Fast Hough Transform on GPUs: Exploration of Algorithm Trade-offs

Gert-Jan van den Braak, Cedric Nugteren, Bart Mesman, and Henk Corporaal
{g.j.w.v.d.braak, c.nugteren, b.mesman, h.corporaal}@tue.nl

Dept. of Electrical Engineering, Electronic Systems Group
Eindhoven University of Technology, The Netherlands

Abstract. The Hough transform is a commonly used algorithm to detect lines and other features in images. It is robust to noise and occlusion, but has a large computational cost. This paper introduces two new implementations of the Hough transform for lines on a GPU. One focuses on minimizing processing time, while the other has an input-data independent processing time. Our results show that optimizing the GPU code for speed can achieve a speed-up over naive GPU code of about 10×. The implementation which focuses on processing speed is the faster one for most images, but the implementation which achieves a constant processing time is quicker for about 20% of the images.

1 Introduction

Computer vision applications are more and more used in every day life. For example in industrial applications like traffic surveillance [2], but also in consumer applications like the augmented reality applications on mobile phones [11]. Detecting shapes like lines and circles is an important and often computational intensive part of these computer vision applications.

Since the end of 2006, with the release of "CUDA" by NVIDIA and "Close to Metal" by AMD, Graphical Processing Units (GPUs) have become more programmable and more usable for other applications than computer graphics. Since then, many computer vision applications have been implemented on GPUs [1].

The Hough transform is a popular technique to locate shapes in images. It is mostly used to find straight lines and circles in images, but it can also be used to detect arbitrary shapes. The Hough transform is a robust technique that works well even in the presence of noise and occlusion. It is used in many computer vision and image processing applications, like robot navigation [4], industrial inspection and object recognition [14].

A complete application for detection shapes in images usually consists of several steps: a) edge detection; b) thresholding; c) voting in Hough space; d) Hough space post-processing; e) displaying detected lines. These steps are illustrated in Fig. 3. In this paper we will only focus on the third step: voting in the Hough space. The first step can be a convolution based edge detection, e.g. Sobel edge detection, as can be found in the NVIDIA CUDA SDK. For the second step Otsu

thresholding can be used. Otsu thresholding makes a histogram of the edge image, processes the histogram and finds the best threshold value. An efficient GPU implementation of making a histogram can be found in [8]. In the Hough space post-processing stage the maximum in the Hough space is located. This maximum can be used to draw the most dominant line in the original image.

This paper is organized as follows. First two parameterizations for lines and their corresponding Hough transform are presented in Section 2. In Section 3 the benchmark setup can be found, together with a brief description of GPU hardware and programming. The different GPU implementations of the Hough transform can be found in Section 4 and the results and evaluation can be found in Section 5. Finally conclusions and future work are presented in Section 7.

2 Hough transform for lines

The Hough transform for lines [6] is a voting procedure where each feature (edge) point in an image votes for all possible lines passing through that point. All votes are stored in the so called Hough space, which is two dimensional for the Hough transform for lines. The size of the Hough space is determined by the size of the input image and the required accuracy for the parameterization of the lines. Two different parameterizations for lines and their corresponding Hough transforms are described in this section.

2.1 Cartesian Hough transform

A straight line can be described in a Cartesian coordinate system with a slope a and some intersect b with the vertical axis by the following equation:

$$y = ax + b \tag{1}$$

In the Hough transform, the characteristics of the straight line are not considered as image points (x_i, y_i), but instead in terms of its parameters a and b. Therefore Eq. 1 can be rewritten to:

$$b = y_i - x_i a \tag{2}$$

For each image point (x_i, y_i) a line of votes is placed in the Hough space for a range of angles θ. Parameter a is calculated as $a = \tan(\theta)$, and the corresponding values for b are calculated with Eq. 2. In Fig. 1(a) the two points (x_p, y_p) and (x_q, y_q) form a line. The two corresponding lines in the Hough space are shown in Fig. 1(b). At the intersect of these two lines the (best approximated) value for the parameters a and b can be found.

The parameters can become an infinite number when the line is vertical. Therefore the Hough space is usually divided into two parts: one part for angles between $-45°$ and $45°$ which uses Eq. 2 and one part for angles between $45°$ and $135°$ which uses Eq. 3.

$$b' = x_i - y_i a' \tag{3}$$

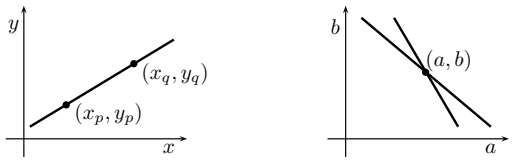


Fig. 1. (a) A line through two points in an image. (b) Corresponding two lines in Hough space.

2.2 Polar Hough transform

In the polar representation a line is parameterized with ρ and θ [3], as shown in Fig. 2. Parameter ρ represents the distance between the line and the origin, and the angle of the vector from the origin to this closest point, as given by Eq. 4. Eq. 1 and Eq. 4 are related by Eq. 5.

$$\rho = x\cos(\theta) + y\sin(\theta) \tag{4}$$

$$a = -\frac{1}{\tan(\theta)} \qquad b = \frac{\rho}{\sin(\theta)} \tag{5}$$

In this polar parameterization the parameters ρ and θ are bounded. The angle θ ranges from 0° to 180° and the radius ρ ranges from 0 to $\sqrt{W^2 + H^2}$, where W and H are the width and height of the image respectively.

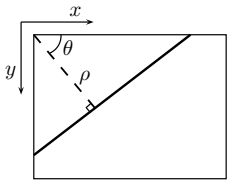


Fig. 2. Polar representation of a line.

3 Benchmark setup

To measure the performance of the different parameterizations for lines in the Hough transform, a number of benchmarks are performed. The CPU used in these benchmarks is an Intel Core i7 930 with four cores running at 2.8 GHz. The CPU implementations use OpenMP to utilize all cores and calculate the Hough transform by iterating over all pixels in the binary input image. If a pixel value is equal to '1', this pixel value is used in the voting process, otherwise it is discarded. In all implementations the trigonometric functions are pre-calculated and stored in an array.

The GPU used in our setup is an NVIDIA GTX 470 with 448 CUDA cores running at 1.2 GHz and has 1280 MB of off-chip global memory. In NVIDIA's latest architecture (Fermi) [9], 32 CUDA cores are grouped into a cluster. Each cluster has an on-chip shared memory (about 48 kB) and a cache.

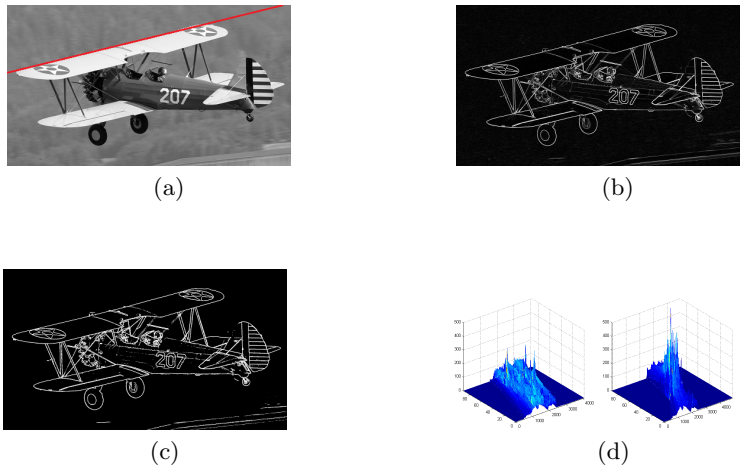


Fig. 3. Test image with resulting line (red) after Hough transform (a); intermediate images after edge detection (b) and thresholding (c); final Hough spaces (d).

The code executed on a GPU is called a kernel. Kernels run on the GPU in thousands or even millions of threads. Each thread executes the same program, but not necessarily the same instruction at the same time. Threads are organized into thread blocks. All threads in a thread block are executed on the same processing cluster and can communicate via its shared memory. Threads within a thread block are arranged in warps of (at most) 32 treads, and each thread in a warp executes the same instruction at the same clock cycle [10].

All images used in the measurements in this paper are gray scale images and have a resolution of 1920×1080 pixels. The chosen resolution for the parameters in the Hough space is one degree for the the angle parameter (a in Eq. 1 and θ in Eq. 4) and 1 pixel for the intersect parameter (b in Eq. 1 and ρ in Eq. 4). First Sobel edge detection and Otsu thresholding are applied on the images. An example test image can be found in Fig. 3. The number of edge pixels after thresholding in this image is 5.9%. Taken an average over the 2550 unique pictures in the Nistér and Stewénius benchmark set [7], the average number of edge pixels after applying Otsu thresholding is 9.6%. The distribution of the number of edge pixels in an image in this benchmark set is shown in Fig. 4.

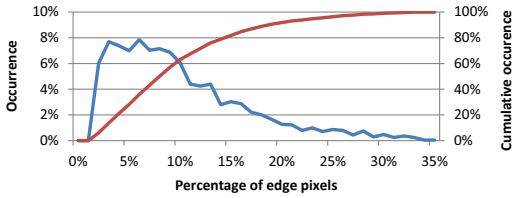


Fig. 4. Distribution of the number of edge pixels in the images of the Nistér and Stewénius benchmark set [7].

4 GPU implementations

The difference in execution time of the two different parameterizations of the Hough transform as described in Section 2 is measured by two implementations on a CPU and a GPU. The GPU versions are based on the fast GPU implementation as described in Section 4.2 below. A comparison between these CPU and GPU implementations is made in Section 5.

Next to the different versions for the different parameterizations, three implementations for the Cartesian parameterization of the Hough transform have been made to explore the trade-offs between execution speed, predictability and code complexity. First a very basic implementation is described, which is used as the reference for the other implementations. The second implementation focuses on processing speed, at the cost of more complex code and a higher memory utilization. The last implementation achieves a constant processing time, which makes its processing time independent of the input image. In all three GPU implementations the trigonometric functions are calculated jointly by all threads in a thread block and stored in an array in the shared, on chip, memory.

4.1 GPU implementation 1 - basic

The first GPU implementation is a based on the CPU implementation, without GPU specific optimizations. The pseudo code for this implementation is given in Fig. 5. First the Hough space in off-chip global memory has to be reset to all zeros. Then a kernel is started with one thread for each pixel in the input image. If the value of the pixel is a '1', the thread places a vote in both Hough spaces (the Hough transform in the Cartesian parameterization consists of two Hough spaces, see Section 2.1) for each possible angle. The Hough spaces are located in the global (off-chip) memory of the GPU and atomic operations have to be used for the voting process. Measurements (see Section 5) show that this implementation is just a bit slower than an optimized CPU implementation. The most time consuming step in this implementation are the atomic additions on the Hough spaces in global memory, since atomic operations which modify values in the same location and which are executed by threads in a warp are all serialized, and global memory latency is typically 400 - 800 clock cycles [10].

```
1  pixel_value = image[x,y]
2  if(pixel_value > threshold) {
3     for i=0:N {
4        a1 = A1[i]; b1 = y - x*a1 // tan() calculations are
5        a2 = A2[i]; b2 = x - y*a2 // stored in arrays A1 and A2
6
7        atomicAdd(HS1[(a1,b1)], 1)
8        atomicAdd(HS2[(a2,b2)], 1)
9     }
10 }
```

Fig. 5. Pseudo code for voting in the Hough space for a single pixel in the basic GPU implementation of the Cartesian Hough transform.

4.2 GPU implementation 2 - fast

The second implementation focuses on processing speed. As mentioned in Section 3, less than 10% of the pixels are actually used in the voting process. This means that most of the threads in the previous solution are waiting for a few threads to finish. Therefore this second solution starts by making an array of all pixels that need to be processed. A second kernel processes this array to create the Hough space. This two step process is illustrated in Fig. 6.

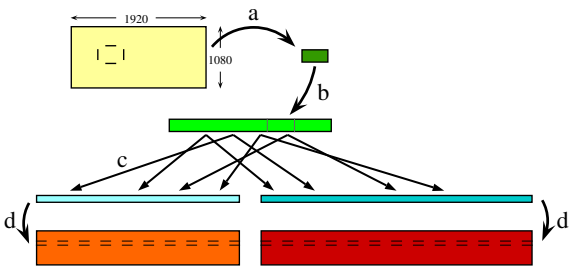


Fig. 6. Fast implementation of the Hough transform on GPU. Each thread block in the first kernel converts a part of the image to an array of pixel coordinates in the shared (on-chip) memory (*a*). The part of the array is added to the main array in global (off-chip) memory (*b*). In the second kernel the array of pixel coordinates is processed by a thread block to create one Hough line in each of the two Hough spaces in the shared memory (*c*). When the complete array of coordinates has been processed, the Hough line is copied to the corresponding Hough space in global memory (*d*).

Creating the array The creation of the array is inspired by the work in [8], where a histogram for each warp in a thread block is made. For the Hough transform an array of only the pixels which have to be used in the voting process is desired. To build this array in a parallel way on the GPU (step *a* in Fig. 6), small arrays are made on a warp-level granularity. How an array per warp is made is summarized in pseudo-code in Fig. 7. Note that all threads in a warp execute the same instruction at the same time in parallel, but some threads may be disabled due to branching conditions.

```
1 pixel_value = image[x,y]
2 if(pixel_value > threshold) {
3    do {
4       index++
5       SMEM_index = index
6       SMEM_array[index] = (x,y)
7    } while(SMEM_array[index] != (x,y))
8 }
9 index = SMEM_index
```

Fig. 7. Building an array of coordinates of edge pixels in shared memory (SMEM) (step *a* in Fig. 6).

Each thread in a warp reads a pixel from the input image (line 1, Fig. 7). As the pixel value is larger than a threshold value (line 2), the pixel coordinates need to be added to the array. The index of where these coordinates are to be stored is increased by one (line 4), to ensure no previously stored coordinates are erased. The new index is also stored in the on-chip shared memory (line 5), so threads in the warp which do not have to store coordinates can update their index value after all coordinates in this iteration have been added to the array (line 9). Now each thread tries to write its coordinate pair (x, y) to the array in shared memory at location *index* (line 6). Only one thread will succeed (line 7), and the others have to retry to write to the next location in the array (line 3-7). On average this loop has to be executed three times before all threads in the warp (32 threads in total) have added their coordinate pair (x, y) to the array, since only 10% of the pixels are above the threshold, as shown in Section 3.

There is a trade-off in the number of pixels each thread has to process. More pixels per threads result in less arrays to combine later, but too many pixels per thread means that there are not enough threads active to keep the GPU fully utilized. Also the maximum number of pixels in each small array is limited by the amount of shared memory available.

Now all small arrays in the shared memory have been made, they have to be combined in one array in the off-chip global memory (step b in Fig. 6). First one thread in each thread block sums the lengths of all warp-arrays of the thread block. This sum is added by this single thread to the global length of all arrays by a global atomic operation. This operation returns the value of the global length before the sum was added. This global length value is now used to tell each warp at which index in the global array their warp-array can be stored.

Voting in Hough space A second kernel is used to vote in the Hough space. Since atomic operations to the off-chip global memory are slow, the voting implementation is improved compared to the voting implementation in Section 4.1. A single thread block is used to create a single line (one value for the angle parameter) in each of the two Hough spaces simultaneously (step c in Fig. 6). The number of lines in the Hough space is determined by the required accuracy of the angle parameterization. This implies that the entire array will be read as many times as there are values for the angle parameter. Each Hough line is first put together in the shared memory, and later copied to the global memory to create the complete Hough space(step d in Fig. 6). This also removes the requirement that the Hough space in global memory has to be reset to zero, as was the case in the implementation in Section 4.1.

4.3 GPU implementation 3 - constant

For the third implementation the relative number of pixels to be processed does not influence the processing time. A graphical representation of this implementation is shown in Fig. 8.

In this implementation, all threads in a thread block will together copy a couple of lines of the input image to the on-chip shared memory (step b in

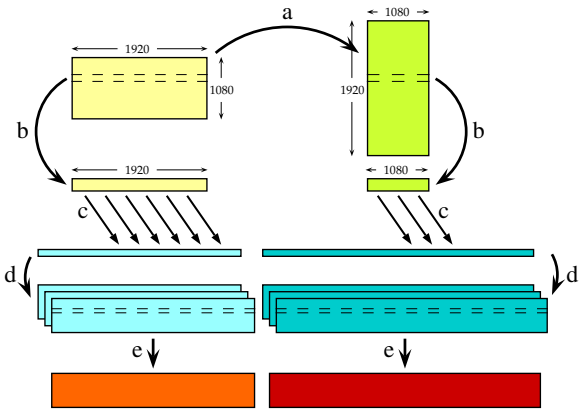


Fig. 8. Input-data independent implementation of the Hough transform on GPU. First the image is rotated by the first kernel (*a*). In a second kernel each thread block copies a part of the input image from the global (off-chip) memory to the shared (on-chip) memory (*b*). Then one Hough line is calculated in shared memory based on the part of the image in the shared memory (*c*). This line is stored in a sub-Hough space in global memory (*d*). This step is repeated to calculate the next Hough line, until one entire sub-Hough space is filled by each thread block. A third kernel sums all sub-Hough spaces together to make the final Hough space (*e*). The last two kernels are implemented in two versions, one for the original image in landscape orientation and one for the rotated image in portrait orientation.

Fig. 8). Then all threads read this part of the input image pixel by pixel, and together produce one Hough line (step *c* in Fig. 8). Here atomic operations are not required, since consecutive threads vote for consecutive bins in the shared memory (since consecutive threads process consecutive pixels). This is only true if threads are working on the same image line, as can be seen in Eq. 3. If threads work on different image lines, they vote for the same value of b and atomic operations would be required. So all threads in a thread block need to synchronize after processing an image line, to remove the need for atomic operations. This method is most efficient when the least amount of synchronizations are required, e.g. the width of the input line is as large as possible.

After one line in the Hough space is created, it is written to the off-chip global memory (step *d* in Fig. 8) and the next Hough line is generated in the same way. After all lines are generated and copied to global memory, a second kernel combines all sub-Hough spaces of parts of the image to one Hough space of the entire image (step *e* in Fig. 8).

To create the second Hough space, the image is first rotated in another kernel (step *a* in Fig. 8). This makes it possible to read the image coalesced and vote for consecutive bins, since consecutive pixels are read according to Eq. 2. Then the same algorithm is used as described above, but now there are more lines which are smaller (since the image now has a portrait orientation instead of a

landscape orientation). This means that creating this second Hough space takes more time than creating the first Hough space.

This implementation is limited by the amount of on-chip shared memory in the GPU. To reduce the number of sub-Hough spaces, a thread block should process a part of the image as large as possible. Since after the thresholding stage the pixels can only have two values (0 or 1, below or above threshold value), each pixel can be packed into a single bit. This means that more pixels can be stored in the shared memory (in comparison to the original approach where each pixels is stored in one byte), and the number of sub-Hough spaces (which have to be added later) is reduced. A second benefit is that the reading of the input image is faster, since the number of bytes required to read the complete image is reduced. Packing the image from bytes to bits can be done in the rotating stage (step *a* in Fig. 8), through what it does not take much extra processing time (about 2% of the total processing time).

5 Results

In this section the results of the CPU and GPU implementation for the two different parameterizations are discussed first. Then the results of the three different GPU implementations (basic, fast and constant) are discussed. All GPU timing measurements only include the execution time of the kernels, data transfer times to and from the GPU are not included since the pre- and post-processing steps are also executed on the GPU.

The performance of the (fast) GPU implementation of the different parameterizations can be found in Table 1. These results are compared to an optimized CPU implementation, where all four cores in the test system are used.

Table 1. Results of the two different parameterizations on the test image.

Hough version	**CPU time**	**GPU time**	**Speed up**
Cartesian	18.2 ms	2.6 ms	7.0×
Polar	23.3 ms	3.3 ms	7.0×

As can be seen in Table 1, both parameterizations have about the same performance, as well on the CPU as on the GPU. The Polar parameterization is a bit slower, since each vote by a pixel requires two (floating point) multiplications instead of one. The average speed-up of the GPU implementations over the CPU implementations is seven times.

For all three implementations (basic, fast and constant) of the Cartesian parameterization as described in Section 4, a GPU implementation has been made. The results can be found in Table 2, which show that the basic implementation is about 20% slower than the optimized CPU implementation. By optimizing the GPU code for speed, the performance can be increased by almost a factor

of 10, but at the cost of more complex code. The code is not only more complex in number of source lines of code, but also in how easy its parameters like image size and Hough space size (which controls the quality of the result) can be adjusted. The code for the constant implementation is even more complex. The input image size is fixed (only multiples of the current image size are easy to implement), to make all optimizations possible.

Table 2. Timing results of the three different GPU implementation on the test image. The results of the fast- and constant implementations are split over the different kernels. The number of source lines of code only includes the GPU kernel code.

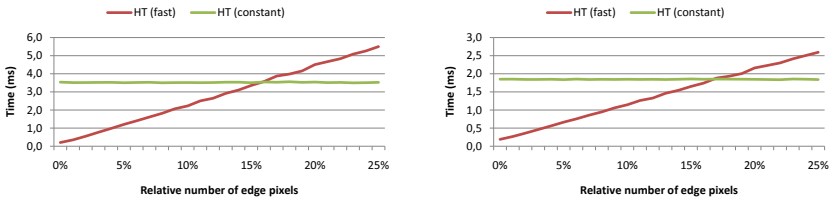
Implementation	**GPU time**	**Source lines of code**
1. Basic	21.6 ms	29
2. Fast	2.6 ms	97
a. Array building	0.3 ms	
b. Voting in Hough space	2.3 ms	
3. Constant	10.6 ms	193
a. Rotate and pack image	0.5 ms	
b. Voting in sub Hough spaces 1	3.9 ms	
c. Summing sub Hough spaces 1	0.4 ms	
d. Voting in sub Hough spaces 2	4.9 ms	
e. Summing sub Hough spaces 2	0.9 ms	

The constant implementation which takes bits as input for the input pixels, executes in 10.6 ms as shown in Table 2. Without packing each pixel into a single bit, but leaving it in a byte, increases the execution time to 18.6 ms. Packing the pixels from bytes to bits only takes 0.2 ms extra in the rotating stage, which makes it well worth the effort, although it also increases the program complexity.

A trade-off can be made between accuracy and processing speed. By reducing the number angles in the Hough space, the execution time is reduced, but so is its accuracy. The execution time of the fast and the constant implementation with a 3 degree and 6 degree accuracy can be found in Fig. 9. This figure shows that the execution time of the fast implementation scales linear with the number of edge pixels in the image. After some value (about 16% of all pixels being edge pixels), the constant implementation is even faster than the fast implementation.

6 Related Work

An OpenGL implementation of the Hough transform on a GPU is presented in [5]. Unfortunately no performance measurements are given, but it is mentioned that an array of all edge pixels is made on the CPU. The circle Hough transform has been implemented on a GPU by [13], also in OpenGL. Both papers use the rendering functions of OpenGL to calculate the Hough space. With the availability of CUDA nowadays, using OpenGL to program GPUs for general



(a) Accuracy 3 degrees (b) Accuracy 6 degrees

Fig. 9. Execution time of the Fast and Constant GPU implementation of the Hough Transform (HT) with 3 degrees and 6 degrees of accuracy.

purpose computations has fallen in disuse. One CUDA implementation of the Hough transform can be found in CuviLib [12], a proprietary computer vision library. It uses the polar representation of a line for the Hough transform. Next to calculating the Hough space, it also finds the maxima in the Hough space at the same time.

7 Conclusion

In this paper we have introduced two new implementations for the Hough transform on a GPU, a fast version and an input-data independent version. We have shown that the parameterization (Cartesian or polar) used for lines in images does not influence the processing speed of the Hough transform significantly. Optimizing the GPU code for speed does result in a significant improvement. Another way to optimize the GPU code is to make it input-data independent. Our result show that the fast-implementation is the quicker of the two for about 80% of the images. The input-data independent implementation has the same processing speed for every image, and is faster if the number of edge pixels exceeds a certain threshold (about 16% in our case). While the effort for making a basic GPU implementation is about an hour, creating the fast implementation can take a couple of days, and the constant implementation even weeks.

The program code for the input-data independent implementation is so complex that it is very hard to make any changes to parameters, like image size. The fast implementation does not suffer from this drawback. Therefore, and because it is the quicker solution for about 80% of input images, it is advisable to select the fast implementation of the Hough transform in every case where the processing time does not have to be fixed.

The input-data independent implementation shows that packing the input-data from bytes to bits can result in a large speed-up of the application. The GPU used in this paper already supports packing standard data types (char, int, float) into vectors of two, three or four elements. Packing bytes into bits would make a good addition to this, at little extra hardware costs.

Future work will include the Hough transform for circles. The corresponding Hough space is much larger than the Hough space for lines, since it has three dimensions instead of two. This will create a new trade-off between the fast and

the input-data independent approach. For the input-data independent implementation the image no longer has to be rotated, which would save over half of the processing time in the Hough transform for lines. But the final Hough space is much larger when detecting circles, which will limit the number of sub-Hough spaces which can be generated and makes them more costly to add together.

References

1. Allusse, Y., Horain, P., Agarwal, A., Saipriyadarshan, C.: GpuCV: An OpenSource GPU-Accelerated Framework for Image Processing and Computer Vision. In: 16th ACM international conf. on Multimedia. pp. 1089–1092. MM '08, ACM (2008)
2. Bramberger, M., Brunner, J., Rinner, B., Schwabach, H.: Real-Time Video Analysis on an Embedded Smart Camera for Traffic Surveillance. In: Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE. pp. 174 – 181 (May 2004)
3. Duda, R.O., Hart, P.E.: Use of the Hough Transformation to Detect Lines and Curves in Pictures. Commun. ACM 15 (January 1972)
4. Forsberg, J., Larsson, U., Wernersson, A.: Mobile Robot Navigation using the Range- Weighted Hough Transform. Robotics Automation Magazine, IEEE (1995)
5. Fung, J., Mann, S.: OpenVIDIA: Parallel GPU Computer Vision. In: Proceedings of the 13th annual ACM international conference on Multimedia. pp. 849–852. ACM, New York, NY, USA (2005)
6. Hough, P.: Method and Means for Recognising Complex Patterns. US Patent No. 3,069,654 (1962)
7. Nistér, D., Stewénius, H.: Scalable recognition with a vocabulary tree. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR). vol. 2, pp. 2161–2168 (June 2006)
8. Nugteren, C., van den Braak, G.J., Corporaal, H., Mesman, B.: High Performance Predictable Histogramming on GPUs: Exploring and Evaluating Algorithm Trade-offs. GPGPU 4 (2011)
9. NVIDIA Corporation: NVIDIA's Next Generation CUDA Compute Architecture: Fermi (2009)
10. NVIDIA Corporation: NVIDIA CUDA C Programming Guide - Version 3.1 (2010)
11. Takacs, G., et al.: Outdoors Augmented Reality on Mobile Phone using Loxel-Based Visual Feature Organization. In: Proceeding of the 1st ACM international conference on Multimedia information retrieval. pp. 427–434. MIR '08, ACM (2008)
12. TunaCode (Limited): Cuda Vision and Imaging Library, http://www.cuvilib.com/
13. Ujaldón, M., Ruiz, A., Guil, N.: On the computation of the Circle Hough Transform by a GPU rasterizer. Pattern Recognition Letters 29(3), 309–318 (2008)
14. Wang, Y., Shi, M., Wu, T.: A Method of Fast and Robust for Traffic Sign Recognition. In: Fifth International Conference on Image and Graphics, ICIG '09 (2009)