GPU Accelerated Hough Transform for Line Detection

Damian Gradziuk[0009−0004−1379−1646], Beata Bylina[0000−0002−1327−9747], and Przemysław Stpiczyński[0000−0001−8661−414X]

Institute of Computer Science and Mathematics
Maria Curie-Skłodowska University
ul. Akademicka 9, 20-032 Lublin, Poland
gradziuk.damian5@gmail.com,
beata.bylina@mail.umcs.pl,
przemyslaw.stpiczynski@mail.umcs.pl

Abstract. The Hough Transform (HT) is a classical technique for detecting straight lines inferred from edge points in binarized images, valued for its robustness to noise and incomplete edge information. However, the numerical nature of the HT, leads to high computational complexity, where performance is dominated by updates and memory access patterns. The main contribution of this work is a GPU-accelerated implementation of the classical HT for line detection, focused on implementation-level optimizations rather than algorithmic modifications. The proposed approach preserves functional equivalence with a widely used OpenCV GPU-based implementation while exploiting the parallelism and memory hierarchy of modern GPUs. An adaptive voting strategy is employed, using shared memory when the accumulator size permits and falling back to global memory for larger parameter spaces. Experimental results obtained across various GPU architectures show that the proposed implementation achieves speedups ranging from approximately 50× up to over 400× compared to a sequential CPU baseline and consistently outperforms the OpenCV GPU-based solution by a factor of about 1.3× to 2.5×.

Keywords: GPU Acceleration · Hough Transform · Image Processing · Discretized Parameter Space · Numerical Voting Scheme

1 Introduction

Originally proposed by Hough [7] and later extended for line detection by Duda and Hart [4], the Hough Transform (HT) is a classical technique for detecting parametric shapes such as straight lines in images. Its robustness to noise, fragmented contours, and missing edge points has made it a standard tool in computer vision. From a computational perspective, the classical voting-based formulation of the HT can be viewed as a dense numerical accumulation over a discretized two-dimensional parameter space, where each edge point contributes

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

votes across a range of angular parameters. As a result, execution time is dominated by accumulator updates and memory access patterns rather than by control flow complexity.

The growing availability of high-resolution images and dense edge maps has magnified the computational cost of the HT, limiting its applicability in time-critical and large-scale scenarios. On conventional CPU architectures, the inherent parallelism of the voting process is only weakly exploited, leading to execution times that scale linearly with the number of edge points and the angular resolution. In contrast, modern GPUs provide massive data parallelism and high memory bandwidth, making them a natural target for accelerating accumulation-based numerical algorithms. Additionally, GPUs offer a hierarchical memory model, including shared memory that can be used to optimize access patterns and reduce latency for frequently updated data structures [2].

In this work, we focus on accelerating the classical HT for straight line detection through implementation-level optimization on GPUs. Rather than introducing algorithmic approximations or probabilistic variants, the proposed approach deliberately retains the classical formulation of the HT in order to focus on the impact of implementation and hardware-level optimizations. This approach allows performance variations to be examined with respect to GPU hardware architecture, memory hierarchy, and input image characteristics, while eliminating variability due to algorithmic design and parameterization.

The main contribution of this paper is a GPU-accelerated implementation of the classical HT that employs an adaptive voting strategy. Depending on the size of the discretized accumulator, voting is performed either in shared memory or directly in global memory, allowing efficient execution across a wide range of image resolutions and parameter-space configurations. The implementation is evaluated on various GPU architectures, demonstrating speedups ranging from approximately 50× to over 400× compared to a single-threaded CPU baseline, while achieving an additional $1.3\times - 2.1\times$ speedup over the wildely-used OpenCV GPU-based implementation.

The remainder of this paper is organized as follows. Section 2 reviews related work on HT variants and GPU-based implementations. Section 3 introduces the theoretical background of the classical HT. Section 4 describes the proposed GPU implementation. Section 5 presents the experimental setup and performance results. Section 6 concludes the paper.

2 Related Work

Due to the high computational cost of the classical HT, numerous approaches have been proposed to reduce its complexity. Probabilistic and randomized variants aim to limit the number of processed edge points while preserving detection performance. The Probabilistic HT (PHT) introduced by Kiryati et al. [10] demonstrated that reliable line detection can often be achieved using only a randomly selected subset of edge points. Similarly, the Randomized HT (RHT) proposed by Xu and Oja [19] reduces both computational and memory complex-

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

ity by sampling minimal point sets and estimating parameters probabilistically. Further extensions of these ideas allowed to improve efficiency by incrementally refining detection results, as shown by Galamhos et al. [6].

Another class of methods focuses on reducing the dimensionality or resolution of the accumulator space. Multiresolution and hierarchical variants, such as the approaches proposed by Atiquzzaman [1] and Espinosa and Perkowski [5], employ coarse-to-fine strategies to limit accumulator growth and improve computational efficiency. While effective, these methods often introduce additional algorithmic complexity and require careful parameter tuning.

Beyond algorithmic modifications, several works have explored hardware-accelerated implementations of the HT. Early efforts targeted reconfigurable and embedded architectures, including FPGA-based implementations optimized for memory locality and parallelism [21]. More recently, GPUs have emerged as a natural platform for accelerating the voting-based structure of the HT. Van den Braak et al. [2] investigated different GPU implementation strategies and highlighted trade-offs between throughput-oriented and input-independent execution models. Additional GPU-based accelerations have been reported for both two-dimensional image data [13] and higher-dimensional Hough spaces, such as 3D plane detection in LiDAR data [17].

The HT has also been extended and adapted for various application domains, including lane detection [12], industrial inspection [11], and medical imaging [20]. Recent works combine classical Hough-based formulations with learning-based components or deploy them on embedded GPU platforms to meet real-time constraints [14].

Direct performance comparisons with existing approaches are challenging, as many rely on probabilistic or randomized variants of the HT that reduce computation by processing subsets of input data. This alters the computational characteristics and makes their performance not directly comparable to the classical HT operating on the full dataset.

This work concentrates on implementation-level optimization of the classical, voting-based HT for straight line detection. By preserving functional equivalence with the widely used OpenCV GPU-based implementation, the proposed approach enables a controlled evaluation of performance scalability with respect to both image resolution and edge point density. Furthermore, the analysis is conducted across various GPU architectures, allowing to focus on architectural effects related to parallel execution and memory hierarchy.

3 Hough Transform Algorithm

In the classical case of line detection, the algorithm maps every edge point from the edge map into a parameter space, where each line is represented by a pair (ρ, θ), where ρ denotes the perpendicular distance from the origin to the line, and θ represents the angle of the line's normal vector with respect to the x-axis.

The key idea of the HT is that collinear points in the image space correspond to curves in the parameter space that intersect at a common point. As a result,

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

the presence of a line in the image is manifested as a prominent peak in the accumulator space, as illustrated in Fig. 1.

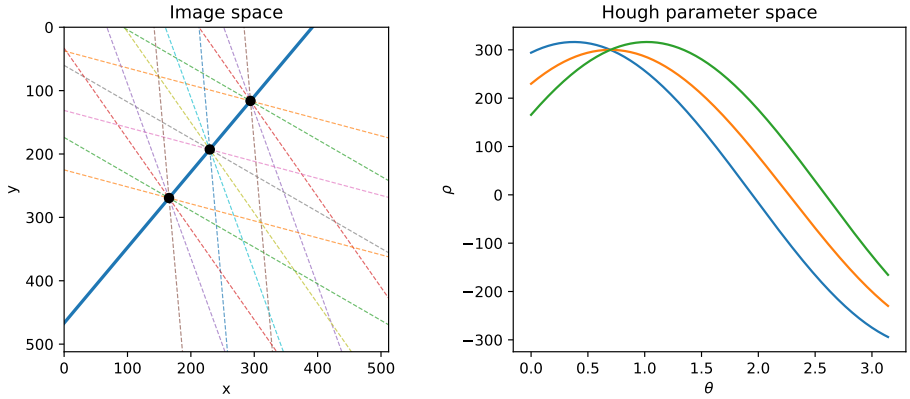

Fig. 1. Principle of the Hough transform: multiple candidate lines passing through individual edge points in the image space (left) and the corresponding sinusoidal mappings intersecting at a common point in the parameter space (right).

For a binary edge map $I(x, y)$ achieved after applying an edge detection algorithm, such as the Canny edge detector [3], on an image, the equation of a line can be expressed in polar coordinates as

$$\rho = x\cos(\theta) + y\sin(\theta), \tag{1}$$

where (x, y) denotes the coordinates of an edge point. For each such point, the algorithm iterates over a discretized range of θ values, typically spanning the interval $[0, \pi]$, and computes the corresponding ρ. The accumulator array is incremented at the discretized location (ρ, θ), effectively casting a vote for a potential line passing through the point.

From a computational perspective, the time complexity of the classical voting-based HT for line detection is given by

$$O(N_e \cdot N_\theta), \tag{2}$$

where N_e denotes the number of edge points in the image and N_θ represents the number of discretized angle samples [4,8]. In the worst case, N_e is bounded by the image resolution, i.e., $N_e \leq W \cdot H$, where W and H denote the image width and height, respectively. As the result, the computational cost grows linearly with both the number of edge points and the angular resolution of the parameter space.

The memory complexity of the algorithm is dominated by the size of the accumulator array, which is given by

$$O(N_\rho \cdot N_\theta), \tag{3}$$

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

where N_ρ denotes the number of discretized distance values. For an image of resolution $W \times H$, the parameter ρ typically spans the range $[-\sqrt{W^2+H^2}, \sqrt{W^2+H^2}]$, implying that N_ρ increases proportionally with the image diagonal. Consequently, both the memory requirements and the computational cost of the accumulator update are strongly influenced by the image resolution and the chosen discretization parameters [8,16].

After all edge points have been processed, the accumulator is scanned for local maxima. These maxima correspond to parameter pairs (ρ, θ) that received a significant number of votes and thus represent detected line candidates. In practice, a threshold is applied to suppress weak responses and reduce false detections [4].

The HT algorithm has been extended to support the detection of other parametric shapes, including circles [9], ellipses [18], and more complex forms. In such cases, the dimensionality of the parameter space increases, leading to higher computational and memory requirements. To address these limitations, the Generalized HT was introduced [15], enabling the detection of arbitrary shapes by replacing the analytical parameterization with a lookup-based representation. While these extensions significantly broaden the applicability of the HT, they also increase its computational cost, making optimised implementations particularly important from practical point of view. For this reason, the present work restricts its scope to straight line detection using the classical voting-based formulation, which remains particularly well-suited for parallelization on modern GPU architectures.

4 Implementation

We have implemented two variants of the HT for line detection: a classical sequential version executed on the single-threaded CPU and a parallel, GPU-accelerated version. All implementations were developed in C++, using the NVIDIA CUDA framework (version 12.8). The source code of the proposed implementation is publicly available at `https://github.com/FoxedGuy/hough-gpu`.

The sequential CPU implementation is intentionally limited to a single-threaded baseline without vectorization or multi-core parallelization. This design choice allows for a clear comparison between the classical execution model and the massively parallel GPU implementation, isolating the impact of hardware-level parallelism on the performance of the voting process. At the same time, the CPU implementation was not left entirely unoptimized: trigonometric values were precomputed using a lookup table, and the code was compiled with compiler optimizations enabled (-O3). These measures reduce avoidable overhead while preserving the fundamentally sequential character of the baseline.

The GPU-accelerated implementation follows a three-stage pipeline executed by dedicated CUDA kernels: extraction of edge point coordinates from the input edge map, voting in the accumulator space and detection of local maxima. In the first stage, non-zero values from the edge map are extracted using the

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

`extract_non_zero_packed` kernel. Each thread examines a single pixel and if it corresponds to an edge point, stores its coordinates using an atomic increment. To reduce memory traffic and improve cache efficiency, edge points coordinates are packed into a 32-bit integer, with horizontal and vertical positions encoded in separate 16-bit fields. The parallel extraction and the packing of non-zero values coordinates kernel is formally summarized in Algorithm 1.

Algorithm 1 CUDA Kernel: Extraction of Non-Zero Pixels with Packed Coordinates

Require: Binary edge map I of size $N \times N$
Ensure: Packed coordinate list P, number of edge points N_e

for each pixel (x, y) **in parallel do**
  if $I(y, x) \neq 0$ **then**
    $idx \leftarrow \textbf{atomicAdd}(N_e, 1)$
    $P[idx] \leftarrow (y \ll 16) \mid x$
  end if
end for

The accumulator is represented as a two-dimensional integer array stored in the global memory, with its dimensions determined by the discretization of the ρ and θ parameters, including padding. For better memory coalescing, the pitch version of CUDA memory allocation function, `cudaMallocPitch` has been used.

The voting stage constitutes the most computationally intensive part of the algorithm. Each CUDA block is responsible for processing a single discretized angle value, while threads within the block iterate over the list of detected edge points and compute the corresponding polar parameter. Trigonometric values are computed on-the-fly using the `__sincosf` intrinsic function and scaled by the reciprocal of the radial discretization step. This approach shows itself to be more efficient than using precomputed lookup tables due to reduced number of memory references and better utilization of the GPU's special function units. The computed radial index is rounded using the `__float2int_rn` intrinsic to ensure consistent mapping to accumulator bins.

Depending on the size of the accumulator and the available shared memory on the target device, two alternative voting strategies are employed. For each discretized angle, a one-dimensional slice of the accumulator corresponding to the radial parameter is processed independently. When the size of this slice fits into shared memory, it is allocated as a shared buffer local to the CUDA block stored in shared memory. The buffer is shared by all threads within the block and represents the accumulator row for a single angle value. The shared memory voting kernel employed in this case is formally summarized as Algorithm 2.

During voting, all threads concurrently update the shared accumulator using atomic operations, accumulating votes for different radial bins. Because of shared memory being physically located on-chip and shared only among threads of the same block, this approach significantly reduces global memory traffic and contention. After all votes for a given angle are processed, a synchronization

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

barrier ensures that the shared accumulator is fully updated before its contents are written back to the corresponding row in global memory.

Algorithm 2 CUDA Kernel: Voting-based Hough Accumulator with Shared Memory

Require: Packed edge point list P of size N_e, N_θ, N_ρ, ρ^{-1}, $\theta_{\min}$, $\Delta\theta$
Ensure: Accumulator array A
for each angle index θ_k **in parallel do**
  Allocate shared array $S[0 \dots N_\rho + 1]$
  Initialize $S \leftarrow 0$
  $\theta \leftarrow \theta_{\min} + k \cdot \Delta\theta$
  $(\cos\theta, \sin\theta) \leftarrow \text{sincos}(\theta)$
  $\cos\theta \leftarrow \cos\theta \cdot \rho^{-1}$
  $\sin\theta \leftarrow \sin\theta \cdot \rho^{-1}$
  $shift \leftarrow \lfloor (N_\rho - 1)/2 \rfloor$
  for each edge point $(x_i, y_i) \in P$ **in parallel do**
    $r \leftarrow \text{round}(x_i \cdot \cos\theta + y_i \cdot \sin\theta)$
    $r \leftarrow r + shift$
    atomicAdd$(S[r+1], 1)$
  end for
  Synchronize threads
  Copy S to global accumulator row $A[k+1][\cdot]$
end for

If the size of the accumulator slice exceeds the available shared memory capacity, the implementation falls back to a global-memory-based strategy. In this case, all threads directly update the global accumulator using atomic operations. It leads to increased memory latency and contention but ensures correctness for larger problem sizes.

After the voting phase, local maxima in the accumulator space are detected using the `find_maxims` kernel. Each thread evaluates a single accumulator cell and performs a comparison with its immediate neighbors in both the radial and angular dimensions. Only cells exceeding a predefined threshold and satisfying the local maximum criterion are considered as valid line candidates. Each CUDA thread is mapped to a single accumulator cell (ρ, θ) using a two-dimensional grid configuration, enabling parallel evaluation of the local maximum criterion across the parameter space. Detected lines are stored using atomic operations to ensure correctness under concurrent writes.

The implementation processes square $N \times N$ edge maps, where N is assumed to be a power of two for implementation convenience. This restriction is not inherent to the algorithm and could be relaxed without affecting correctness. Detected lines are visualized using OpenCV drawing functions and saved for qualitative inspection.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

5 Experiments

All experiments have been performed on three hardware platforms with varying CPU and GPU capabilities, referred to as H1, H2, and H3. The platforms have beed selected to represent different generations of server-class processors and GPU architectures, allowing the scalability of the proposed implementation to be evaluated across a wide performance spectrum.

Platform H1 is equipped with an Intel Xeon Gold 5218R CPU and an NVIDIA RTX A2000 GPU, representing a lower-end workstation-class configuration. Platform H2 consists of an Intel Xeon Platinum 8358 CPU paired with an NVIDIA A100 PCIe GPU, offering substantially higher computational throughput and memory bandwidth. Platform H3 employs an AMD EPYC 9825 processor together with an NVIDIA H200 NVL GPU, representing a high-end configuration with a next-generation GPU architecture. An overview of the GPUs' specifications is provided in Table 1.

Table 1. Hardware platforms used for experimental evaluation.

Platform	GPU	Memory	CUDA Cores
H1	NVIDIA RTX A2000	12 GB	3328
H2	NVIDIA A100 PCIe	40 GB	6912
H3	NVIDIA H200 NVL	141 GB	16896

All platforms support CUDA-based execution and provide sufficient memory resources to accommodate the accumulator sizes used in the experiments. This selection enables a consistent comparison of CPU and GPU execution characteristics across different architectural classes.

5.1 Test Data

To ensure controlled and reproducible experimental conditions, synthetic edge maps were generated and used as test data. The edge maps contain multiple horizontal line segments placed symmetrically with respect to the map center. Each line has a fixed thickness of one pixel, while the number of lines and the edge map resolution are varied across experiments.

For experiments analyzing the impact of edge map resolution, square edge maps of size $N \times N$ were generated, with N ranging from 512 to 8192 pixels. In these edge maps, a fixed number of horizontal line segments was drawn, each with a predefined length centered along the horizontal axis. This setup allows the number of edge points to be controlled independently of the edge map resolution.

All generated edge maps are binary, with foreground pixels representing line segments and background pixels set to zero. The use of synthetic data enables precise control over the number and spatial distribution of edge points, facilitating the systematic evaluation of the computational behavior of the HT under varying input conditions.

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

5.2 Evaluation Metrics

The primary evaluation metric used in this work is the execution time, as the main objective is to assess the computational efficiency and scalability of the proposed GPU-accelerated HT implementation. All reported timings correspond to the line detection stage, excluding image loading and disk I/O operations.

Detection quality is not treated as a primary metric. Instead, qualitative and functional correctness was verified by ensuring that, for identical parameter settings, the proposed implementation produces line detection results consistent with those obtained using the OpenCV reference implementation as shown in Fig. 2. It ensures that performance comparisons are conducted between functionally equivalent configurations.

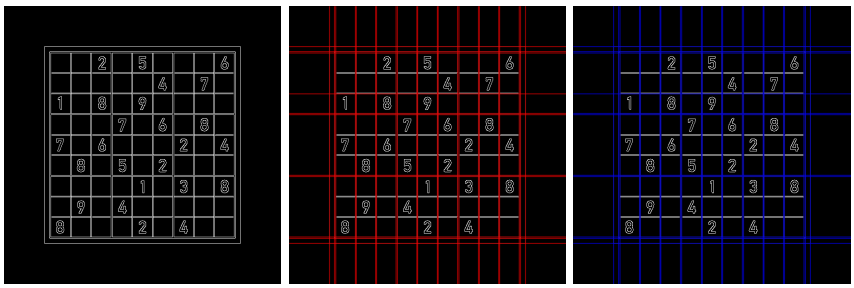

Fig. 2. Comparison of line detection results on a sudoku edge map (left) between the proposed GPU implementation (center, detected lines coloured red) and the OpenCV reference implementation (right, detected lines coloured blue). Both implementations yield identical detected lines, confirming functional equivalence.

Execution times were measured using high-resolution timing mechanisms appropriate to each execution context. For GPU execution, NVIDIA Nsight profiling tools were additionally employed to validate timing measurements and to analyze kernel-level execution behavior, memory access patterns, and synchronization overheads.

5.3 Results

Table 2 reports average execution times for edge maps with a fixed resolution of 1024×1024 pixels and an increasing number of edge points. For all three platforms (H1–H3), the sequential CPU implementation has exhibited an approximately linear growth in execution time as the number of edge points has increased, which is consistent with the $O(N_e \cdot N_\theta)$ complexity of the voting stage.

On platform H1, increasing the number of edge points from 10,240 to 163,840 results in the execution time increase from 8.17 ms to 80.80 ms, corresponding to a factor of 9.9×. A similar trend can be observed on H2, where execution time growns from 8.19 ms to 85.04 ms (10.4×), and on H3, from 4.96 ms to 54.71 ms

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

Table 2. Average calculation times for edge maps with 1024 × 1024 resolution and different numbers of edge points. Threshold for line detection set to 400.

Edge Points	Hardware								
	H1			H2			H3		
	CPU	GPU	CV	CPU	GPU	CV	CPU	GPU	CV
10240	8.17	0.16	0.29	8.19	0.14	0.24	4.96	0.09	0.15
20480	13.04	0.18	0.31	13.14	0.14	0.25	8.46	0.09	0.15
40960	22.86	0.24	0.36	23.85	0.16	0.26	14.69	0.10	0.16
81920	42.89	0.33	0.44	43.50	0.18	0.29	27.80	0.11	0.17
163840	80.80	0.51	0.60	85.04	0.23	0.34	54.71	0.13	0.19

(11.0×). These results confirm that the CPU performance is primarily dominated by the number of processed edge points, while differences between platforms reflect their single-core performance and memory subsystem efficiency.

In contrast, the proposed GPU implementation shows only a weak dependence on the number of edge points across all platforms. For H1, execution time increases from 0.16 ms to 0.51 ms when the number of edge points grows by a factor of 16. On H2, the corresponding increase is from 0.14 ms to 0.23 ms, while on H3 it ranges from 0.09 ms to 0.13 ms. As a result, the GPU execution time has varied by at most 0.35 ms across the entire tested range, compared to increases exceeding 70 ms for the CPU.

The resulting speedup over the CPU implementation depends strongly on the edge density. For sparse edge maps (10,240 points), the speedup ranges from 51× on H1 to 55× on H3. For the densest configuration (163,840 points), the speedup increases to 158× on H1, 370× on H2, and 421× on H3. These trends are visualized in Fig. 3, which highlights the widening performance gap between CPU and GPU implementations as the number of edge points has increased.

The relatively weak dependence of GPU execution time on input size for smaller resolutions can be attributed to the high degree of parallelism and the ability to overlap memory operations with computation. However, as the resolution increases, performance becomes increasingly dominated by memory access patterns and accumulator size, particularly when shared memory capacity is exceeded. In contrast, the CPU implementation processes edge points sequentially, resulting in a nearly linear increase in execution time.

The influence of image resolution is summarized in Table 3. For the CPU implementation, execution time increases noticeably with resolution on all platforms. On H1, execution time grows from 45.23 ms at 512×512 pixels to 99.69 ms at 8192 × 8192 pixels, corresponding to a 2.2× increase. Comparable scaling can be observed on H2 (42.61 ms to 111.28 ms, 2.6×) and H3 (29.02 ms to 68.01 ms, 2.3×).

For the GPU-accelerated implementation, execution times remains nearly constant for resolutions up to 2048 × 2048 pixels. Across all platforms, GPU execution time in this range varies by less than 0.1 ms. A far more significant

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

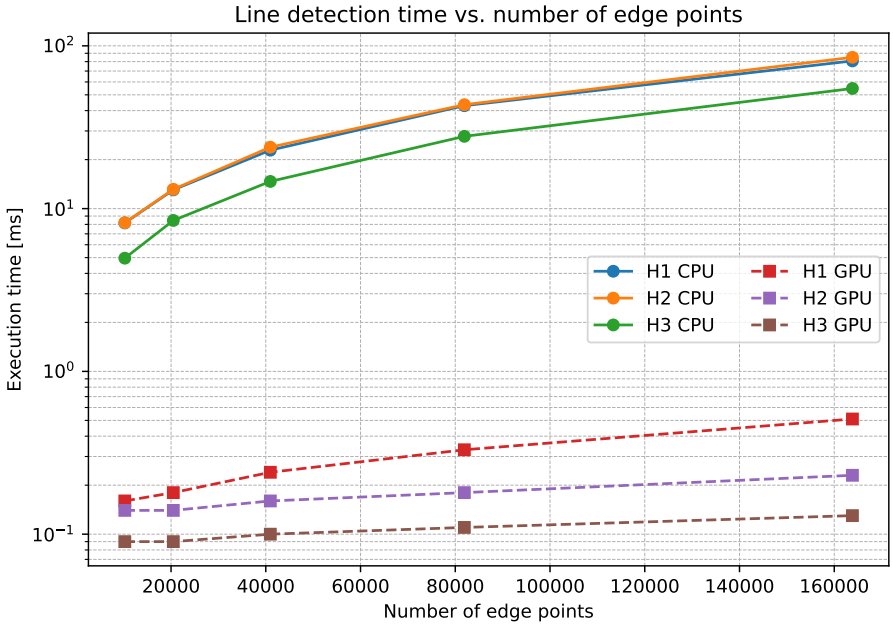

Fig. 3. Line detection execution time as a function of the number of edge points for CPU and GPU implementations across platforms H1, H2, and H3. The vertical axis is shown on a logarithmic scale.

increase can be observed for 4096×4096 pixels and above, where execution time rises to 1.03 ms on H1, 0.54 ms on H2, and 0.26 ms on H3. At the largest resolution of 8192×8192 pixels, GPU execution time reaches 2.03 ms on H1, 0.98 ms on H2, and 0.43 ms on H3. This noticeable increase in execution time can be explained by the growing size of the accumulator, which no longer fits into shared memory. Despite this increase, the GPU implementation remains from $34\times$ to $230\times$ faster than the CPU, depending on the platform. These scaling trends are illustrated in Fig. 4.

The observed increase in execution time for larger resolutions is primarily associated with the growth of the accumulator, which exceeds the capacity of shared memory and forces the use of global memory. This transition leads to increased memory latency and contention, which becomes the dominant performance factor for large parameter spaces.

Across all evaluated configurations, the proposed GPU implementation consistently outperforms the OpenCV GPU-based solution. For the fixed-resolution experiments in Table 2, speedups over OpenCV have ranged from $1.8\times$ to $2.1\times$ on H1, from $1.7\times$ to $2.5\times$ on H2, and from $1.6\times$ to $1.9\times$ on H3. Similar ratios can be observed for varying resolutions in Table 3. Since both approaches rely on comparable algorithmic principles, these differences can be attributed to

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

Table 3. Average calculation times for edge maps with the same number of edge points and different resolutions. Threshold for line detection set to 400.

Resolution	Hardware								
	H1			H2			H3		
	CPU	GPU	CV	CPU	GPU	CV	CPU	GPU	CV
512×512	45.23	0.32	0.33	42.61	0.19	0.22	29.02	0.12	0.13
1024×1024	41.20	0.33	0.44	42.72	0.18	0.29	27.15	0.11	0.17
2048×2048	43.22	0.40	0.54	47.80	0.21	0.32	29.10	0.12	0.18
4096×4096	53.77	1.03	1.07	60.32	0.54	0.63	36.94	0.26	0.31
8192×8192	99.69	2.03	2.06	111.28	0.98	1.05	68.01	0.43	0.51

implementation-level factors such as kernel specialization, reduced synchronization overhead, and more efficient memory access patterns.

It should be emphasized that the OpenCV GPU implementation is designed as a general-purpose solution supporting a wide range of configurations and use cases. In contrast, the proposed implementation focuses on a specialized, performance-critical scenario with fixed parameterization. Therefore, the observed performance gains should be interpreted as the result of targeted, implementation-level optimizations rather than a general superiority over more flexible frameworks such as OpenCV.

Finally, a consistent performance hierarchy can be observed across all experiments. Platform H3 has achieved the lowest execution times for both CPU and GPU implementations, followed by H2 and H1. While CPU performance on H1 and H2 remains relatively close, GPU execution times on H2 have been consistently $1.5\times$ to $2.0\times$ lower than on H1. These results confirms that the proposed implementation benefits directly from increased GPU compute capability and memory bandwidth, and that newer GPU architectures provide measurable advantages even for highly optimized, short-running kernels. The cost of the numerical voting process is effectively amortized across massively parallel accumulator updates, significantly reducing sensitivity to input density compared to the CPU implementation. As a result, performance becomes increasingly dominated by parameter-space resolution rather than the number of edge points.

6 Conclusions

This paper investigates the computational performance of the classical HT for straight line detection from the perspective of numerical accumulation on modern GPU architectures. Rather than modifying the underlying algorithm, the proposed approach focuses on implementation-level optimization of the voting process, which constitutes the dominant computational and memory-intensive component of the method. In particular, the results demonstrate that careful exploitation of GPU memory hierarchy can significantly improve the efficiency

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

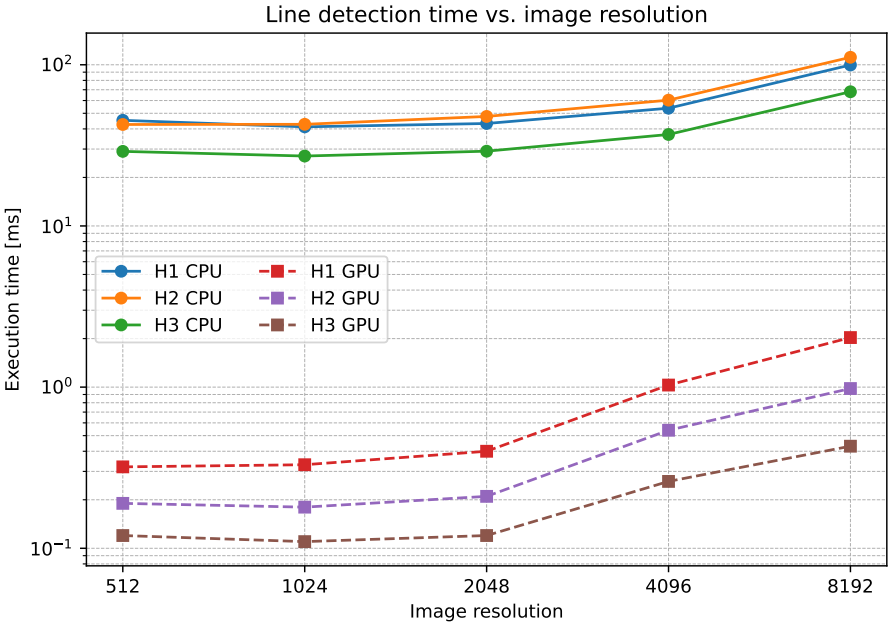

Fig. 4. Line detection execution time as a function of the resolution of the edge map for CPU and GPU implementations across platforms H1, H2, and H3. The vertical axis is shown on a logarithmic scale.

of classical accumulation-based algorithms without introducing algorithmic modifications or approximations.

A GPU-accelerated implementation preserving full functional equivalence with a widely used OpenCV reference has been presented. By exploiting the hierarchical memory architecture of GPUs and employing an adaptive voting strategy that dynamically selects between shared and global memory, the implementation effectively reduces memory traffic and contention during accumulator updates. This design allows the numerical voting process to scale efficiently across a wide range of edge point densities and image resolutions.

Experimental results obtained on multiple hardware platforms have demonstrated substantial performance gains over a sequential CPU baseline. Depending on the input characteristics and the target architecture, speedups range from approximately 50× for sparse edge maps to over 400× for dense configurations. Across all tested scenarios, the proposed implementation consistently outperforms the OpenCV GPU-based solution, confirming that careful implementation-level optimization can yield performance improvements even for well-established algorithms.

The results further indicate that, on GPUs, the execution time of the classical HT becomes increasingly dominated by the resolution of the parameter space

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

rather than by the number of edge points. This contrasts with CPU-based execution, where performance scales almost linearly with input density. Such behavior highlights the effectiveness of massive parallelism and fast on-chip memory in amortizing the cost of numerical accumulation.

Future work will focus on further refinement of the GPU implementation using advanced CUDA features and programming abstractions. In particular, the use of high-level parallel primitives provided by libraries such as `thrust` may simplify selected stages of the processing pipeline while preserving performance. Beyond raw execution time, an important direction for future research is the evaluation of energy efficiency, allowing performance gains to be analyzed in terms of computational cost per watt on different GPU architectures. An important direction for future research is the development of architecture-aware optimizations tailored to specific GPU architectures, which may further improve performance by leveraging low-level hardware characteristics such as warp scheduling and memory access patterns. Finally, the proposed implementation framework may be extended to higher-dimensional parameter spaces to support the detection of circles and ellipses, enabling an assessment of how the observed numerical and architectural scaling properties generalize to more complex HT variants.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Atiquzzaman, M.: Multiresolution hough transform-an efficient method of detecting patterns in images. IEEE Transactions on Pattern Analysis and Machine Intelligence **14**(11), 1090–1095 (1992). https://doi.org/10.1109/34.166623
2. van den Braak, G.J., Nugteren, C., Mesman, B., Corporaal, H.: 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. pp. 611–622. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23687-7_55
3. Canny, J.: A computational approach to edge detection. IEEE Transactions on Pattern Analysis and Machine Intelligence **PAMI-8**(6), 679–698 (1986). https://doi.org/10.1109/TPAMI.1986.4767851
4. Duda, R.O., Hart, P.E.: Use of the hough transformation to detect lines and curves in pictures. Commun. ACM **15**(1), 11–15 (Jan 1972). https://doi.org/10.1145/361237.361242
5. Espinosa, C., Perkowski, M.: Hierarchical hough transform based on pyramidal architecture. In: Eleventh Annual International Phoenix Conference on Computers and Communication [1992 Conference Proceedings]. pp. 743–750 (1992). https://doi.org/10.1109/PCCC.1992.200515
6. Galamhos, C., Matas, J., Kittler, J.: Progressive probabilistic hough transform for line detection. In: Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149). vol. 1, pp. 554–560 Vol. 1 (1999). https://doi.org/10.1109/CVPR.1999.786993
7. Hough, P.V.C.: Method and means for recognizing complex patterns. US Patent 3,069,654 (1962)

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46

8. Illingworth, J., Kittler, J.: A survey of the hough transform. Computer Vision, Graphics, and Image Processing **44**(1), 87–116 (1988). https://doi.org/10.1016/S0734-189X(88)80033-1
9. Ioannou, D., Huda, W., Laine, A.F.: Circle recognition through a 2d hough transform and radius histogramming. Image and Vision Computing **17**(1), 15–26 (1999). https://doi.org/10.1016/S0262-8856(98)00090-0
10. Kiryati, N., Eldar, Y., Bruckstein, A.: A probabilistic hough transform. Pattern Recognition **24**(4), 303–316 (1991). https://doi.org/10.1016/0031-3203(91)90073-E
11. Li, H., Ma, Y., Bao, H., Zhang, Y.: Probabilistic hough transform for rectifying industrial nameplate images: A novel strategy for improved text detection and precision in difficult environments. Applied Sciences **13**(7) (2023). https://doi.org/10.3390/app13074533
12. Marzougui, M., Alasiry, A., Kortli, Y., Baili, J.: A lane tracking method based on progressive probabilistic hough transform. IEEE Access **PP**, 1–1 (05 2020). https://doi.org/10.1109/ACCESS.2020.2991930
13. Patil, P.R., Patil, M.D., Vyawahare, V.A.: Acceleration of hough transform algorithm using graphics processing unit (gpu). In: 2016 International Conference on Communication and Signal Processing (ICCSP). pp. 1584–1588 (2016). https://doi.org/10.1109/ICCSP.2016.7754427
14. Rossi, F.D.: Boosting performance of computer vision applications through embedded gpus on the edge (2025), https://arxiv.org/abs/2511.01129
15. Samal, A., Edwards, J.: Generalized hough transform for natural shapes. Pattern Recognition Letters **18**(5), 473–480 (1997). https://doi.org/10.1016/S0167-8655(97)00023-8
16. Stockman, G., Shapiro, L.G.: Computer vision. Prentice Hall PTR (2001), https://dl.acm.org/doi/10.5555/558008
17. Tian, Y., Song, W., Chen, L., Sung, Y., Kwak, J., Sun, S.: Fast planar detection system using a gpu-based 3d hough transform for lidar point clouds. Applied Sciences **10**(5) (2020). https://doi.org/10.3390/app10051744
18. Xie, Y., Ji, Q.: A new efficient ellipse detection method. In: 2002 International Conference on Pattern Recognition. vol. 2, pp. 957–960 vol.2 (2002). https://doi.org/10.1109/ICPR.2002.1048464
19. Xu, L., Oja, E.: Randomized hough transform (rht): Basic mechanisms, algorithms, and computational complexities. CVGIP: Image Understanding **57**(2), 131–154 (1993). https://doi.org/10.1006/ciun.1993.1009
20. Zhang, Y., Liu, T., Zhou, H.: Automatic detection based on deep hough transform for b-lines in ultrasound image. In: 2025 8th International Symposium on Big Data and Applied Statistics (ISBDAS). pp. 641–644 (2025). https://doi.org/10.1109/ISBDAS64762.2025.11117024
21. Zhou, X., Ito, Y., Nakano, K.: An efficient implementation of the gradient-based hough transform using dsp slices and block rams on the fpga. In: 2014 IEEE International Parallel & Distributed Processing Symposium Workshops. pp. 762–770 (2014). https://doi.org/10.1109/IPDPSW.2014.88

ICCS Camera Ready Version 2026
To cite this paper please use the final published version:
DOI: 10.1007/978-3-032-29915-4_46