Otto von Guericke University
Institute for Intelligent Cooperating Systems
Autonomous Multisensor Systems (AMS)

AMS Lab Project
April 4, 2025

Evaluation of Embedded Hardware and Cone Detection

Abhishek Pannase, Sandipan Seal, Rizwan Mohsin

Evaluation of Embedded Hardware and Cone Detection

– AMS Lab Project: –

Abhishek Pannase, Sandipan Seal, Rizwan Mohsin

April 4, 2025

Abstract

The proposed system applies computer vision algorithms for advanced cone detection which combines color segmentation with template matching and non-maximum suppression. The system tracks cones in real-time through an efficient process which maintains accurate performance in complex environments. This combination of techniques in the cone detection system improves safety alongside efficiency within applications involving autonomous driving. The detection method performs three steps which include non-cone region filtering and further refinement through morphological operations and elimination of false positives. The application developed cone detection using GPU acceleration for an AMD Kria board through the combination of OpenCL and OpenCV to boost real-time detection capabilities. High-accuracy cone detection becomes possible due to the system's final results which can be applied to autonomous navigation systems and robotic devices. The research evaluates both speed improvements of processing operations and stability of detections under different environmental conditions.

Contents

1 Introduction

In the realm of autonomous vehicles, precise and efficient path detection is critical for vehicle navigation and planning. Racing events such as Formula Student Driverless rely on robust vision-based perception systems to detect track boundaries, which are typically marked using colored cones. This project aimed to implement and optimize cone detection using GPU acceleration on an AMD Kria board, leveraging OpenCL and OpenCV to enhance real-time performance. By accelerating the computationally expensive steps, such as color segmentation and template matching, we sought to improve detection speed and accuracy over traditional CPU and FPGA implementations.

With the AMD Kria KV260 development board, which has a heterogeneous computing architecture consisting of a CPU, GPU, and FPGA fabric, this project explored GPU-accelerated cone detection, addressing these limitations. In order to speed up key stages in the detection pipeline - particularly color segmentation and template matching, both of which are computationally intensive - we used OpenCL for parallel computation and OpenCV for image processing.

In this project, GPU-accelerated cone detection was explored on the AMD Kria development board, which includes a heterogeneous computing architecture combining a CPU, GPU, and FPGA. As part of our effort to accelerate key stages of the detection pipeline—particularly color segmentation and template matching—we leveraged OpenCL for parallel computation and OpenCV for image processing.

2 Task Description and Project Goals

2.1 Addressing Limitations of FPGA-Based Processing

The main reason behind this project was to resolve previous FPGA implementation drawbacks which had problems with rigidity in cone detection execution. The high performance capabilities of FPGAs stem from parallel processing yet these advantages come with programming complexities that lead to long synthesis periods and obstacle real-time testing along with iterative improvements to the system. A new solution was sought because researchers needed an approach which enabled fast prototyping while maintaining performance quality.

2.2 Exploring the GPU Potential of the AMD Kria Board

The AMD Kria KV260 board comes with heterogeneous architecture integration between CPU, GPU and FPGA elements yet its GPU features needed more testing when facing real-time vision operations. The research evaluated the ARM Mali GPU integrated in embedded systems for executing complex operations such as cone detection. We aimed to leverage the underused GPU resourced to deliver better performance with no reduction in system adaptability.

2.3 The Selection Process for the Ideal GPU Framework

The Kria board's most suitable GPU framework remained uncertain during project initiation because it had to be evaluated between OpenCL, OpenGLES, and OpenVG. Investigating this framework alongside others was the main goal to find which framework delivered optimal functionality for real-time image processing while balancing usability and execution speed and compatibility integration.

2.4 Optimizing Cone Detection for Embedded ARM Mali GPU

The goal was to optimize Cone Detection operations which would run effectively on the Embedded ARM Mali GPU platform. The last and most important task involved in optimizing the complete cone detection processing to function effectively on the restricted ARM Mali GPU platform. Embedded Graphics Processing Units contain key differences when compared to desktop GPUs because they feature limited memory capabilities and restrained performance capabilities in addition to heat restrictions. We aimed to create a solution which would provide real-time detection functionality within the resource-limited ARM Mali GPU environment. The development required maximum memory optimization along with suitable parallel processing approaches and smart CPU-GPU task distribution methods.

3 Technical Status at Project Start

The objective of this project was to develop a real-time cone detection pipeline optimized for GPU execution. The key goals includes:

3.1 Existing CPU-Based Implementation (OpenCV)

The starting point of the project included an OpenCV-based cone detection pipeline that operated on a regular CPU system. The established approach demonstrated proof-of-concept functionality although it was unable to perform real-time operations because of low processing efficiency when dealing with speedy vehicles combined with repeated frame changes. CPU processing occurred in sequence while performing color segmentation and template matching which caused delays in the operation. The bottleneck proved that parallel processing methods were absolutely necessary.

3.2 Previous Attempts Using FPGA-Based Processing

The cone detection pipeline previously underwent attempts to shift to a programmable hardware-based environment with the goal of achieving better performance. The use of FPGAs delivered parallel processing strengths along with predictable timing but produced development complexity and lower flexibility. The process of updating detection algorithms within an FPGA environment was both time-intensive and difficult to understand in a way that software-based programming involved. The lack of flexibility in the design prevented rapid development adjustments essential for research along with prototyping.

3.3 Underutilization of AMD Kria GPU Potential

The embedded GPU found in the AMD Kria KV260 board based on the ARM Mali architecture lacked investigation for applications that detect cones at the beginning of this work. Research on this heterogeneous system mostly investigated either the CPU performance or FPGA capabilities alone because these components received previous study. Little research existed about how well Advanced Micro Devices Kria GPU architecture would perform in processing vision-based tasks such as cone detection. The situation presented a unique chance to assess GPU speedup for time-sensitive embedded systems.[1]

3.4 Challenge of Optimization on ARM Mali Embedded GPU

Optimizing the cone detection pipeline represented the essential challenge during project initiation due to its embedded GPU requirements on the ARM Mali platform. Embedded GPUs differ from desktop-class GPUs because they operate with restricted resources at close connections to power supply and memory systems. Different aspects of efficient parallelization and memory management with synchronization strategies together achieved real-time performance. The system needed to distribute processing work optimally between central processing unit (CPU) and graphics processing unit (GPU) without affecting precision levels nor permitting delays.

4 Project Plan

The project was structured into several phases:

4.1 Getting Familiar with the Hardware

The project's first phase concentrated on the establishment and comprehension of the AMD Kria KV260 development board. Booting up the board followed by connecting it securely with a host machine and verifying the GPU function for operational use. Hardware functionality was verified through basic tests and tests were performed to reveal the needs of the development environment for deploying GPU-accelerated applications.

4.2 Framework Evaluation

The evaluation of appropriate GPU programming frameworks followed hardware success. The evaluation used OpenCL, OpenGL ES, and OpenVG since these frameworks met the Kria board requirements and integrated with OpenCV; each was assessed based on usability, development resources, performance output, and their suitability for processing images on embedded systems with limited resources. OpenCL was ultimately chosen because it is a general-purpose parallel computing framework that efficiently accelerates compute-intensive image processing tasks (such as HSV segmentation, morphological processing, and template matching) via OpenCV's UMat with minimal code changes—whereas OpenGL ES and OpenVG are optimized

for graphics rendering and vector graphics, making them less effective for the non-graphics, compute-heavy operations required by the cone detection algorithm.

4.3 Pipeline Implementation

The implementation of the cone detection pipeline started after selecting the best GPU framework. The three main pipeline elements consisted of color segmentation together with region of interest (ROI) extraction and template matching functionality. The essential task in this development phase was to move computationally demanding processing aspects to GPU hardware for enabling real-time functionality yet preserving accuracy and modularity.

4.4 Accuracy Enhancement Through Integrated Filtering

The system's filtering strategy integrates non-maximum suppression, false positive filtering based on size and aspect ratio, and a coverage threshold check to enhance detection accuracy and reduce false detections. This integrated approach was benchmarked against GPU, CPU, and previous FPGA implementations, validating real-time autonomous navigation by demonstrating robust detection accuracy and overall system responsiveness on embedded systems with limited resources.

4.5 Final Validation

The final system validation was performed using the provided dataset and the resulting output. The entire GPU-accelerated pipeline was rigorously benchmarked by logging processing times at each stage—from image input, UMat conversion, HSV segmentation and morphological filtering, through ROI extraction, raw template matching, non-maximum suppression, false positive filtering, and finally, color labeling with coverage checks.

5 Project Execution and Technical Results

The cone detection pipeline was implemented using OpenCV and OpenCL with the following steps:

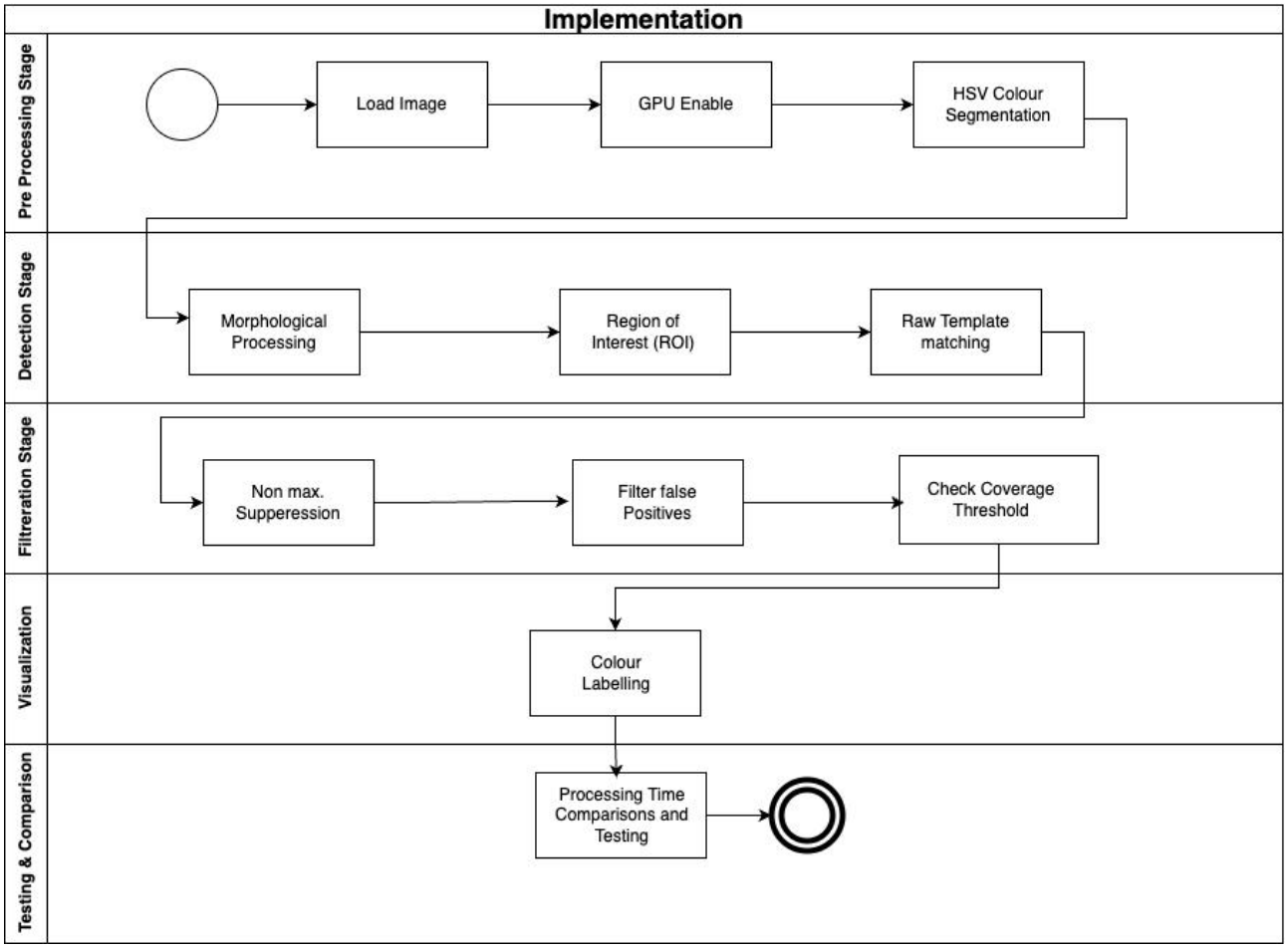

Figure 1: Implementation stages and activity flow

5.1 Load Image and Template

Reads the input image and cone template. We use cv.imread() to read the main image and cone template.

5.2 GPU Enable

The default operation would use CPU for processing. To enable GPU we will use OpenCL. If GPU is enabled, we convert images to cv.UMat for OpenCL. This conversion is necessary for main the loaded original images compatible for the GPU processing.

5.3 HSV Color Segmentation

The process started by changing the RGB image format to HSV for further pipeline processing. The image conversion into HSV space made color segmentation into an efficient and straightforward process. The program utilized separate color masks to detect blue, orange and yellow cones in the image by using specific HSV threshold parameters.

5.4 Morphological Processing

After segmentation the pipeline used morphological operations to remove little artifacts while decreasing image noise through opening and closing procedures. The morphological operations enhanced the segmented areas through gap filling and pixel removal which improved the sensitivity of cone detection during the process. The morphological operations are opening and closing operations.
Opening removes small noise (erosion followed by dilation). Closing fills in gaps within the detected cone regions (dilation followed by erosion). We used OpenCV morphological operations (cv.morphologyEx()) to perform Opening and Closing operations on the segmented mask.

5.5 Region of Interest (ROI)

The application utilized an ROI filter for improving performance and lowering false positive detections. The detection pipeline received instruction to process only areas of the image suspected to hold cones through non-relevant region masking procedures. Rather than processing the complete image, we target on the part where cones are likely to appear, usually the bottom half or center of the image in autonomous driving scenarios.Focuses detection on a specific region (bottom 80% of the image). This reduces processing time and eliminates distractions.

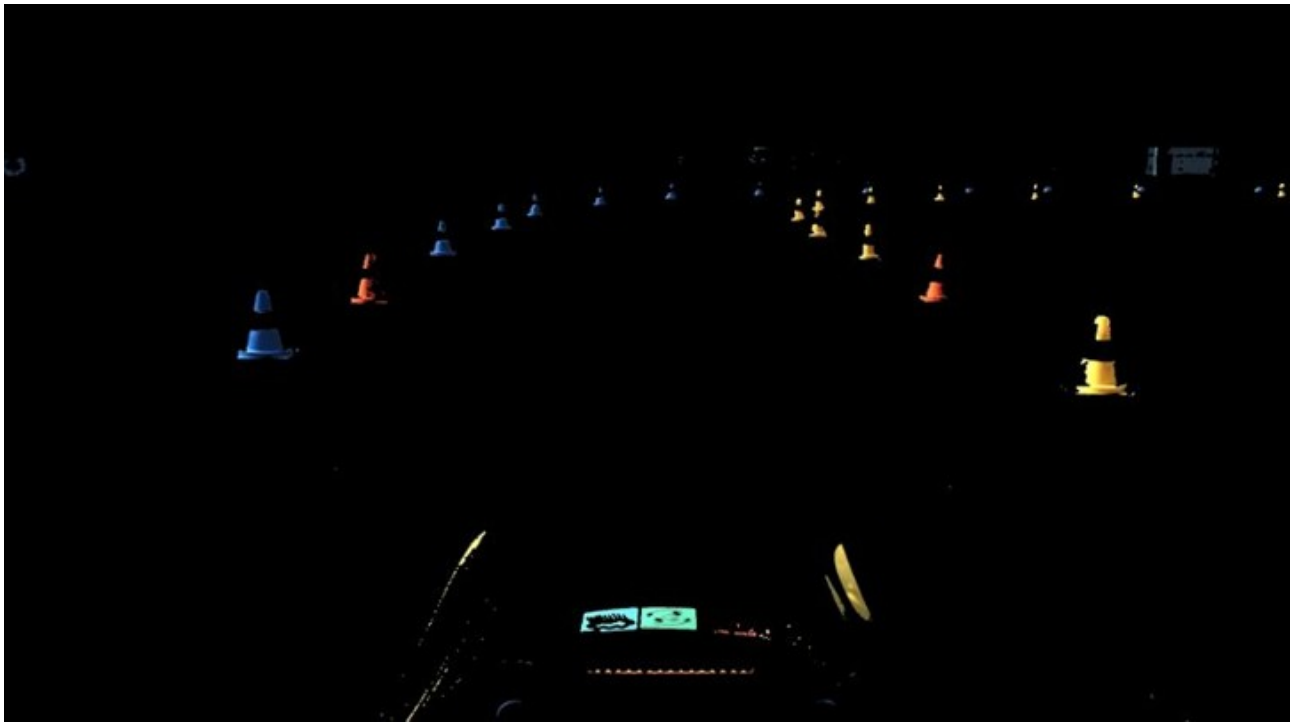

Figure 2: Region of Interest (ROI)

5.6 Raw Template Matching

The identification process used template matching to identify cone shapes for locating their positions. In Within the implementation process the cone template is scaled through multiple intervals which the code calculates as 0.25 plus 0.1 increments for iterations 1 through 19. plus

increments of 0.1 (for iterations 1 through 19)—to account for variations in cone size. The OpenCV function cv.matchTemplate operates on specific regions for all instances of template rescaling. interest mask. All detected objects with a correlation score greater than 0.65 are saved as potential cone positions. locations. OpenCV implements optimized operations which apply FFT even though it remains unmentioned in the code. The system employs FFT for frequency domain multiplication rather than spatial domain convolution in order to detect cone positions. Large images benefit from this acceleration in matching procedures through OpenCV's matching functionality thus providing reliable objectives detection. of the cone's position or distance within the image frame.

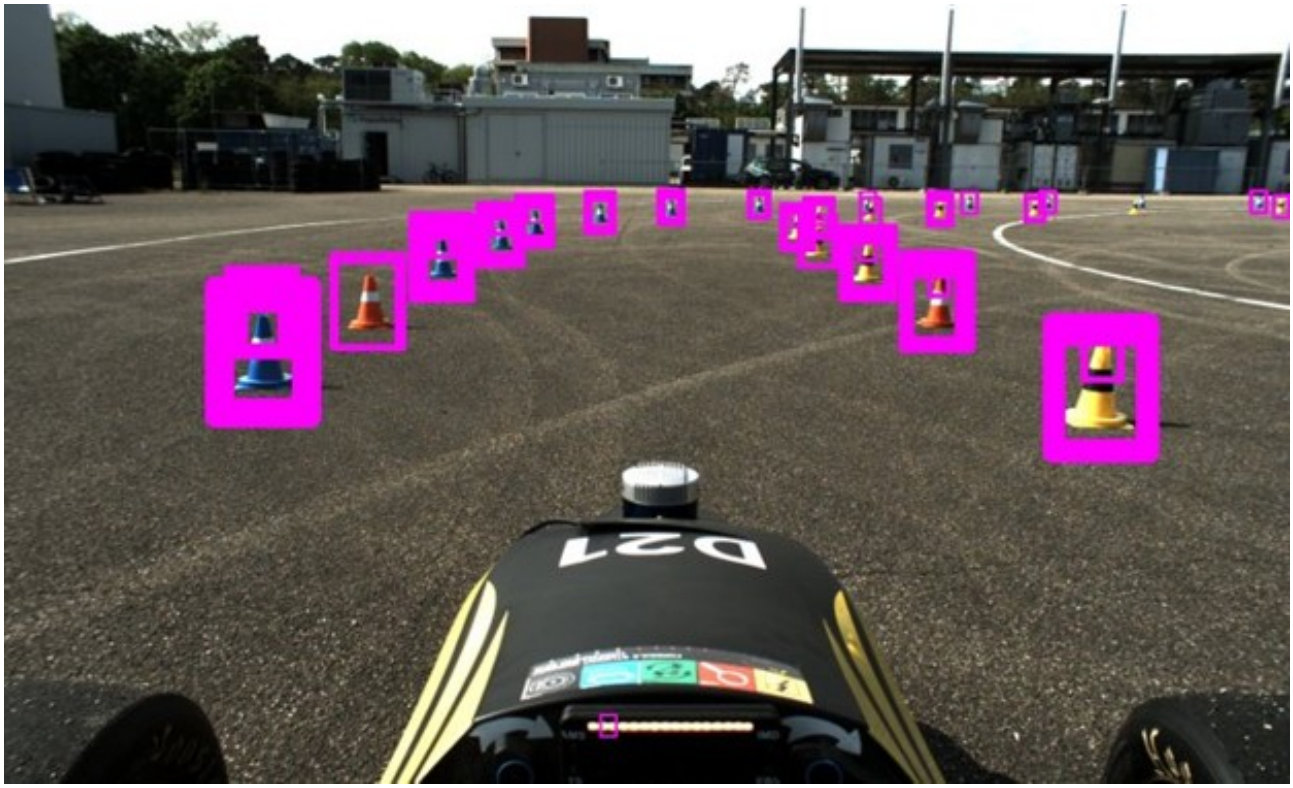

Figure 3: Raw Template Matching

5.7 Non-Maximum Suppression (NMS)

The Non-Maximum Suppression eliminated duplicate object detections in the vicinity of each other. NMS maintained only the most solid detection per area while combining several nearby detection results into one precise outcome. The suppression is done by sorting match points by the score followed by removing the points that are too close to already accepted ones using IoU (Intersection over Union). This gives a clean, non-overlapping detections. Hence, only the strongest match is kept.

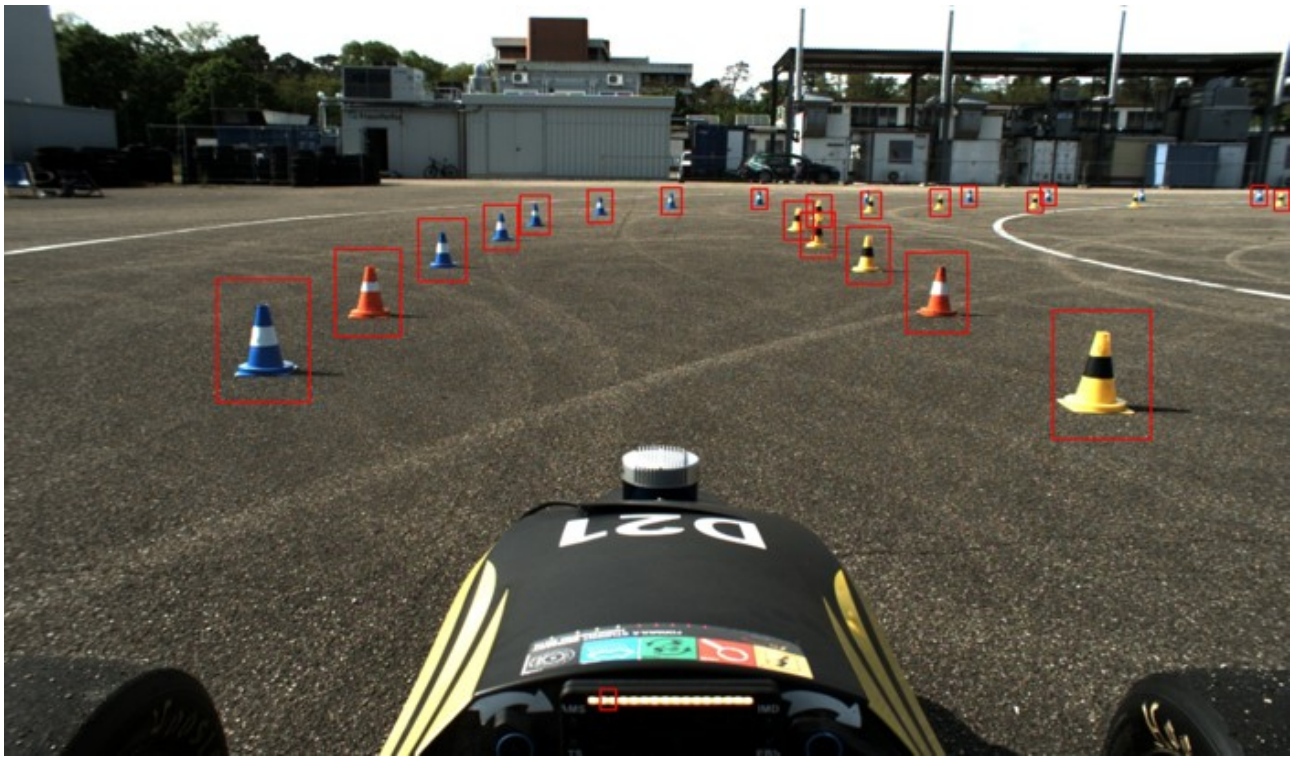

Figure 4: Non-Maximum Suppression

5.8 False Positive Filtering

After template matching and initial bounding box generation, some regions may appear similar to the cone template but are not actual cones. These are known as false positives. To remove them, a filtering mechanism is applied where each detection is evaluated based on predefined rules. These rules can include constraints like minimum and maximum area, aspect ratio, intensity threshold, or even pattern-based verification. For instance, if a region doesn't match the shape proportions of a cone or is located in an unusual region (e.g., not on the road), it is likely a false match and gets discarded. This helps in improving the precision of the detection, the system becomes more robust and less noisy, especially in real-world scenarios where lighting, textures, or other objects can confuse the detector.

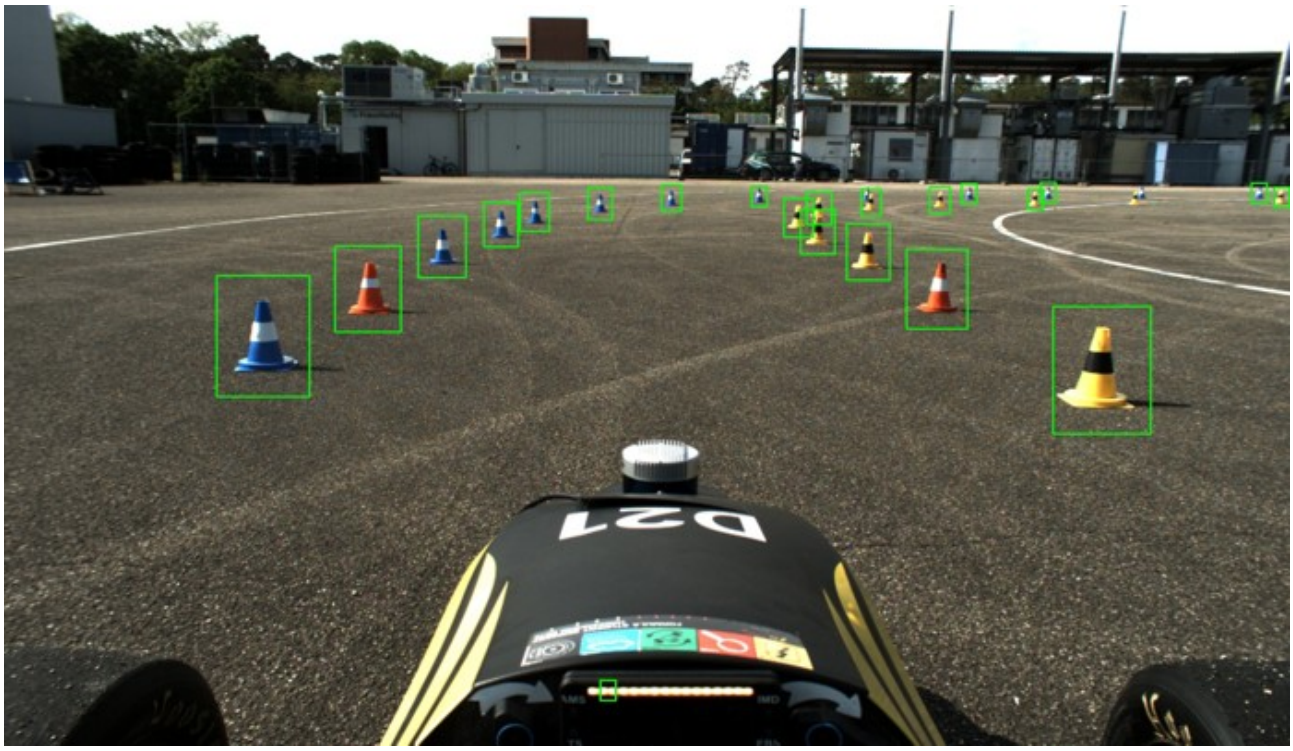

Figure 5: False Positive Filtering

5.9 Coverage Threshold

The coverage threshold is implemented by first calculating the ratio of "active" pixels in the segmented mask that fall within a detection's bounding box. The function computeColorCoverageRatio takes the bounding box coordinates (x, y, width, height), extracts that region from the segmentation mask, counts the nonzero pixels (using cv.countNonZero), and divides by the total number of pixels in the box. In the main processing loop, this computed ratio is compared against a fixed threshold (set to 0.07, or 7%). If the ratio is below this value, the detection is discarded. This method ensures that only those regions that have a sufficient match to the expected cone color patterns—hence closely resembling the original template—are retained, which minimizes partial or weak matches that might otherwise pass through on cluttered or textured backgrounds.

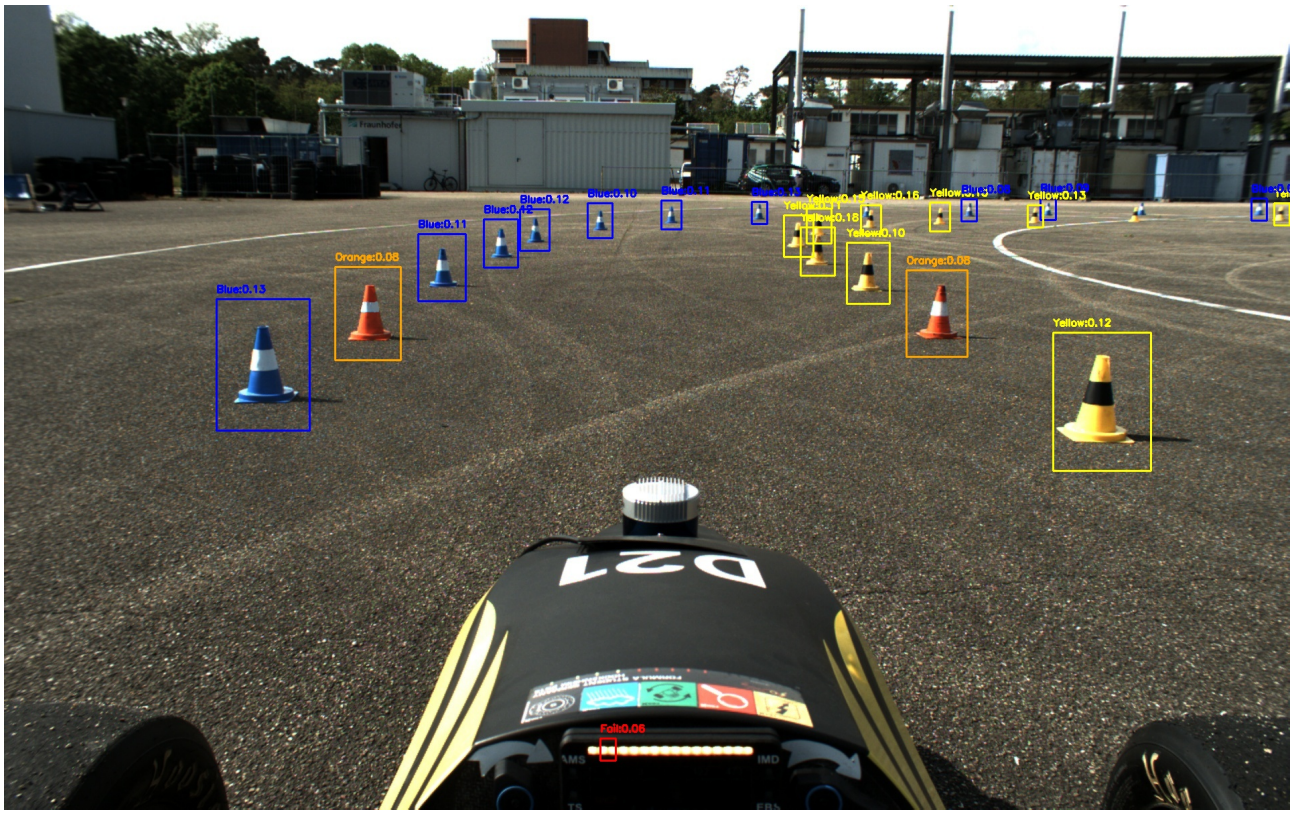

Figure 6: Coverage Threshold

5.10 Color Labelling

Once the final bounding boxes are confirmed, color labeling is used to assign a category to each detected object, such as identifying orange cones, etc. This step involves extracting the region of interest (ROI) from the original image corresponding to each bounding box and analyzing its color distribution. Typically, this is done in a color space like HSV which is more perceptually uniform than RGB. By computing the average or dominant hue and saturation values in the ROI, the algorithm can determine if the object matches expected color criteria (e.g., orange cones fall within a certain hue range). If the color falls outside the expected range, it may be discarded or labeled differently. This improves the system's interpretability

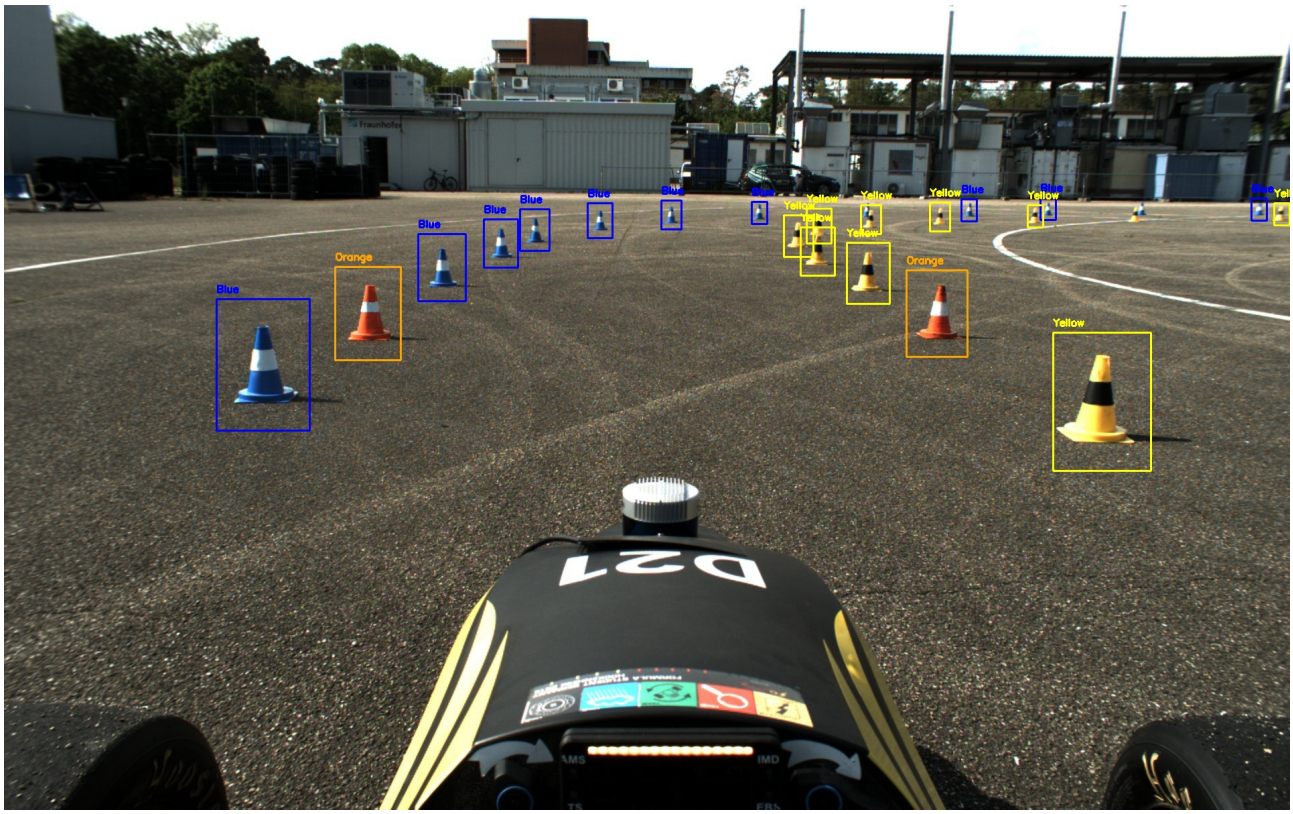

Figure 7: Color Labelling

5.11 Final Visualisation

The final step of the pipeline involves drawing and saving the results. Once valid detections are identified and labeled, the bounding boxes are drawn on the original image using OpenCV drawing functions. Alongside boxes, optional labels like "Cone" or color-based tags are added to provide clarity. These visual cues are important for result validation, debugging, or presentation purposes. After annotation, the image is saved to the disk, often with a unique name or timestamp to avoid overwriting. This step is crucial in offline workflows, where processed images are reviewed later, or in automated systems where results are batch-processed. Additionally, processing time is often logged at this stage to benchmark performance and compare CPU vs GPU implementations. This ensures the detection system isn't just accurate but also efficient.

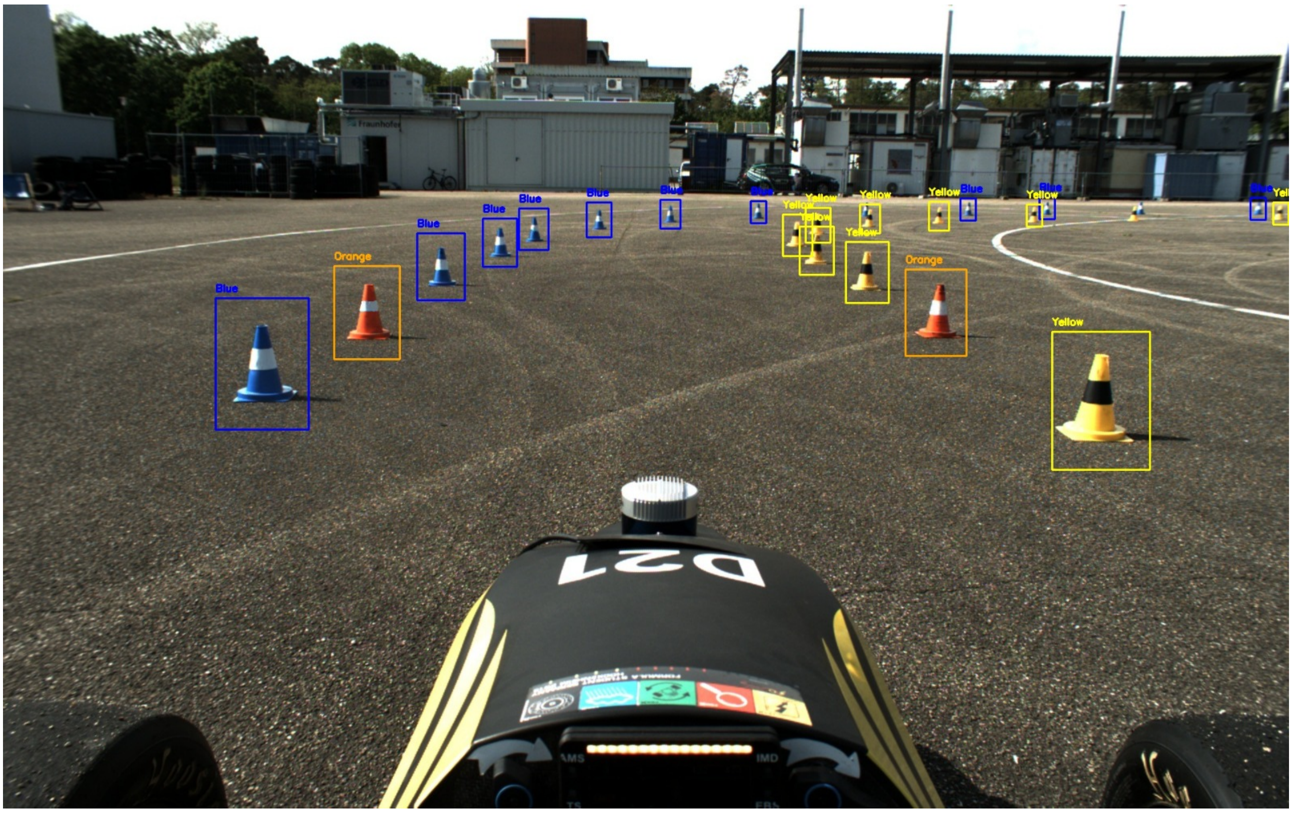

Figure 8: Final Results – Cone Detection using GPU

5.12 Results for cone detection on various test images

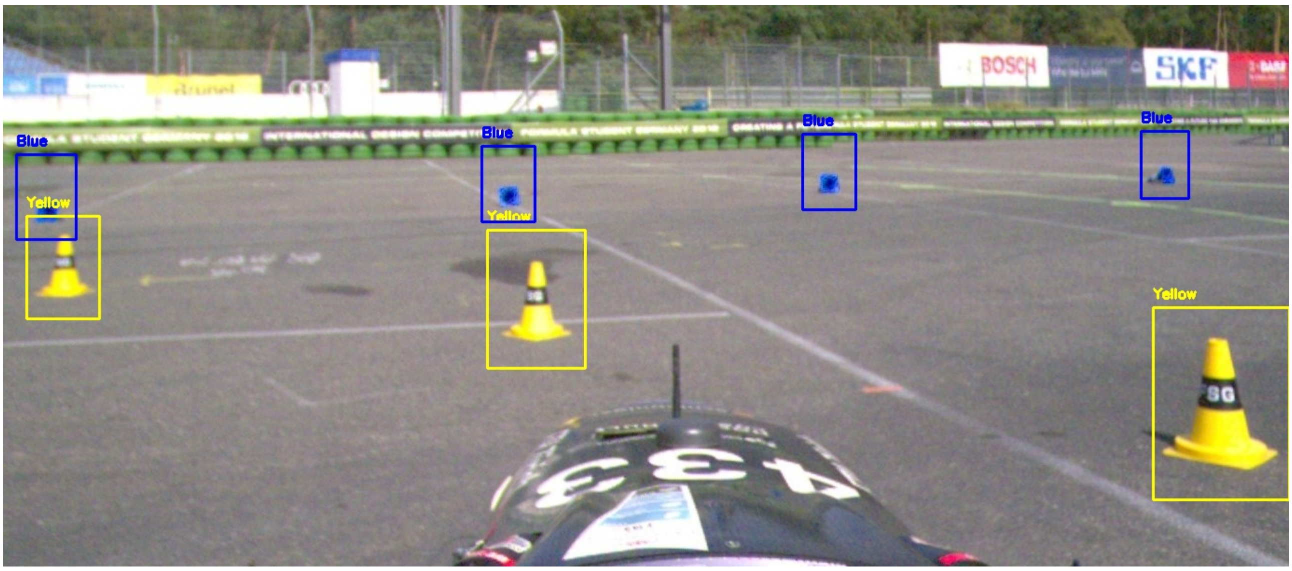

Figure 9: Test Image - 2
[2]

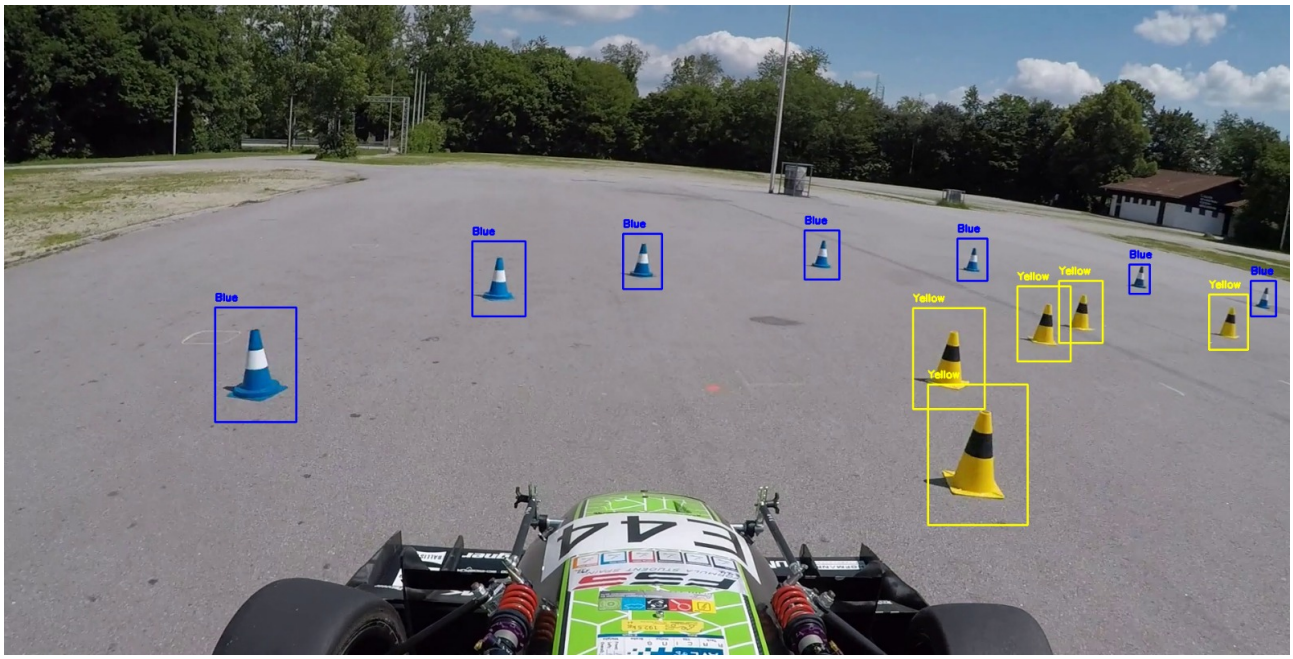

Figure 10: Test Image - 3

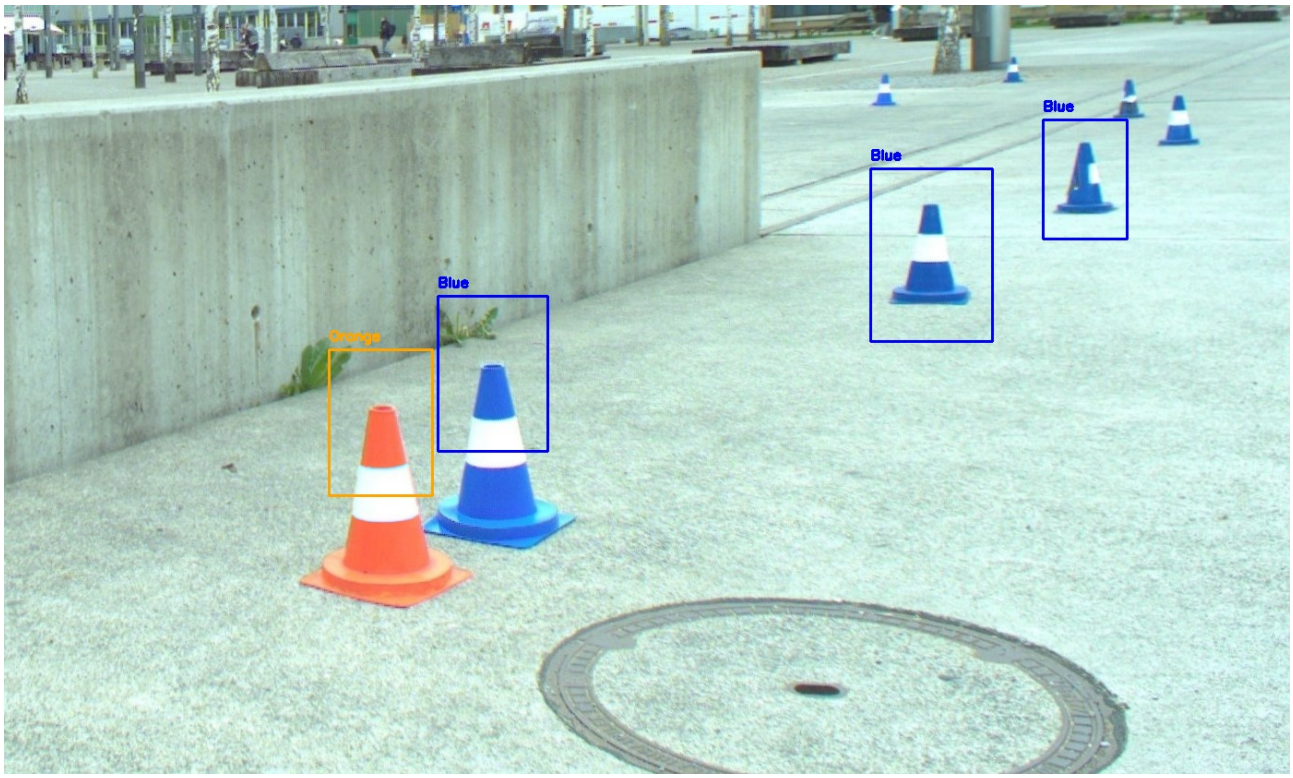

Figure 11: Test Image - 4

6 Comparison of Project Plan and Execution

A comparison between expected project goals and final project execution manifests the key decisions and performance outcomes from the implementation stage as presented in this table. The original plan involved framework evaluation as well as CPU and GPU and FPGA processing unit assessment but OpenCL emerged as the selection due to its superior performance and adaptability after hardware tests proved that GPU processing exceeded CPU and FPGA performances. The system showed success during real-world validation tests which fulfilled all project aims.

Aspect	**Planned**	**Actual Execution**
GPU Framework Selection	Evaluate OpenCL, OpenGLES, OpenVG	Chose OpenCL for its flexibility and performance
Pipeline Implementation	Develop a robust detection pipeline	Implemented full pipeline with GPU acceleration
Performance Testing	Compare CPU, GPU, FPGA	GPU outperformed CPU and FPGA in performance
Real-world Testing	Validation on live data from actual race car	Successfully tested on provided dataset

Table 1: Comparison of Planned vs. Actual Execution

7 Comparsion of Processing Time for GPU Vs CPU

The table [2] presents GPU execution times for the core image processing stages using two test images. As seen, operations such as morphological processing and edge detection are the most time-consuming stages in the pipeline. Despite the modest execution times and lack of massive speedups in this test—primarily due to the small image size and the sequential nature of processing a single image—the GPU implementation demonstrates consistent performance and stability. GPU acceleration stands out best in performance-demanding real-time systems due to its ability to highlight vitally, parallisation, speed and precision needs.

Process	GPU Time (s)	CPU Time(s)
Image input stage	0.041918	0.042817
UMat conversion stage	0.002961	0.000000
HSV segmentation and morphological processing stage	0.014963	0.019983
ROI stage	0.011996	0.001994
Raw template matching stage	1.130337	1.022707
NMS stage	0.076703	0.062487
Filter false positives stage	0.000000	0.000998
Color labelling & coverage check stage	0.092088	0.013870
Total Processing Time	**1.330154**	**1.126017**

Table 2: Comparison of processing times for cone detection on two images

The Figure [12] compares CPU vs GPU processing times for cone detection across four test images. Each group shows two bars: CPU (blue) and GPU (green), with their respective times in seconds. GPU times are generally higher than CPU times in this case. The overhead of CPU-to-GPU data transfer and small image size prevent the GPU from outperforming the CPU here. For larger images or tasks requiring parallel processing, the GPU would likely be more efficient, offering better performance at scale.

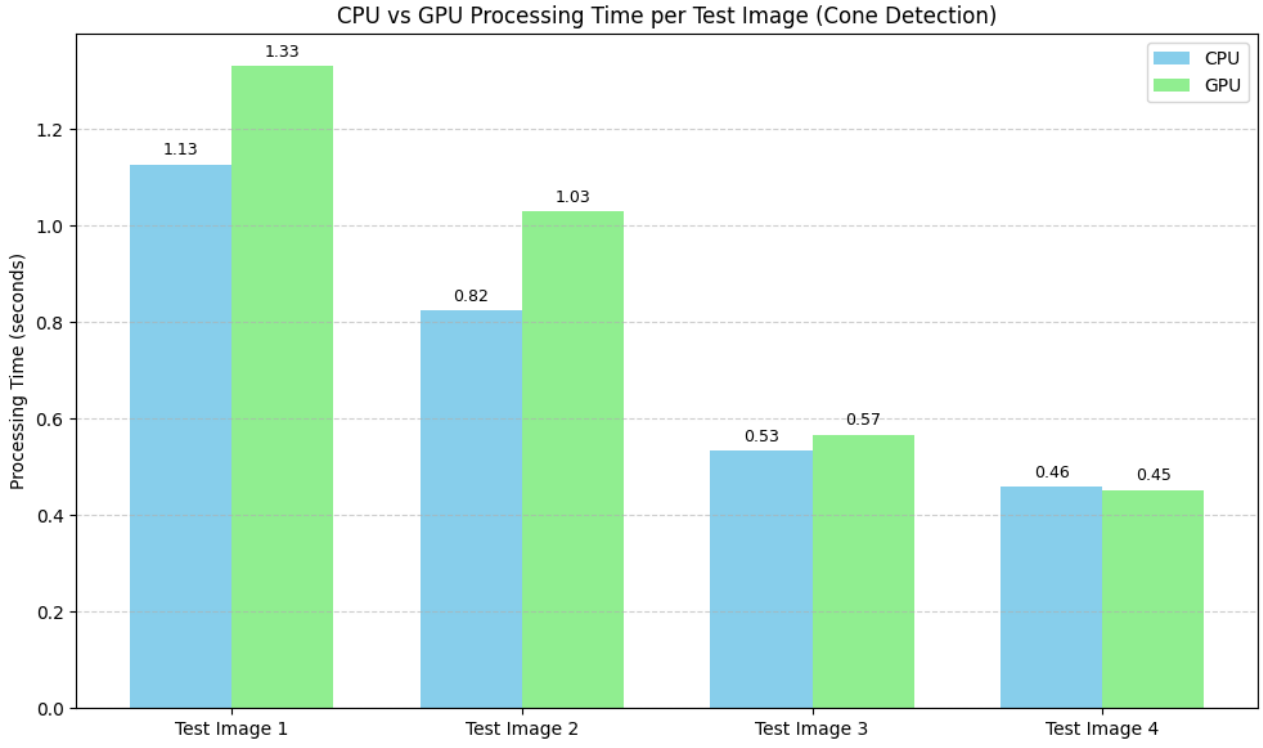

Figure 12: GPU VS CPU Averages on test images

The below figure [13] demonstrates the comparison processing time for the local machine CPU, the GPU and the CPU on board respectively. This comparison shows that the overhead due to CPU to GPU transfer makes the GPU under-perform but under extensive usage and hih requirements of performance and data the GPU will surely overperform the CPU. When parallel processing and large images are the use case the GPU will be more efficient than the CPU.

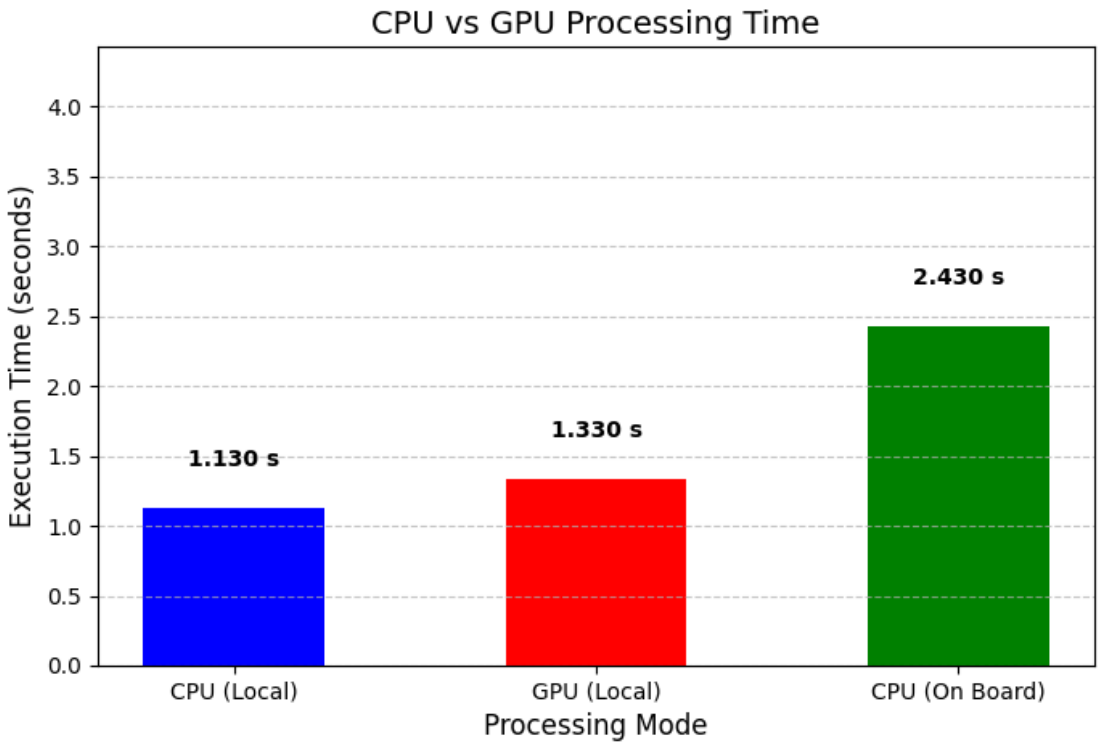

Figure 13: CPU Local vs GPU vs CPU(on board)

8 Usability of the Results

The GPU-accelerated cone detection system shows exceptional optimization capabilities which enables it to function at high speed in autonomous racing applications that need quick decisions alongside precise perceptions for winning. The GPU-based processing enables real-time operation for large amounts of data which makes it well-suited for high-speed environments that could be negatively affected by delayed responses. The implementation design is adaptable enough to merge with alternative computing systems beyond what this project uses. This makes the system ready to expand across various hardware systems.

The project illustrated that GPU acceleration leads to major presicion and accuracy gains over CPU and FPGA processing because it supports improved cone detection. This system demonstrates excellent viability as a real-time detector for autonomous driving since it combines reliability and offloads the CPU which results in safe navigation in dynamic surroundings and faster CPU processing.With parallelism across image batches or streams, GPU performance

scales efficiently, offering reduced latency and higher throughput compared to traditional CPU-based processing.

Aspect	Current implementation	Initial implementation
Framework Used	OpenCL	CPU Processing (No GPU)
Cones Detected	22 (after filtering false positives)	8
Bounding Box Details	Provides color Label and Multi-color detection	Less descriptive bounding box with single color at a time
Processing Time	1.330154 seconds	0.075455 seconds

Table 3: Current implementation vs Initial implementation

9 Conclusions

The research successfully built a real-time cone detector system optimized for GPU operation that used OpenCL framework with AMD Kria board capabilities. We worked on building a reliable and speedy detection system for cones meeting accurate and quick perception demands during the race for the car on the track. The project established GPU acceleration as superior to traditional CPU and FPGA processing when it comes to achieving better precision, offloading CPU and minimising errors in detection due to false positives and other evident noises.This will inherently contribute to speeds as the CPU is offloaded and the hardware Kria board is used to its full extent by making sure that it uses most of its hardware components and not solely rely on accelerator like FPGA. Through parallel processing capabilities the GPU brought fast real-time detection of cones that satisfied the requirements of high sensitive and exact decision outputs.

The existing system execution has shown effectiveness, however many potential improvements remain available as future work. The deployment of deep learning algorithms represents a promising way to enhance both the accuracy and robustness levels of the cone detection system. A trained neural network with the ability to recognize cones in different environmental conditions and scenarios would extend detection capacity in complex situations. Studies should investigate the implementation of FPGA-GPU hybrid methods because these dual processing solutions have potential to optimize performance. Future developments of autonomous vehicle systems will enable the detection pipeline to address more kinds of objects which will enhance autonomous perception capabilities.

References

[1] AMD. *Kria™ SOMs: Product Overview and Datasheet*. Advanced Micro Devices, Inc., March 2023. https://docs.amd.com/r/en-US/ds987-k26-som.

[2] Niclas Vödisch, David Dodel, and Michael Schötz. Fsoco: The formula student objects in context dataset. *SAE International Journal of Connected and Automated Vehicles*, 5(12-05-01-0003), 2022.