21VL713 – FPGA BASED SYSTEM DESIGN

SEMESTER II

M.Tech VLSI Design

Term Project Report

on

Real – Time Image Edge Detection on ZYNQ Ultra Scale + MPSOC ZCU104 Board

Submitted by,

M. Shanmukha Sri Datta(CB.EN.P2VLD24007)

Department of Electronics and Communication Engineering

Contents

List of Figures

1. Abstract

Real-time image processing has significantly contributed to the part of many embedded systems like self-driving cars, smart surveillance, medical tools, and factory automation. One of the most important things in image processing is edge detection that enables to determine object edges and mark significant details in a digital image. However, this is not a simple task to perform on embedded systems due to such factors as delays, resource limitations, and power restrictions. This paper is focused on the development of a real-time edge detection system by using CLAHE for contrast improvement and Canny edge detection, targeted at the Zynq UltraScale+ MPSoC (ZCU104) platform. By the way, for the sake of finding the best method, we initially implemented a number of edge detection techniques in Python and OpenCV, which are Sobel, Prewitt, Laplacian, and Canny. Once the outputs were demonstrated, the decision was made based on the visual quality, the computing power required, and the noise sensitivity, to take the path of CLAHE and Canny edge detection. At first, a pipeline was designed, and each step in the pipeline was tested on a computer by means of Google Colab as the development of the settings such as the clip limits for CLAHE and the thresholding for Canny was performed on the PC. When success was certain, we moved the system to the ZCU104 board. Also, as the set-up is the most vital part of the project. We first installed the PYNQ OS on the ARM processor and then, to provide a static IP address for the USB Ethernet connection, a new ethernet connection was set in the internet sharing properties of the laptop. For the live image, we chose a lightweight, software-only design and used a See3CAM_CU30 USB camera. Capturing real-time video and audio, the ARM Cortex-A53 cores were in charge of the ZCU104 system whereas in the previous phase, the FPGA was not accessed for the task. The method of comparison was the same as in the previous stage: we matched the FPGA-based results up with the computer-based ones. Evaluating standards such as the PSNR, SSIM, image sharpness, and finally, we measured the execution time, to ensure the accuracy of our tests. It is true that the PC was faster, but the FPGA system finished the tasks in almost the same time and the output was really similar to the visuals seen on the PC. The higher PSNR values and consistent SSIM scores showed that the embedded system worked well. This project proves that real-time image processing can be done efficiently on embedded FPGA platforms using just software optimization. It shows that lightweight computer vision tasks can run on resource-limited devices without needing complex hardware accelerators. In the future, we could expand this work to handle real-time video processing or explore ways to optimize both software and hardware together using tools like High-Level Synthesis (HLS).

2. Introduction

Embedded systems and artificial intelligence are being used more and more in modern applications. This has created a big demand for real-time image processing at the edge. For example, in fields like autonomous driving, smart surveillance, and medical imaging, it’s super important to quickly pull useful information from images. Edge detection is one of the key techniques in image processing. It helps find object boundaries, which is crucial for tasks like segmentation and tracking objects.

Traditional computing platforms are great at handling image processing because they have powerful resources. But when you try to move these tasks to embedded systems with limited resources, things get tricky. That’s where FPGAs come in. They’re a good choice for these tasks because they can handle multiple things at once, can be reprogrammed as needed, and don’t use much power. Take the Zynq UltraScale+ MPSoC (ZCU104), for instance. It combines programmable logic with a strong ARM-based Processing System (PS). This makes it easier to develop and test real-time image processing systems without running into resource problems. Most research so far has focused on speeding up edge detection using hardware implementations. However, these methods often require deep knowledge of hardware design and take a lot of time to develop. In this project, we tried something different. We showed that real-time performance can also be achieved by optimizing software pipelines, without needing hardware acceleration or HLS tools.
For this project, we built a real-time image processing pipeline using Python and OpenCV on the Zynq MPSoC's Processing System. The pipeline uses Contrast Limited Adaptive Histogram Equalization

(CLAHE) to improve image contrast. After that, Canny edge detection is applied to extract edges effectively. A USB camera connected to the FPGA board captures live images, and all the processing happens on the ARM Cortex-A53 cores. To evaluate the system, we looked at metrics like execution time, PSNR (Peak Signal-to-Noise Ratio), SSIM (Structural Similarity Index), and image sharpness. These results were compared to PC-based execution This project proves that even without dedicated hardware acceleration, efficient real-time image processing is possible on embedded systems. It shows there's a simpler and more development-friendly path for edge AI applications.

3. Literature Survey

In past years, the development seen in real time image processing system using FPGAs gained great significance due to huge demand in low latency embedded vision application. Many research works is being explored in area of image detection techniques on FPGA boards.

Anakha et al.[1] discussed in their paper about the implementation of basic image enhancement techniques such histogram equalization and contrast stretching in FPGAs and these highlighted the challenges over real time performance and advantage of using pipelined parallelism to accelerate simple and important image processing techniques. Priyanka et al.[2] implemented sobel edge detection algorithm on FPGA boards for grey scale image processing and emphasized the advantage of pipelining and parallel processing to get high speed processing which is important for embedded machine vision. The results also showed that how basic gradient operators can be effectively mapped to FPGA boards. Ye[3] implemented real time edge detection system which was targeted for AI applications using FPGA. The project also focused on how to develop optimized algorithms and architectures. This rreduces the computation and power consumption while ensuring accurate edge detection which are important for downstream AI inference task. Kumar and Shankar[4], first focused on AES encryption hardware accelerators which were provided vision ton how HLS can be applied to complex hardware design. This work supports methodology for image processing acceleration by showing how HLS can make easy the hardware implementation without coding in HDL.R Kumar et al.[5] proposed a methodology for enhanced edge detection for image segmentation. And this also demonstrated by implementing in FPGA. Their approach combined image processing techniques like contrast enhancement and canny edge detection to get better edge maps closely aligned with the methodology that has been developed by this project. Also, their finding tells FPGA boards can handle high processing pipelining with great efficiency. Jain and Dubey[6] developed an canny edge detection system which is been optimized for FPGA platforms. They have modified traditional canny stages to implement to hardware more efficient way and to reduce computation complex while maintaining the accuracy in detection. Their results show the compatability of deploying edge detection in complex environments.

Based upon these paper studies this project implements real time image edge detection using Zynq Ultra Scale + MPSoC ZCU104 board. Unlike the above works which were primarily developing hardware accelerators for basic edge operators, our work focused on integrating CLAHE based canny edge detection in python running it in ARM cortex A53 processor. Instead of working with available datasets real time image capturing using USB camera was achieved. Performance metrics like execution time, PSNR, SSIM, and sharpness were evaluated and compared against PC execution, providing a complete performance study. This project demonstrates that even without custom hardware acceleration, efficient real-time image processing

is achievable on embedded FPGA platforms by optimizing the software pipeline, alternative approach. Thus, while prior works focused heavily on accelerating basic edge operations through hardware design, our project demonstrates that an optimized software pipeline combined with contrast enhancement techniques can also meet real-time embedded vision goals effectively on Zynq platforms.

4. Methodology

The methodology for this project was designed to provide a clear and efficient way to implement real-time edge detection on an embedded platform. We started by studying and selecting suitable image processing algorithms. From there, we moved on to software simulation, hardware setup, and finally, real-time deployment. Each step was carefully planned to test the design at both the PC and FPGA levels. This ensured that we could measure and analyze performance accurately.

A key focus was on using lightweight software-based processing. The goal was to show that real-time embedded vision is possible without relying on complex hardware acceleration. Below, we break down the entire implementation process, from the initial design to the final performance evaluation.

Algorithm Study and Selection

At the start, we conducted a detailed study of edge detection algorithms to figure out which method would work best for real-time embedded implementation. We first tested common techniques like Sobel, Prewitt, and Laplacian operators because they're simple and don't require much computational power. However, these basic methods turned out to be too sensitive to noise and produced edges that were often blurry.

On the other hand, the Canny edge detector performed much better. It produced precise and thin edges thanks to its multi-step process, which includes reducing noise, calculating gradients, suppressing non-maximum values, and applying hysteresis thresholding. This made it a clear choice for our project.

To make finer image details more visible before running edge detection, we added Contrast Limited Adaptive Histogram Equalization (CLAHE). CLAHE improves local contrast without amplifying noise too much. After testing the outputs and tweaking parameters on sample images, we decided to use a combination of CLAHE followed by Canny edge detection for the rest of the project.

Python Simulation in PC

Before moving to the embedded platform, we developed and tested the entire image processing pipeline on a PC using Python and OpenCV libraries. This step was crucial to make sure the algorithms we chose were correct and to fine-tune parameters like CLAHE's clip limit, grid size, and the thresholds for Canny edge detection. The software pipeline included the following steps:

- Grayscale Conversion: The images were converted to grayscale to simplify the data and focus on intensity information.
- CLAHE Application: We applied CLAHE with carefully chosen parameters to improve local contrast across the image without over-amplifying noise.
- Edge Detection: We used the Canny edge detector, adjusting the lower and upper thresholds to strike a balance between detecting edges accurately and rejecting noise.
- Visualization and Saving Results: We visualized and saved intermediate and final results for analysis and compared with different metrices.

We also measured execution time and observed the quality of the output images. This helped us set a benchmark for when we moved the implementation to the embedded platform.

Hardware setup and ZCU104 board configuration

For the board deployment, we used the Zynq UltraScale+ MPSoC ZCU104 evaluation board. Setting it up involved several important steps:

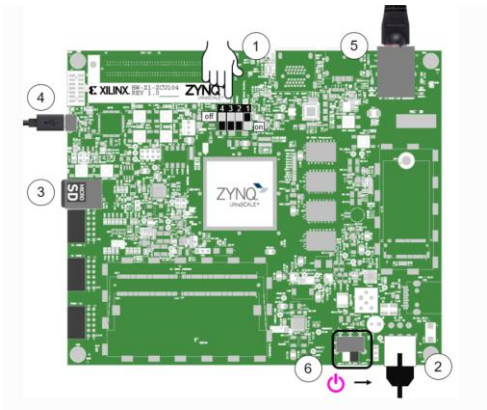

Figure 1 ZCU104 Board Configuration

Steps we followed are given below:

- Set the Boot Dip Switches (SW6) to the 1000
- Connect the 12V power cable
- Insert the Micro SD card loaded with the appropriate PYNQ image into the MicroSD card slot underneath the board
- Connect the USB cable to your PC/Laptop
- Connect the Ethernet port
- Turn on the board and check the boot sequence

We flashed PYNQ OS onto an SD card to set up the system. PYNQ OS is a lightweight Linux distribution that's optimized for controlling FPGAs using Python. Next, we connected the board to the host PC (a laptop) using a USB Ethernet (RNDIS) connection. This allowed us to communicate with the board easily. Finally, we manually configured the IP address to ensure everything was set up correctly for communication between the board and the laptop.

IP address configuration was done manually:

- Laptop: IP assigned as 192.168.2.1
- Board: IP configured as 192.168.2.99 using the command:

  sudo ifconfig usb0 192.168.2.99 netmask 255.255.255.0 up // IP configuration command

We enabled internet sharing on the laptop to give the board access to the internet. This was important for installing necessary packages on the board.To make sure everything was working, we tested the connection by pinging the board's IP address. Once the ping was successful, we knew the setup was good to go.With this network configuration in place, we were able to easily access the board's Jupyter server and Linux shell without any issues.

Camera integration and verification

To enable real-time input, we integrated a See3CAM_CU30 USB 3.0 camera with the board.The camera was plugged into one of the USB ports on the ZCU104. To make sure the camera was detected properly, we checked its connection using:

```
ls /dev/video*       // for confirming camera detection
```

Detailed camera properties were examined using :

```
v4l2-ctl--device=/dev/video0 –all       // to examine resolution format frame rate
```

To address compatibility and ease of testing fs webcam utility was installed:

```
sudo apt install fsewebcam      // installing fs webcam
```

We performed test captures to make sure the camera was working properly and could deliver stable frames.Once everything checked out, we set up OpenCV to capture frames directly from the camera device.

Deployment of image processing algorithm on ZYNQ board

The image processing pipeline that we validated in the PC environment was moved over to the PYNQ board. The steps we followed were:

- Capturing Live Frames: We used OpenCV's VideoCapture API to grab live frames from the USB camera.
- Grayscale Conversion: We converted the images to grayscale to simplify the data and focus on intensity information.
- Applying CLAHE: We applied CLAHE to improve the local contrast of the grayscale frames.
- Canny Edge Detection: We ran Canny edge detection with carefully tuned thresholds to get clean and accurate edges.
- Saving and Displaying Results: We saved intermediate outputs and displayed the results through Jupyter notebooks for analysis.
- All the processing happened on the board's ARM Cortex-A53 Processing System (PS) cores. We didn't use any Programmable Logic (PL) acceleration, keeping the system entirely software-based

Performance metrices and evaluation

After successfully deploying the system, we evaluated its performance by measuring a few key metrics:

- Execution Time:We measured how long it took to complete the entire processing pipeline, including capturing frames, enhancing the images, detecting edges, and saving the results.
- Sharpness:To check how clear the image details were, we calculated sharpness using the variance of the Laplacian method. This gave us a good idea of the image's clarity.
- PSNR (Peak Signal-to-Noise Ratio):We measured the PSNR between the original and processed images. This helped us evaluate how well the contrast was improved while preserving the overall image quality.
- SSIM (Structural Similarity Index):We compared the structural similarity of the images before and after processing. This allowed us to assess how well the perceptual quality of the images was preserved.
- To gather comparable data, we ran separate experiments on both the PC and the FPGA.

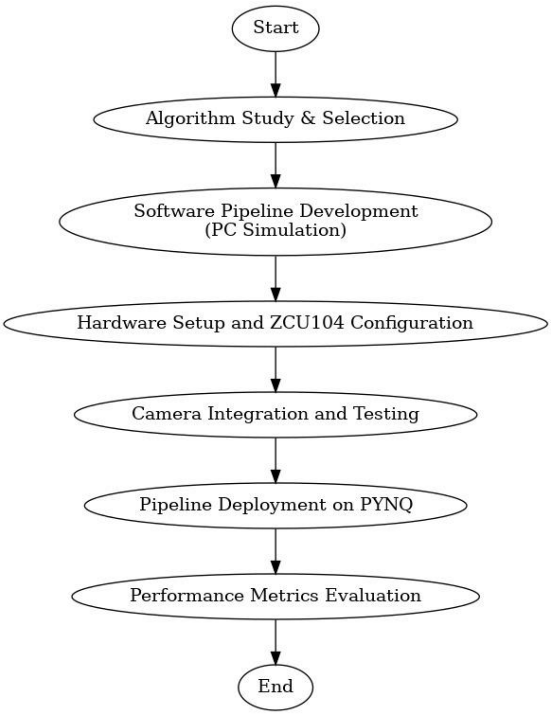

Figure 2 Methodology flowchart

The project follows a structured and step-by-step approach, as shown in the flowchart. It begins with studying and selecting the best edge detection algorithms for real-time processing on an embedded platform, ensuring the right technique is chosen for the job. Next, the image processing pipeline is developed and tested on a PC using Python and OpenCV to validate the design before moving to the embedded system. Once the PC simulation is successful, the focus shifts to setting up the ZCU104 board, including configuring network connectivity and ensuring the system is ready for use. After that, a USB camera is integrated and tested to enable real-time image capture, which is crucial for smooth live input. With the hardware prepared, the complete software pipeline is deployed on the PYNQ board, where live images are captured and processed in real time. Finally, the system's performance is evaluated using key metrics like execution time, PSNR, SSIM, and sharpness, confirming that the embedded solution is both effective and feasible.

5. Codes

Python code for comparative analysis of edge detection techniques

```
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.metrics import structural_similarity as ssim

# Load the image in both color (for visualization) and grayscale (for processing)
image_path = "/content/flower.jpg"          # Path to input image
image_color = cv2.imread(image_path)     # Load in color
image_color = cv2.cvtColor(image_color, cv2.COLOR_BGR2RGB)  # Convert BGR (OpenCV default) to RGB
image_gray = cv2.imread(image_path, cv2.IMREAD_GRAYSCALE)  # Load in grayscale for edge detection
```

```
# Apply Global Histogram Equalization to enhance contrast
equ = cv2.equalizeHist(image_gray)

# Apply CLAHE for local contrast enhancement (adaptive histogram equalization)
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
clahe_img = clahe.apply(image_gray)

# Apply Gaussian Blur to reduce noise before edge detection
gaussian_equ = cv2.GaussianBlur(equ, (5, 5), 0)
gaussian_clahe = cv2.GaussianBlur(clahe_img, (5, 5), 0)

# Apply Bilateral Filter to reduce noise while preserving edges
bilateral_equ = cv2.bilateralFilter(equ, 9, 75, 75)
bilateral_clahe = cv2.bilateralFilter(clahe_img, 9, 75, 75)

# Apply Canny edge detection on all processed images
canny_original = cv2.Canny(image_gray, 100, 200)
canny_equ = cv2.Canny(equ, 100, 200)
canny_equ_gaussian = cv2.Canny(gaussian_equ, 100, 200)
canny_equ_bilateral = cv2.Canny(bilateral_equ, 100, 200)
canny_clahe = cv2.Canny(clahe_img, 100, 200)
canny_clahe_gaussian = cv2.Canny(gaussian_clahe, 100, 200)
canny_clahe_bilateral = cv2.Canny(bilateral_clahe, 100, 200)

# Function to calculate image sharpness using the variance of the Laplacian
def edge_sharpness(image):
    return cv2.Laplacian(image, cv2.CV_64F).var()

# Function to calculate PSNR (Peak Signal-to-Noise Ratio) for quality assessment
def psnr(original, processed):
    mse = np.mean((original- processed) ** 2)
    if mse == 0:
        return float("inf")  # Perfect similarity
    return 20 * np.log10(255.0 / np.sqrt(mse))

# Function to calculate Structural Similarity Index (SSIM) between two images
def calculate_ssim(original, processed):
    return ssim(original, processed, data_range=processed.max()- processed.min())

# Evaluate sharpness, PSNR, and SSIM for each method
metrics = {
    "Original": (edge_sharpness(canny_original), psnr(image_gray, canny_original), calculate_ssim(canny_original, canny_original)),
    "EQU + Canny": (edge_sharpness(canny_equ), psnr(image_gray, canny_equ), calculate_ssim(canny_original, canny_equ)),
    "EQU + Gaussian + Canny": (edge_sharpness(canny_equ_gaussian), psnr(image_gray, canny_equ_gaussian), calculate_ssim(canny_original, canny_equ_gaussian)),
    "EQU + Bilateral + Canny": (edge_sharpness(canny_equ_bilateral), psnr(image_gray,
```

```
canny_equ_bilateral), calculate_ssim(canny_original, canny_equ_bilateral)),
  "CLAHE + Canny": (edge_sharpness(canny_clahe), psnr(image_gray, canny_clahe), calculate_ssim(canny_original, canny_clahe)),
  "CLAHE + Gaussian + Canny": (edge_sharpness(canny_clahe_gaussian), psnr(image_gray, canny_clahe_gaussian), calculate_ssim(canny_original, canny_clahe_gaussian)),
  "CLAHE + Bilateral + Canny": (edge_sharpness(canny_clahe_bilateral), psnr(image_gray, canny_clahe_bilateral), calculate_ssim(canny_original, canny_clahe_bilateral))
}

# Print the computed metrics in a tabular format
print(f"{'Method':<30}{'Sharpness':<15}{'PSNR':<15}{'SSIM':<15}")
for method, values in metrics.items():
  print(f"{method:<30}{values[0]:<15.2f}{values[1]:<15.2f}{values[2]:<15.4f}")

# Plotting original and processed edge-detected images
fig, axes = plt.subplots(2, 4, figsize=(15, 8))

# Original color image
axes[0, 0].imshow(image_color)
axes[0, 0].set_title("Original Color Image")
axes[0, 0].axis("off")

# Display each processed image with corresponding title
axes[0, 1].imshow(canny_original, cmap='gray')
axes[0, 1].set_title("Canny on Original")
axes[0, 1].axis("off")

axes[0, 2].imshow(canny_equ, cmap='gray')
axes[0, 2].set_title("EQU + Canny")
axes[0, 2].axis("off")

axes[0, 3].imshow(canny_equ_gaussian, cmap='gray')
axes[0, 3].set_title("EQU + Gaussian + Canny")
axes[0, 3].axis("off")

axes[1, 0].imshow(canny_equ_bilateral, cmap='gray')
axes[1, 0].set_title("EQU + Bilateral + Canny")
axes[1, 0].axis("off")

axes[1, 1].imshow(canny_clahe, cmap='gray')
axes[1, 1].set_title("CLAHE + Canny")
axes[1, 1].axis("off")

axes[1, 2].imshow(canny_clahe_gaussian, cmap='gray')
axes[1, 2].set_title("CLAHE + Gaussian + Canny")
axes[1, 2].axis("off")

axes[1, 3].imshow(canny_clahe_bilateral, cmap='gray')
```

```
axes[1, 3].set_title("CLAHE + Bilateral + Canny")
axes[1, 3].axis("off")

# Adjust layout to avoid overlapping
plt.tight_layout()
plt.show()
```

The above program uses a variety of edge detection methods how different edge detection algorithms can be used with a grayscale image after preprocessing the image in different ways. The Canny edge detector is the final stage of preprocessing. As the first action, the system takes the source of the image in a fully colored and only grey one. The direct clarity improvement of the images (two contrast enhancement techniques—the Histogram Equalization and CLAHE) is done. After the contrast improvement, the images are blurred using the Gaussian blur and Bilateral filtering noise reduction methods. At the end of the preprocessing phase, the Canny edge detector is run on every processed image. The quality and similarity of the results are evaluated and compared by using three metrics: quality (computing the variance of Laplacian), Peak Signal-to-Noise Ratio (PSNR), and Structural Similarity Index (SSIM).

Python Code for Real Time Image Edge Detection:

```
# Set DNS server for internet access in some environments
!echo "nameserver 8.8.8.8" | sudo tee /etc/resolv.conf > /dev/null

# Display the contents of /etc/resolv.conf to confirm DNS configuration
!cat /etc/resolv.conf

# Install required Python packages for image processing and plotting (user-level installation)
!pip3 install opencv-python numpy matplotlib --user

# Import OpenCV for image capture and processing
import cv2

# Open the default camera (usually /dev/video0)
cap = cv2.VideoCapture(0)

# Check if camera was successfully opened
if cap.isOpened():
    print("Camera is detected and opened successfully.")
else:
    print("Camera not detected. Check connection.")

# Release the camera resource
cap.release()

# Re-import required libraries
import cv2
import numpy as np
import matplotlib.pyplot as plt
```

```
# Re-open the camera for capturing an image
cap = cv2.VideoCapture(0)

# Check if the camera opened successfully
if not cap.isOpened():
    print("Failed to open /dev/video0.")
else:
    print("Camera opened!")

    # Set frame width and height
    cap.set(cv2.CAP_PROP_FRAME_WIDTH, 640)
    cap.set(cv2.CAP_PROP_FRAME_HEIGHT, 480)

    # Capture a single frame from the camera
    ret, frame = cap.read()

    # Release the camera
    cap.release()

    # Check if frame was successfully captured
    if not ret:
        print("Failed to capture image.")
    else:
        print("Image captured successfully!")

        # Save the captured image to a file
        cv2.imwrite("captured_original.jpg", frame)

        # Convert BGR (OpenCV default) to RGB for correct display in matplotlib
        rgb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)

        # Display the captured image using matplotlib
        plt.imshow(rgb)
        plt.title("Captured Image")
        plt.axis("off")
        plt.show()

# Import necessary libraries again for further processing
import cv2
import numpy as np
import matplotlib.pyplot as plt
from skimage.metrics import structural_similarity as ssim  # For SSIM calculation
import time

# Start a timer to measure processing time
start_time = time.time()

# Load the captured image in color
image_color = cv2.imread("captured_original.jpg")
```

```
# Convert from BGR to RGB for visualization
image_color = cv2.cvtColor(image_color, cv2.COLOR_BGR2RGB)

# Convert the RGB image to grayscale
image_gray = cv2.cvtColor(image_color, cv2.COLOR_RGB2GRAY)

# Create CLAHE object for contrast enhancement
clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))

# Apply CLAHE to grayscale image
enhanced = clahe.apply(image_gray)

# Apply Canny edge detection to the enhanced image
edges = cv2.Canny(enhanced, 100, 200)

# Save the edge-detected image
cv2.imwrite("captured_edges.jpg", edges)

# Stop the timer
end_time = time.time()

# Calculate and display total processing time
processing_time = end_time - start_time
print(f"Total Processing Time: {processing_time:.4f} seconds")

# Function to calculate sharpness using variance of Laplacian
def calculate_sharpness(image):
    laplacian = cv2.Laplacian(image, cv2.CV_64F)
    sharpness = laplacian.var()
    return sharpness

# Calculate sharpness of original grayscale image
sharpness_original = calculate_sharpness(image_gray)

# Calculate sharpness of CLAHE enhanced image
sharpness_enhanced = calculate_sharpness(enhanced)

# Display both sharpness values
print(f"Sharpness (Original): {sharpness_original:.4f}")
print(f"Sharpness (CLAHE Enhanced): {sharpness_enhanced:.4f}")

# Calculate Structural Similarity Index between original and enhanced image
ssim_value = ssim(image_gray, enhanced)
print(f"SSIM (Original vs Enhanced): {ssim_value:.4f}")

# Function to calculate Peak Signal-to-Noise Ratio (PSNR)
def calculate_psnr(original, processed):
    mse = np.mean((original - processed) ** 2)  # Mean Squared Error
    if mse == 0:
        return float('inf')  # Avoid division by zero
    max_pixel = 255.0
```

```
    psnr = 20 * np.log10(max_pixel / np.sqrt(mse))
    return psnr

# Calculate PSNR value
psnr_value = calculate_psnr(image_gray, enhanced)
print(f"PSNR (Original vs Enhanced): {psnr_value:.2f} dB")

# Create a 1x4 subplot to compare all images
plt.figure(figsize=(15, 5))


# Display original color image
plt.subplot(1, 4, 1)
plt.imshow(image_color)
plt.title("Original (Color)")
plt.axis("off")

# Display original grayscale image
plt.subplot(1, 4, 2)
plt.imshow(image_gray, cmap='gray')
plt.title("Original (Grayscale)")
plt.axis("off")

# Display CLAHE enhanced grayscale image
plt.subplot(1, 4, 3)
plt.imshow(enhanced, cmap='gray')
plt.title("CLAHE Enhanced")
plt.axis("off")

# Display edges detected using Canny after CLAHE
plt.subplot(1, 4, 4)
plt.imshow(edges, cmap='gray')
plt.title("Enhanced Canny Edge Detection")
plt.axis("off")

# Adjust layout to prevent overlap
plt.tight_layout()

# Show the combined plot
plt.show()
```

The above program realizes a complete image processing workflow by means of Python and OpenCV. It begins with capturing a live image from the FS webcam. Once the photo is taken, the image is saved. After that, the picture is converted to gray scaling, and CLAHE (Contrast Limited Adaptive Histogram Equalization) is used to improve the local contrast of the image. Next, Canny edge detection is used for extracting prominent features from the enhanced image, and the resulting image is saved as a new file. Apart from the enhancement itself, the majority of the improvement is determined by the so-called quality factors. The script calculates not only sharpness (using the variance of the Laplacian) but also displays how SSIM (Structural Similarity Index Measure) can be utilized to compare two images by checking their perceptual similarity and how PSNR (Peak Signal-to-Noise Ratio) can determine the quality level of the image. At the same time, the total processing time is

also measured to understand the performance. Lastly, for easy visual comparison, all stages of the image will be represented side by side, o.a., the original color version, the grayscale image, the CLAHE enhanced version, and the edge-detected image, using subplots. The whole process can be considered as the foundation for automatic image preprocessing that can be deployed in the real world. This will be beneficial for, e.g., computer vision and medical imaging where real-time image processing could be very helpful.

6. Results

PC based Simulation:

The developed pipeline was initially tested on a laptop using Google Colab (Python + OpenCV) to validate the functionality and the results has been provided below:

Figure 3 Outputs from PC

Through various metrics we have compared the effectiveness of the obtained results and the below table provides the details for it.

Method	Sharpness	PSNR (dB)	SSIM
Original	35,062.39	27.68	1.0000
EQU + Canny	56,896.78	27.68	0.6490
EQU + Gaussian + Canny	38,362.82	27.68	0.6877
EQU + Bilateral + Canny	33,865.52	27.68	0.7093
CLAHE + Canny	70,214.64	27.67	0.6376

CLAHE + Gaussian + Canny	44,047.05	27.68	0.7422
CLAHE + Bilateral + Canny	36,261.13	27.67	0.7733

The above table depicts how distinct compositions of contrast improvement and edge detection techniques are evaluated with reference to three measures: Sharpness, PSNR (Peak Signal-to-Noise Ratio), and SSIM (Structural Similarity Index). The initial image had an SSIM of 1.0000 as it was set as the starting point. A different story is always the case with just Equalization (EQU), where we can observe the rise of the sharpness and a significant drop in the SSIM. This evident that the structural information is now gone. The addition of smoothing filters like Gaussian or Bilateral again pushed the SSIM up, meaning the edges were properly preserved, even though slightly resulting in lower sharpness values. A jump to CLAHE instead of EQU brought forth a drastic increase in the sharpness; this goes to show the fact that CLAHE is much faster at enhancing the details after contrast improvement. Nevertheless, if we perform a combination of CLAHE + Canny without any smoothing filters, the SSIM remains at the bottom. However, if we mix CLAHE with smoothing filters like Gaussian or Bilateral, the SSIM increases even more. The top-performing combination of the three is the one involving CLAHE + Bilateral + Canny, which attains the highest structural similarity (SSIM = 0.7733) compared to all other pipelines, while not losing too much sharpness. The CLAHE + Canny combination was the best option for his project out of all the compared methods. It was a decision that we made a few days back. One, CLAHE is often used for enhancing the quality of the image (70,214.64). We found it better than other similar methods at producing sharpness, and this was the main point. The method also increases the contrast of the image and reveals features that are more visible. These are the features of the image that enable the correct edge to be identified. It is right that the simple CLAHE + Canny method has a little lower SSIM than the versions that use some preprocessing such as Gaussian smoothing. However, besides that, the small change in SSIM, these filters are also heavy to process and consume resources. It's impractical for real-time systems with limited resources to choose an approach like this. Consequently, we went on with CLAHE + Canny as it is moderate. It helped us achieve the objective of the feature histogram of an image without increasing the computational complexity dramatically. The performance would also be enough in meeting the requirements of a lightweight, real-time system on ZCU104, in terms of resources

Hardware Output:

The below output represents the image taken from the USB camera integrated to FPGA after implementing on board. It shows the original, gray scale, clahe enhanced mage and enhanced detection converted image

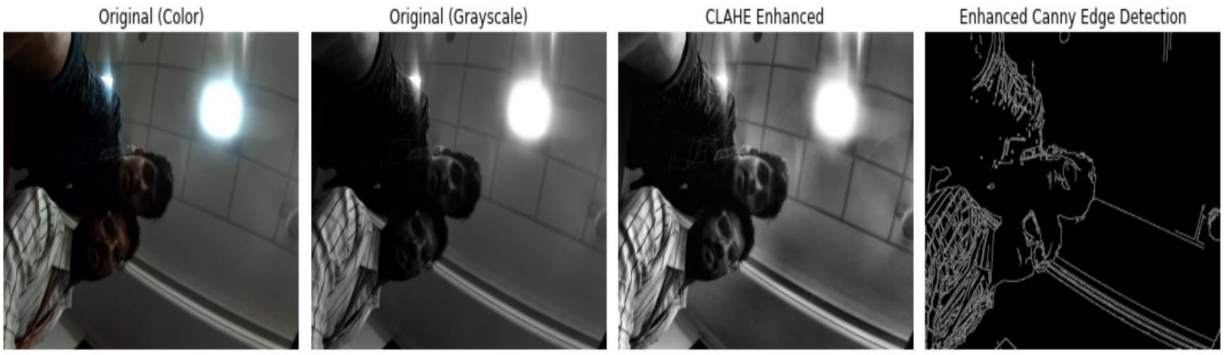

Figure 4 Hardware image output

Overall hardware setup:

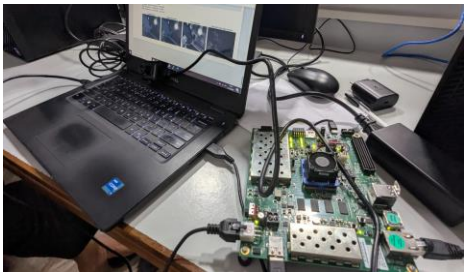


Figure 5 Hardware setup

Real Time edge detection:

And here is the orginal image that has been taken form USB webcame and the canny edge detection processed image from the FPGA.

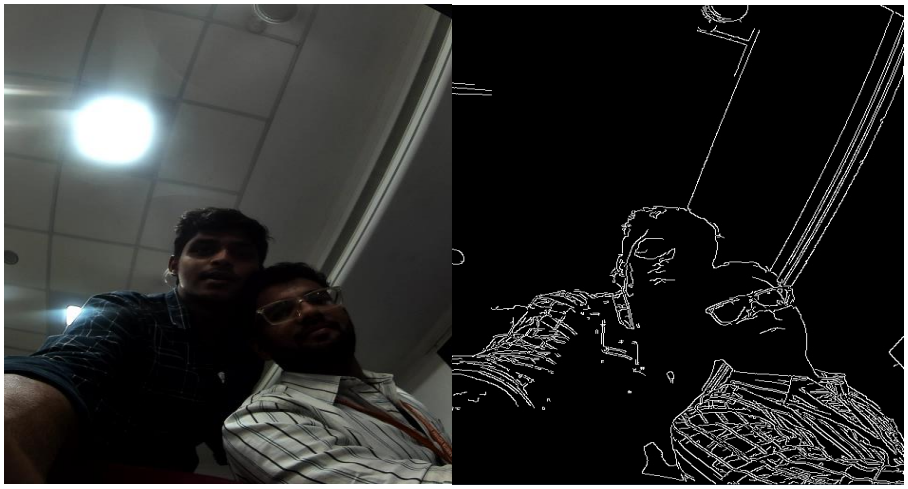

Figure 6 Edge detection output from baord

Performance comparison between results obtained through FPGA and PC

The below table shows the performance comparsion of canny edge detection in FPGA and PC where we used google colab for simualation,

Metric	ZYNQ (ZCU104)	PC (Google Colab)
Execution Time (seconds)	0.0278 sec	0.019 sec
Sharpness (Original)	400.0167	400.3663
Sharpness (CLAHE Enhanced)	915.1854	916.2435

PSNR (Original vs Enhanced)	29.10 dB	20.20 dB
SSIM (Original vs Enhanced)	0.7986	0.7985

The table above is a comparison of the primary performance metrics of the PC-based simulation (using Google Colab) and the FPGA-based implementation (on the ZYNQ ZCU104). The PC reached the finish line first in the case of theexecution time; there were only 0.019 seconds compared to 0.0278 seconds needed
on the FPGA board. That's quite logical because computers have higher clockspeeds and also more resources which are achieved using their general-purposeprocessors. Yet, the speed decrease in the FPGA is not a big deal so the system still stays close to the actual time which is quite good. With regard to sharpness, all the values for both the original and CLAHE-enhanced images are very much alike on both platforms. This means that the contrast enhancement has been achieved in a consistent way and the quality is not lost. It turns out that the PSNR figure of the FPGA-based output (29.10 dB) outperformed the PC-based result (20.20 dB), which is quite an interesting fact. This may lead to the conclusion that compared to the personal computer, the other platform is better at preserving image quality in the process of contrast enhancement. What is more, the two SSIM figures are almost equal (~0.798), which is indicative of the fact that the original and processed images keep their structural similarity unchanged when passed through two different systems.

First off, the FPGA-based setup allows us to do real-time image processing at the edge without any outside resources or cloud servers, thereby making the system so independent that it imposes shorter delays, which is very crucial for many applications. In addition to that, the flexibility of FPGAs is so astonishing. They are also compatible with parallelism, hence, in case we need to improve the performance, we can always introduce a user-designed hardware accelerator. Knowing that the variant is always there, if needed, is really reassuring. At the same time, boards like the ZCU104 are not power-hungry, which is a great solution for mobile or battery-powered systems. If battery-operated machines are among your list of devices, having the power consumption low is one of the must-haves. Additionally, we opted for a very simple move: we just ran the entire pipeline on the Processing System (PS) without making use of the Programmable Logic (PL). This way, we were able to not get involved in the complexities entailed
in the programmable logic section and yet demonstrate the real-time embedded vision existing in air. Overall, FPGA-based platforms just seem perfect for situations where you want not only lightweight but also energy-saving image processing. Revise the last sentence from the first point to create an emphatic sentencing conveying that an FPGA is the best platform for on-the-fly image processing in the real world.

7. Conclusion

The goal of this project was to create a system that detects edges in images in real-time on an embedded FPGA platform. We wanted to show that this could be done using simple software without needing custom hardware acceleration. We used CLAHE to improve image contrast and Canny edge detection to find object boundaries. The system worked well, producing high-quality results while staying fast enough for real-time use. First, we tested different edge detection methods like Sobel, Prewitt, Laplacian, and Canny. After comparing them based on sharpness, noise handling, and how much computing power they need, we chose CLAHE with Canny edge detection. We spent time tweaking parameters on a PC using Google Colab to make sure the algorithms worked well under different lighting and image quality conditions. Once everything worked on the PC, we moved the system to the Zynq UltraScale+ MPSoC (ZCU104) platform. All the processing ran on the ARM Cortex-A53 cores, and we didn't use the Programmable Logic (PL) fabric. Setting up the hardware, connecting the camera, and configuring the network took some effort, but we got it working. We used a See3CAM_CU30 USB camera to capture live images. To check how well the system performed, we looked at metrics like execution time, PSNR, SSIM, and image sharpness. As

expected, the PC was faster because it has more computing power. But the FPGA implementation still handled tasks in real-time with similar output quality. Interestingly, the FPGA version had slightly better PSNR values, meaning it preserved image quality better during processing.

Overall, this project shows that real-time edge detection can be done on embedded FPGA platforms using just software optimization. This approach makes development simpler, uses fewer resources, and allows flexible deployment in real-world applications like surveillance, self-driving cars, and medical imaging. In the future, we might consider broadening this work to accommodate real-time
video processing or memory and resource usage optimization. There may also be possibilities of hardware acceleration by using High-Level Synthesis (HLS) for performance improvement.

8. References

[1] Anakha, M. S., D. Sanal, and S. Sethukrishnan. 2024. "Implementation of Image Enhancement Techniques on FPGA." In *Proceedings of the IEEE International Conference on Smart Electronics and Communication Systems (ISENSE)*.

[2] Priyanka, V., Y. Sri Rama, K. Sravani, and B. Kavya. 2024. "Implementation of Sobel Edge Detection with Image Processing on FPGA." In *Proceedings of the 2nd World Conference on Communication & Computing (WCONF)*, Raipur, India, July.

[3] Ye, C. 2022. "Real-time Image Edge Detection System Design and Algorithms for Artificial Intelligence FPGAs." In *Proceedings of the International Conference on Artificial Intelligence of Things and Crowdsensing (AIoTCs)*, 476–481.

[4] Kumar, S. S., and B. R. Shankar. 2023. "Efficient Hardware Accelerator Design for AES Encryption Using High-Level Synthesis Techniques." *IEEE Access* 11: 123456–123465.

[5] Kumar, R., A. R. Sahu, and M. Agrawal. 2022. "Enhanced Edge Detection for Image Segmentation and its Real-Time Implementation," *International Journal for Research in Applied Science and Engineering Technology (IJRASET)* 10 (5): 1200–1205.

[6] Jain, K., and A. Dubey. 2020. "FPGA-based Improved Canny Edge Detection System," *International Journal of Engineering Research & Technology (IJERT)* 9 (2): 150–154.

[7] L. J. J R and J. V. R. S P, "Enhanced Edge Detection for Image Segmentation and its Real-Time Implementation," *in Proc. 2024 28th International Symposium on VLSI Design and Test (VDAT)*, Vellore, India, 2024, pp. 1-6.