GPU Accelerating an Image Processing Pipeline for Lane Detection in CUDA

Cael Black

January 26, 2026

Abstract

Autonomous robotics applications, such as lane-following vehicles, can be significantly accelerated by taking advantage of GPU hardware capable of highly parallel computation. This paper describes how CUDA code was written to GPU-accelerate an image processing pipeline traditionally executed on a CPU.

The resulting acceleration led to a 4x speedup when run on a lab machine equipped with an NVIDIA RTX 3070 graphics card.

All code for this project can be found in the following GitHub repository:
https://github.com/blackcael/DuckieTown-CUDA

1 Introduction

1.1 Background: DuckieTown

DuckieTown is an introductory robotics course designed to teach the fundamentals of autonomous navigation. A large track is set up in a classroom or lab space where each student operates a small two-wheeled robot equipped with various sensors, most notably a camera. The primary goal of these labs is to use the camera's images to determine the robot's position relative to lane markings, then apply a PID controller to stay within the lanes and navigate turns.

The DuckieTown robot runs on an NVIDIA Jetson Nano (available in 2GB and 4GB variants), which provides 128 CUDA cores for parallel computation. However, due to time and scope constraints, the course typically presents solutions written in simple, serial CPU code, missing the substantial gains that parallel computing can provide—especially in computer vision tasks.

1.2 Goal: Acceleration

One limitation of PID systems is their sampling rate. The more frequently a system can sample its error, the quicker and more precisely it can correct deviations, improving stability and reducing oscillation.

By leveraging the GPU to accelerate the image processing pipeline, we can increase the effective sampling rate of the controller and improve overall robot performance. This is accomplished by implementing the pipeline as a sequence of CUDA kernels, each operating on pixels in parallel to achieve massive speedups.

2 Approach and Implementation

The image processing pipeline consists of the following five elements:

1. Color Filtering
2. Erosion / Dilation
3. Canny Edge Detection — Gaussian Blur
4. Canny Edge Detection — Sobel Masks

5. Canny Edge Detection — Non-Maximum Suppression (NMS)
6. Bitwise-ANDing Color Masks with Edges

Once these steps are complete, the user is left with two arrays of pixels containing only the border lines of the yellow and white lanes. These pixel arrays are then converted into line segments via the Hough Lines Transform to determine the robot's position. Although the Hough Transform would also benefit significantly from GPU acceleration, it is outside the scope of this project for now.

2.1 Color Filtering

Color filtering is the first step in the pipeline. For each input RGB pixel, we convert it to HSV (Hue, Saturation, Value). Each component of HSV is compared against threshold values to determine whether the pixel fits within the definition of yellow or white. This isolates the lane colors from the background, while all other pixels are discarded as zero.

Because each pixel is processed independently and requires no information from other pixels, this step is embarrassingly parallel and maps naturally to assigning one GPU thread per pixel.

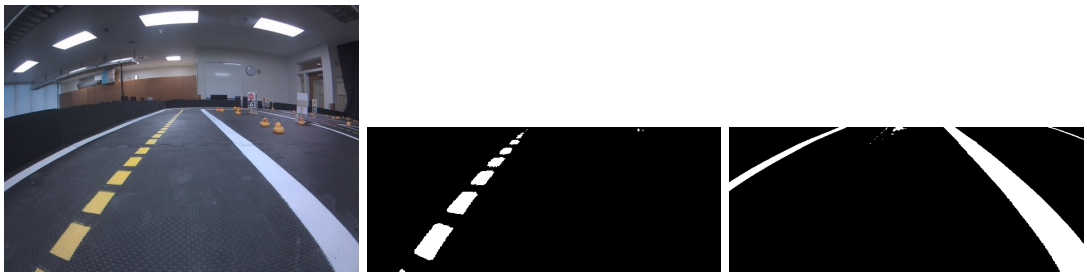

Figure 1: A view of the road in DuckieTown after Color Filtering. (The image is also cropped—sky pixels are not useful for lane detection.)

2.2 Erosion / Dilation

Color filtering may leave behind isolated pixels that match the threshold but are not part of the lane. Because these stray pixels are usually sparse, we remove them using erosion. In erosion, a pixel checks the values of its eight neighbors and assumes the minimum value among them. Erosion typically reduces the thickness of lane markings, so it is followed by dilation, its inverse, which replaces each pixel with the maximum value among its neighbors.

Although dilation must follow erosion, each individual erosion or dilation operation is still embarrassingly parallel.

Figure 2: Yellow lane pixels after Color Filtering (left), Erosion (middle), and Dilation (right).

2.3 Canny Edge Detection — Gaussian Blur

Gaussian blur is the first step of the Canny Edge Detection algorithm. Because Canny looks for sharp intensity changes, small pixel-level noise can be mistaken as edges. By blurring the image, we suppress these false positives. Gaussian blur is applied via digital convolution using a 3×3 Gaussian kernel:

$$K = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

As with previous steps, each pixel can be processed independently and mapped to a unique GPU thread.

2.4 Canny Edge Detection — Sobel Edge Masks

Sobel edge detection identifies locations where pixel intensities change sharply. We convolve the blurred image with the Sobel kernels:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The gradient magnitude and direction are computed as:

$$\text{magnitude} = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

Again, each pixel computation is independent and parallelizable.

2.5 Canny Edge Detection — Non-Maximum Suppression (NMS)

The final stage of Canny Edge Detection is Non-Maximum Suppression. Using the gradient angles, we determine the local direction of the edge. A pixel is kept only if it is the maximum magnitude along the direction normal to the edge. If either of its neighboring pixels in that direction has a larger magnitude, the pixel is suppressed to zero.

Because this operation is performed independently per pixel, it is also embarrassingly parallel.

Figure 3: A DuckieTown road passed through Canny Edge Detection (Left to Right: Grayscale, Blurred, Sobel + NMS).

2.6 Bitwise-ANDing Color Masks with Edges

The final step is to bitwise-AND the processed yellow and white color masks with the final edge map. This step ensures that only lane-colored edges are retained. This operation is trivially parallel, as it is executed independently for each pixel.

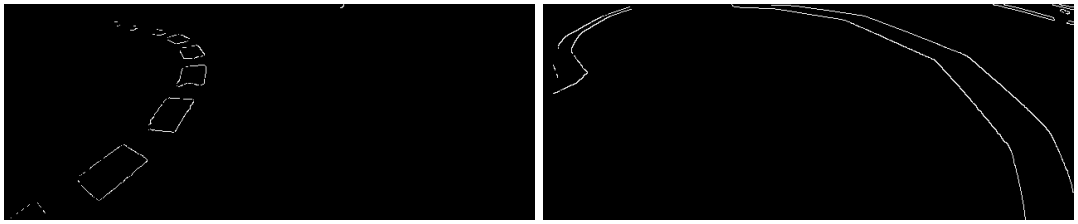

Figure 4: Final lane edges after bitwise-ANDing (Left: Yellow, Right: White).

3 Analysis and Results

Because each step of the pipeline is embarrassingly parallel, each was implemented as a CUDA kernel. We compared the total and per-stage runtimes of the CUDA pipeline with a Python implementation using the CPU-bound OpenCV library. The following chart shows the results.

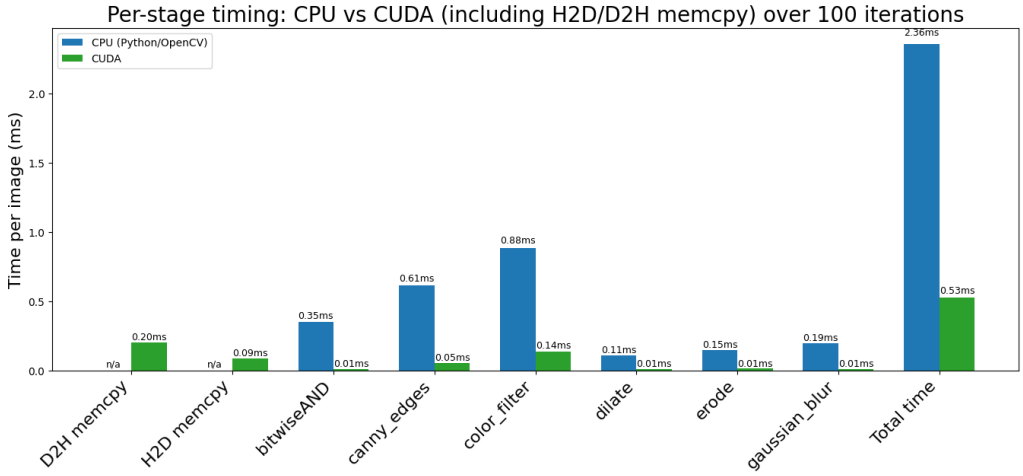

Figure 5: Performance comparison of each stage of the pipeline.

This chart shows that each individual stage of the CUDA implementation substantially outperforms the CPU OpenCV version. However, the CUDA pipeline incurs overhead from copying memory between host and device, which the CPU-only version avoids.

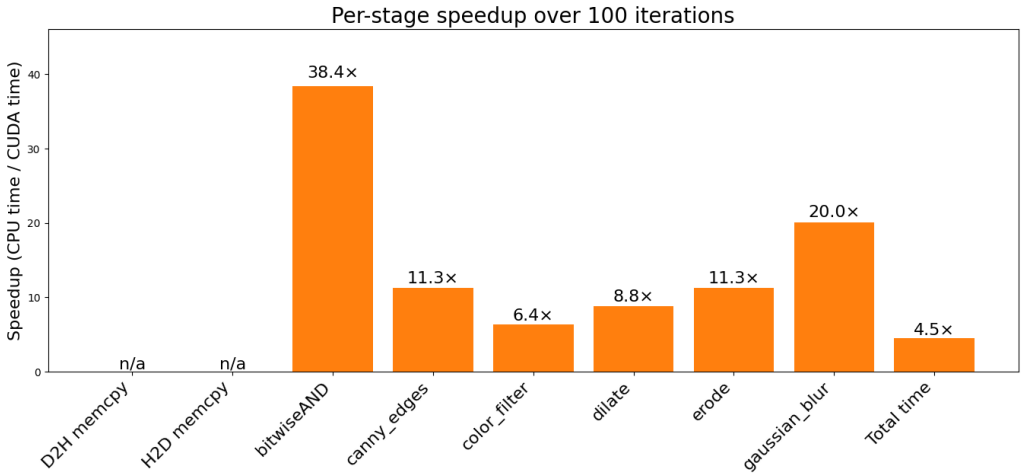

Figure 6: Speed-up factor for each stage of the image processing pipeline.

Even with memory transfer overhead, the total speedup remains significant. Beyond raw performance improvements, GPU acceleration also frees the CPU to perform other tasks while the GPU handles the heavy computation.

4 Conclusion

This project demonstrates the considerable potential that discrete graphics cards offer in accelerating robotics workloads. Even though these algorithms could benefit from even more optimizations (using Look-Up tables, asynchronous CUDA calls, tiling in shared-memory), it still had an enormous speed-up factor of 4.5!

Moving forward, the next steps will be to port this code over to a DuckieBot and run it on the NVIDIA Jetson Nano. Further expansion of the scope of this project (Hough Lines Transform, Path Planning Algorithms, Machine Learning) could also yield insights to how parallel processing can enhance robotics even in a small, simple environment.