Fast Hough Transform on GPUs

MADHUR TANDON and MUDIT GARG, Indraprastha Institute of Information Technology, Delhi

1. ABSTRACT

Feature Extraction is an important step in Image Processing, Computer Vision and other related fields. One of the major features in an image are detection of instances of objects of a certain type. Hough Transform is a technique that helps to identify these objects via a voting mechanism. The objects or shapes are detected as a local maximum in the accumulator space which is implicitly constructed by the underlying algorithm.

Hough Transform has been initially applied to detect lines in an image. Further developments also include the detection of circles and in recent years, the Hough Transform has also been generalized to detect any arbitrary shape.

The algorithm usually involves many steps in the pipeline and thus takes a very long time to get good results. Since detection of these geometrical figures is only one step of a larger problem - say classification, it becomes difficult and quite impossible to be able to carry out real-time image processing pipelines.

With the advent of NVIDIA's GPUs and the CUDA API, efficient computations on matrices and vectors can be carried out exploiting the massively parallel capabilities of the GPU's hardware. In fact, almost every step of the pipeline can be parallelized and significant speedups can be obtained.

2. ANALYSIS OF ALGORITHM

The Hough Transform Algorithm includes several steps in the pipeline. These include

—Conversion To Grayscale
—Blurring of an Image
—Edge Detection
—Creation of Accumulator Array
—Thresholding
—Non-Maxima Suppression

Each of the steps in the pipeline are highly parallelizable. We discuss the above steps below:

2.1 Conversion To Grayscale

Coloured Images are primarily 3D in nature. There are 3 channels for Red, Green and Blue each. The conversion formula to grayscale is given by

$$0.299 * R + 0.587 * G + 0.114 * B$$

This is a pixelwise operation and this can be applied on each pixel independent of other pixels. Thus, a highly efficient implementation using GPU can be achieved.

2.2 Blurring of the Image

The images need to be blurred to remove some noises so that only those edges are kept which are relevant. An averaging kernel or a gaussian blurring kernel can be used to perform this operation. Regardless of which kernel is used, this process is essentially a convolution operation and the convolution of a kernel with an image can be performed parallely with the values being computed for each pixel as the kernel's center by a separate thread.

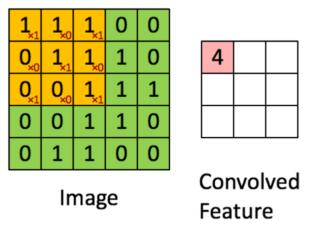
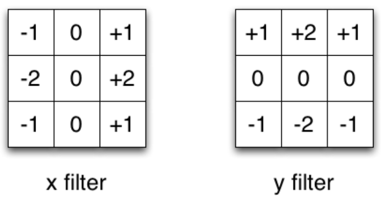


Fig. 1. Convolution Operation

2.3 Edge Detection

The edges in an image are the points for which there is a sharp change of color. These features can be captured by the gradient of an image. Gradient of Images are calculated using the Sobel Operators. This in fact is another convolution operation just like above.

The magnitude and orientation of the gradient can be computed per pixel independent of other pixels and thus, a GPU based implementation can help speed up this operation.


x filter
y filter

-1	0	+1
-2	0	+2
-1	0	+1

+1	+2	+1
0	0	0
-1	-2	-1

Fig. 2. Sobel Operators

2.4 Creation of Accumulator Array & Thresholding

2.4.1 *Algorithm for Circles.*

An equation of a circle is given by

$$(x-a)^2+(y-b)^2=r^2$$

. But this equation is of little use and the more general parametric equation

$$x=a+r*cos(\theta)$$

and

$$y=b+r*sin(\theta)$$

is used where θ ranges from 0 to 360 degrees.

The edge points ”x”, ”y” in the image domain are put in the above equation for a fixed radius ”r” and for ”θ” varying from 0 to 360. The corresponding values of ”a” and ”b” are found out and if these centers lie within the height and width of the image, the accumulator array at the index

$$a,b,r$$

is incremented. (Voting Mechanism)

For the filled accumulator array, only those ”a, b, r” are retained which have at least 40% of the total points as their votes. This threshold can be changed according to one’s convenience.

2.4.2 *Algorithm for General Geometric Shapes.*

The algorithm for general geometric shapes differs as no predefined mathematical equation exists for the curve of the object which is to be found out.

For the purpose of our implementation, we consider the case where the target object has a fixed orientation and size.

Following the above assumption, the Hough transform builds up an R Table for the target object. The algorithm for building up the R-Table proceeds as follows:

—Pick a reference point (ideally the centroid of the object (x_c, y_c))
—Draw a line from the reference point to the boundary.
—The vector r is simply the a vector from the centroid to the boundary point. It is calculated for every boundary point.
—The angle ϕ (that the tangent at boundary point makes with the X-Axis is found out) for every boundary point.
—The angle α that the vector r makes with the X-axis is also found out for every boundary point.
—The relation between the centroid (x_c, y_c) and the boundary point (x, y) is then given by $x_c = x + rcos(\alpha)$ and $y_c = y + rsin(\alpha)$
—For each unique angle ϕ, the vectors r along with their respective angles α are stored into the list indexed by ϕ.

—Since there can be multiple boundary points (with different r's and different α's) having the same angle ϕ, the creation of this R-Table is dynamic in nature.

This procedure is illustrated by the figures below:

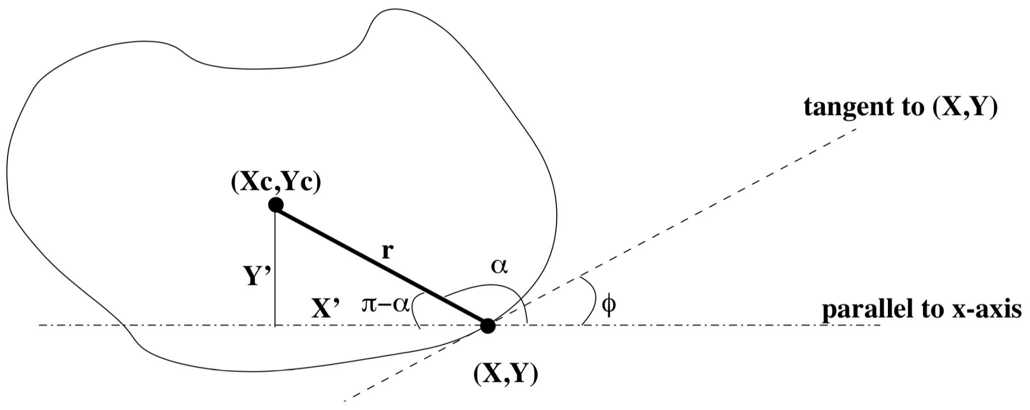


Fig. 3. General Shape Detection

ϕ_1:	(r_1^1, α_1^1), (r_2^1, α_2^1), ...
ϕ_2:	(r_1^2, α_1^2), (r_2^2, α_2^2), ...
ϕ_n:	(r_1^n, α_1^n), (r_2^n, α_2^n), ...

Fig. 4. Filling the R-Table

The procedure ultimately results in the reference point (x_c, y_c) stored as a function of these ϕ's.

Thus, the R-Table allows us to use the contour edge points and gradient angle to recompute the location of the reference point.

Having build up the R-Table, the voting mechanism takes place as follows:

—For Each Edge Point (x, y)

—Use the gradient angle ϕ to retrieve from R-Table, all the (r, α) values indexed under ϕ.

—For each (r, α), compute the candidate reference points using the equations $x_c = x + rcos(\alpha)$ and $y_c = y + rsin(\alpha)$.

—Increase the voting count i.e. $P[x_c][y_c] += 1$

3. PARALLELIZATION STRATEGY AND IMPLEMENTATION NOTES

3.1 For Circles

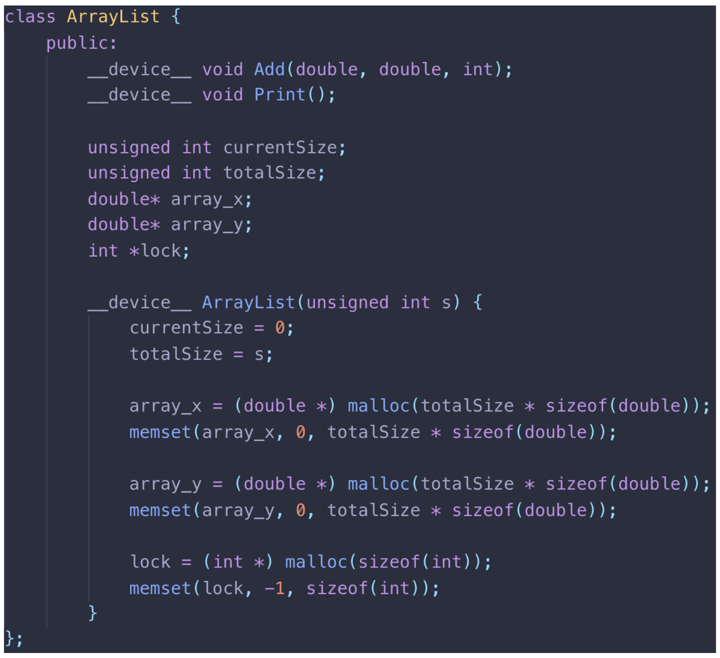
Several Kernels are made which perform respective steps of the pipeline. Specifically for Circle Detection using Hough Transform, the kernels implemented are mentioned below:

—grayscaleKernel
—convKernel
—sobelKernel
—accumulatorKernel
—thresholdingKernel

These perform the relevant operations and each kernel is run one after another to produce the final output. Out of the above, the kernels ”accumulatorKernel” and ”thresholdingKernel” are 3D kernels i.e. they use the ”z” dimension of *blockIdx*, *blockDim* and *threadIdx* as well. The other kernels are 2D Kernels.

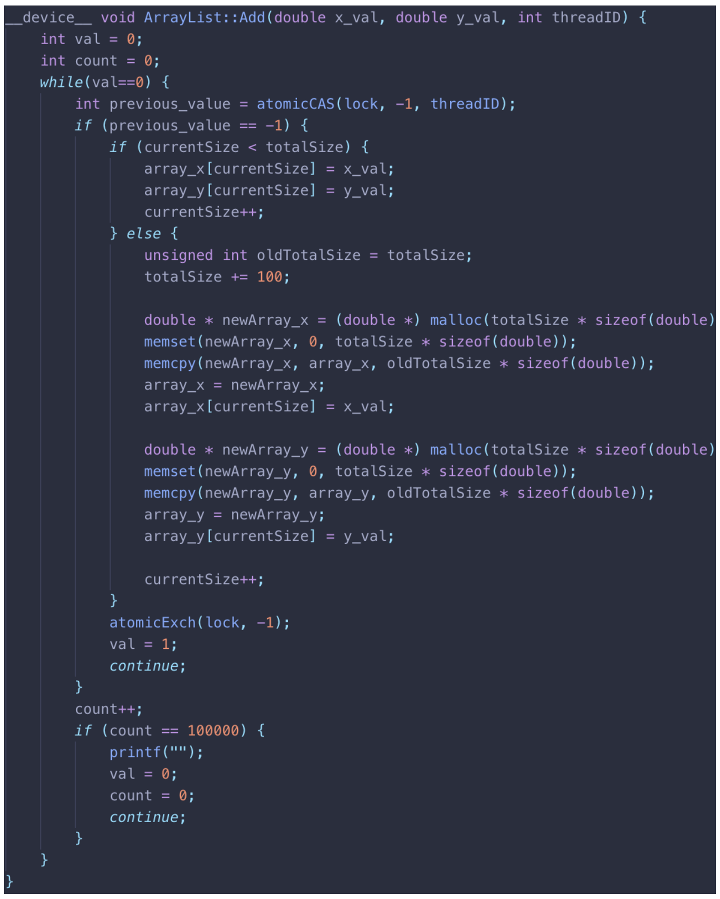
3.2 For General Geometric Shapes

Since in The R-Table, the list of (r, α) indexed for each ϕ can be arbitrarily long, a need for an ArrayList implementation inside the GPU model arises. After having experimented with several locking mechanisms and atomic operations, we have been able to build our own ArrayList library - the snippets of which are shown below:



```
class ArrayList {
    public:
        __device__ void Add(double, double, int);
        __device__ void Print();

        unsigned int currentSize;
        unsigned int totalSize;
        double* array_x;
        double* array_y;
        int *lock;

        __device__ ArrayList(unsigned int s) {
            currentSize = 0;
            totalSize = s;

            array_x = (double *) malloc(totalSize * sizeof(double));
            memset(array_x, 0, totalSize * sizeof(double));

            array_y = (double *) malloc(totalSize * sizeof(double));
            memset(array_y, 0, totalSize * sizeof(double));

            lock = (int *) malloc(sizeof(int));
            memset(lock, -1, sizeof(int));
        }
};
```

Fig. 5. Declaration of ArrayList



```
__device__ void ArrayList::Add(double x_val, double y_val, int threadID) {
    int val = 0;
    int count = 0;
    while(val==0) {
        int previous_value = atomicCAS(lock, -1, threadID);
        if (previous_value == -1) {
            if (currentSize < totalSize) {
                array_x[currentSize] = x_val;
                array_y[currentSize] = y_val;
                currentSize++;
            } else {
                unsigned int oldTotalSize = totalSize;
                totalSize += 100;

                double * newArray_x = (double *) malloc(totalSize * sizeof(double)
                memset(newArray_x, 0, totalSize * sizeof(double));
                memcpy(newArray_x, array_x, oldTotalSize * sizeof(double));
                array_x = newArray_x;
                array_x[currentSize] = x_val;

                double * newArray_y = (double *) malloc(totalSize * sizeof(double)
                memset(newArray_y, 0, totalSize * sizeof(double));
                memcpy(newArray_y, array_y, oldTotalSize * sizeof(double));
                array_y = newArray_y;
                array_y[currentSize] = y_val;

                currentSize++;
            }
            atomicExch(lock, -1);
            val = 1;
            continue;
        }
        count++;
        if (count == 100000) {
            printf("");
            val = 0;
            count = 0;
            continue;
        }
    }
}
```

Fig. 6. Add Operation of ArrayList

Further, we also make 3 new kernels that help us build the R-Table, these are given by

—rtable_init_kernel
—rtable_create_kernel
—accumulator_general

4. RESULTS OF IMPLEMENTATION

Given the inability of drawing shapes from the CUDA API, we restrict ourselves to the output of the Hough Space. This can then be taken as an input by the python program that can draw the detected circles OR mark the centroids of the general shape detected.

The Results are given below:

4.1 For Circles

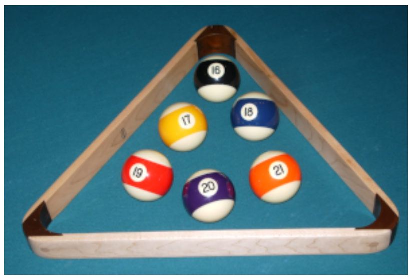


Fig. 7. Input Image

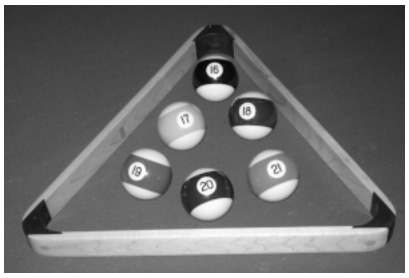


Fig. 8. RGB to GrayScale

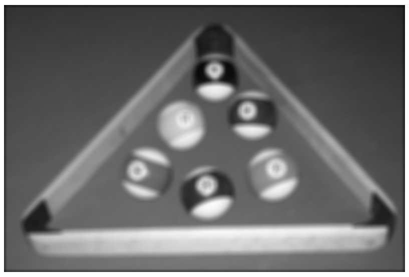


Fig. 9. Blurred Image

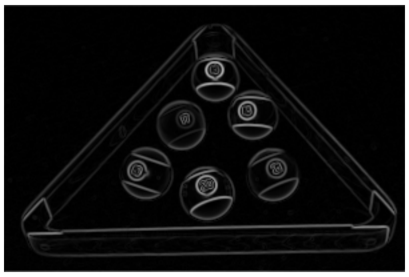


Fig. 10. Edge Detection

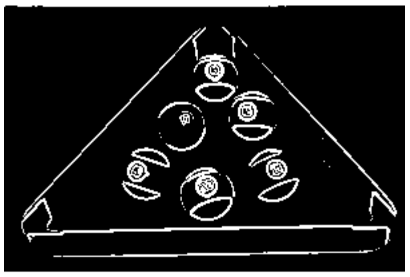


Fig. 11. Thresholding

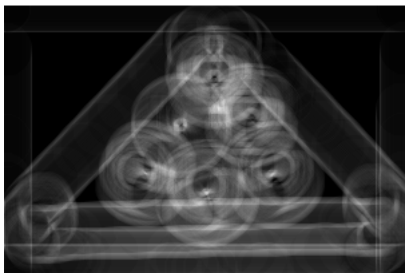


Fig. 12. Hough Space for a particular radius

4.1.1 *Performance Analysis for Circles.*

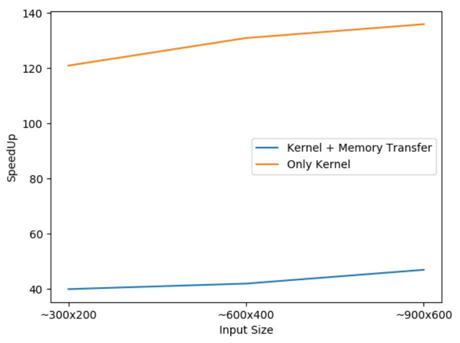


Fig. 13. SpeedUps for different Input Sizes

4.2 For a General Geometric Shape (Key)

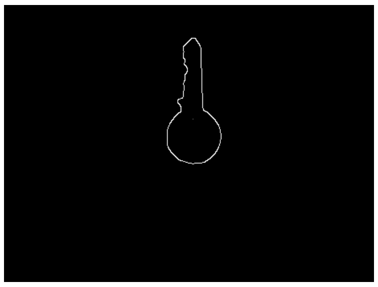


Fig. 14. A given Key

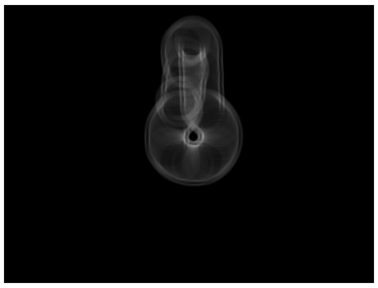


Fig. 15. Corresponding Hough Space

4.2.1 *Performance Analysis for General Geometric Shapes.*

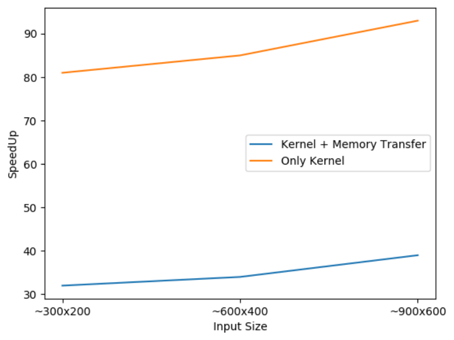


Fig. 16. SpeedUps for different Input Sizes

5. REFERENCES

[1] Su Chen and Hai Jiang
Accelerating the Hough Transform with CUDA on GraphicsProcessing Units