N$ image and for each of the n values of θ, the value of ρ is computed based on (1) and the sum corresponding to the particular (θ, ρ) is accumulated as given in the following algorithm. In the algorithm, ρ_{res} is the resolution along the ρ direction; and gray-value(x, y) is the intensity of the pixel at location (x, y).

for each black pixel at location (x, y) in an image do
for $\theta = \theta_0, \theta_1, ..., \theta_{n-1}$ do

```
begin
(* parameter computation *)
ρ := (x cos θ + y sin θ)/ρ_res
(* accumulation *)
sum[ θ, ρ ] := sum[ θ, ρ ] + gray-value(x, y)
end;
```

Obviously, for an $N \times N$ image, and n values of θ, a sequential computer calculates the Radon (Hough) transform in $O(nN^2)$ time since the number of black pixels is $O(N^2)$. The computation time is too long for many applications, especially for real-time applications, as N and n can be very large.

Recently, several constant-time algorithms for computing the Hough transform have been proposed for the reconfigurable mesh model [1, 5, 6, 7, 9, 10]. Among them, the ones described in [1, 10] are the most efficient. For a problem with an $N \times N$ image and an $n \times n$ parameter space, the algorithm in [1] runs in a constant time on a 3D $nN \times N \times N$ reconfigurable mesh, and the algorithm in [10] runs in a constant time on a 3D $n^2 \times N \times N$ reconfigurable mesh. Besides computing Hough transform, the algorithm in [10] can also compute the Radon transform in a constant time using the same number of processors. In this paper, a more efficient Hough transform algorithm for binary images on a 3D reconfigurable mesh is proposed. For the same problem, our algorithm runs in constant time on a 3D $n \log^2 N \times N \times N$ reconfigurable mesh. We also show that the algorithm can be adapted to computing the Radon transform of gray-level images in constant time on a 3D $n \log^3 N \times N \times N$ reconfigurable mesh. Clearly, our algorithm uses the fewest number of processors to achieve the same objectives and is the most efficient one compared to existing results in the literature [1, 5, 6, 7, 9, 10].

2 The Computational Model

A reconfigurable mesh consists of a bus in the shape of a mesh which connects a set of processors, but which can be split dynamically by local switches at each processor. By setting these switches, the processors partition the bus into a number of subbuses through which the processors can then communicate. Thus the communication pattern between processors is flexible, and moreover, it can be adjusted during the execution of an algorithm. The reconfigurable mesh has begun to receive a great deal of attention as both a practical machine to build, and a good theoretical model of parallel computation.

A 2D reconfigurable mesh consists of an $N_1 \times N_2$ array of processors which are connected to a grid-shaped reconfigurable bus system. Each processor can perform arithmetic and logical operations and is identified by a unique index (i, j), $0 \leq i < N_1$, $0 \leq j < N_2$. The processor with index (i, j) is denoted by $PE(i, j)$. Each processor can communicate with other processors by broadcasting values on the bus system. We assume that the bus width is $O(\log N)$ and each broadcast takes $O(1)$ time. The arithmetic operations in the processors are

performed on $O(\log N)$ bit words. Hence, each processor can perform one logical and arithmetic operation on $O(1)$ words in unit time.

A high dimensional reconfigurable mesh can be defined similarly. For example, a processor in a 3D $N_1 \times N_2 \times N_3$ reconfigurable mesh is identified by a unique index (i, j, k), $0 \leq i < N_1$, $0 \leq j < N_2$, $0 \leq k < N_3$. The processor with index (i, j, k) is denoted by $PE(i, j, k)$. Within each processor, 6 ports are built with every two ports for each of the three directions: i-direction, j-direction, and k-direction. In each direction, a single bus or several subbuses can be established.

A subarray is denoted by replacing certain indices by $*$'s. For example, the ith row of processors in a 2D reconfigurable mesh is represented by $ARR(i, *)$. Similarly, $ARR(*, j, k)$, $0 \leq j < N_2$, $0 \leq k < N_3$, is a 1-dimensional subarray in a 3D reconfigurable mesh, and these $j \times k$ subarrays can execute algorithms independently and concurrently. Finally, a memory location L in $PE(i, j, k)$ is denoted as $L(i, j, k)$.

3 The Constant-Time Algorithm

In this section, we propose a constant time algorithm for computing the Hough transform of an $N \times N$ image on a 3D $n \log^2 N \times N \times N$ reconfigurable mesh. In the following discussion, we partition the image into parallel bands and these bands run at an angle of θ with respect to the horizontal axis, and then sum the pixel values contained in each band. If a pixel is contained in two or more bands, then it will be counted in only the band that contains its center. If the center of a pixel lies on the boundary between two bands, then it is counted only in the uppermost of the two bands. For example, we have computed a $\pi/4$ angle Hough transform for an 8×8 pixel array in Figure 1, where the bands are one pixel-width wide. Clearly, there are 10 different ρ's in the figure. In the figure, the number of 1-pixels contained in each band is displayed at the upperright end of the band. For a particular angle θ and a particular distance ρ, only the values of the pixels lying in the band specified by θ and ρ need to be added together.

In our algorithm, since all pixels in an image are used as the input, we can exploit easily the geometric features and relations of pixels in an image. Clearly, for a given pair of ρ and θ, we do not need to consider all the pixels in an image. Instead, only those pixels that are centered in the band will contribute to the count value of that band. In this way, we can improve the efficiency of the algorithm during computation. Before we describe the algorithms, several observations will be made. In order to speedup the computation, we need to connect together all processors which have computed and stored the same ρ values. In order to do so, we rely on several results obtained in [8]. Although the results are made for θ_i such that $0 \leq \theta_i \leq \pi/4$, they can easily generalized to other θ values. In the following discussion, we assume that $0 \leq \theta_i \leq \pi/4$.

Lemma 1. For any j, $0 \leq j \leq N-1$, the ρ-distances satisfy $\rho_{i,j} \leq \rho_{i+1,j}$ for $0 \leq i \leq N-2$. It can also be shown that no more than two consecutive values of ρ in row j can be equal.
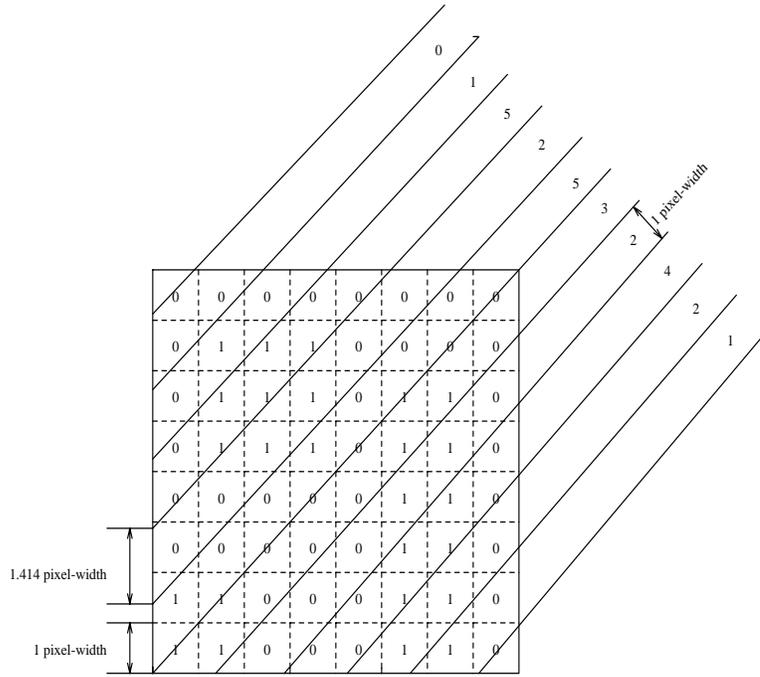
0

1

5

2

5

3

1 pixel-width

2

4

2

1

0	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0
0	1	1	1	0	1	1	0
0	1	1	1	0	1	1	0
0	0	0	0	0	1	1	0
0	0	0	0	0	1	1	0
1	1	0	0	0	1	1	0
1	1	0	0	0	1	1	0

1.414 pixel-width

1 pixel-width

Fig. 1. Parallel Bands for $\theta = \pi/4$ in an 8×8 Image.

Lemma 2. The values of ρ computed using equation (1) by two consecutive processors in a row j, differ by at most 1. More formally, for all i, j, $0 \leq j \leq N-1$ and $0 \leq i \leq N-2$, $0 \leq \rho_{i+1,j} - \rho_{i,j} \leq 1$.

Lemma 3. For all values of i, j, $0 \leq i, j \leq N-2$, $\rho_{i,j} \neq \rho_{i+1,j+1}$.

Lemma 4. If $\rho_{i,j} = \rho_{i,j+2}$ for $0 \leq i \leq N-1$ and $0 \leq j \leq N-3$, then $\rho_{i,j} = \rho_{i,j+1} = \rho_{i,j+2}$. If two ρ-values in a column i are equal and they are placed two rows apart, then the ρ-value in between should have the same value.

The above lemmas will be used in our algorithm to connect related processors together to calculate the number of black pixels in the bands. The following result is also used in our algorithm to compute binary sums efficiently and is due to [?].

Lemma 5. Let a binary sequence of length S stored in the first row of a 2D $S \times \log^2 S$ reconfigurable mesh, the sum of the binary sequence can be computed in a constant time on the array.

For the Radon transform, we need the following result to add integer values. The detailed proof of the lemma is described in [11].

Lemma 6. Given S ($\log S$)-bit integers, these numbers can be added in $O(1)$ time on a 2D $S \times \log^3 S$ reconfigurable mesh.

Assume that the reconfigurable mesh used here is configured as a 3D $n * \log^2 N \times N \times N$ array. The 3D mesh is divided into n layers along the i direction with each layer having a 3D $\log^2 N \times N \times N$ array as shown in Figure 2. Each layer

is responsible for computing the Hough transform for a particular projection (corresponding to a θ value). Now, we formally describe the algorithm.
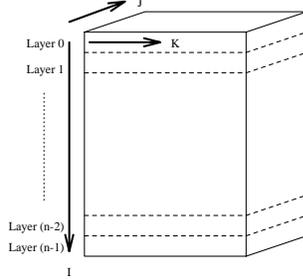

Fig. 2. The 3D mesh is divided into n layers along the i direction.

Input: An $N \times N$ image and an $n \times n$ parameter space, and a constant ρ_{res} which is the resolution along the ρ direction. Assume that each pixel value $a(x, y)$ is stored in processors $PE(0, x, y)$, for $0 \leq x, y < N$, and ρ_{res} is known to all processors initially. Denote $ARR(0, *, *)$ as the base submesh. It is clear that the initial image is stored in the base submesh. The algorithm consists of the following steps.

Step 1. In this step, we copy the whole image from the base submesh to all the other submeshes $ARR(i, *, *)$. All processors $PE(0, j, k)$, $0 \leq j < N$, $0 \leq k < N$, broadcast the image pixels $a(j, k)$ concurrently through its subbuses in direction i such that processors $PE(i, j, k)$, $0 \leq i < n * \log^2 N$, each receives a pixel from $PE(0, j, k)$. At the end of step 1, all processors in subarray $ARR(*, j, k)$, where $0 \leq j < N$, $0 \leq k < N$, contain the pixel value $a(j, k)$ at location (j, k) in the original image. Since only local switch settings and broadcast operations are involved in this step, the time used is $O(1)$.

Step 2. As mentioned before, the whole 3D mesh is divided into n layers with each layer having a 3D $log^2 N \times N \times N$ submesh. Each layer is responsible for computing the Hough transform for a particular projection. Thus, the top $\log^2 N$ 2D submeshes $ARR(i, *, *)$, $0 \leq i < \log^2 N$, are assigned to computing the Hough transform for θ_0. Similarly, the next $\log^2 N$ 2D submeshes $ARR(i, *, *)$, $\log^2 N \leq i < 2\log^2 N$, are in charge of computing for θ_1, and so on. Thus, each processor can calculates its local θ value easily based on its local indix i since it initially knows the resolutions of θ and ρ. This requires $O(1)$ time.

Step 3. In this step, all processors computes its local ρ value independently and in parallel. Here, layer t uses θ_t for $0 \leq t \leq n$ as shown in Figure 2; i.e., submesh $ARR(i, *, *)$ uses $\theta_{\lfloor i/ \log^2 N \rfloor}$, for $0 \leq i \leq n \log^2 N$, to calculate their ρ values. In other words, $PE(i, j, k)$ computes $\rho_{j,k} = j \cos \theta_{\lfloor i/ \log^2 N \rfloor} +$

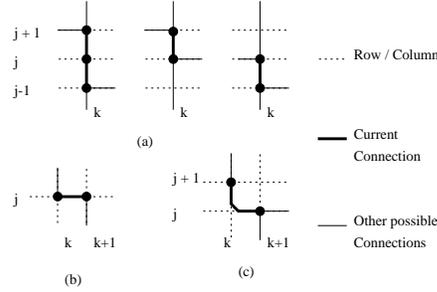$j \sin \theta_{\lfloor i/\log^2 N \rfloor}$. This step involves only local computations, and hence takes $O(1)$ time.

Fig. 3. Possible connections of a processor to neighboring processors for $0 \le \theta_k \le \pi/4$.

Step 4. All processors which have computed the same value of the normal distance ρ in the same layer (for a particular angle) can be connected in a 2D submesh. The idea is to count the number of black pixels in the same band (same ρ value for a particular θ value. Since all layers perform a similar job, in the following discussion we concentrate on layer 0. This operation requires only local communications and some setup of local switches. More specifically, the possible cases are depicted in Figure 3. The following connection schemes are based on lemmas 1-4. If $PE(i,j,k)$ computes some value ρ for the normal distance associate to pixel (j,k), and the same value is obtained for pixel $(j,k-1)$ and/or $(j,k+1)$, then $PE(i,j,k)$ should be connected to $PE(i,j,k-1)$ and/or $PE(i,j,k+1)$, as depicted in Figure 3(a). When two adjacent processors in a row have the same ρ value, the connection can be made as shown in Figure 3(b). In case that processors $PE(i,j,k+1)$ and $PE(i,j+1,k)$ have to be connected, a third intermediate processor $PE(i,j,k)$ is used as depicted in Figure 3(c). Using the above rule, a processor in submesh $ARR(i,j,k)$ is connected to at most two buses at a time and no two distinct buses are connected to the same port of a processor in the same submesh. Figure 4 shows the switch and bus configuration for a 11×11 mesh for $\theta_k = \pi/6$. Since all processors in the same layer have the same θ value, the mesh configuration is the same for all 2D submeshes $ARR(i,*,*)$ in the same layer. Thus, $\log^2 N$ 2D submeshes in layer k will have the same configuration as the one depicted in Figure 5. In effect, many 2D vertical submeshes are established. In Figure 5, we show a vertical submesh formed in a layer after the above configurartions and vertical buses are configured along the i direction. In fact, many vertical submeshes exist in the same layer (not shown in the figure). Of course, submeshes in different layers have different shapes. In this step, processors only exchange information with neighboring processors, and then decide on their switch settings. It is obvious that this step also takes constant time.
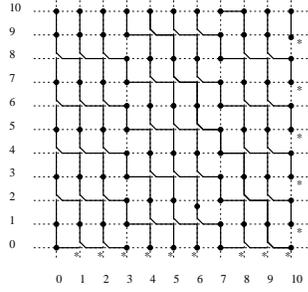
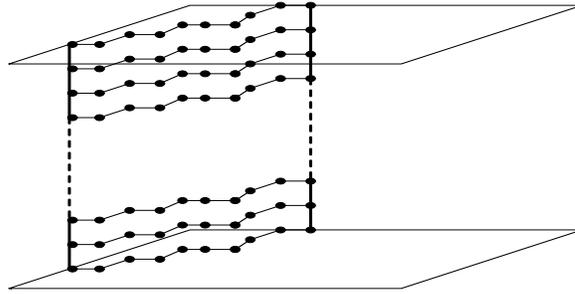**Fig. 4.** Switch and bus configuration for $\theta_k = \pi/6$.

Fig. 5. A 2D vertical submesh is established in a layer after bus reconfiguration in step 4.

Step 5. Accumulate all the pixel values in a band using the corresponding submesh established in the last step in parallel. Notice that each submesh has a size of $\log^2 N \times S$, where S is not fixed and depends its position. As shown in Figure 4, many subbuses of different lengthes are formed and hence their S values are different. However, S is less than $\sqrt{2}N$ and is equal to the number of pixels contained in the band. For S binary values, we can use Lemma 5 to add these binary values in $O(1)$ time on a 2D $\log^2 N \times S$ mesh. Since all submeshes satisfy the above condition, and they can perform the accumulation concurrently, this step uses $O(1)$ time.

Step 6. Each submesh elects a leader and the leader stores the local count from the last step. Notice that this step is necessary since not all boundary processors are the last processors in the reconfigured submesh as indicated in Figure 4. Only those processors with a "*" are leaders. The leaders can be elected easily by simply checking its neighbors and deciding if it should become a leader or not. Clearly, it also takes $O(1)$ time.

The final results are stored in the leaders distributed among different submeshes. Since each step uses $O(1)$ time, the total time used in the algorithm is $O(1)$. To summarize the above discussion, we have:

Theorem 1. For an $N \times N$ binary image and an $n \times n$ parameter space, the

Hough transform can be computed in constant time on a 3D $n \log^2 N \times N \times N$ reconfigurable mesh.

Our result clearly improves the Hough transform algorithms in [1, 10] where a 3D $nN \times N \times N$ reconfigurable mesh and a 3D $n^2 \times N \times N$ reconfigurable mesh are used, respectively, to achieve constant time.

References

1. K.-L. Chung and H.-Y. Lin, "Hough transform on reconfigurable meshes," Computer Vision and Image Understanding, vol. 61, no. 2, 1995, pp. 278-284.
2. P. V. C. Hough, "Methods and means to recognize complex patterns," U.S. Patent 3069654, 1962.
3. H.A.H. Ibrahim, J.R. Kender, and D.E. Shaw, "The analysis and performance of two middle-level vision tasks on a fine-grained SIMD tree machine," Proc. IEEE Computer Society Conf. on Computer Vision and Pattern Recognition, pp. 387-393, June 1985.
4. J. F. Jeng and S. Sahni, "Reconfigurable mesh algorithms for the Hough transform," International Conference on Parallel Processing, vol. III, pp. 34-41, Aug. 12-16, 1991.
5. T.-W. Kao, S.-J. Horng, Y.-L. Wang, "An O(1) time algorithm for computing histogram and Hough transform on a cross-bridge reconfigurable array of processors," IEEE Transactions on Systems, Man and Cybernetics, Vol. 25, No. 4, April 1995, pp. 681-687
6. S.S. Lin, "Constant-time Hough transform on the processor arrays with reconfigurable bus systems," Computing, vol. 52, pp. 1-15, 1994.
7. M. Merry and J. W. Baker, "Constant time algorithm for computing Hough transform on a reconfigurable mesh," Image and Vision Computing, Vol. 14, pp. 35-37, 1996.
8. S. Olariu, J. L. Schwing, and J. Zhang, "Computing the Hough transform on reconfigurable meshes," Image and Vision Computing, vol. 11, no.10, pp.623-628, Dec. 1993.
9. Y. Pan, "A More Efficient Constant Time Algorithm for Computing the Hough Transform," Parallel Processing Letters, vol. 4, no. 1/2, pp. 45-52, 1994.
10. Y. Pan, K. Li, and M. Hamdi, "An improved constant time algorithm for computing the Radon and Hough transforms on a reconfigurable mesh," IEEE Transactions on Systems, Man, and Cybernetics: (Part A), Vol. 29, No. 04, July 1999, pp. 417-421. (A preliminary version also appeared in Proceedings of the 8th International Conference on Parallel and Distributed Computing and Systems, 1996, pp. 82-86.)
11. H. Park, H. J. Kim, and V. K. Prasanna, "An O(1) time optimal algorithm for multiplying matrices on reconfigurable mesh," Information Processing Letters Vol. 47, August 1993, pp. 109-113.