ABSTRACT

This project focuses on optimizing computationally intensive functions using ARM NEON intrinsics for embedded auto-control systems. NEON is ARM's SIMD (Single Instruction, Multiple Data) extension that enables parallel processing. The primary goal was to accelerate functions such as histogram calculation, grayscale conversion, Sobel edge detection, and box blurring. These are crucial in real-time perception systems like ADAS (Advanced Driver Assistance Systems).
NEON-based optimizations significantly reduced processing time across all functions. For instance, grayscale conversion showed 2× speedup, and Sobel filtering showed a 3× improvement. These results demonstrate that NEON intrinsics can greatly enhance system responsiveness, lower power consumption, and improve performance in embedded vision applications.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In modern embedded systems, particularly within automotive electronics, the ability to process visual data in real time is critical for applications such as Adaptive Cruise Control, Lane Keeping Assist, Obstacle Detection, and Driver Monitoring Systems. These applications demand high-performance image processing under strict latency and power constraints, especially on edge-AI platforms like the NVIDIA Jetson AGX Orin.

Traditional scalar implementations of image processing algorithms—such as grayscale conversion, histogram analysis, edge detection, and blurring—often become performance bottlenecks due to the sheer volume of pixel data and the need for real-time responsiveness. To address these limitations, this project proposes a function-level optimization framework leveraging ARM's NEON SIMD (Single Instruction, Multiple Data) architecture.

The project focuses on rewriting critical image processing functions using NEON intrinsics, enabling data-level parallelism to dramatically accelerate execution on Jetson's ARM Cortex-A78AE CPUs. NEON provides 128-bit vector registers that allow simultaneous processing of multiple pixels or operations in a single instruction cycle, significantly improving throughput without increasing CPU frequency or power consumption.

The optimized system is benchmarked against baseline (non-NEON) implementations for four key functions: grayscale conversion, Sobel filter-based edge detection, histogram calculation, and box blurring. Each function was carefully profiled and restructured using NEON load-store operations, vectorized arithmetic, and approximation techniques (e.g., bit-trick for exponential functions) to demonstrate real-world performance gains.

The results highlight substantial improvements, such as a 2× speedup in grayscale conversion and a 3× reduction in edge detection latency. These optimizations directly translate into better frame rates and reduced energy consumption—essential for real-time ADAS systems deployed in resource-constrained embedded environments.

The modular design of this optimization framework and its compatibility with common edge-AI platforms make it highly reusable for a wide range of vision-based applications. Future extensions could include deep learning model acceleration, GPU-CPU hybrid optimization, or integration with real-time sensor fusion pipelines for complete ADAS solutions.

1.1 Motivation

Embedded systems for ADAS and autonomous driving require real-time image processing with minimal latency. Conventional scalar code struggles under these constraints. NEON intrinsics allow multiple data points to be processed simultaneously, making them ideal for speeding up image preprocessing tasks like histogram calculation and edge detection.

1.2 Objectives

- To identify and profile key image processing functions in automotive control systems that act as performance bottlenecks (e.g., grayscale conversion, histogram calculation, edge detection, and box blurring).

- To implement NEON-optimized versions of these functions using ARM SIMD intrinsics on the NVIDIA Jetson AGX Orin platform.

- To leverage 128-bit NEON vector registers and parallel data processing techniques to achieve significant reductions in execution time and latency.

- To conduct side-by-side benchmarking and performance analysis between baseline (scalar) and NEON-optimized implementations across 1000+ iterations.

- To incorporate additional mathematical optimizations, such as bit-trick approximations for exponential operations used in histogram compression.

- To demonstrate the functional correctness and real-time gains of NEON-optimized code through quantifiable metrics like speedup factor, execution time (µs), and resource utilization.

- To develop a modular, reusable, and low-cost software optimization framework for accelerating vision-based embedded systems in edge AI, ADAS, and robotics.

1.3 Problem statement

Neon Optimization of Function from Auto Control

Real-time functions in ADAS like lane detection, object tracking, and sensor fusion require fast, low-power processing. Without optimization, these algorithms risk missing deadlines or overloading embedded CPUs. This project aims to close that gap using NEON intrinsics.

1.4 Application in Real-Life Context

- Adaptive Cruise Control and Lane Keeping
- Real-time video processing in drones and robotics
- Surveillance systems with edge AI processing
- Smart traffic monitoring and autonomous navigation

1.5 Project Planning and Bill of Materials

1.5.1 Project Planning

This project focuses on optimizing core image-processing functions using ARM NEON intrinsics on the Jetson AGX Orin platform. These functions—such as histogram calculation, grayscale conversion, and edge detection—are essential in real-time auto-control systems. The workflow includes NEON-based optimization, benchmarking, and performance comparison with baseline implementations to validate speed and efficiency improvements.

System Setup:

- Jetson AGX Orin with a computer.

Data Validation:

- Performance of each optimized function is benchmarked against its baseline implementation.
- Speedup and correctness are verified using execution time analysis across multiple iterations.

1.5.2 Bill of Material

Component	Description	Quantity	Cost (INR)
Jetson AGX Orin Dev Kit	Controller Device (CPU)	1	2,50,000

Table 1.1: Bill of Material

1.6 Organization of the report

This report is organized into five chapters, each covering a distinct aspect of the I2C Camera Emulator project. Below is an overview of the chapter structure:

- **Chapter 2 – System Design:** This chapter describes the overall system architecture, including the target hardware platform (Jetson AGX Orin), the role of NEON SIMD optimization, and the selection of functions chosen for acceleration.
- **Chapter 3 – Implementation Details:** This chapter provides the implementation specifics for each optimized function—such as histogram calculation, grayscale conversion, Sobel filtering, and box blurring. It includes the use of ARM NEON intrinsics, code-level techniques, and algorithmic strategies.
- **Chapter 4 – Results and Discussions:** This chapter presents performance benchmarks comparing NEON-optimized functions with their baseline versions. It includes speedup factors, execution time analysis, and observations on computational efficiency.
- **Chapter 5 – Conclusions and Future Scope:** The final chapter summarizes key outcomes, emphasizes the impact of NEON optimization, and discusses potential extensions such as support for higher resolution data, GPU-based optimization, or real-time deployment in ADAS systems.

Chapter 2

System Design

The **Neon Optimization framework** is designed to accelerate critical image-processing functions in embedded auto-control systems using ARM's NEON SIMD architecture. Implemented on the NVIDIA Jetson AGX Orin platform, this project targets real-time performance bottlenecks in operations like histogram calculation, grayscale conversion, edge detection, and box blurring. By leveraging NEON intrinsics, the system enables high-throughput, low-latency processing suitable for applications such as ADAS. This setup provides a reproducible and performance-focused environment for evaluating the computational efficiency of optimized functions, allowing developers to enhance the responsiveness and reliability of embedded vision pipelines without the need for additional hardware or external accelerators.

2.1 Functional Block Diagram

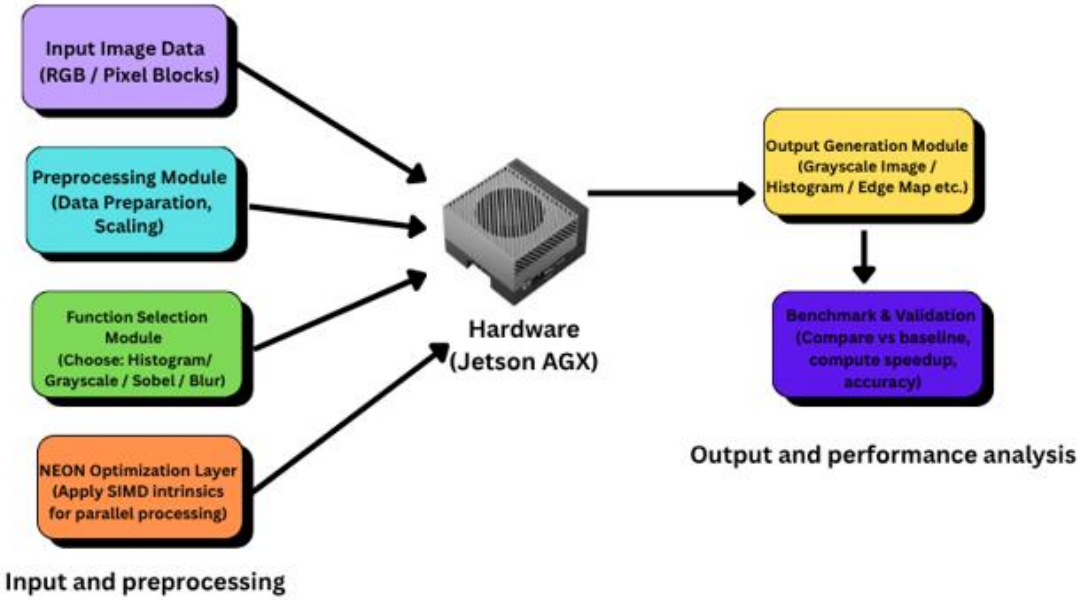

Figure 2.1: Block diagram depicting overview of the system operation.

2.2 Design Alternatives

In the development of the **Neon Optimization framework**, several design alternatives were explored for optimizing key image-processing functions before finalizing the current implementation. These alternatives focused on **optimization strategies, instruction-level techniques, and performance validation methods**, all evaluated based on execution speed, resource efficiency, and compatibility with the Jetson AGX Orin platform.

Key alternatives considered:

- **Manual Loop Unrolling vs NEON Intrinsics:**
  Loop unrolling was initially considered to reduce loop overhead, but NEON intrinsics provided better parallelism and scalability with SIMD registers.
- **Taylor Series vs Bit-Trick for Exponential Calculations:**
  For functions involving exponential compression (e.g., histogram normalization), bit-level tricks were preferred over Taylor expansion due to superior execution speed and minimal memory overhead.
- **Inline Assembly vs Compiler Intrinsics:**
  While inline assembly offered finer control, NEON compiler intrinsics were chosen for better readability, portability, and ease of integration into existing C/C++ codebases.
- **Function-specific vs Generic SIMD Modules:**
  Initially, a generic SIMD template was considered for reuse across functions, but dedicated NEON routines were found to be more efficient due to their tailored memory access patterns.

2.3 Final Design

After evaluating various approaches, the following design decisions were adopted to maximize performance and maintain code clarity:

- **NEON Intrinsics:**
  ARM NEON intrinsics were selected over inline assembly for optimizing image-processing functions due to their portability, ease of use, and effective parallelism.
- **Bit-Trick Optimization:**
  For exponential operations (used in histogram normalization), bit-trick methods were chosen for faster execution over traditional Taylor series expansion.
- **Dedicated SIMD Paths:**
  Instead of using a generic SIMD module, each function was optimized individually to take advantage of specific data patterns, improving efficiency.
- **Function-Specific Benchmarks:**
  Each function was profiled and benchmarked independently to accurately measure performance improvements and avoid cross-function interference.

These choices enabled a clean, modular implementation that delivered significant speedups while remaining scalable for embedded deployment on Jetson AGX Orin.

Chapter 3

Implementation Details

3.1 Specifications and System Architecture

The Neon Optimization framework is developed and tested on the NVIDIA Jetson AGX Orin platform to accelerate performance-critical functions in embedded vision systems. These functions include histogram calculation, grayscale conversion, Sobel edge detection, and box blurring. Each function is optimized using ARM NEON intrinsics to enable parallel processing and reduce execution time.

This system follows a modular software architecture where each function is profiled, optimized, and benchmarked independently. The design ensures flexibility, allowing easy integration of future functions and scalable performance analysis.

Hardware Specifications:

- **Jetson AGX Orin Dev Kit:**
  High-performance ARM Cortex-A78AE-based SoC with 12 CPU cores at 2.2 GHz, NEON SIMD support, and up to 64GB LPDDR5 memory.

Software Stack:

- C/C++ source code:

  Implements both baseline and NEON-optimized versions of each function using ARM intrinsics.

- GCC Compiler with NEON Flags:

  Uses -mfpu=neon or -march=armv8-a+simd flags for building optimized binaries.

- Benchmarking Scripts:

  Measure execution time and speedup for 1000 iterations of each function.

- Terminal/Console Output:

  Logs performance metrics, function outputs, and validation results.

3.2 Software Stack Details

The software stack for the Neon Optimization project consists of tightly integrated components developed in C/C++ and executed on the Jetson AGX Orin platform. The primary goal is to implement, optimize, and benchmark various image-processing functions using ARM NEON intrinsics.

Each function—such as histogram computation, grayscale conversion, Sobel edge detection, and box blurring—is written in two variants: a baseline scalar version and a NEON-optimized version using SIMD instructions. These are compiled with NEON support enabled and executed on the Jetson CPU.

For performance analysis, each function is profiled across 1000 iterations, and metrics such as execution time and speedup factor are logged. The results help compare the efficiency of NEON optimizations against traditional implementations.

Additionally, terminal outputs and optional log files are used to validate functional correctness and summarize benchmarking data for analysis

3.2.1 Histogram calculations

Histogram Calculations "CalcColorHistogramsFromStatsBlock()"

- Loading RGB averages and weights in parallel.
- Computing the maximum color intensity per window.
- Compressing values (like squeezing a range) using custom power logic.
- Rounding and clamping indices for histogram bins.
- Updating the histogram by accumulating weights based on intensity.

The NEON SIMD pipeline replaces 4 iterations of scalar math with 1 instruction, giving major speedups.

Figure 3.1: Linear data loading

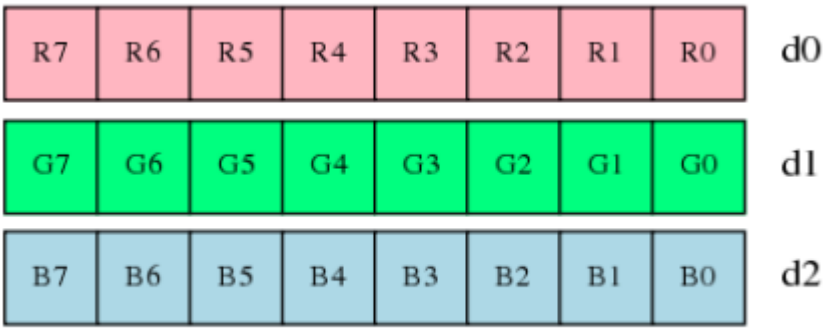

Figure 3.2: Structural data loading

```
for (UInt32 i = 0; i < ROI_NUM; i++)
{
    if (ROIEnable[i])
    {
        for (UInt32 j = 0; j < NumWindows[i]; j++)
        {
            // Inlined implementation of GetWeightTableIndexForStatsBlock
            UInt32 idx;
            if (roiWidth == 0)
            {
                idx = 0;
            }
            else
            {
                UInt32 wtIndexForRoi = roiWidth * (i % 2) + roiWidth * wtWidth * (i / 2);
                idx = wtIndexForRoi + j % roiWidth + wtWidth * (j / roiWidth);
            }

            const Float32 weight = weightTablePtr[idx];
            const Float32 r = pAverage_R[i * MAX_ROI_WINDOWS + j];
            const Float32 g = pAverage_G[i * MAX_ROI_WINDOWS + j];
            const Float32 b = pAverage_B[i * MAX_ROI_WINDOWS + j];
            const Float32 maxValueFloat =
                static_cast<Float32>(powf(MAX_OF_THREE(r,g,b), histogramCompressionPower) *
                    (HISTOGRAM_SIZE - 1));

            const UInt32 maxValueInt = static_cast<UInt32>(maxValueFloat + 0.5f);
            if (maxValueInt > (HISTOGRAM_SIZE - 1))
            {
                printf("Calculated histogram index exceeds limit\n");
            }

            histPtrOutput[maxValueInt] += weight;
            totalMass += weight;
        }
    }
}
```

Figure 3.3: Before optimization

```
// Process ROIs and accumulate histogram values
for (UInt32 i = 0; i < ROI_NUM; i++) {
    if (ROIEnable[i]) {
        for (UInt32 j = 0; j < NumWindows[i]; j += 4) { // Process in chunks of 4 (vectorized)
            UInt32 idx[4];
            for (int k = 0; k < 4; k++) {
                UInt32 wtIndexForRoi = roiWidth * (i % 2) + roiWidth * wtWidth * (i / 2);
                idx[k] = wtIndexForRoi + (j + k) % roiWidth + wtWidth * ((j + k) / roiWidth);
            }

            float32x4_t weights = vld1q_f32(&weightTablePtr[idx[0]]);
            float32x4_t r = vld1q_f32(&pAverage_R[i * MAX_ROI_WINDOWS + j]);
            float32x4_t g = vld1q_f32(&pAverage_G[i * MAX_ROI_WINDOWS + j]);
            float32x4_t b = vld1q_f32(&pAverage_B[i * MAX_ROI_WINDOWS + j]);


            // Compute maximum of r, g, b
            float32x4_t max_rg = vmaxq_f32(r, g);
            float32x4_t max_rgb = vmaxq_f32(max_rg, b);

            // Apply compression power
            float32x4_t max_value_float = vmulq_n_f32(
                vector_pow(max_rgb, histogramCompressionPower),
                (HISTOGRAM_SIZE - 1)
            );

            // Round to nearest integer and clamp
            uint32x4_t max_value_int = vcvtq_u32_f32(
                vaddq_f32(max_value_float, vdupq_n_f32(0.5f))
            );
            uint32x4_t clamped_indices = vminq_u32(
                max_value_int,
                vdupq_n_u32(HISTOGRAM_SIZE - 1)
            );

            // Update histogram bins
            for (int k = 0; k < 4; k++) {
                UInt32 index = vgetq_lane_u32(clamped_indices, k);
                histPtrOutput[index] += vgetq_lane_f32(weights, k);
                totalMass += vgetq_lane_f32(weights, k);
            }
        }
    }
}
```

Figure 3.4: After optimization

- All the possible operations involving load and store were optimized using **NEON** Intrinsics
- Loading and storing intrinsics:

vld1q_f32(&array[index]) //Loads 4 floats at once

vst1q_f32(result, [1.0, 2.0, 3.0, 4.0]); //Stores 4 floats at once

3.2.2 GrayScale Conversion

Convert RGB pixels to grayscale using:

GRAY=0.299·R+0.587·G+0.114·B

Neon optimization algorithm:

- Preload Weights as NEON Vectors
- Process Pixels in Chunks of 8
- Multiply-Accumulate and Downscale
- Stores 8 grayscale pixels at once

Figure 3.5: RGB image

Figure 3.6: Gray scale image

3.2.3 Box blurring

A box blur is a simple image filter where each output pixel is the average of a surrounding box of pixels in the input image.

The optimized function applies a box blur using ARM NEON SIMD intrinsics on grayscale image data (uint8_t values), processing 16 pixels at once horizontally.

Figure 3.7: Blurred image

3.2.4 Sobel filter for edge detection

- When we apply this mask on the image it prominent vertical edges. It simply works like as first order derivative and calculates the difference of pixel intensities in a edge region.

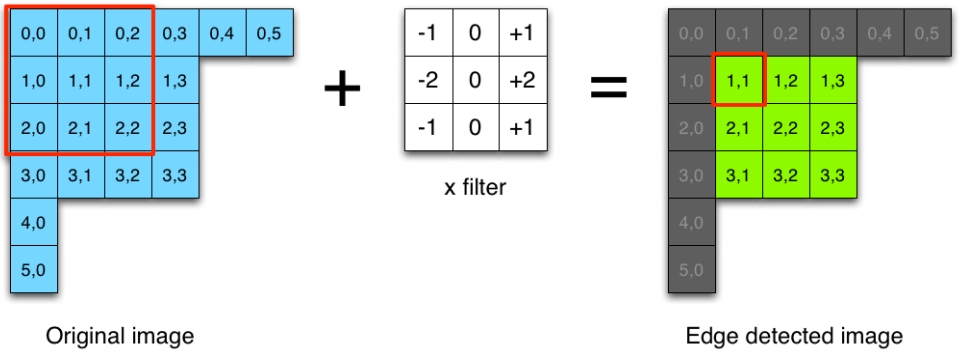

Figure 3.8: Sobel Filter

SobelNeon Function:

- It processes 8 pixels in parallel using NEON Registers (uint8x8_t)
- Each 3x3 neighbourhood around a pixel is loaded using vld1_u8

Figure 3.9: Edge detection

3.3 Algorithm

Jetson AGX:

- **Select the target image-processing function**
  (e.g., Histogram, Grayscale Conversion, Sobel Filter, or Box Blur).
- **Load the input data**
  (e.g., pixel array or statistical block) into memory.
- **Execute the baseline (non-NEON) version**
  and record the execution time.
- **Execute the NEON-optimized version**
  using ARM intrinsics with SIMD instructions.
- **Benchmark and compare**
  both implementations by calculating speedup:
  Speedup = Baseline Time / NEON Time.
- **Validate output accuracy**
  by comparing outputs from both versions for correctness.
- **Log the performance data**
  (execution time, speedup, memory usage, etc.) for reporting.

3.4 Flowchart

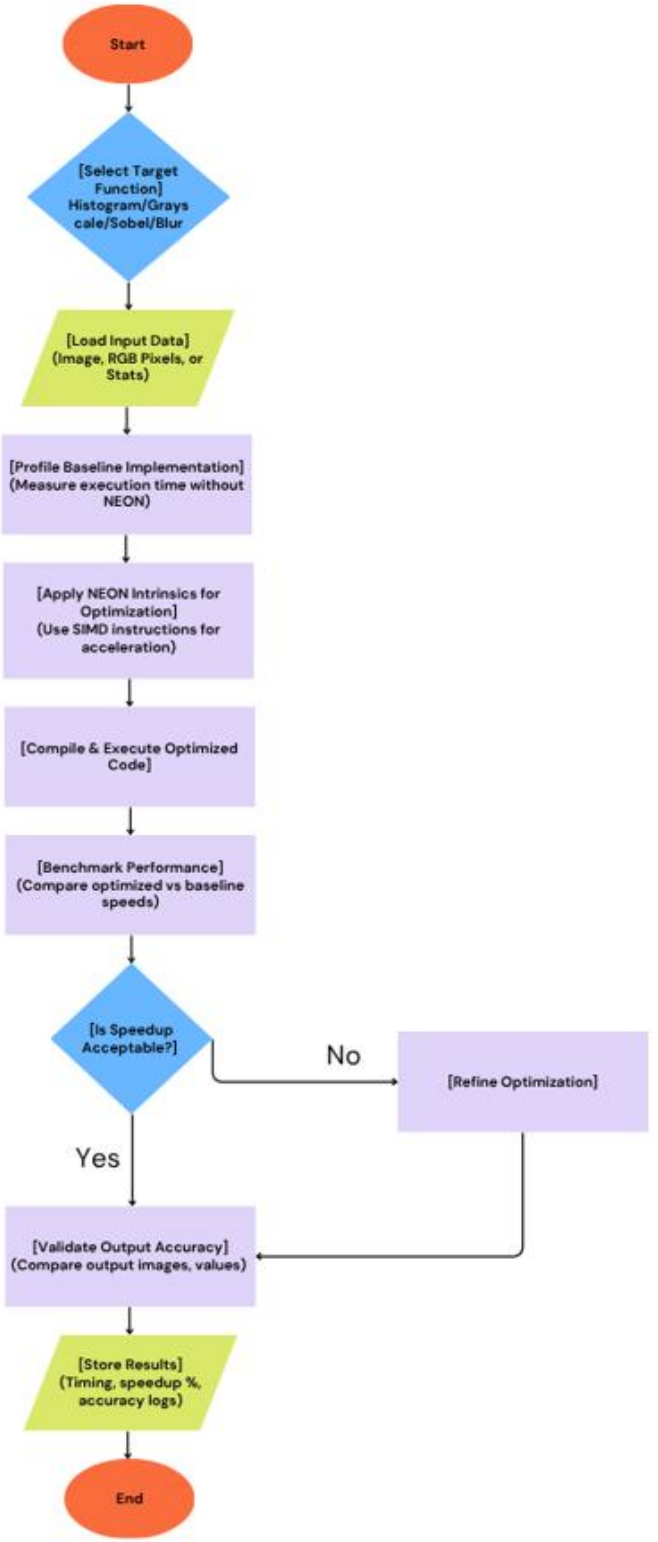

Figure 3.10: Flowchart

Chapter 4

Results and Discussions

4.1 Performance analysis

To evaluate the performance impact of NEON optimization, each target function—Histogram Calculation, Grayscale Conversion, Box Blurring, and Sobel Edge Detection—was implemented in both scalar (baseline) and NEON-intrinsic versions. The benchmarking was performed on the NVIDIA Jetson AGX Orin, which features a 12-core ARM Cortex-A78AE CPU with NEON SIMD support.

Each function was executed over **1000 iterations**, and the execution times were averaged for both the baseline and NEON-optimized versions. The **speedup factor** was calculated by comparing these two execution times, giving a quantitative measure of performance improvement.

Function	Baseline time (µs)	NEON Time (µs)	Speedup Factor
Histogram	127	86	1.5x
Grayscale	2919	1451	~2x
Box blurring	1040.73	287.49	~3.6x
Edge detection	19723	6592	~3x

Table 4.1: Performance analysis

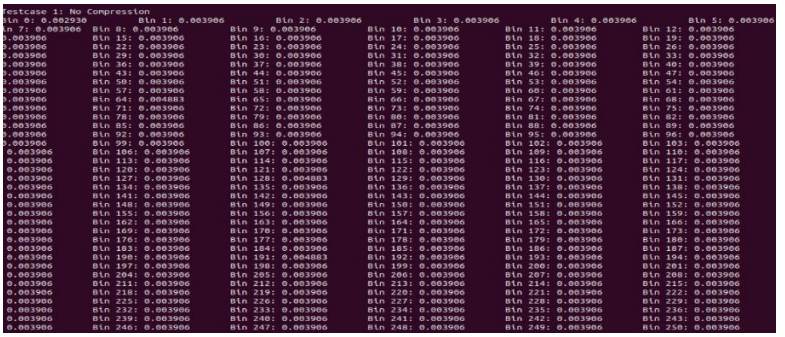

Figure 4.1: Statistical block data

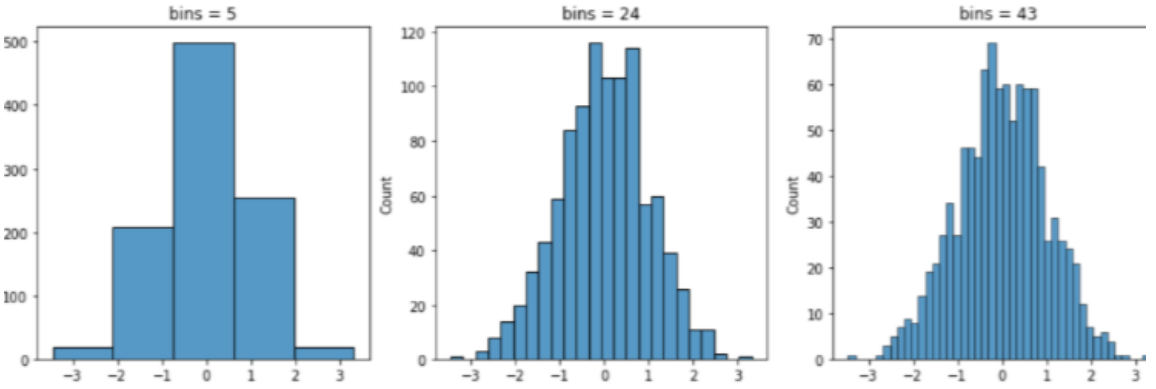

Figure 4.2: Histogram

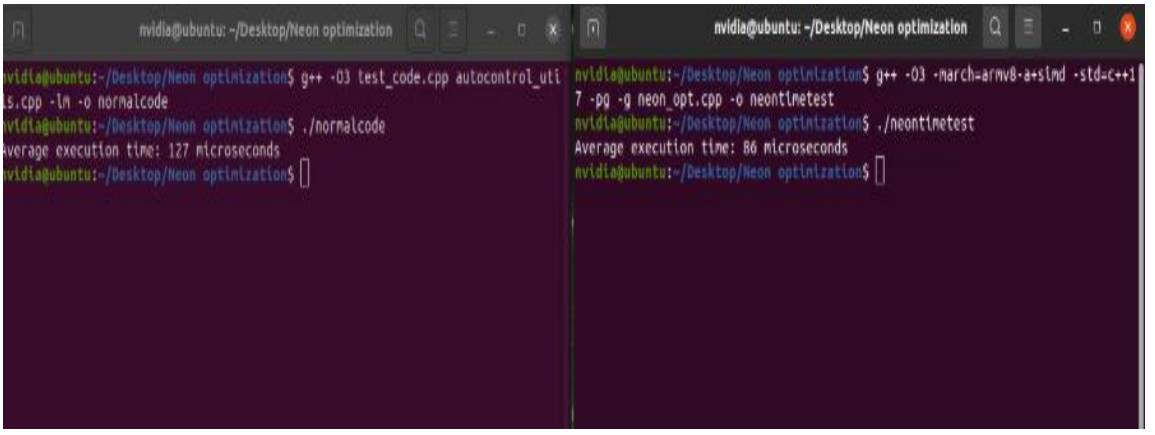

Figure 4.3: A snapshot of terminal

Chapter 5

Conclusions and Future Scope

5.1 Conclusion

The Neon Optimization of Functions from Autocontrol project demonstrates the effectiveness of using ARM NEON intrinsics to accelerate core image-processing operations in embedded vision systems. Implemented on the NVIDIA Jetson AGX Orin platform, the project focused on optimizing functions such as histogram calculation, grayscale conversion, Sobel edge detection, and box blurring—each critical for real-time perception in applications like driver-assistance systems. By leveraging SIMD parallelism, significant reductions in execution time were achieved, validating the potential of NEON for enhancing performance under strict latency and power constraints. The optimized implementations maintain functional accuracy while offering substantial speedup over traditional scalar code. This project provides a scalable and practical framework for embedding high-efficiency vision algorithms in resource-constrained environments, establishing a strong foundation for future work in real-time embedded optimization.

5.2 Future Scope

While the current implementation fulfils its core objectives, several improvements and extensions are possible:

• **Dynamic Optimization Selection:** Integrate a configuration interface to toggle between baseline and NEON-optimized functions at runtime for adaptive performance evaluation.

• **Support for Additional Functions:** Expand the optimization framework to include other vision-based operations like median filtering, dilation, erosion, and thresholding.

• **Automated Benchmarking Suite:** Develop a GUI or script-based tool for auto-profiling, logging, and visualizing speedup and performance metrics across multiple functions.

• **Cross-Platform NEON Portability:** Extend the implementation to support other NEON-capable platforms (e.g., Raspberry Pi 4, i.MX series) for broader applicability.

• **Integration with Real-Time Systems:** Incorporate NEON-optimized modules into a live ADAS testbench or ROS-based robotic platform to evaluate real-world latency benefits.