Fall 2022 ME/CS/ECE759 Final Project Report
University of Wisconsin-Madison

Hough Transform Implementation Using CUDA

Maninder Kumar

December 14, 2022

Abstract

Hough Transform is an important algorithm in the object detection process. It serves the purpose of detecting predefined shapes in the image frame. The basic type of Hough Transform serves to find the equations of straight lines in the image frame. Like any image processing algorithm, it also has a potential for pixel-by-pixel parallelization. Such parallel nature of the algorithm can be leveraged for speedups using GPU. However, unlike image filtering algorithms, Hough Transform needs to maintain a data structure called the Accumulator Matrix or Hough Space which can be incremented by multiple parallel threads dealing with distinct pixels. This creates a problem as these increments need to be serialized causing a bottleneck in the potential speedup that can be achieved. This project aims to explore different methods to implement Hough Transform for high potential speedups as compared to serial implementation using Nvidia GPUs. This report will discuss the algorithm, a basic implementation of the Hough Transform, a fast implementation of the Hough Transform and the speedups obtained for Polar Hough Transform on a set of four images of different resolutions.

Link to Final Project `git` repo:
https://git.doit.wisc.edu/MKUMAR46/repo759.git
in FinalProject759 sub-directory

Contents

1. General information

1. **Home Department:** Electrical and Computer Engineering.
2. **Current status:** MS Student.
3. I am not interested in releasing my code as open-source code.

2. Problem statement

The aim of this project can be summarized in three bullet points:

1. To successfully implement Hough Transform in CUDA.
2. To find ways to optimize the implementation for higher speedups.
3. To benchmark the implementations against the serial implementation for speedup estimation.

Hough Transform is a perfect example of an imperfect parallel algorithm. If offers high parallelization over the processing of individual pixels but the computation needs to be serialized over multiple parallel threads. Such serialization is done in GPU using atomic operations. However, there are ways to reduce the cost of these atomic operations [1]. This project is an exploration to understand this bottleneck and its potential solution.

3. Solution description

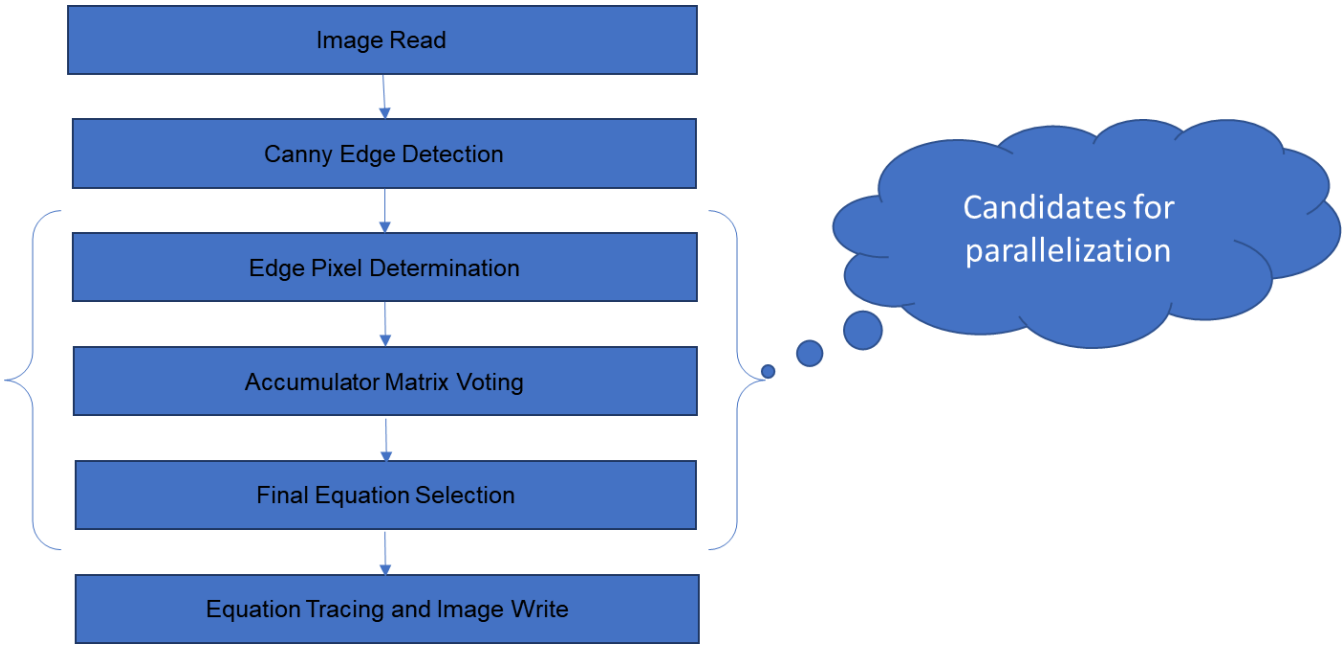


Figure. 1

Figure. 1 demonstrates the setup created for implementation and evaluation. The individual steps are briefly discussed below:

Image Read: Read the image in JPEG Format and store it in monochrome (8-bit) format in an internal array. It has been implemented using an open-source library [2].

Canny Edge Detection: Use Canny Edge detection to contrast edges in the image, the output is in form of a matrix with value=255 for edge pixels and 0 otherwise. The code for Canny Edge detection has been used from [3].

Edge Pixel Determination: Find out coordinates of the edge pixels.

Accumulator Matrix Voting: Accumulator Matrix for polar lines is a 2-D matrix in θ and ρ dimensions. Where $\theta \in$ [-90, 90] and $\rho \in$ [-*diag*, *diag*] where *diag* is the length of the diagonal of the image. Generally, θ and ρ are taken in steps of 1. This gives 181 columns and 2*diag+1 rows in the accumulator matrix.

Using the polar equations, a line can be represented as:

$$\rho = x\cos\theta + y\sin\theta$$

For a particular coordinate set (x,y), for each theta between -90 to 90, the value of ρ can be determined and correspondingly each (θ, ρ) can be incremented by 1 in the accumulator matrix. This process is repeated for all the edge pixels.

Final Equation Selection: For each entry in the Accumulator Matrix, check whether the number of votes for that entry exceeds a particular threshold. If yes, then this is a line in the image, add it into a list of polar line equations.

Equation Tracing and Image Write: For every equation of line found, trace that line over original image in red and output the resulting image into .jpeg format. The image write function has been taken from [2].

As shown in the Figure. 1, the Edge Pixel Determination, Accumulator Matrix Voting and Final Equation Selection are candidates for parallelization on GPU. These operations are the core of Hough Transform algorithm.

Baseline Implementation:

```
__global__ void voting_kernel(uchar * edge_array, int* hough_space,  int width, int height, int diag) {

        //Pixel indices
        int i, j;

        int rho;
        double theta_rad;


        int index = blockDim.x*blockIdx.x + threadIdx.x;

        i = index/width;
        j = index%width;

        if(index< width* height){
                if(*(edge_array + index) == 255) {
                        for(int theta = 0; theta <= 180; theta += BIN_WIDTH) {

                                theta_rad = ((double)((double)(theta -90) /180)) * PI;
                                rho = round(j * cos((double)(theta_rad)) + i * sin((double)(theta_rad))) + diag;

                                //Atomically vote in the hough space
                                atomicAdd((hough_space + rho*(NUM_BINS+1) + theta), 1);
                        }
                }

        }
}
```

Figure 2. Edge Pixel Determination and Accumulator Matrix Voting.

Figure. 2 shows the code snippet for CUDA implementation of Edge Pixel Determination and Accumulator Matrix Voting. Here, each thread examines one pixel and if that pixel is an edge pixel, its coordinates are used for voting in the Accumulator Matrix. Here, the Accumulator Matrix is present in the global memory. So, the latency cost for atomic accesses will accrue in terms of global memory accesses.

```
__global__ void equation_forming_kernel (int* hough_space, int input_length, int* rho_vals, int* theta_vals, int* output_length, int threshold, int diag) {


        extern __shared__ int s [];
        //Shared Memory Arrays
        int* arr_size = &s[0];
        int* rho_arr =  &s[1];
        int* theta_arr =  &s[1+ blockDim.x];
        int rho, theta;
        int smem_array_size;
        int store_index;
        int index = blockDim.x*blockIdx.x + threadIdx.x;

        if(threadIdx.x ==0) {
                //Shared Memory variables are always uninitialized, initialize it here:
                *arr_size=0;
        }

        __syncthreads();

        if(index< input_length) {
                if(*(hough_space + index) > threshold) {
                        rho =   index/(NUM_BINS+1);
                        theta = index%(NUM_BINS+1);
                        rho = rho - diag;
                        theta = theta -90;

                        store_index = atomicAdd(arr_size, 1);

                        rho_arr[store_index] = rho ;
                        theta_arr[store_index] = theta;
                 }
        }

        __syncthreads();

        smem_array_size = *arr_size;
        if(threadIdx.x ==0) {
                *arr_size = atomicAdd(output_length, smem_array_size);
        }

        __syncthreads();

        if(threadIdx.x < smem_array_size){
                rho_vals[(*arr_size)+ threadIdx.x] = rho_arr[threadIdx.x];
                theta_vals[(*arr_size)+ threadIdx.x] = theta_arr[threadIdx.x];
        }
}
```

Figure 3. Final Equation Selection

In Figure. 3, the CUDA implementation of Final Equation Selection step is shown. Here, each thread tests one entry in the Accumulator Matrix and if the value of that entry is greater than a line_threshold, that entry is selected for the final equation list. The next step is to form an array of all the (θ, ρ) pairs so that it can be passed to the host processor for further processing. Here, every positive thread in the thread-block first adds (θ, ρ) value pair over an array in the shared memory. The index is read and updated using atomic fetch-and-add operation, the value read is used as an index to insert the equation in the array. After a thread block collects all the equations in its shared memory, a single thread reads and increments a counter in the global memory. The value read then is used as an index to copy the shared memory array as a slice of a global memory array.

Fast Implementation Using Shared-Memory Voting

The cost of atomic increments during Accumulator Matrix voting can be optimized. [1] suggests one possible optimization for images with low edge percentage (as the case with sample input images as discussed in the result section). To achieve this, first the edge pixel coordinates are filtered separately (Figure. 4), the algorithm to do so is same as discussed for equation_forming_kernel in Figure. 3. Once an array for all the edge pixel coordinates is formed in the global memory, the voting step can be optimized by replacing atomic additions in the global memory with atomic additions in the shared memory.

```
__global__ void found_edges_kernel(uchar * edge_array, int width, int height, int* x_coor_g, int* y_coor_g, int* edge_array_size) {

        int index;
        extern __shared__ int s[];
        int* str_index_local = &s[0];
        int* x_coor_s = &s[1];
        int* y_coor_s = &s[blockDim.x+1];
        int str_index =0;
        int x;
        int y;
        int input_size = height*width;

        index = blockIdx.x * blockDim.x + threadIdx.x;

        if(threadIdx.x ==0) {
                *str_index_local=0;
        }
        __syncthreads();

        if(index< input_size) {

                x = index %width;
                y = index /width;
                if(edge_array[index] ==255) {
                        str_index = atomicAdd( str_index_local  ,1);
                        x_coor_s[str_index] = x;
                        y_coor_s[str_index] = y;
                }
        }

        __syncthreads();
        str_index = *str_index_local;

        if(threadIdx.x ==0) {
                *str_index_local = atomicAdd(edge_array_size , str_index);
        }
        __syncthreads();

        if(threadIdx.x< str_index) {

                x_coor_g[*str_index_local + threadIdx.x ] = x_coor_s[threadIdx.x];
                y_coor_g[*str_index_local + threadIdx.x ] = y_coor_s[threadIdx.x];
        }
}
```

Figure 4. Separate implementation of Edge Pixel Determination step.

Figure. 5 shows the fast implementation of the Accumulator Matrix voting step. Here, each thread-block votes for a particular value of θ. This slices the Accumulator Matrix and reduces the shared memory requirement for storing the Accumulator Matrix during the voting step. Each thread in the block handles one edge pixel and based on its coordinates calculates ρ, then the value corresponding to that ρ is incremented in the Accumulator Matrix. There are two points to note here:

1. The amount of shared memory required is dependent on the size of image, numerically (size of integer)*2**diag*, where *diag* is the length of the diagonal of the image and is equal to $\sqrt{w^2 + h^2}$.

2. Each thread may do the evaluation for more than one edge pixel if the size of the threadblock is smaller than the number of edge pixels.

After all the pixels are done voting for the Accumulator Matrix slice in the shared memory, it is copied into the global memory. It can be noted that here we do not need to read and increment a counter in the global memory as the position of this slice is fixed (based on the value of θ) in the complete Accumulator Array that resides in the global memory.

```
__global__ void voting_kernel(int* x_coor, int* y_coor, int* found_edges_device, int* hough_space,   int diag) {

        //Pixel indices
        int i, j;
        extern __shared__ int s [];
        int* hough_sub_s = &s[0];
        int rho;
        double theta_rad;
        int input_size = *found_edges_device;
        int theta = blockIdx.x;
        int iters;

        iters = ((2*diag) + blockDim.x -1)/blockDim.x;

        //Initialize the shared memory local address space.
        for( int p=0; p< iters; p++) {
                if((blockDim.x*p + threadIdx.x) < 2*diag) {
                        hough_sub_s [blockDim.x*p + threadIdx.x] =0;
                }
        }

        __syncthreads();

        iters= (input_size + blockDim.x-1)/blockDim.x;
        for( int p=0 ; p< iters ; p++) {
                if((blockDim.x*p + threadIdx.x)< input_size) {
                        i = y_coor[blockDim.x*p + threadIdx.x];
                        j = x_coor[blockDim.x*p + threadIdx.x];
                        theta_rad = ((double)((double)(theta -90) /180)) * 3.141592653589793238;
                        rho = round(j * cos((double)(theta_rad)) + i * sin((double)(theta_rad))) + diag;
                        //Atomically vote in the hough space
                        atomicAdd((hough_sub_s+ rho), 1);
                }
        }

        __syncthreads();

        iters = ((2*diag) + blockDim.x -1)/blockDim.x;
        for( int p=0; p< iters; p++) {
                if((blockDim.x*p + threadIdx.x) < 2*diag) {
                        hough_space[theta*2*diag + blockDim.x*p + threadIdx.x ] = hough_sub_s [blockDim.x*p + threadIdx.x];
                }
        }
}
```

Figure 5. Fast implementation of the Accumulator Voting step.

4. Overview of results. Demonstration of your project

Both CUDA based implementations along with the serial implementation have been timed on **Euler on Nvidia Geforce 1080 GPU.** The benchmarking has been done for four different images, specifications of which are shown in Table. 1.

Image	Dimensions	Edge %
Zebracrossing.jpg	2400x1600	0.588
Sudoku.jpg	558 x 563	3.71
Texture.jpg	512 x 512	1.49
Triangles.jpg	4096 x 3112	1.38

Table 1.

Table. 2 tabulates the final evaluations.

Image	Time for Serial Implementation (ms)	Baseline		Fast	
		Time (ms)	Speedup	Time (ms)	Speedup
Zebracrossing.jpg	253.365057	34.376831	7.37	5.364096	47.26
Texture.jpg	37.520016	5.614400	6.68	0.942368	39.83
Sudoku.jpg	105.848696	7.778880	13.6	2.044288	51.78
Triangles.jpg	1662.309107	193.675354	8.58	30.130304	55.17

Table 2.

Baseline implementation gives roughly 8X speedup while the Fast implementation gives a rough speedup of 50X.

5. Deliverables:

The following deliverables are part of the project:

All the paths mentioned start from the **FinalProject759** subdirectory of the *git* repo:

https://git.doit.wisc.edu/MKUMAR46/repo759.git

Input Images: Four input images as mentioned in Table. 1 are given. These images can be found in sample_images/ subdirectory.

Source Code:

Baseline:

Source File: hough_baseline.cu
To Compile: make baseline

Fast:

Source File: hough_voting_in_shared_mem.cu
To Compile: make voting_in_shared_mem

Run/Sbatch script: run.sh -> Contains the commands for all the 4 inputs.

Standalone Run Command:

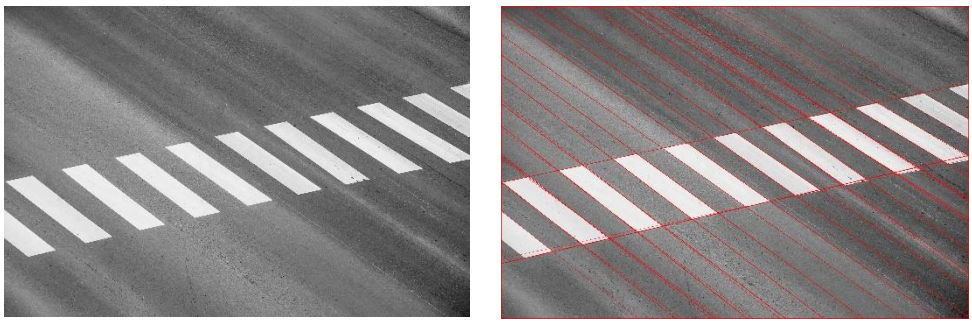
./hough <Image Path> <line_threshold> <canny_threshold>

./hough sample_images/zebracrossing.jpg 160 100

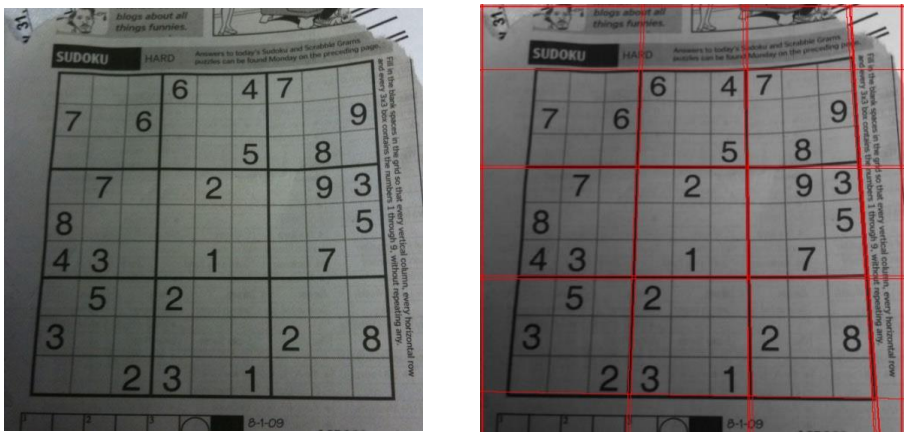
<line_threshold> is the threshold to select the equations after voting is done while the <canny_threshold> is the threshold used for Canny edge detection.

Output Images: The resulting outputs for all the four inputs are present in the outputs/ subdirectory. For each implementation, there is a separate subdirectory in the outputs/ directory.

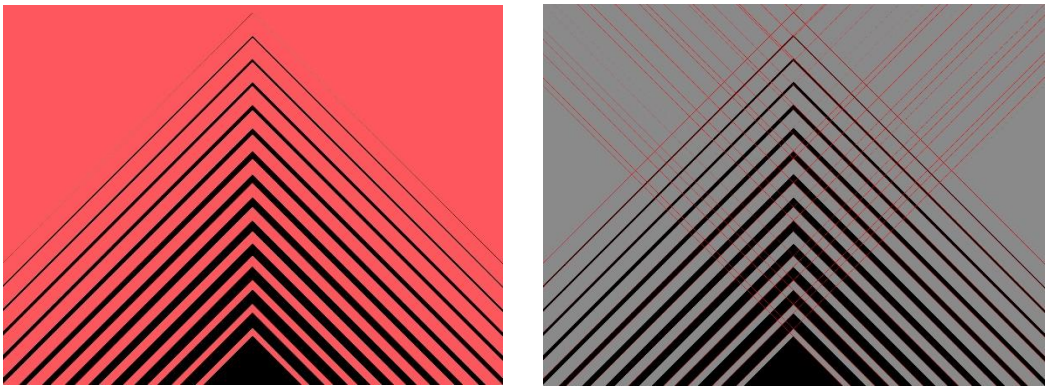
Demonstration of the Outputs:



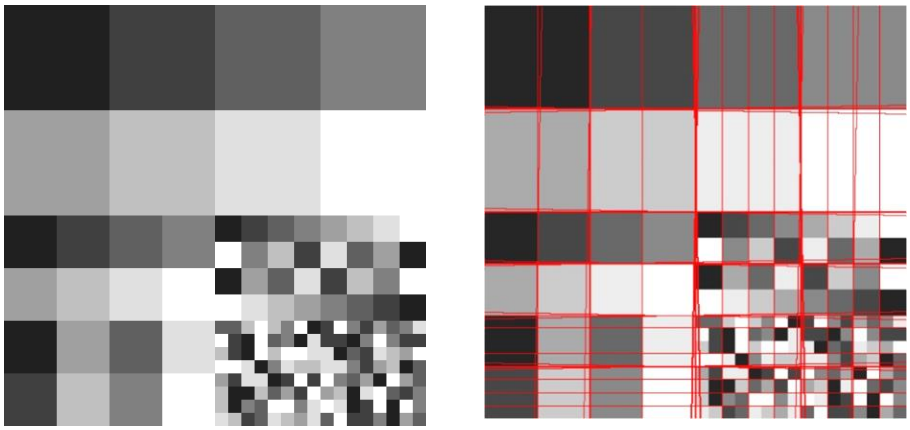
Zebracrossing.jpg



sudoku.jpg



Triangles.jpg



texture.jpg

6. Conclusions and Future Work

The project builds upon the CUDA Programming concepts taught in the class, specifically shared memory accesses and atomic operations. This project successfully achieves an initial analysis and implementation of two different Hough Transform CUDA implementations. It also times the serial and CUDA implementations to evaluate the performance gains.

The voting kernel in the fast method is dependent on the number of edge pixels and shines well with smaller edge pixel percentage. [1] also discusses another fast implementation of the Hough Transform which has an execution time independent of the edge pixel percentage. This input independent solution, however, is slower than the fast implementation discussed in this report for smaller number of edge pixels. A polymorphic solution can be devised which either breaks the image into smaller sub images or dynamically switches the method of computation.

Also, there are several common optimizations like non-conflicting shared memory accesses, aligned and coalesced global memory accesses, SM occupancy analysis-based optimization and usage of lookup tables stored in the constant memory to compute sin and cosine mathematical functions. Exploration of such optimizations was planned but couldn't be done due to time constraints and remain as a scope for future work.

References

[1] van den Braak, GJ., Nugteren, C., Mesman, B., Corporaal, H. (2011). Fast Hough Transform on GPUs: Exploration of Algorithm Trade-Offs. In: Blanc-Talon, J., Kleihorst, R., Philips, W., Popescu, D., Scheunders, P. (eds) Advanced Concepts for Intelligent Vision Systems. ACIVS 2011. Lecture Notes in Computer Science, vol 6915. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-23687-7_55

[2] https://github.com/nothings/stb.git (last visited December 10, 2022)

[3] https://rosettacode.org/w/index.php?title=Canny_edge_detector&oldid=329226 (last visited December 10, 2022)

[4] https://git.doit.wisc.edu/MKUMAR46/repo759.git (last visited December 14, 2022)

Special Note:

[4] hasn't been mentioned in the text but provided an initial support as a reference model in debugging and planning the code.