# Real-Time Line Detection via GPU-Based Hough Transform

Xián García Veiga, José Rodrigo Sanjurjo Amado, and Margarita Amor López

Grupo de Arquitectura de Computadores (GAC), Faculty of Computer Science, Universidade da Coruña, 15071 A Coruña, Spain
Centro de Investigación CITIC, Universidade da Coruña, 15071 A Coruña, Spain
Correspondence: x.veiga@udc.es

DOI: *https://doi.org/10.17979/spu.23.c29*

*Abstract*: The presence of noise and other imperfections in images or point clouds often hampers the accurate detection of geometric shapes, patterns, and objects. The Hough transform is a robust feature detection technique that identifies shapes through a voting mechanism, making it particularly resilient to noise and discontinuities, especially effective for detecting straight lines. This robustness makes it a valuable tool in a wide range of applications, such as industrial inspection or autonomous driving systems. However, its traditional implementation is computationally intensive, limiting its suitability for real-time tasks. Our approach addresses this limitation by offloading the most compute-intensive components to the GPU (Graphics Processing Unit), significantly improving execution time.

## 1 Introduction

Object detection is a fundamental operation for many digital signal processing and computer vision applications. The line is one of the most fundamental geometric primitive that can be found in many places in our world; therefore, its detection and characterization is a basic and essential process for many applications.

The Hough transform is a digital image processing technique for finding parametric shapes, such as lines, circles, and ellipses, in digital images through a voting mechanism. The main advantage of this method is its robustness against noise and interruptions in the image, being able to detect shapes even with the presence of substantial noise, partial occlusion or discontinuities. It is a widely used technique in many fields, such as manufacturing engineering, for instance for the detection of in-situ welding defects [Wang (2024)]. In forestry engineering, the Hough transform is used to identify key elements, such as conductors on powerlines based on high density point-cloud data [Yermo et al. (2024)]. In territorial analysis, usually a high number of tiled images need to be processed in order to identify key features. For example, in [Widyaningrum et al. (2020)] it is used to perform building boundary detection from airborne LiDAR data. However, it is a computationally intensive operation, which traditionally makes it unsuitable for real-time applications.

On the other hand, GPUs excel on highly parallel computation, which makes the Hough transform a prime candidate to be solved using GPUs. In [Yermo et al. (2024)], a CUDA implementation of an improved Hough transform algorithm is presented, for airborne LiDAR point clouds. In [Lin et al. (2009)], the Hough transform is used as a component for a video segmentation pipeline, and in [Gómez-Luna et al. (2011)], the Generalized Hough Transform is presented in the context of load balancing and occupancy maximization. In [Aldama (2025)] a complete GPU implementation for small images of equal size is presented, using bitonic-sort

as the sorting algorithm on one of the steps. However, with bigger and a higher number of images, other sorting algorithms can scale more efficiently, achieving higher performance.

In this work, we present the preliminary stages of a Hough line detection proposal specifically designed to be run efficiently on GPUs while handling a large number of images or tiles of variable sizes. First, the Hough Transform for line detection is presented, and the fundamental parts of the algorithm are described, as well as the notations used to express the lines in polar form. Next, the specific details of implementationse are explained, highlighting the specific challenges found when targeting execution on GPUs. Then, experimental results are shown, comparing the execution times against other existing implementations. Finally, a series of conclusions are drawn, focusing on possible improvements and future work.

## 2 Hough Transform

The Hough transform is a widely known technique in image processing for the detection of geometric shapes. The technique was presented in [Paul V C Hough (1962)] as a method for recognizing complex patterns in physics research. In [Duda and Hart (1972)] the methodology was extended by introducing a parametric representation using polar coordinates, and later popularized in the computer vision industry [Ballard (1981)]. Many variations have been created since then, and the technique was further extended to detect more complex shapes [Fernandes and Oliveira (2012)].

The algorithm is based on transforming the problem of line detection from the Cartesian coordinate domain to a parametrized space based on polar coordinates. The algebraic representation of a line in cartesian coordinates is $y = m \cdot x + n$, with $m$ being the slope and $n$ the intercept at the origin. However, this form is unable to represent vertical lines. Therefore, the Hough transform uses the polar representation

$$\rho = x \cdot cos(\theta) + y \cdot sin(\theta) \tag{20.1}$$

where $\theta$ and $\rho$ are the angle of the line and the distance from the line to the origin, respectively. Therefore, any line in a cartesian coordinate system can be represented by a single point in the Hough polar space, corresponding to a unique set of $\theta$ and $\rho$. As a consequence, a point on the input image is represented as a sinusoid on the Hough space. The line between two, three or more distinct points on the cartesian space is the intersection between the sinusoids representing those points on the Hough domain. The figure 1 illustrates a point and a line on the cartesian domain being transformed to the Hough space.

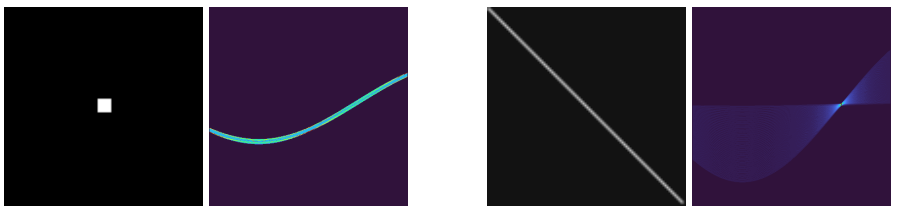

Figure 1: Point (left) and line (right) on the cartesian space transformed to the Hough domain

The fundamental part of the algorithm is the accumulator matrix. It stores votes in the Hough space for each set of parameters that correspond to a potential line on the original image. For each point on the input, the cells in the accumulator are incremented for all possible lines that could pass through that point. The cells with the highest values represent a higher probability of a line being present in the original image. Then, the values are filtered with a configurable threshold $\tau$, which eliminates undesirable noise, invalid shapes or very short segments. The remaining values are then interpreted as lines, expressed in polar coordinates and with infinite length, which can be converted to any other representation afterward.

The number of cells in the accumulator dictates the precision with which the lines can be detected. Usually, $\theta$ comprises an interval from 0° to 180° (or equivalently -90° to 90°), and is normally divided into bins of 0.5° or 1°. The line distance $\rho$, is measured in pixels, and therefore is dependent on the image size. Each cell represents a set of values for $\theta$ and $\rho$. Both the number of angular divisions $\Delta\theta$ and the pixel resolution per cell $\Delta\rho$ are configurable parameters, and their selection is very important in order to balance accuracy and efficiency. The representation in polar coordinates and the voting mechanism makes the algorithm very resilient to discontinuities or noise in the input image. However the size of the accumulator matrix notably increases the computation cost, both in terms of memory and compute time.

## 3 Implementation on a CUDA GPU

CUDA GPU's are composed of a set of SM (Streaming Multiprocessors). Each SM contains several cores, which follow a SIMT (Single Instruction Multiple Thread) paradigm, where several threads can swap execution in order to optimize resource usage. This strategy hides the latency of memory transactions. The code that is executed on the GPU is denoted *kernel*, which is launched with a configurable number of threads, which in turn are grouped in blocks. A block resides on the same SM until its execution is completed. Upon kernel instantiation, the number of blocks and threads per block is set by the programmer. The optimal value depends on the kernel resource usage and the GPU model. On the other hand, on CUDA GPUs, the memory hierarchy is composed of several levels. Global memory is accessible from all threads, it has the most capacity, but also the highest latency. Shared memory is considerably faster, but it can only be accessed within a thread block, and its limited size constraints its usage.

The Hough transform takes a preprocessed edge image as input, and attempts to locate those edges that are placed as a straight line. The entire process can be divided into four distinct stages: Accumulation matrix (voting), maximum local search, sorting and polar coordinate calculation.

Each step has data dependencies on the previous stage, so a synchronization barrier with `__syncthreads()` is enforced between steps. In order to minimize kernel launch overhead, all of the steps are consolidated into a single kernel. Each image is processed on a single GPU block, such that multiple images can be processed in *batch* on concurrent running thread blocks.

The accumulation matrix, or voting, is calculated over every pixel of the edge image, computing the transformation to the Hough space, and incrementing the cells corresponding to that point sinusoidal on the accumulator. It is the most computationally demanding step, as each pixel requires updating multiple cells. The accumulation matrix is usually larger than the image itself, requiring large amounts of memory, especially with high-resolution images. For this reason, the accumulator values are stored on *global* memory on the GPU, which is required due as the size of the accumulator matrix and derived structures far exceed the maximum capacity of *shared* memory. In order to maximize memory coalescence access [Kirk and mei W. Hwu (2016)] and efficiency, consecutive threads process a row of consecutive pixels, looping over rows of the available image, such that the data read by threads is part of the same data word wherever possible. This way, memory transactions are minimized. The accumulator matrix values are incremented using `atomicAdd()`, as it ensures no race conditions can occur if two threads need to write to the same cell in global memory.

For the local maximum search, threads iterate over the entire space of the accumulator and check whether the value of each cell is greater than a preset threshold $\tau$. For those that fulfill this condition, the values of the four neighboring cells are checked to determine whether the current value is a local maximum. In that case, the cell is added to a list, representing a detected line in that location. As the threads are processing multiple cells in parallel, storing the maximum values on a list-like data structure requires synchronization. A fixed-size array is allocated, whose size is configurable as a runtime parameter and constrains the maximum number of lines that can be found on the image. On this structure, the values of the highest cells is stored,

and a counter keeps track of the next free position in the array. When a local maximum is found, the thread first reserves the next position by increasing the counter using `atomicAdd()` and then writes the value, warranting that no other thread attempts to write to the the same location.

Next, a sorting process is introduced in order to rank the lines by their number of votes. This ensures that the lines that had the most prominent features, or had the largest amount of contributing elements are placed first. Efficient sorting algorithms for GPUs require being able to divide the work into several lockstep parallel operations. Initially, we chose to use radix sort as provided by the *Cub* [Nvidia (2025)] library. However, this function must be called from the CPU, so it requires splitting the algorithm in two kernel launches in order to perform the sorting, which increased overhead significantly.

For this reason, we chose to implement the sorting directly as part of a single kernel. Bitonic sort was selected, which is a parallel sorting algorithm that can be efficiently performed on GPU architectures. It is a comparison algorithm that builds bitonic sequences, in ascending or descending order, and then merged into a sorted sequence. However, other alternatives are being explored to increase performance further. Radix sort is still a prime candidate for future iterations, as it offers reduced complexity with a large number of elements, compared to other sorting algorithms.

Finally, a polar coordinate calculation step is performed, where the position of the cells that had the highest values in the Hough space is converted back to polar coordinates, representing the final lines found. Once the algorithm is finished, the line positions can be drawn and overlaid over the original input image, in order to verify that line detection is performed correctly. An example is shown on Figure 2.

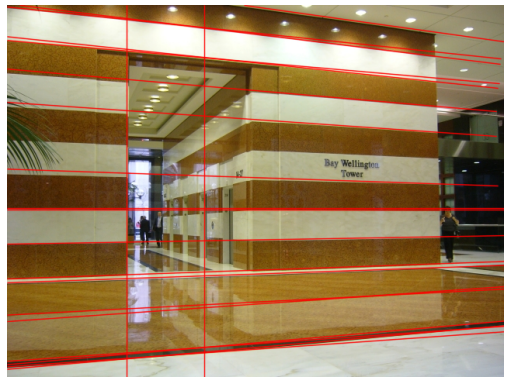


Figure 2: Example of a Hough lines detection result on an image from the York Urban Line Segment Database [Denis et al. (2008)]. Detected lines are overlaid over the original image

### 3.1 Batch processing

The implementation is designed to perform line detection on several images or tiles at once. This is useful, for example, when processing tiles of a large satellite or point-cloud image. For this reason, each image is processed on an independent GPU thread block, which in turn is executed on a separate SM on the GPU hardware. This design allows independent processing of images of different sizes, or with different parameters, without requiring separate kernel launches or reconfiguration of the underlying data structures, reducing overhead. Prior to the GPU kernel launch, the memory footprint for all of the auxiliary data structures is calculated and allocated, taking into account the dimensions of each image. This functionality requires storing the offsets for the data structures belonging to each image into separate arrays that are passed to the GPU kernel as arguments. Code in the kernel is then responsible for calculating

the corresponding offsets and accessing its allocated memory regions, depending on the image index assigned to each block.

## 4 Results

Preliminary benchmarks were run on a Nvidia GeForce RTX 4090, based on the Ada Lovelace architecture. Input images were taken from the York Urban Line Segment Database [Denis et al. (2008)], which consists of 102 images with a resolution of $640 \times 480$ of indoor and outdoor urban environments, whose lines are validated to satisfy a 3D orthogonal relationship with the rest of the environment in the frame.

In order to test our variable-size approach, we cropped ten independent slices for each image. Their size is randomized on the range $w = [320, 640]$ for the width, and $h = [240, 480]$ for the height to test the performance with differing image sizes. A preprocessing step is performed first, using OpenCV's Canny edge detection. Then, the images are preloaded in the system RAM. The total time is measured as the interval between launching the function until all line information is returned. Before the *kernel* launch, the allocation and initialization of data structures is performed sequentially on the CPU, as well as the copy of the results once the execution is finished. With the data set described, the initialization time on CPU was 0.453 s, the kernel compute took 0.351 s, and the result copy took 0.153 seconds. Upon kernel launch, an important consideration is the number of threads per block. A higher value helps hide memory latencies whenever a *warp* stalls on memory transactions. However, performance can be constrained by resource availability, so finding the optimal number for each use case often requires empirical testing. Table 2 shows the kernel execution time with varying numbers of threads per block. We show results for both our initial proposal using the *Cub* library sorting function, and our own single-kernel bitonic sort implementation. Each thread is responsible for computing a part of the accumulation matrix, so a higher number of threads incurrs more parallelism. Memory transactions are also a limiting factor. For these reasons, the optimal value of threads is the highest available on the hardware, 1024.

We compared our best configuration against OpenCV's Hough lines detection, with both the CPU [OpenCV (2025a)] and the CUDA GPU [OpenCV (2025b)] variants. The OpenCV API does not currently have a batch processing function, so the `Hough Lines` function is called once per image in a loop. Therefore, the reported execution times for GPU include the time taken for memory copies, both for OpenCV and for our implementation. The results were obtained by applying a Hough threshold $\tau = 150$. Additionally, we used $\Delta\theta = \pi/180$ steps. Benchmarks with the set of images of variable size described earlier are shown in table 1. One important difference, is that the GPU version of OpenCV internally uses CUDA streams, which can overlap compute and memory transactions, and reducing total memory consumption and latency. Ours, currently stores the images in memory first, and then performs the compute.

We also compared performance against the implementation presented in [Aldama (2025)], which is designed for batch processing of images of fixed size. Internally, it also uses bitonic sort, and tries different configurations, including single a kernel one. However, the most optimal configuration performs two kernel separate launches. For this comparison, the images of the dataset were duplicated nine times to increase computational complexity. Execution times are shown on table 3, where our implementation is faster, in part due to its lower kernel launch overhead.

| | OpenCV CPU | OpenCV GPU | Ours |
|---|---|---|---|
| Time(s) | 27.563 | 1.761 | 0.977 |

Table 1: Total execution time for a set of 5100 variable size tiles

| Threads per block | 32 | 64 | 128 | 256 | 512 | 1024 |
|---|---|---|---|---|---|---|
| With `DeviceSegmentedRadixSort` | 8.755 | 5.673 | 4.587 | 4.387 | 3.196 | 2.570 |
| Single kernel with radix sort | 3.732 | 3.849 | 3.411 | 2.701 | 1.786 | 0.977 |

Table 2: Total execution time in seconds for different threads per block configurations

| | OpenCV CPU | OpenCV GPU | Aldama (2025) | Ours |
|---|---|---|---|---|
| Time (s) | 4.899 | 0.350 | 0.889 | 0.292 |
| Speedup (CPU) | - | 14.00 | 5.51 | 16.78 |

Table 3: Execution time and speedup for various implementations using 918 images with a fixed resolution of $640 \times 480$

## 5 Conclusion

In this work, a preliminary version of a real-time line detection algorithm is presented, based on the Hough Transform, that runs on CUDA GPUs. The preliminary results show reduced execution time over the comparable OpenCV Hough lines implementation on GPU, while being able to process images of different sizes.

The current version of the algorithm extensively uses the GPU's *global* memory. However, most of the data in the accumulator matrix is composed of zero values. Future optimizations are planned on the topic of using sparse representations to reduce the memory size and bandwidth requirements on the voting and local maximum search steps, and also on using other levels of the memory hierarchy to improve performance further. Other improvements may arise from memory alignment, and other storage optimizations. Another optimization comes from the sorting algorithm, which as shown on the experimental results, can have a considerable impact on performance. Testing of other sorting algorithms, such as radix sort, is planned for subsequent evolutions of our implementation. We also plan to extend the implementation's flexibility by overlapping memory transfers and compute operations using CUDA streams.

Although the implementation is based on raster images, the same algorithm can be applied to laser scanned data. We plan to extend our implementation to perform line detection on 3D point clouds.

## Bibliography

V. N. B. Aldama. Design and implementation of the hough transform on gpu using cuda. *https://hdl.handle.net/2183/45592*, 2025. [Online; accessed September 24, 2025.

D. Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111–122, 1981. ISSN 0031-3203. doi: https://doi.org/10.1016/0031-3203(81)90009-1. URL *https://www.sciencedirect.com/science/article/pii/0031320381900091*.

P. Denis, J. H. Elder, and F. J. Estrada. Efficient edge-based methods for estimating manhattan frames in urban imagery. In D. Forsyth, P. Torr, and A. Zisserman, editors, *Computer Vision – ECCV 2008*, pages 197–210, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-88688-4.

R. O. Duda and P. E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11–15, Jan. 1972. ISSN 0001-0782. doi: 10.1145/361237.361242. URL *https://doi.org/10.1145/361237.361242*.

L. A. Fernandes and M. M. Oliveira. A general framework for subspace detection in unordered multidimensional data. *Pattern Recognition*, 45(9):3566–3579, 2012. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2012.02.033. URL *https://www.sciencedirect.com/science/article/pii/S0031320312001112*. Best Papers of Iberian Conference on Pattern Recognition and Image Analysis (IbPRIA'2011).

J. Gómez-Luna, J. González-Linares, J. Benavides, E. Zapata, and N. Guil. Load balancing versus occupancy maximization on graphics processing units: The generalized hough transform as a case study. *IJHPCA*, 25:205–222, 05 2011. doi: 10.1177/1094342010383998.

D. B. Kirk and W. mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 2016. ISBN 978-0-12-811986-0.

D. Lin, X. Huang, Q. Nguyen, J. Blackburn, C. Rodrigues, T. Huang, M. do, S. Patel, and W.-m. Hwu. The parallelization of video processing. *Signal Processing Magazine, IEEE*, 26:103 – 112, 12 2009. doi: 10.1109/MSP.2009.934116.

Nvidia. Cuda core compute libraries documentation. *https://nvidia.github.io/cccl/cub/api/structcub_1_1DeviceSegmentedRadixSort.html*, 2025. [Online; accessed September 24, 2025.

OpenCV. Hough line transform tutorial. *https://docs.opencv.org/4.12.0/d9/db0/tutorial_hough_lines.html*, 2025a. [Online; accessed September 24, 2025.

OpenCV. Hough lines detector class reference. *https://docs.opencv.org/4.12.0/d2/dcd/classcv_1_1cuda_1_1HoughLinesDetector.html*, 2025b. [Online; accessed September 24, 2025.

Paul V C Hough. Method and means for recognizing complex patterns. *https://patents.google.com/patent/US3069654A*, 1962. [Online; accessed September 24, 2025.

Z. Wang. The active visual sensing methods for robotic welding: Review, tutorial, and prospect. *IEEE Transactions on Instrumentation and Measurement*, 73:1–19, 2024. doi: 10.1109/TIM.2024.3485460.

E. Widyaningrum, R. Y. Peters, and R. C. Lindenbergh. Building outline extraction from als point clouds using medial axis transform descriptors. *Pattern Recognition*, 106:107447, 2020. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2020.107447. URL *https://www.sciencedirect.com/science/article/pii/S0031320320302508*.

M. Yermo, R. Laso, O. G. Lorenzo, T. F. Pena, J. C. Cabaleiro, F. F. Rivera, and D. L. Vilariño. Powerline detection and characterization in general-purpose airborne lidar surveys. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 17:10137–10157, 2024. doi: 10.1109/JSTARS.2024.3396522.