2011 Second International Conference on Networking and Computing

Fast Ellipse Detection Algorithm using Hough Transform on the GPU

Yasuaki Ito, Kouhei Ogawa, and Koji Nakano
Department of Information Engineering,
Hiroshima University
1-4-1 Kagamiyama, Higashi Hiroshima, Hiroshima, 739-8527, Japan
{*yasuaki, kogawa4, nakano*}*@cs.hiroshima-u.ac.jp*

***Abstract*—GPUs (Graphics Processing Units) are specialized microprocessors that accelerate 3D or 2D graphics operations. Recent GPUs, which have many processing units connected with a global memory, can be used for general purpose parallel computation. To utilize the powerful computing ability, GPUs are widely used for general purpose computing. The main purpose of this paper is an ellipse detection algorithm with Hough transform. The feature of our algorithm is that to reduce computational time and space, the parameter spaces in the Hough transform are decomposed for each parameter and each parameter is computed in series. Also, we implemented our algorithm on a modern GPU system. The experimental results show that, for an input image with size of 2040×2040, our GPU implementation can achieve a speedup factor of approximately 64 times over the sequential implementation without the GPU support.**

***Keywords*-Ellipse detection; Hough transform; GPU; CUDA;**

I. Introduction

Random bin picking is one of the systems with highest interest of the industry, in order to automate the production process. It is to locate and move parts randomly placed, and jumbled in generic bins with a computer vision system and a robot arm. However, there are some problems such as extreme part overlap and occlusion, significant lights variability and shadowing, shortage of distinct features on parts, and collision avoidance with other parts, tools, and bins. To reduce the difficulty, parts with markers are utilized to locate them. Especially, circle markers are used for robot vision [1] because a circle must be seen as an ellipse from any angle as shown in Figure 1. In this paper, we focus on the ellipse detection to find such circle markers.

Hough transform is a technique that finds shapes in images [2]. It has been used to extract lines, circles, ellipses and arbitrary shapes. The Hough transform defines a mapping from the image into a parameter space that is 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 the detected shapes. Therefore, the elements that are voted intensively correspond to the parameters of the shapes in the image space.

In general, the technique of the Hough transform can be directly used for any parameterized curve. Let us consider the detection of ellipses using Hough transform. An ellipse can be defined by the five parameters, its center (c_x, c_y), the major axis a, the minor axis b, and the slope θ as shown in Figure 2. To detect ellipses, a five-dimensional parameter space is necessary, that is, $O(N^5)$ space is necessary to store the parameter space, where N is the size of each dimension of the parameter space. Also, it takes $O(N^5)$-time to vote for each edge point and search intensive elements in the accumulate array. Therefore, the generalization of the Hough transform implies an exponential increase in computational time and space requirements. For this reason, to reduce its computational complexity, most of methods based on the original Hough transform have been proposed [3]. One of the important ways to reduce the computation has been the use of geometric properties of shapes to decompose the parameter space.

Many algorithms for ellipse detection have been proposed in the past. Ming *et al.* have shown an ellipse detection algorithm using bounding boxes [4]. Yao *et al.* have proposed a robust GA-based ellipse detection [5]. Also, ellipse detection algorithms using Hough transform have been proposed using parameter space decomposition [6], [7], [8]. To reduce the number of votes to the accumulate array, randomized Hough transform is utilized by voting for random selected points [9], [10], [11], [12], [13].

The main contribution of this paper is to present a new ellipse detection algorithm based on Hough transform. The idea of this algorithm is to reduce the computing time and space using the parameter space decomposition in the Hough transform. Specifically, center coordinates, a slope, and two axes that define an ellipse are detected in series. In each detection, based on the edge image, voting to the parameter space is used same as the Hough transform. In the center detection, middle points of every two edge points are voted. To detect the slope, perpendicular bisectors of every two edge points are voted. Voting the lengths of two axes, a major axis and a minor axis are found. In each computation, it is not always to detect the correct parameters for the effect of noise or other objects. Therefore, we find totally P^3 ellipse candidates, P center candidates, P^2 slope candidates and P^3 axes candidates, as shown

978-0-7695-4569-1/11 $26.00 © 2011 IEEE
DOI 10.1109/ICNC.2011.61

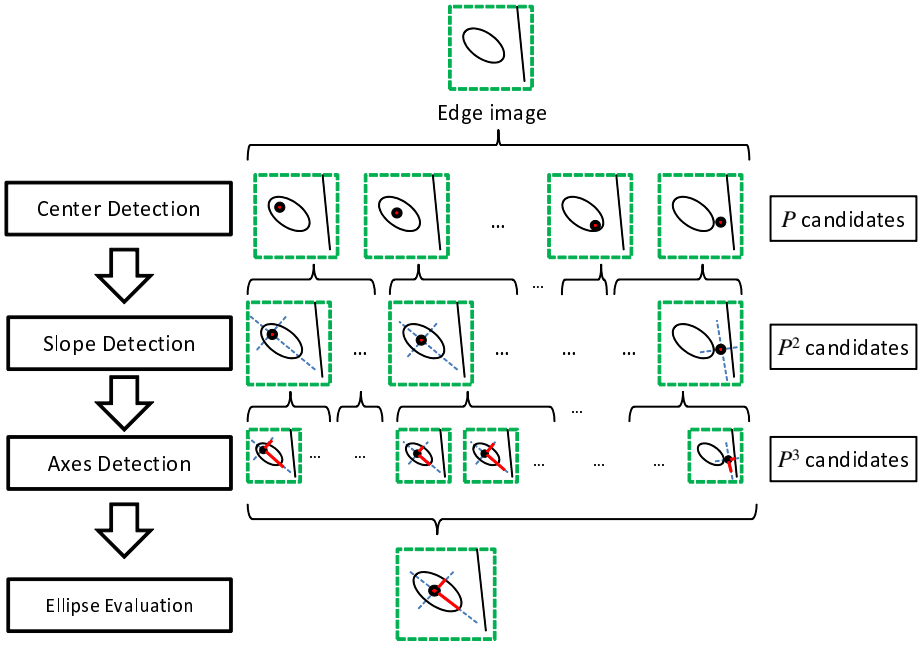


Figure 3. Overview of the proposed ellipse detection algorithm

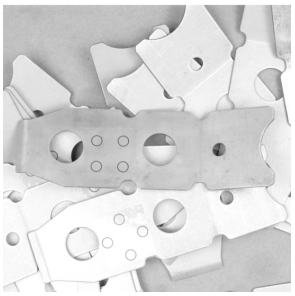


Figure 1. Industrial parts with circle markers

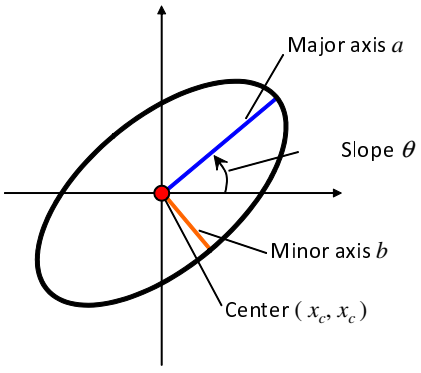


Figure 2. Five parameters of an ellipse

in Figure 3. After detecting P^3 ellipse candidates, each candidate is checked whether it is a true ellipse or not, using Euclid distance map of the edge image. Recall that, when the Hough transform is used directly, it takes $O(N^5)$ time using an accumulate array of size $O(N^5)$. However, in the proposed algorithm, a two-dimensional parameter space for each computation is used. The proposed algorithm runs in $O(N^2 + NE)$ using an accumulate array of size $O(N^2)$, where E is the number of the edge points in the edge image from the input image.

The proposed algorithm is similar to the Hough transform based algorithm with parameter space decomposition [6], [7], [8]. The difference is that in the proposed algorithm, the parameter space is decomposed into three two-dimensional spaces. Also, after voting, each detected ellipse is checked whether it is a true ellipse or not using Euclid distance map.

GPUs (Graphics Processing Units) are specialized microprocessors that accelerate 3D or 2D graphics operations. Recent GPUs, which have many processing units connected with a global memory, can be used for general purpose parallel computation. CUDA (Compute Unified Device Architecture) [14] is an architecture for general purpose parallel computation on GPUs. Using CUDA, we can develop parallel algorithms to be implemented in GPUs. Therefore, many studies have been devoted to implement parallel algorithms using CUDA [15], [16].

Also, we have implemented it in a modern GPU system, Nvidia GeForce GTX 480. In this GPU implementation, we have considered many programming issues of the GPU

system such as coalescing access of global memory, utilization of a shared memory that is an on-chip memory. Our GPU implementation has achieved approximately 64 times speedup over a CPU implementation without the GPU support.

The reminder of this paper is organized as follows: Section II shows the proposed ellipse detection algorithm. Section III briefly describes about CUDA architecture. The GPU implementation is shown in Section IV. Section V exhibits the performance of our proposed algorithm on the GPU. Finally, Section VI offers concluding remarks.

II. Ellipse Detection Algorithm based on Hough Transform

This section describes our proposed ellipse detection algorithm. The idea of this algorithm is to reduce the computing time and space by decomposition of the parameter space in the Hough transform. Given an input image of size $M \times M$, the details of this algorithm are spelled out as follows:

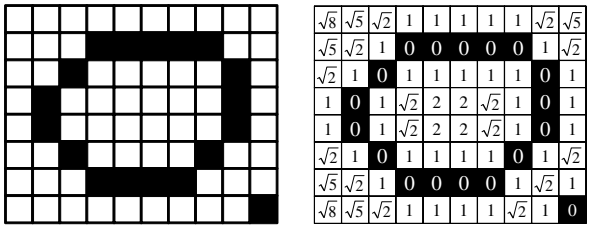
Ellipse Detection Algorithm

Step 1. An edge image is obtained from an input image.
Step 2. Euclid distance map is generated from the edge image.
Step 3. An edge point list is generated from the edge image.
Step 4. Center candidates are detected by voting midpoints of every two edge points.
Step 5. Slope candidates are detected by voting perpendicular bisectors of every two edge points.
Step 6. Axes candidates are detected by the result of voting the length of axes for every edge point.
Step 7. Every candidate is evaluated with Euclid distance map.

This algorithm is divided into two parts, preprocess and vote parts. These two parts correspond to Steps 1 to 3 and Steps 4 to 7, respectively. The details of each step are shown, as follows.

Step 1: To obtain an edge image, we utilize Canny edge detection algorithm [17]. Canny edge detection algorithm is the most commonly used image processing algorithm that detects edges in images. It is known that we can obtain accurate edges of input images. The computing time is $O(M^2)$.

Step 2: Given an edge image, Euclidean Distance Map (EDM) is a 2-D array of the same size such that each element is storing the Euclidean distance to the nearest edge point as shown in Figure 4. The array is used for Step 7. The computing time is $O(M^2)$ whose details are shown in [15].



(a) Edge image (b) Euclid distance map

Figure 4. Euclid Distance map

Step 3: In this algorithm, the edge image is scanned again and again to utilize edge information. Since usually the number of edge points is much smaller than that of non-edge points in the image, an *edge list* that is a series of coordinates of all edge points is generated to avoid scanning non-edge points. It takes $O(M^2)$-time to generate it.

Step 4: In this step, centers of ellipses in the image are detected. To find the center of the ellipse, coordinates of midpoints for every two edge points are voted to an accumulator that is a 2D array for the (x, y)-space. The coordinates of the midpoints (x_m, y_m) of the two edge points (x_1, y_1) and (x_2, y_2) are computed by the formulas, $x_m = \frac{x_1+x_2}{2}, y_m = \frac{y_1+y_2}{2}$. If there is an ellipse in the image, the votes for the center of the ellipse are concentrated. That is, it is possible that the coordinate whose number of the votes is large is the center of the ellipse. However, if there are line segments, the votes on the segments are concentrated. To avoid it, the voting for two points that are too close or too far is not performed. In this voting method, it is not always to detect the center of the ellipse for the effect of noise or other objects. Therefore, after voting, we find the largest P peaks that correspond to the P points are detected as center candidates. Let N be the size of each dimension of the (x, y)-parameter space. To store the parameter space, an accumulate array of size $O(N^2)$ is necessary. Since a middle point is voted for each two edge points, it takes $O(E^2)$-time to vote for all the pairs of edge points, where E is the number of the edge points. Also, if $P \ll N$, to search P center candidates, $O(N^2)$-time is necessary. The total computing time in Step 4 is $O(E^2 + N^2) = O(N^2)$ because $E^2 < N^2$.

Step 5: In this step, the slope of ellipses in the image are detected. Since an ellipse is symmetry for its two axes, if we can find the slope of the axes, the slope of the ellipse is determined. To find the symmetry axes of the ellipse, perpendicular bisectors for every two edge points are voted to an accumulator that is a 2D array for the (ρ, θ)-space. Let us consider polar consideration of a line. A line is represented by the formula $\rho = x\cos\theta + y\sin\theta$, where ρ represents the distance between the line and the origin, while θ is the angle of the line normal to the line (Figure 5). Therefore, a perpendicular bisector for two points (x_1, y_1) and (x_2, y_2) is computed by the formulas:

$$\theta = \tan^{-1}\frac{y_2 - y_1}{x_2 - x_1},$$

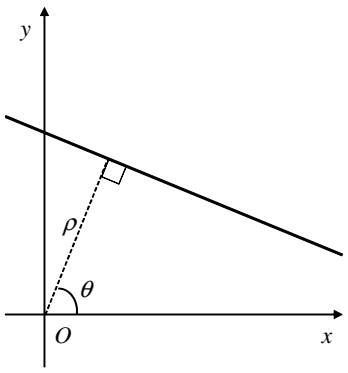


Figure 5. Polar consideration of a line

$$\rho = \frac{(x_1 + x_2)\cos\theta + (y_1 + y_2)\sin\theta}{2}.$$

Perpendicular bisectors for every two edge points are voted to an accumulator that is a 2D array for the (ρ, θ)-space. In the (ρ, θ)-space, parameters whose center is (c_x, c_y) are on the curve $\rho = c_x \cos\theta + c_y \sin\theta$. Since two axes of an ellipse, a major axis and a minor axis, are orthogonal, the number of the votes on the curve for two angles whose difference is 90 degree is large. The two angles whose number of the votes is large are found as the slope of the axes. After voting, we find the largest P peaks for each candidate detected in Step 4 and the slopes are detected as the slope candidates. That is, P^2 candidates are found in this step. To store the (ρ, θ)-parameter space, an accumulate array of size $O(N^2)$ is necessary. Since a perpendicular bisector is voted for each two edge points, it takes $O(E^2)$-time to vote for all the pairs of edge points. Since there are P center candidates, $O(PN)$-time is necessary because, for each center candidate, the scan in the parameter space is executed along the curve obtained by the center candidate. Because P is much smaller than E and N, it can be considered as a constant. Therefore, the total computing time in Step 5 is $O(PN + E^2) = O(N + E^2)$.

Step 6: In this step, the remainder parameters a and b that are the length of the axes are detected. Let us consider an ellipse through a point (x, y), whose center is (c_x, c_y) and slope is θ. The length of the axes is computed by the formula

$$b = \sqrt{\frac{a^2(-(c_x - x)\sin\theta + (c_y - y)\cos\theta)^2}{a^2 - ((c_x - x)\cos\theta + (c_y - y)\sin\theta)^2}}.$$

To find the length of the axes, using the above formula, for each edge point, the length b is voted for every a to an accumulator that is a 2D array for the (a, b)-space. According to the voting, the parameter whose number of the votes is large is detected as the length of the axes. After voting, we find the largest P peaks for each candidate detected in Step 4 and 5 and the axes are detected as the axes candidates. In this step, P^3 candidates are found. To store the (a, b)-parameter space, an accumulate array of size $O(N^2)$ is necessary. Since for each edge point, the length b is voted for every a, it takes $O(NE)$-time to vote for all the edge points. Since there are P^2 candidates, $O(P^2NE)$-time is necessary. Because P^2 can be considered as a constant, the total computing time in Step 6 is $O(N^2 + P^2NE) = O(N^2 + NE)$.

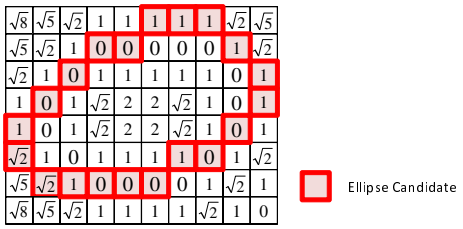


Figure 6. Evaluation of the estimated ellipse

Step 7: In the above steps, P^3 ellipse candidates that are defined by the five parameters are found. To confirm whether the candidates are true ellipses or not, we evaluate them with the Euclid distance map generated in Step 2. For each ellipse, it is plotted on the Euclid distance map. the sum of the distances on the plotted ellipse is computed (Figure 6). If the ellipse is close to the edge points, the sum is small. Therefore, in our algorithm, the ellipse whose sum is the smallest in the P^3 ellipse candidates is detected as a true ellipse. Let k denote the number of the plotted points for each ellipse. For each ellipse candidate, it takes $O(k)$-time to evaluate the candidate. The computing time for P^3 ellipse candidates is also $O(kP^3)$-time.

According to the above, the total computing time of the proposed algorithm is $O(M^2 + N^2 + NE + kP^3) = O(M^2 + N^2 + NE)$, because kP^3 is much smaller than other terms. Considering the time of the ellipse detection, it takes $O(N^2 + NE)$-time.

Using the above method, we can detect only one ellipse with combinatorial Hough transform. However, if there are multiple ellipses in the image, only one of them is detected. Therefore, the input images are divided into overlapped subimages as shown in Figure 7. The size of each subimage is large enough to include every ellipse. To detect ellipses that stretch over the subimages, they overlap one-third horizontally and vertically. Perform the above steps for each subimage, we can detect multiple ellipses in the image.

III. 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

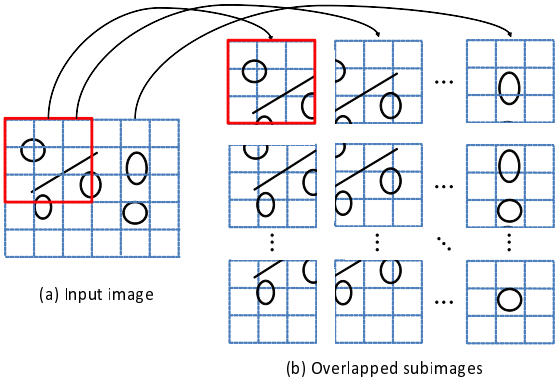


Figure 7. Division of the input image

the processing cores of a GPU, numerous threads are required. Compute Unified Device Architecture (CUDA) [14] 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 half warp of a thread block can be executed by the assigned streaming multi-core processor concurrently. The grid will launch a segment of codes, named a *kernel*, to occupy a GPU device at a time. Actually, CUDA is a new parallel programming model and instruction set architecture. CUDA comes with a software environment that allows developers to use C-like high-level programming language.

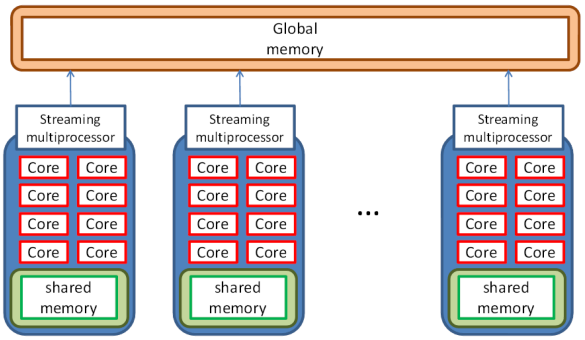


Figure 8. CUDA Hardware Architecture

On the other hand, GPUs can provide different memory spaces for different applications and each memory space has its own advantages and drawbacks. In CUDA architecture, each memory space has a corresponding specification. In this paper we only introduce few of them shown as follows.

Global memory is a main device memory of GPUs and which is off-chip memory. Therefore it has heavy access latency to each processing core. Fortunately, CUDA provides a technique known as *coalescing* [18] to hide the access latency of the global memory. When 16 sequential threads access 16 sequential and aligned values in the global memory, the GPU will automatically combine them into a single transaction.

Shared memory is a sort of on-chip memory and which is located within each streaming multi-core processor. It has almost no access latency and only visible to the thread block which is executed by the corresponding streaming multi-core processor. In practice, the shared memory can be used as a cache to hide the access latency of the global memory.

IV. GPU Implementation

In this section, we show an implementation of parallel execution of ellipse detection using a GPU. The idea of our GPU implementation is to reduce the number of access to the global memory using the shared memory and coalescing.

In Step 1 and Step 2 shown in Section II, we use our previous works that are implementations of Canny edge detection and Euclid distance map algorithm on the GPU [16] and [15], respectively.

The process of Steps 3 to 7 consists of three parts, that is, we implement three kernels, *EdgeListKernel*, *VoteKernel*, and *EvaluationKernel*. We explain the detail of them as follows.

EdgeListKernel: In this kernel, an edge list shown in Step 3 is generated, as follows. The edge image made in Step 1 is divided into subimages that correspond to squares of broken lines in Figure 7(a). For each subimage, a thread block that consists of threads is performed in parallel. The number of the threads in each thread block is the number of pixels of a side of the subimage. The threads read the pixel values of the edge image in column wise concurrently to utilize the coalescing as shown in Figure 9. If edge pixel is found, its coordinate is stored to the edge list in the shared memory. Since the data in the shared memory is not utilized over kernels, after all edge pixels in each subimage, the edge list is stored to the global memory.

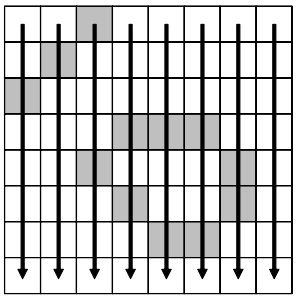


Figure 9. Coalescing access

VoteKernel: This kernel performs the process in Step 4 to Step 6. In this kernel, the process for each subimage shown in Figure 6 is assigned to one thread block. Each thread block is performed in parallel. Furthermore, every

thread block consists of multiple threads and the process for each subimage is performed in parallel using the threads. The voting process in Step 4 to Step 6 is performed on the global memory using the edge lists that are made in EdgeListKernel. To perform voting concurrently, the *atomic function* that performs a read-modify-write atomic operation provided by CUDA [18] are used.

EvaluationKernel: This kernel evaluates the ellipse candidates detected in Step 7. That is, the candidates are checked whether each of them is a true ellipse or not using the edge list and Euclid distance map. This kernel also assigns one thread block for each subimage shown in Figure 6. Each thread block evaluates the ellipse candidates detected in each subimage in parallel. Each thread block is composed by threads. Each of the threads evaluates one or more candidates in parallel.

V. Performance Evaluation

We have implemented and evaluated our proposed ellipse detection algorithm on the GPU. For the purpose of comparison, we have also implemented a conventional software approach without support of the GPU. We have used Nvidia GeForce GTX 480 with 480 processing cores (15 streaming multi-core processors which has 32 processing cores) running in 1.4GHz and 3GB memory. To evaluate software approach, we have used Intel Xeon E5540 running in 2.0GHz and 6GB memory.

Table I shows performance of our proposed ellipse detection on the GPU. We have used an input image of the size 2048×2048 that has industrial parts with circle markers shown in Figure 1. Figure 10 is the edge image obtained in Step 2. Note that to focus on the performance of the ellipse detection, we have evaluated the performance except the edge detection in Step 1 and generating Euclid distance map in Step 2. The size of subimage has been set to 96×96, which is large enough to include a circle marker that looks a ellipse. Also, we have executed our algorithm for $P = 5$, where P is the number of detected candidates in each step for every subimage. That is, in Step 4 of the algorithm, 5 center candidates are detected. In Step 5, 5 slope candidates for each center candidate are detected. Similarly, in Step 6, 5 axes candidates for each center and slope candidate are detected. In total, $P^3 = 5^3 = 125$ ellipse candidates are found for each subimage. After that, one ellipse in the candidates is detected with the evaluation of the ellipses in Step 7. According to the result, all the ellipses in the image are detected. Also, the result of the GPU implementation is equivalent to that of the CPU implementation. According to the table, the computing time of every kernel of the GPU implementation is shorter than that of the CPU implementation. We have achieved 64.79 times speedup in total

Table I
Performance of ellipse detection on the GPU

	CPU [ms]	GPU[ms]	Speed-up
EdgeListKernel	8.72	0.70	12.45
VoteKernel	33373.40	298.78	111.70
EvaluationKernel	14942.42	446.42	33.47
Total	48324.54	745.90	64.79

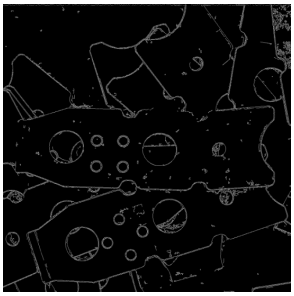


Figure 10. Result of the edge detection

VI. Conclusions

In this paper, we have presented a fast ellipse detection algorithm based on Hough transform. The parameters that define an ellipse are computed by voting for each parameter. Also, we have implemented it with a modern GPU system, Nvidia GeForce GTX 480. Comparing to the performance of the CPU implementation, the GPU implementation have achieved 64.79 times speedup.

References

[1] Y. Mochizuki, A. Imiya, and A. Torii, "Circle-marker detection method for omnidirectional images and its application to robot positioning," in *Proceedings of International Conference on Computer Vison*, 2007, pp. 1–8.

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

[3] J. Illingworth and J. Kittler, "A survey of the Hough transform," *Computer Vision, Graphics, and Image Processing*, vol. 44, pp. 87–116, 1988.

[4] C.-M. Chang, "Detecting ellipses via bounding boxes," *Asian Journal of Health and Information Sciences*, vol. 1, no. 1, pp. 73–84, 2006.

[5] J. Yao, N. Kharma, and P. Grogono, "Fast robust GA-based ellipse detection," in *Proceedings of the 17th International Conference on Pattern Recognition*, vol. 2, 2004, pp. 859–862.

[6] A. S. Aguado, M. E. Montiel, and M. S. Nixon, "On using directional information for parameter space decomposition in ellipse detection," *Pattern Recognition*, vol. 29, no. 3, pp. 369–381, 1996.

[7] R. Krishnapuram and D. Casasent, "Hough space transformations for discrimination and distortion estimation." *Computer Vision, Graphics, and Image Processing*, vol. 38, pp. 299–316, 1987.

[8] P. S. Nair and A. T. Saunders, Jr., "Hough transform based ellipse detection algorithm," *Pattern Recognition Letters*, vol. 17, no. 7, pp. 777–784, 1996.

[9] L. Xu, E. Oja, and P. Kultanen, "A new curve detection method: randomized hough transform (RHT)," *Pattern Recognition Letters*, vol. 11, no. 5, pp. 331–338, 1990.

[10] R. A. McLaughlin, "Randomized Hough transform: Improved ellipse detection with comparison," *Pattern Recognition Letters*, vol. 19, no. 3-4, pp. 299–305, 1998.

[11] C. A. Başca, M. Taloş, and R. Brad, "Rondomized Hough transform for ellipse detection with result clustering," in *Proceedings of International Conference on Computer as a Tool*, vol. 2, 2005, pp. 1397–1400.

[12] K. Hahn, Y. Han, and H. Hahn, "Ellipse detection using a randomized Hough transform based on edge segment merging scheme," in *Proceedings of the 6th WSEAS International Conference on Signal Processing, Robotics and Autoaimton*, 2007, pp. 1–6.

[13] J. K. Lee, B. A. Wood, and T. S. Newman, "Very fast ellipse detection using GPU-based RHT," in *Proceedings of 19th International Conference on Pattern Recognition*, 2008, pp. 1–4.

[14] NVIDIA Corporation., *NVIDIA, CUDA Architecture*, http://www.nvidia.com/object/cuda_home_new.html.

[15] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of parallel computation of Euclidean distance map in multicore processors and GPUs," in *Proceedings of International Conference on Networking and Computing*, November 2010, pp. 120–127.

[16] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *Proceedings of International Workshop on Advances in Networking and Computing*, 2010, pp. 279–280.

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

[18] NVIDIA, *NVIDIA CUDA Programming Guide*, July 2009.