SSE PRACTICE SYNTHESIS

Final Report

Course Code: EIEN6711P
Group: FPGA Image Processing-2
Teacher Name: Dr. Ding Qing
Email: raoha@mail.ustc.edu.cn

中国科学技术大学
University of Science and Technology of China

Design and Implementation of a Hardware-Accelerated Image Filtering System using Zynq FPGA

Date: 2025 June 30

Submitted by:

Student Name: RAOHA BIN MEJBA (李一含)
Student ID: SL24225002

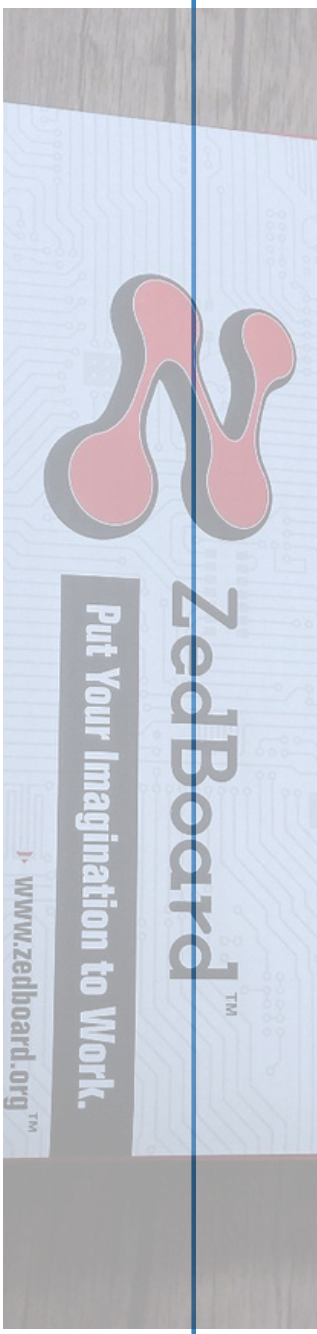
Student Name: MARJAN UR RAHMAN REMO (雷漠)
Student ID: SL24225005

Student Name: MD HEZBULLAH (李龙)
Student ID: SL24225009

摘要

本项目基于 ZedBoard Zynq-7000 FPGA 平台，设计并实现了一套实时图像处理系统。该系统通过一个名为 multi_filter 的统一硬件 IP 核，对灰度图像依次应用五种常见滤波算法：均值模糊、边缘检测、锐化、高通滤波和低通滤波。该 IP 核使用 Vitis HLS 开发，并通过 Vivado 集成至 Zynq 处理系统和 AXI DMA 总线中。在 ARM 处理器上运行的裸机 C 应用程序负责主机与 FPGA 之间的 UART 通信和 DMA 数据传输控制。

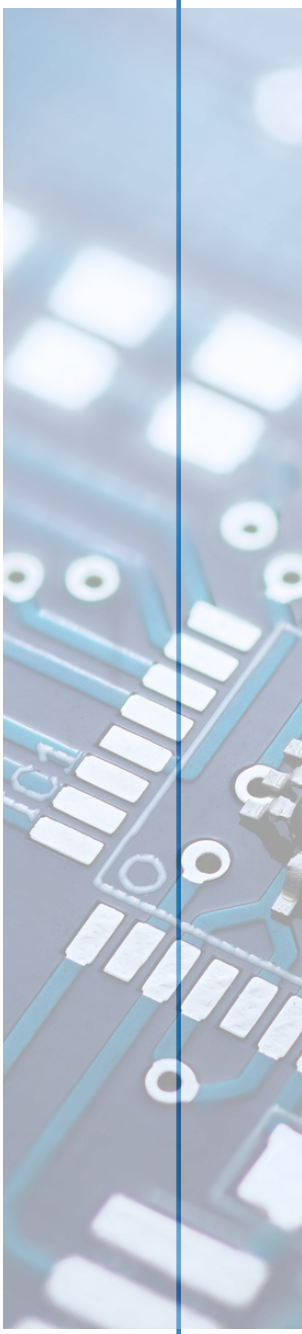
为了提升系统的可用性，我们采用 Python 和 Tkinter 构建了图形化用户界面（GUI），用户可通过界面上传图像、查看滤波结果并保存处理后的图像文件。系统整体体现了软硬件协同设计的高效性，既发挥了 FPGA 的并行处理优势，又兼顾了界面的易用性。通过仿真测试、串口日志输出和图像结果验证，系统各项功能运行稳定，滤波效果准确可靠。本项目充分展示了基于 FPGA 的图像处理系统的实用性，并为后续扩展，如视频处理、彩色图像支持和实时显示，奠定了良好的基础。



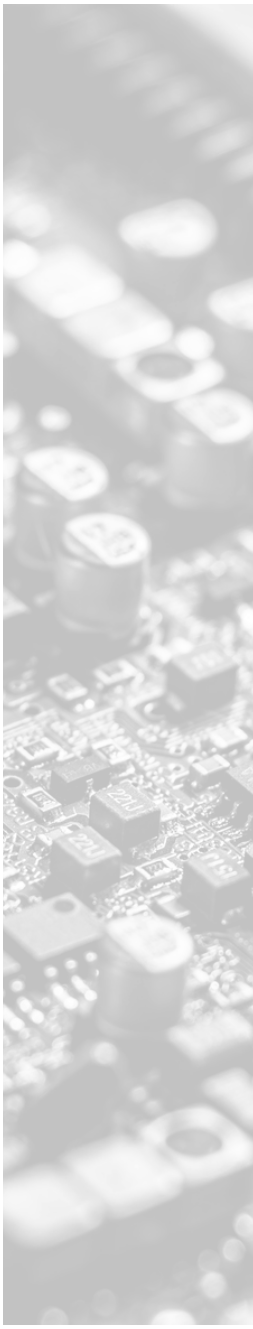
Abstract

This project presents the design and implementation of a **real-time image processing system using the ZedBoard Zynq-7000** FPGA platform. The system applies five common filters; **Box Blur, Edge Detection, Unsharp Masking, High Pass, and Low Pass** to grayscale images through a unified hardware IP core named **multi_filter**. The IP was developed using Vitis HLS and integrated with the Zynq Processing System and AXI DMA via Vivado. A bare-metal C application running on the ARM processor manages UART communication and DMA transfers between the host and the custom IP.

To ensure usability, a Python-based GUI was developed using Tkinter. The interface allows users to upload images, view the filtered outputs in a structured layout, and save outputs to their local device. The complete system demonstrates efficient hardware-software co-design, leveraging FPGA acceleration for performance and GUI integration for ease of use. The system was validated through simulation, UART logs, and visual output analysis, confirming the accuracy of each filter. This project highlights the practicality of FPGA-based image processing and provides a scalable foundation for future enhancements such as video filtering, color image support, and real-time display.



Contents



INTRODUCTION

Image processing is a field that deals with analyzing and manipulating images using computational techniques. It plays a central role in many real-world applications such as medical diagnostics, industrial inspection, satellite imaging, machine vision, biometric identification, and video surveillance. As image quality and resolution increase, the volume of data to be processed also becomes very large. This creates a strong need for systems that can perform image processing operations in real-time, without delay.

Traditionally, most image processing tasks are carried out using software on general-purpose processors. While this method works for many cases, it becomes inefficient when the task requires high-speed or real-time performance. Software-based solutions usually follow a sequential execution model and are often limited by the processor's clock speed and memory bandwidth. As a result, they struggle to meet the needs of real-time systems, particularly in embedded or edge-computing environments.

To overcome these limitations, hardware-based image processing has gained popularity. Among various hardware options, Field Programmable Gate Arrays (FPGAs) are widely used due to their reconfigurable nature, parallelism, and high computational power. FPGAs allow developers to implement custom processing pipelines tailored to specific tasks. This is especially useful for image processing, where many operations such as convolution, edge detection, and filtering can be executed in parallel, significantly increasing the speed of computation.

In this project, we have developed a complete FPGA-based image processing system using the **ZedBoard Zynq-7000** development board. The ZedBoard is built on the **Xilinx Zynq-7000 SoC (System on Chip)**, which integrates a dual-core ARM Cortex-A9 Processing System (PS) with programmable logic (PL) on the same chip. This combination allows us to design both the control logic in software and the high-speed processing functions in hardware, leading to a flexible and efficient co-design system.

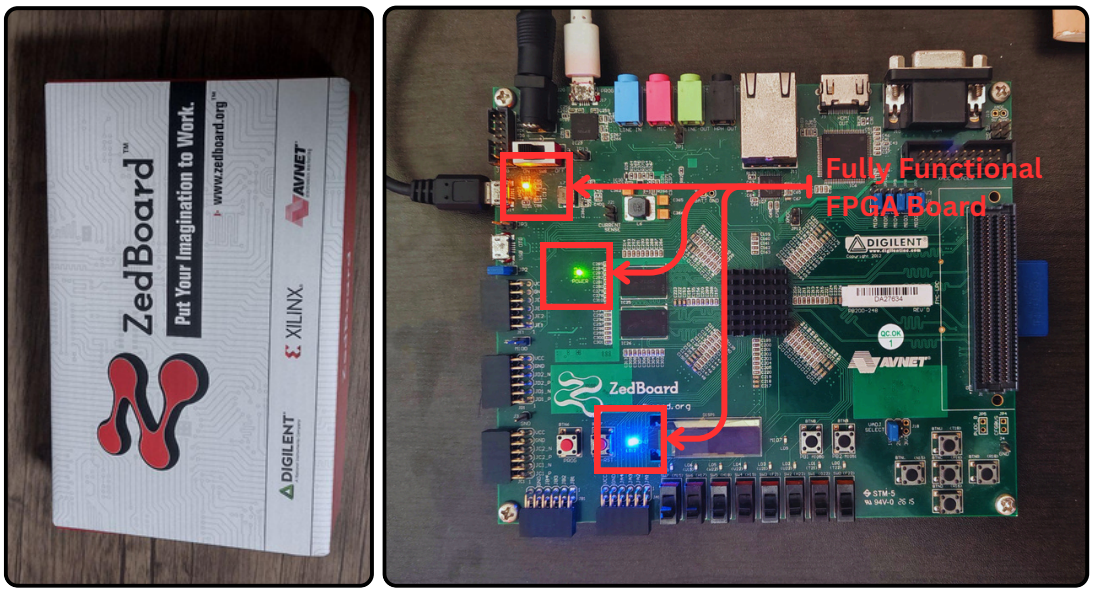


Fig 1: ZedBoard Zynq-7000 Development Kit used in the project.

The primary goal of our project is to apply five different image filtering operations on grayscale images using FPGA hardware acceleration. The filters implemented in our system are:

1. **Box Blur**
2. **Edge Detection (using Sobel operator)**
3. **Unsharp Masking**
4. **High Pass Filtering**
5. **Low Pass Filtering**

These filters are commonly used in image preprocessing and enhancement tasks. Box Blur is a basic smoothing filter; Edge Detection highlights boundaries; Unsharp Masking sharpens images by enhancing edges; and the High and Low Pass Filters control image frequency content. In our design, all these filters are implemented inside a single custom hardware IP, which we named **multi_filter**. This IP core is developed using **Vivado High-Level Synthesis (HLS)**, where the behavior is described in C/C++ and then converted into synthesizable HDL code.

The multi_filter IP core is connected to the rest of the system using AXI4-Stream interfaces, which allow high-throughput communication between modules. We use AXI Direct Memory Access (DMA) to transfer image data between DDR memory and the IP core. The Processing System (PS) in the Zynq SoC is responsible for controlling the data flow, setting up the DMA engine, and handling UART communication with the host PC.

To make the system more accessible and user-friendly, we have also developed a graphical user interface (GUI) using **Python** and **Tkinter**. The GUI allows the user to upload an input image, which is automatically converted to grayscale and resized to **128×128 pixels**. This image is then sent over a UART serial connection to the ZedBoard. After processing, the FPGA sends back five filtered images, one for each filter. These output images are displayed in a grid layout within the GUI. The GUI also includes options to save the processed images and select the COM port used for communication.

The overall system can be divided into four major parts:

1. **Hardware Platform Setup:** Using Vivado, we created a block design with ZYNQ7 Processing System, AXI SmartConnect, AXI DMA, and our custom multi_filter IP. The design was synthesized, implemented, and exported with the bitstream.
2. **HLS Design of Image Filters:** We developed the multi_filter IP using Vitis HLS. This IP takes in a grayscale image and applies all five filters in sequence, outputting five separate images.
3. **Software Application:** Using Vitis IDE, we developed a bare-metal C application that runs on the ARM Processing System. This application sets up UART and DMA peripherals, sends the input image to the IP core, and receives the filtered outputs.
4. **Python GUI Development:** A GUI application using Python was built to allow simple interaction with the hardware. It performs image preprocessing, sends and receives data, and shows the final result.

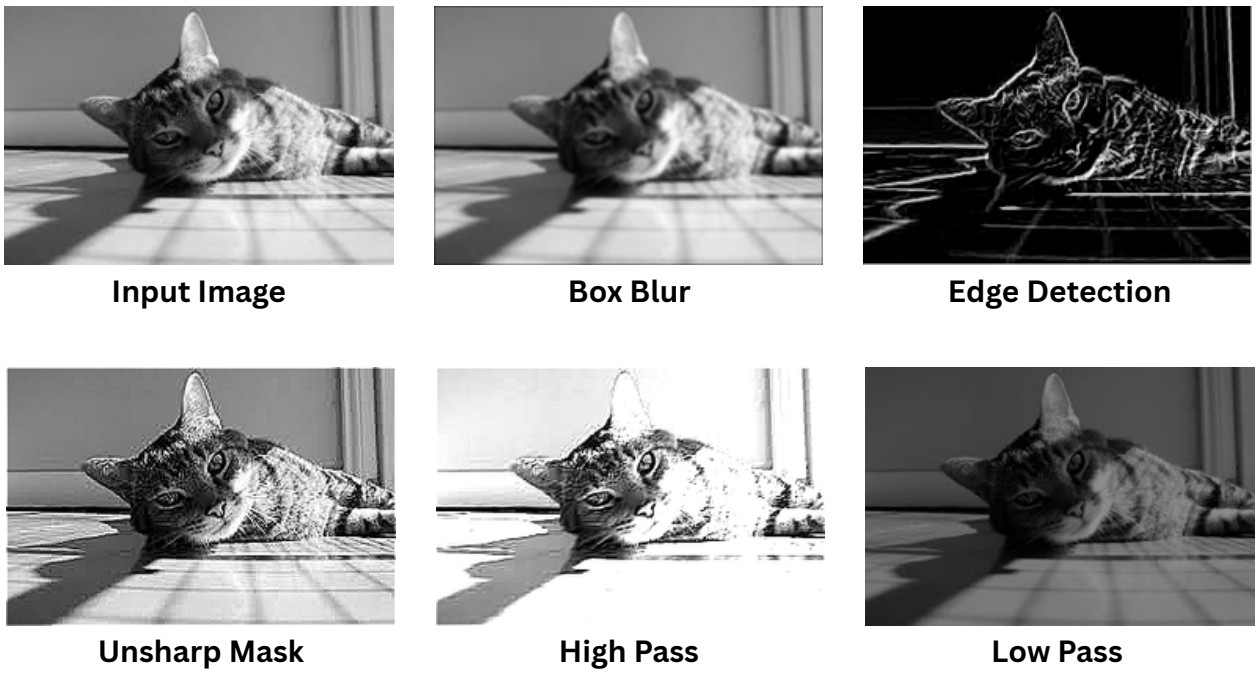


Fig 2: Example of input image and five filtered outputs Image we get from the GUI App.

This project showcases the benefits of using FPGA for image processing. One of the most important advantages is the **speed gained by hardware parallelism**. Instead of processing each pixel one by one in software, the FPGA processes data in a stream, allowing multiple pixels to be handled in parallel. This leads to faster processing even on low-resolution images like 128×128, and the design can be scaled to larger resolutions in the future.

Another key advantage is **modularity**. Our unified IP can be reused for other filters or extended to support color images. The software part is also modular and can be adapted for new user features like real-time webcam input or saving result logs.

Moreover, the system is highly portable. It can run on any Zynq-7000 compatible development board with minor adjustments. The use of a GUI makes the project accessible even to those who do not have technical knowledge of FPGAs or embedded systems.

In summary, this project represents a successful example of **hardware-software co-design applied to real-time image processing**. It demonstrates how FPGA hardware acceleration can be combined with a software-based user interface to build an efficient and interactive application. The design methodology followed in this project ensures clarity, modularity, and scalability. It provides a solid foundation for future projects involving real-time video processing, neural network acceleration, or smart camera systems.

TOOLS AND TECHNOLOGIES USED

This project involved both hardware and software development tools to design, implement, and test a real-time image processing system. The tools were selected based on compatibility with the ZedBoard Zynq-7000 platform and the ability to support hardware-software co-design. This section explains all the tools and technologies used in detail, along with their specific purposes in the project.

2.1 Hardware Platform

ZedBoard Zynq-7000 Development Kit: The ZedBoard Zynq-7000 is a development board manufactured by Digilent and based on the **Xilinx Zynq-7000 SoC (System on Chip)**. This SoC combines a **dual-core ARM Cortex-A9 Processing System (PS)** with programmable logic (PL) on a single chip. It allows users to develop both hardware and software applications in one unified environment.

We selected the ZedBoard for this project because of its **strong support for image processing, ease of integration with Vivado tools, and availability of AXI interfaces**. It provides multiple I/O options including UART, Ethernet, USB, and HDMI, making it suitable for high-speed data transfer and embedded development.

2.2 Software Tools

Xilinx Vivado Design Suite (2024.2): Vivado is the official FPGA design environment provided by Xilinx. It is used for designing digital logic circuits, configuring IP blocks, and generating the bitstream for FPGA programming.

In our project, Vivado was used to:

- Create a block design consisting of the Zynq PS, AXI SmartConnect, AXI DMA, and custom HLS IP.
- Configure the AXI interfaces and clock signals.
- Generate the bitstream file required to program the FPGA.
- Export the hardware specification (.xsa file) for Vitis.

We used **Vivado 2024.2**, which is compatible with our Zynq-7000 board and provides full support for modern AXI-based designs.

Vivado HLS (Vitis HLS 2024.2): Vivado HLS (now known as Vitis HLS) is a tool used to convert high-level C/C++ code into HDL (Hardware Description Language). It allows hardware designers to describe complex logic in C and generate synthesizable RTL for FPGAs.

We used **Vitis HLS** to design our unified image processing IP, named **multi_filter**. This IP applies five different filters sequentially. The IP was synthesized, validated using testbenches, and packaged for integration in Vivado.

Xilinx Vitis IDE (2024.2): Vitis is Xilinx's unified software development environment for embedded applications. It supports bare-metal, FreeRTOS, and Linux software development on Zynq SoCs.

We used Vitis IDE to:

- Import the Vivado-generated hardware platform (.xsa file).
- Create a bare-metal C application that runs on the ARM Cortex-A9 processor.
- Initialize UART and DMA peripherals.
- Control the IP block and manage image transfer between PC and FPGA.

2.3 Programming Languages and Libraries

C Language (Bare-metal Application): We used the C programming language to write the embedded application for the Processing System. The application runs without an operating system and directly manages peripheral drivers and IP configuration. It handles UART initialization, DMA transactions, and filter execution commands.

Python (GUI Development): The user interface was developed using Python due to its simplicity and strong support for GUI frameworks and image processing libraries.

Tkinter: Tkinter is the standard GUI toolkit for Python. It was used to design the layout of the application, including buttons for image upload, execution, and saving outputs. Tkinter also allowed us to display multiple images in a grid format.

Pillow: Pillow (Python Imaging Library) was used to load, convert, and resize images. We resized all input images to 128×128 and converted them to grayscale before sending to the FPGA. Pillow was also used to open and save the received filtered images.

PySerial: PySerial is a Python library that enables serial communication over UART. It was used in our backend to send image data to the FPGA and receive processed images back. The communication was established at a **baud rate of 115200 bps over COM4**.

2.4 Communication Interface

UART (Universal Asynchronous Receiver/Transmitter): UART is a serial communication protocol used to transmit data between two devices. In our system, UART is used to establish a connection between the host computer and the Zynq Processing System. We **mapped UART1 to MIO pins 48 and 49** on the board and used a USB-to-serial cable to connect to the PC.

On the PC side, the Python GUI uses PySerial to open the COM port and send the image data. On the FPGA side, the C application receives the data and triggers the IP for processing.

2.5 Data Format and Image Handling

All images processed in this project are grayscale BMP files resized to 128×128 pixels. Each pixel is represented using **8 bits**, resulting in a total of **16,384 bytes** per image. Since five output images are generated, the system handles **81,920 bytes in total per full processing cycle**.

The image format and size were chosen to balance between processing speed and data transfer time over UART.

2.6 Summary Table

Category	Tool / Language	Purpose
FPGA Design	Vivado 2024.2	Block design, bitstream generation
IP Development	Vitis HLS 2024.2	Custom filter IP design
Embedded Software	Vitis IDE (C Language)	UART, DMA, IP control
GUI Development	Python (Tkinter, Pillow, PySerial)	User interface, image display, UART communication
Communication Interface	UART (115200 bps, COM4)	Host ↔ FPGA data transfer
Image Format	128×128, 8-bit grayscale BMP	Fast processing and transfer

This combination of tools and technologies allowed us to design a reliable, fast, and user-friendly image processing platform. Each component was carefully selected to ensure seamless integration between the hardware and software layers of the system.

SYSTEM OVERVIEW

This project is designed using a hardware-software co-design approach. The system is divided into two main parts: **a hardware-based filtering engine built using FPGA and a software-based graphical user interface (GUI) developed in Python.** The filtering engine performs five types of image processing tasks using a unified custom IP core, while the GUI handles user interaction, image upload, data transfer, and output visualization.

The system uses the ZedBoard Zynq-7000 platform, which integrates a dual-core ARM Processing System (PS) and programmable logic (PL) in a single chip. This allows seamless communication between software and hardware components.

This section gives a complete overview of the working pipeline, including data flow, hardware-software communication, and internal architecture.

3.1 High-Level System Workflow

The following steps explain the full operation of the system from start to finish:

1.Image Upload: The user opens the Python-based GUI and uploads an image file. The image is converted to grayscale and resized to 128×128 pixels.

2. Data Transmission to FPGA: The image is converted to raw byte format and sent to the FPGA using UART serial communication. The Python backend handles this process.

3. Image Processing in Hardware: The FPGA receives the image data and stores it in DDR memory. The ARM processor initializes the DMA controller and triggers the custom multi_filter IP to process the image.

4. Filter Execution: The IP core applies all five filters sequentially (Box Blur, Edge Detection, Unsharp Masking, High Pass, and Low Pass) and writes the output results back to DDR memory using DMA.

5. Receiving Results: After processing, the ARM processor sends the filtered images back to the host PC using UART.

6. Displaying Output: The GUI receives and decodes the filtered images. It displays all five output images in a grid layout and allows the user to save the results.

3.2 Hardware-Software Partitioning

The system is divided into the following components:

1. Software (Host Side – Python GUI):

- Provides a user interface to upload and display images.
- Sends the image bytes to the FPGA via UART.
- Receives processed image data and visualizes it.
- Provides options to save filtered results.

2. Hardware (ZedBoard Zynq-7000):

- Zynq PS (Processing System):
  - Manages UART communication with the host PC.
  - Controls the AXI DMA controller for memory-to-stream and stream-to-memory transfers.
  - Sends commands and data to the custom IP and receives the processed results.
- AXI DMA Controller:
  - Transfers image data between DDR memory and the custom IP core using AXI4-Stream protocol.
  - Operates in simple mode (Scatter-Gather disabled) for ease of use.
  - Supports both MM2S (memory-to-stream) and S2MM (stream-to-memory) directions.
- Custom HLS IP (multi_filter):
  - Processes one 128×128 grayscale image and applies all five filters.
  - Output includes five filtered images, each stored in separate buffers.

3.3 Internal Hardware Design (Block Diagram)

The hardware design includes the following connections:

- Zynq PS is connected to AXI SmartConnect.
- AXI SmartConnect connects the PS to the AXI DMA.
- AXI DMA is connected to the multi_filter IP core using AXI4-Stream interfaces.
- All modules share the same clock and reset lines from FCLK_CLK0 and FCLK_RESETN.

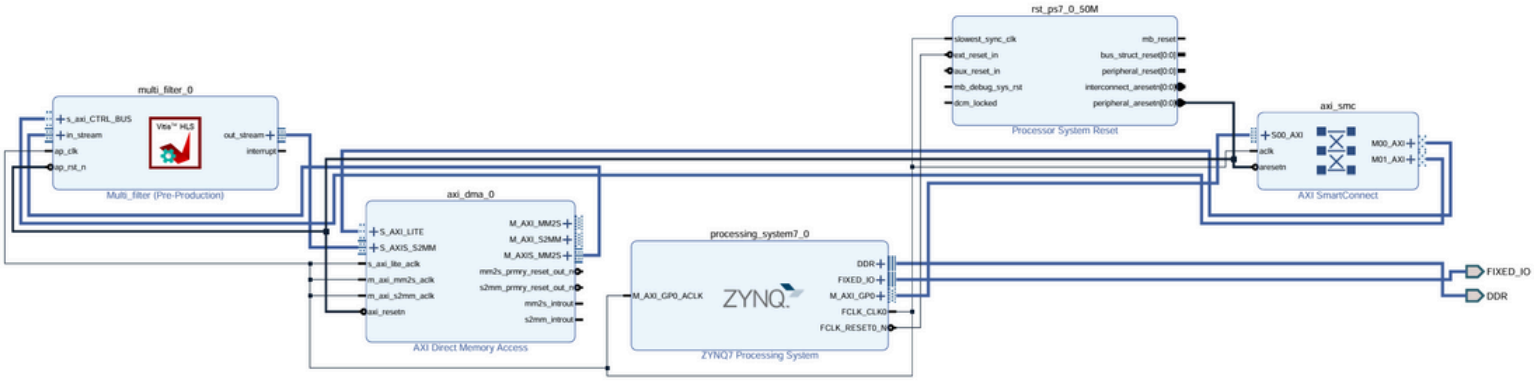


Fig 3: Block diagram of the hardware design in Vivado. This figure shows the Zynq PS, AXI DMA, SmartConnect, and multi_filter IP, along with their connections.

3.4 Data Flow and Buffer Handling

To ensure fast and smooth operation, we use DMA to transfer data between the Zynq PS and the custom IP core. DMA allows large blocks of image data to be moved without processor involvement, which improves speed and efficiency.

- The input image is first stored in DDR memory.
- AXI DMA reads this image and streams it to the multi_filter IP.
- The IP processes the image and streams back five outputs.
- These outputs are stored in memory and then sent to the PC.

Each image (input or output) is represented using 16,384 bytes (128×128 pixels). Since there are five output images, the system handles a total of 81,920 bytes in one cycle.

3.5 Communication Architecture

The following interfaces are used in the system:

- UART:
  - Used for communication between PC and ZedBoard.
  - Operates at a baud rate of 115200.
  - Mapped to MIO pins 48 and 49.
- AXI4-Stream:
  - Used for fast data transfer between AXI DMA and the multi_filter IP.
  - Allows continuous streaming of pixel data.
- AXI4-Lite (Internal IP Control):
  - Used for configuration and control registers (if needed in future versions).

3.6 Control Logic Flow

The overall flow of operations on the FPGA side is managed using simple control logic:

1. UART receives image bytes and stores them in DDR.
2. DMA MM2S is configured to send the image to the IP.
3. IP processes the image and streams five filtered results.
4. DMA S2MM stores each result in memory.
5. After all results are available, they are sent back over UART.

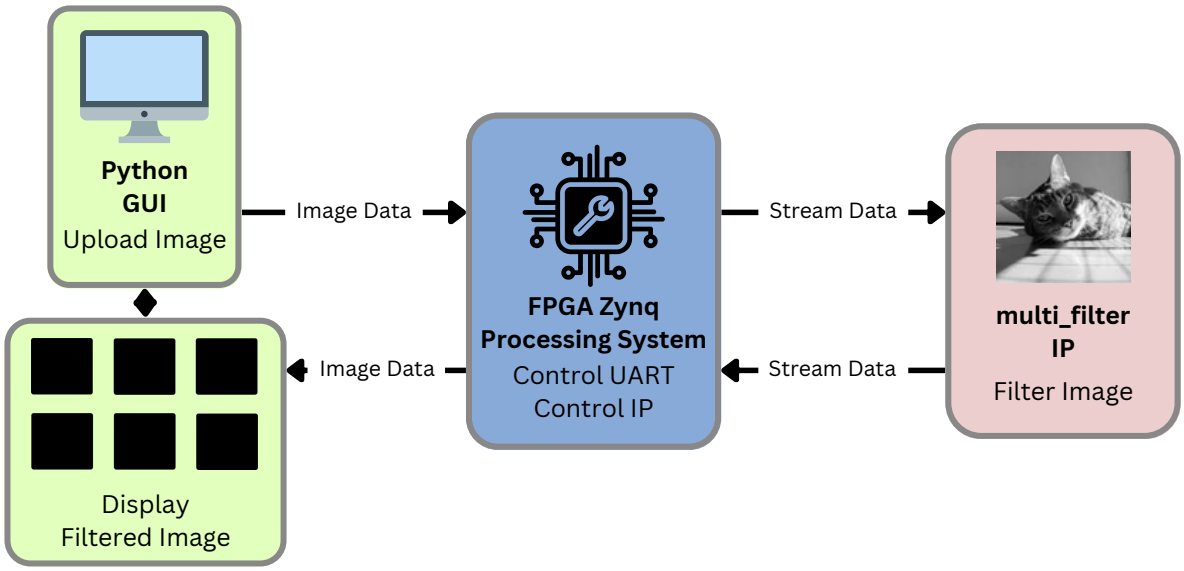


Fig 4: General data flow between Python GUI, Zynq PS, DMA, and multi_filter IP.

HARDWARE DESIGN METHODOLOGY

The hardware portion of this project was designed and implemented using the Vivado Design Suite and Vitis HLS. The design is based on the Zynq-7000 SoC, which contains both a Processing System (PS) and Programmable Logic (PL). The PS is used for system control and UART communication, while the PL is used to implement a high-speed custom image processing IP. This section explains each step of the hardware design in a systematic and descriptive manner.

4.1 Platform Definition and Setup

The hardware development began by creating a Vivado project targeting the ZedBoard Zynq-7000 (XC7Z020CLG484-1) FPGA. The project was configured to include the following components:

- **ZYNQ7 Processing System (PS):** Configured to enable UART1 for communication with the host PC. The UART was mapped to MIO pins 48 and 49, which are physically connected to the USB-UART interface on the board. The Processing System was also set to provide the system clock and reset signals for all other components.

- **AXI DMA (Direct Memory Access):** This IP core enables high-speed data transfer between DDR memory and the custom HLS IP block using AXI4-Stream interfaces. Both directions of transfer were enabled:
  - MM2S (Memory to Stream)
  - S2MM (Stream to Memory)
  - The Scatter-Gather mode was disabled to simplify the design and reduce control complexity.

- **AXI SmartConnect:** Used to interconnect the Zynq Processing System, AXI DMA, and the custom IP block. It allows multiple AXI masters and slaves to communicate using a single, centralized switch.

- **Clocking and Reset Signals:** All components were connected to the FCLK_CLK0 (system clock) and FCLK_RESETN (active-low reset) generated by the Processing System.

Once the block diagram was complete and validated, the design was synthesized and implemented. The final bitstream was generated and exported along with the .xsa hardware description file, which was later imported into Vitis for software development.

4.2 Custom IP Design Using Vitis HLS

The core functionality of the system—applying five image processing filters; was implemented using a single unified HLS IP, named multi_filter.

IP Design Objectives:

- Apply all five filters in sequence to a grayscale image.
- Handle streaming input and output using ap_axiu<8,1,1,1> data types.
- Keep internal memory and logic minimal and reusable to conserve FPGA resources.

Implemented Filters:

1. Box Blur
2. Edge Detection (Sobel Operator)
3. Unsharp Masking
4. High Pass Filter
5. Low Pass Filter

Each filter was applied to the same input image, and the five results were stored in output buffers.

Data Types and Ports:

- **Input/Output Streams:** Used hls::stream<ap_axiu<8,1,1,1>> to send and receive pixel data.
- **Control Interface:** The IP used AXI4-Lite interface for control, though no dynamic parameters were configured in this version.
- **Image Buffers:** Used ap_uint<8> arrays to store and process pixel data internally.

After validating the C code using C/RTL simulation, the design was synthesized using Vitis HLS. A custom IP package was generated (multi_filter.zip) and imported into the Vivado project.

4.3 AXI4-Stream-Based DMA Integration

The custom IP was connected to the AXI DMA IP via AXI4-Stream interfaces:

- **Input Stream:** Connects the in_stream port of multi_filter to the MM2S channel of AXI DMA.
- **Output Stream:** Connects the out_stream port to the S2MM channel, sending the filtered results back to DDR memory.

The IP also outputs an interrupt signal, which notifies the Processing System once all filtering operations are complete.

4.4 Interrupt and Control Flow

To manage the flow of data and coordinate execution, a simple interrupt-driven design was used:

1. Image data is sent from the PC and stored in DDR.
2. DMA is configured to transfer data from DDR to the custom IP.
3. Once filtering is complete, the IP asserts an interrupt signal.
4. The interrupt is handled by the PS, which then initiates the return of data back to the PC.

This approach ensures synchronization between the host, processor, and IP core without requiring complex polling mechanisms.

4.5 Synthesis, Implementation, and Resource Utilization

After completing the block design and IP integration, the project was synthesized and implemented. During synthesis, Vivado optimized the design by analyzing the IP structure, logic dependencies, and interconnects.

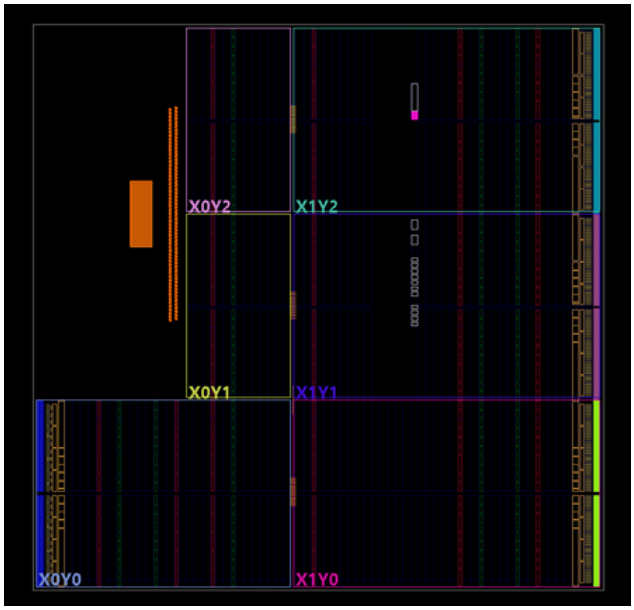


Fig 5: Synthesis Design Layout in Vivado.

Once synthesis was complete, the design was implemented to map it onto the FPGA resources.

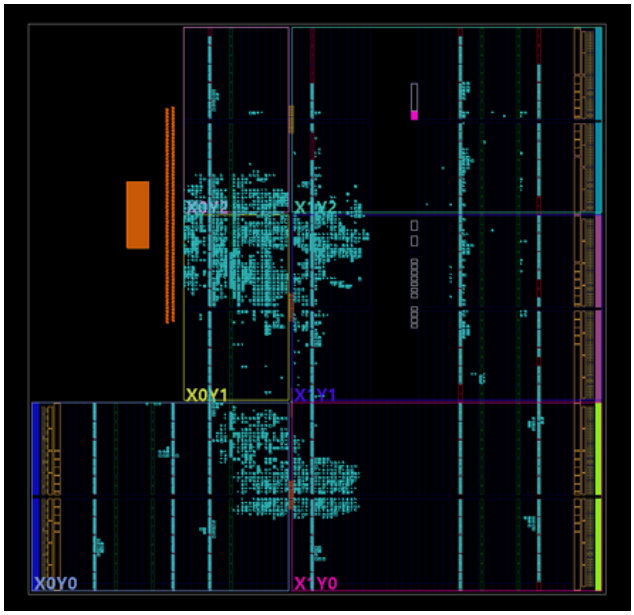


Fig 6: Implementation Design View in Vivado.

These views show how the design was placed and routed within the FPGA fabric. The logic blocks, routing paths, and I/O connections were all automatically optimized by Vivado.

4.6 Simulation and Functional Validation

To confirm the correct behavior of the multi_filter IP, a testbench was written in C and simulated using the Vitis HLS simulator. The waveform view shows internal signal transitions such as:

- Valid input signals
- Pixel processing activity
- Image data output
- Interrupt assertion

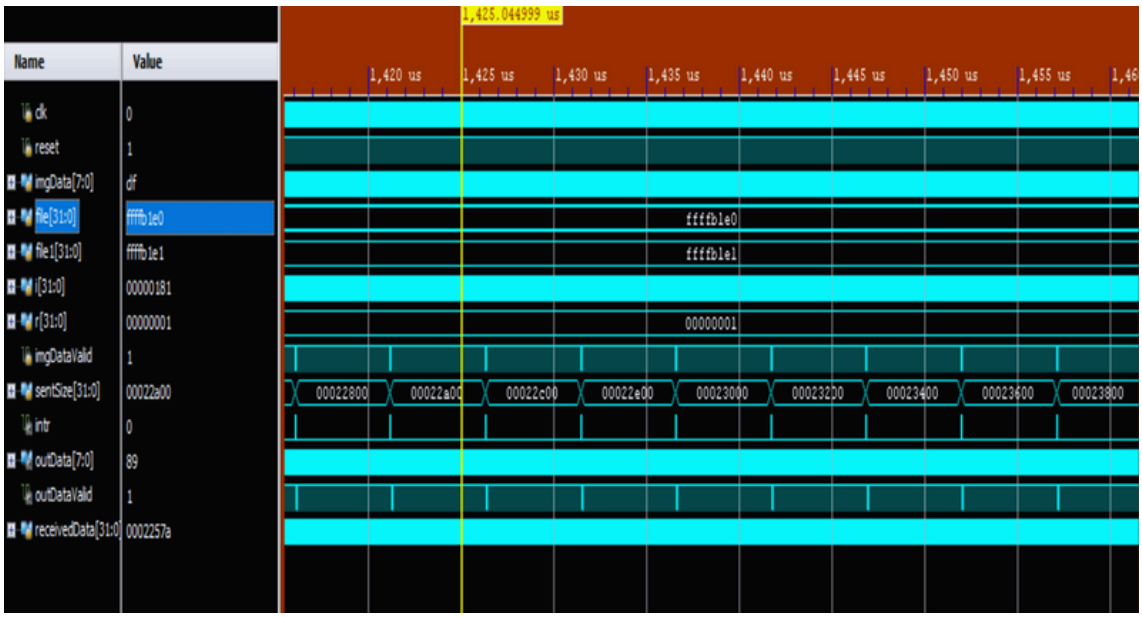


Fig 7: Simulation waveform showing functional behavior of the image processing IP.

In summary, he hardware design of this project involved building a modular and efficient dataflow system using Xilinx tools. The Zynq-7000 SoC allowed for tight integration between software and hardware components. By using HLS for filter implementation and AXI DMA for high-speed data transfer, the system achieved both flexibility and speed. The final design is compact, scalable, and well-suited for real-time embedded image processing.

SOFTWARE DESIGN METHODOLOGY

While the hardware is responsible for the image filtering operation, the software controls the overall system, manages data transfer, and ensures smooth user interaction. In this project, the software design has two major components:

1. A bare-metal C application developed in Vitis IDE for the ARM Cortex-A9 processor on the Zynq Processing System (PS).
2. A Python GUI application developed on the host PC, used to upload images, send and receive data, and display filtered results.

This section explains how both of these software parts were designed and how they interact with the hardware system.

5.1 Bare-Metal C Application (Zynq PS Software)

The software running on the Zynq PS was written in C using the Vitis IDE (2024.2). This application runs without any operating system (bare-metal) and directly interacts with the memory and peripherals.

5.1.1 Main Functions of the Embedded Application:

- Initialize the UART peripheral for communication with the host PC.
- Initialize and configure the AXI DMA controller.
- Wait for image data to arrive via UART.
- Store the image in DDR memory.
- Configure the DMA for MM2S transfer to send the image to the multi_filter IP.
- Wait for the IP to finish processing (indicated by the interrupt signal).
- Configure the DMA for S2MM transfer to receive the processed image back to memory.
- Send the filtered image data back to the host PC via UART.

5.1.2 UART Communication:

- UART1 was used for serial communication.
- Baud rate: 115200 bps.
- UART is configured to use MIO pins 48 and 49.
- The data is transmitted and received byte by byte in a synchronous loop.

5.1.3 DMA Configuration and Execution:

- DMA is operated in simple mode with one transaction at a time.
- The size of each image is 128 × 128 = 16,384 bytes.
- Since there are 5 output images, total transfer size = 81,920 bytes for output.

DMA handles data efficiently by transferring large blocks with minimal CPU intervention, which makes the system fast and responsive.

5.1.4 Interrupt Handling:

- The application checks for a done interrupt from the multi_filter IP.
- Once the interrupt is received, it knows the IP has finished processing.
- This event is used to trigger the next data transmission (result transfer).

5.2 Python GUI Application (Host PC Software)

The host software was developed in Python using the Tkinter library for GUI and PySerial for UART communication. It runs on the user's computer and serves as the interface to interact with the FPGA board.

5.2.1 GUI Layout and User Interaction:

- The GUI was built using Tkinter, which is Python's built-in GUI toolkit.
- Users can click a button to upload an image from their computer.
- The uploaded image is automatically resized to 128 × 128 pixels and converted to grayscale using the Pillow library.

- The GUI shows:
  - The input image.
  - The five output images arranged in a 3×3 grid.
  - Buttons to send the Upload Image, and Save Outputs.

Fig 8: User Interface of Python GUI App.

5.2.2 Image Processing Functions (Pre-Transfer):

- The selected image is:
  - Converted to grayscale.
  - Resized to 128×128.
  - Flattened into a one-dimensional byte array.
- This ensures compatibility with FPGA data format and transfer size limits.

5.2.3 UART Communication with FPGA:

- Communication is done using the PySerial library.
- Serial port is set to COM4, with a baud rate of 115200.
- The GUI sends the input image as a stream of 16,384 bytes.
- It then waits for 5 × 16,384 = 81,920 bytes of output image data.
- The output is sliced into five images and displayed using Tkinter Canvas or Label widgets.

5.2.4 Code Structure:

The Python application is divided into three key files for modularity:

- app.py – Handles GUI layout, button callbacks, and display logic.
- image_utils.py – Handles image resizing, grayscale conversion, and flattening.
- usb_interface.py – Manages serial communication and image transfer logic.

This modular approach helps in debugging and makes future upgrades easier, such as adding webcam input, real-time streaming, or filter selection.

5.3 Workflow Summary

Below is a simplified overview of the software workflow:

1. User uploads image using GUI.
2. Image is preprocessed and sent to FPGA via UART.
3. FPGA performs filtering and sends back five output images.
4. GUI receives, reshapes, and displays all results.
5. User can optionally save the filtered images.

5.4 Advantages of the Software Design

- **User-Friendly Interface:** The GUI allows easy use of FPGA functionality without any technical knowledge of hardware.
- **Real-Time Communication:** The use of UART enables smooth data flow between host and FPGA.
- **Modular and Scalable:** Python code is structured into clear modules. The embedded C code is also modular and reusable for future applications.
- **Efficient Control Flow:** The interrupt-based logic reduces processor load and ensures fast execution of operations.

The software design of this project enables efficient interaction between the user and the hardware system. The bare-metal C application controls the data transfer and IP execution, while the Python GUI provides a friendly interface for uploading images and viewing results. Together, these two components form a complete, user-accessible, and high-performance image filtering system. The system is well-structured and ready for future improvements such as support for real-time streaming or additional filter types.

IMAGE FILTER LOGIC

Image filtering is a fundamental technique in digital image processing used to enhance visual features or extract important details from images. In this project, five widely used filters were implemented using a single unified hardware module, developed in Vitis HLS and integrated into the FPGA. Each filter operates by applying a convolution kernel to a grayscale image of size 128×128 pixels.

The filtering logic is implemented inside the custom HLS IP named multi_filter. This module takes the input image stream, applies all five filters sequentially, and generates five output images. Each filter is explained in detail below along with its functionality and the convolution kernel used.

6.1 Box Blur Filter

The Box Blur filter is used to smooth an image by averaging the values of neighboring pixels. It reduces noise and small details, making the image appear soft and less sharp.

Purpose:

- To reduce image sharpness and smooth out minor variations.
- Used in preprocessing to suppress noise.

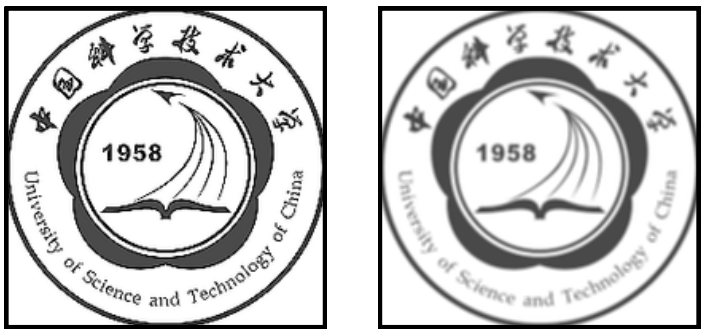
Kernel:

$$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$$

Each output pixel is the average of its 3×3 neighborhood.

Effect on Image:

- Smooth and blurred appearance.
- Softens edges and details.



Original Image Box Blur

Fig 9: Example of Box Blur filter applied to the input image.

6.2 Edge Detection (Sobel Operator)

Edge detection is used to highlight the boundaries between objects in an image. This project uses the Sobel operator, which calculates gradients in both horizontal and vertical directions.

Purpose:

- Detect sharp changes in pixel intensity.
- Identify object outlines and shapes.

Kernel:

Horizontal Sobel Kernel (Gx):

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Vertical Sobel Kernel (Gy):

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

The magnitude of the edge is calculated using:

$$G = \sqrt{G_x^2 + G_y^2}$$

Effect on Image:

- Shows edges as bright lines.
- Removes flat or uniform regions.

Original Image Edge Detection

Fig 10: Image after applying Edge Detection using Sobel filter.

6.3 Unsharp Masking

Unsharp masking enhances the sharpness of an image by subtracting a blurred version from the original image. It increases contrast around edges, making them more visible.

Purpose:

- To sharpen image details.
- Useful in post-processing to improve visual clarity.

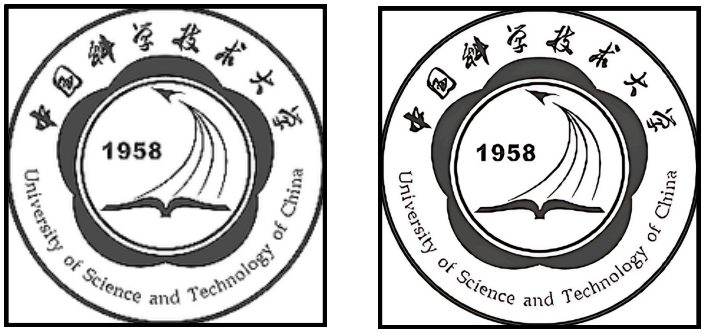
Kernel:

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

This kernel boosts the intensity of the center pixel while reducing the surrounding ones.

Effect on Image:

- Edges and fine details appear more distinct.
- Image appears more vivid and defined.



Original Image Unsharp Mask

Fig 11: Result of Unsharp Masking filter.

6.4 High Pass Filter

A high pass filter emphasizes high-frequency components like edges and fine textures. It suppresses low-frequency content like smooth regions and background areas.

Purpose:

- Highlight edges, lines, and fine details.
- Suppress gradual changes in intensity.

Kernel:

$$\begin{bmatrix} 1 & -\frac{7}{9} & -\frac{7}{9} \\ 0 & 5 & 0 \\ 0 & -\frac{7}{9} & -\frac{7}{9} \end{bmatrix}$$

The sum of kernel values is zero, which preserves edge differences but eliminates uniform areas.

Effect on Image:

- Emphasizes outlines and small patterns.
- Makes the image appear brighter and more textured.

Original Image High Pass

Fig 12: Image after High Pass Filtering.

6.5 Low Pass Filter

A low pass filter smooths the image by reducing high-frequency content. It is often used to remove sharp transitions and noise.

Purpose:

- Reduce sharpness and smooth edges.
- Eliminate small unwanted variations.

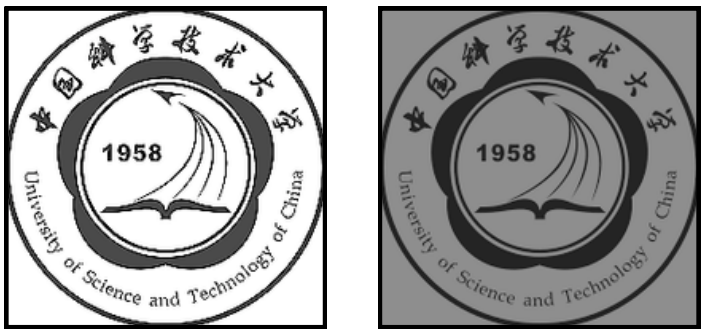
Kernel:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0 & \frac{5}{9} & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

This kernel gives more weight to the center pixel and gradually less to the neighbors.

Effect on Image:

- Appears smooth and slightly blurry.
- Removes edge noise and sharp transitions.



Original Image Low Pass

Fig 13: Output of Low Pass Filtering on the input image.

6.6 Combined Filtering in Unified IP

All five filters are implemented inside a single hardware block. The multi_filter IP is designed to:

- Accept one input image stream.
- Store and process it internally.
- Apply each filter one after the other.
- Output five separate filtered streams.

This unified approach:

- Saves hardware resources.
- Ensures that only one DMA transmission is required for input.
- Reduces overall latency.

The five filters used in this project represent a range of basic but powerful image processing techniques. From blurring to sharpening, and edge detection to smoothing, these filters demonstrate the flexibility of hardware-accelerated image processing using FPGAs. Implementing them in a unified IP core helps achieve modularity, speed, and scalability, all within a compact and reusable design.

SYSTEM VALIDATION AND RESULTS

Once the hardware and software components were fully implemented, we conducted thorough testing to validate the functionality, reliability, and performance of the entire system. The validation process involved confirming that all filters produced correct results, that data transfer between the host and FPGA functioned without errors, and that the graphical interface correctly displayed and managed all output images.

This section explains how the system was tested, presents the results obtained from different stages of the design, and discusses the observed outputs.

7.1 UART Communication Validation

The first step in system validation was to ensure proper communication between the Python GUI and the FPGA board through the UART interface.

Tera Term UART Console Output: During testing, we used Tera Term, a terminal emulator, to monitor the UART output from the Zynq Processing System. Upon startup and successful initialization, the following debug messages were printed:

- `UART Initialized`
- `DMA Initialized`
- `Image Received`
- `Filter Started`
- `Filter Completed`
- `Sending Output Images`

These debug messages confirmed that the system was executing each step as expected, from image reception to processing and result transmission.

7.2 Functional Test with Sample Images

To verify the correctness of the filtering logic, we tested the system using sample grayscale images. The workflow for each test case was as follows:

1. The user uploads a grayscale image via the Python GUI.
2. The image is resized to 128×128 and sent to the FPGA.
3. The FPGA processes the image and applies all five filters.
4. The filtered images are returned to the host and displayed in the GUI.

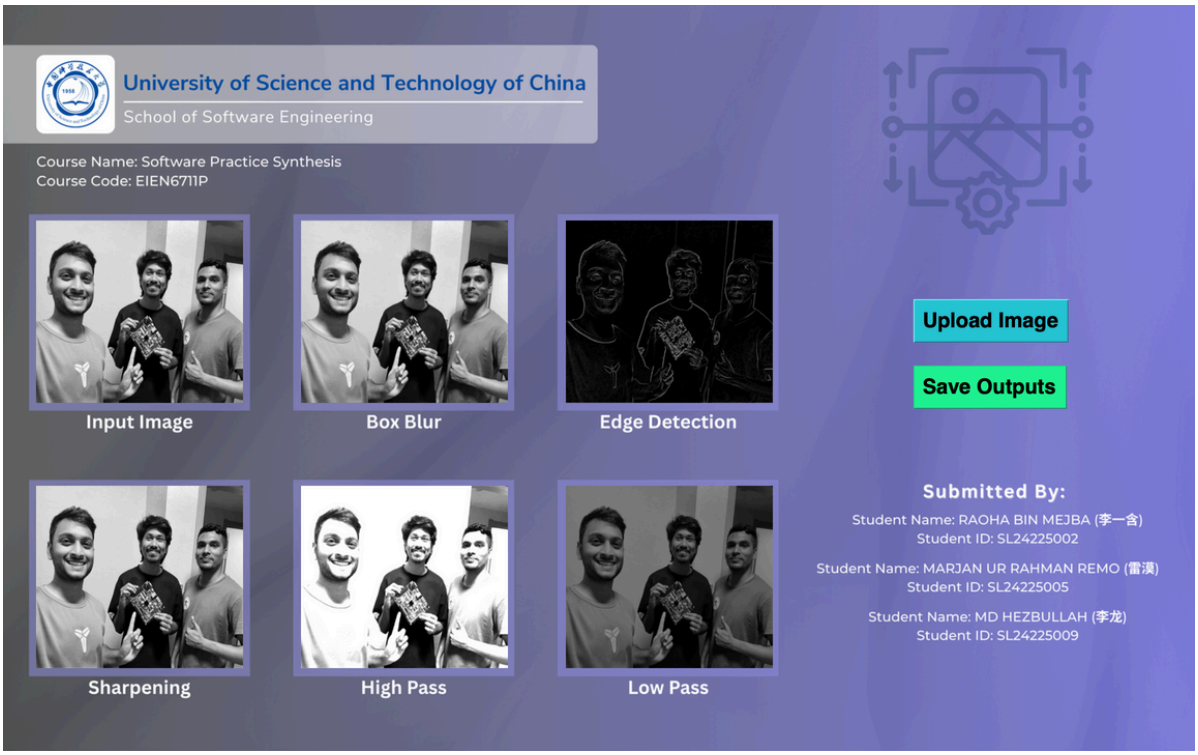


Fig 14: Filtered results displayed in the Python GUI.

Visual Comparison:

We visually compared the output images generated by the FPGA with the expected results from software simulations (C testbench in Vitis HLS). The outputs matched as expected, confirming the functionality of:

- **Box Blur:** Softened and blurred the image.
- **Edge Detection:** Detected clear object boundaries.
- **Unsharp Masking:** Sharpened details and edges.
- **High Pass:** Highlighted fine structures.
- **Low Pass:** Reduced brightness and noise.

7.3 Simulation Waveform Analysis

We validated the multi_filter IP core using behavioral simulation in Vitis HLS. A C testbench was written to provide synthetic input and verify the output.

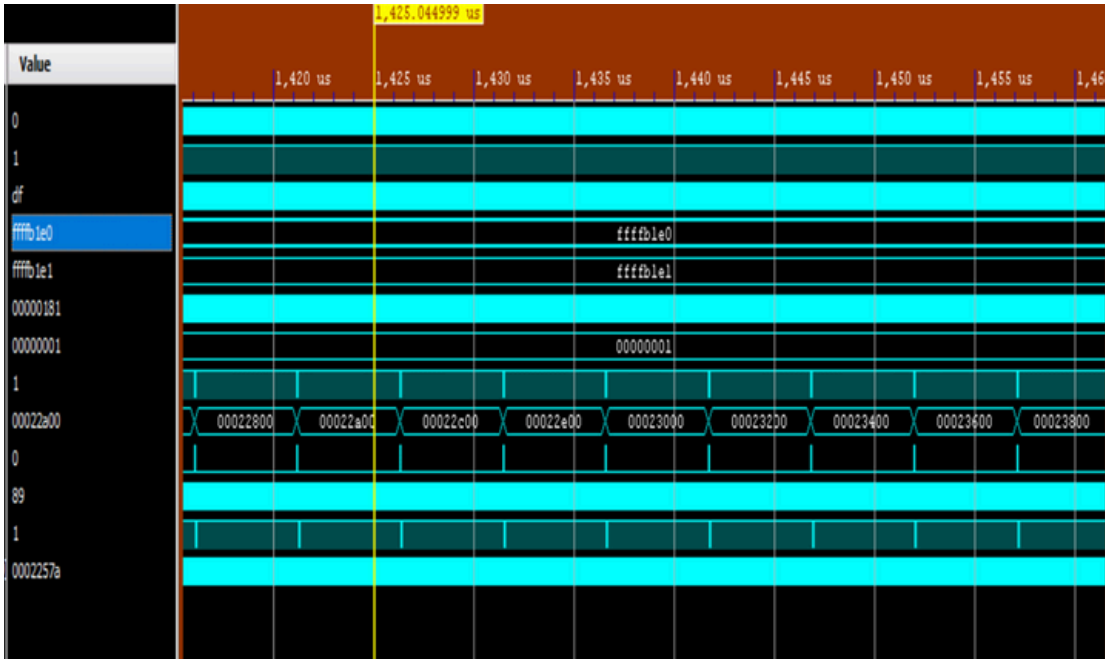


Fig 15: Simulation waveform showing valid data signals, image flow, and interrupt behaviour.

Waveform Insights:

- **imgDataValid:** Goes high when valid input pixel data is sent.
- **outDataValid:** Goes high when processed output data is ready.
- **intr:** Indicates the end of processing and triggers the processor.
- Memory indices and buffer updates were visible and consistent with expectations.

7.4 Hardware Resource Utilization

We also analyzed the FPGA resource usage after synthesis and implementation.

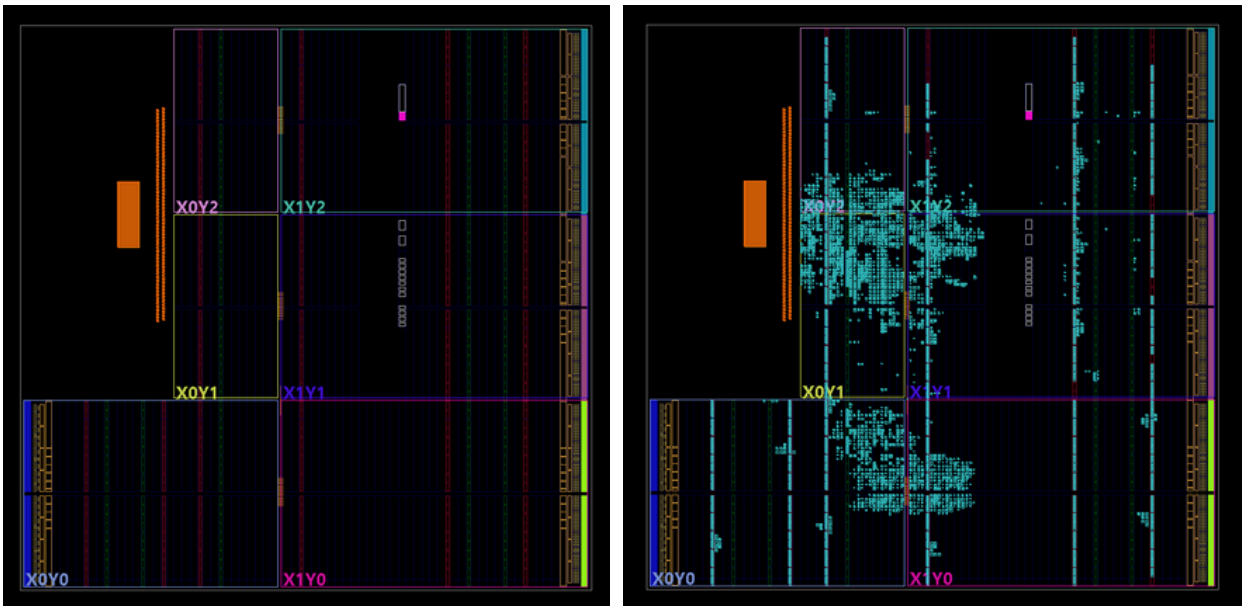


Fig 16: Synthesis and implementation layout in Vivado.

The design used moderate FPGA resources. By combining all filters into one multi_filter IP, resource consumption was kept efficient and within available limits on the Zynq-7000 device.

7.5 Physical Hardware Validation

We validated the final implementation by flashing the bitstream to the ZedBoard using JTAG and running the software application from Vitis. The board showed proper activity with LED indicators and terminal logs.

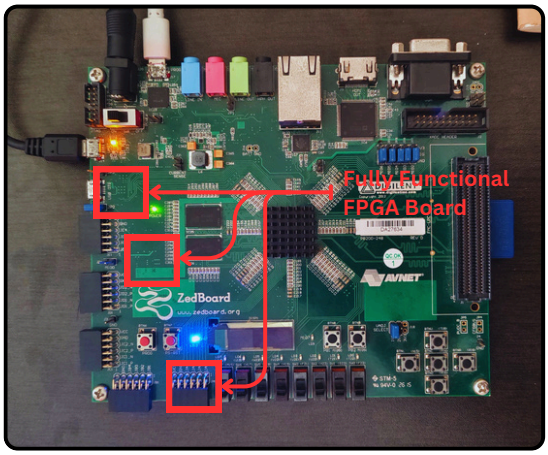
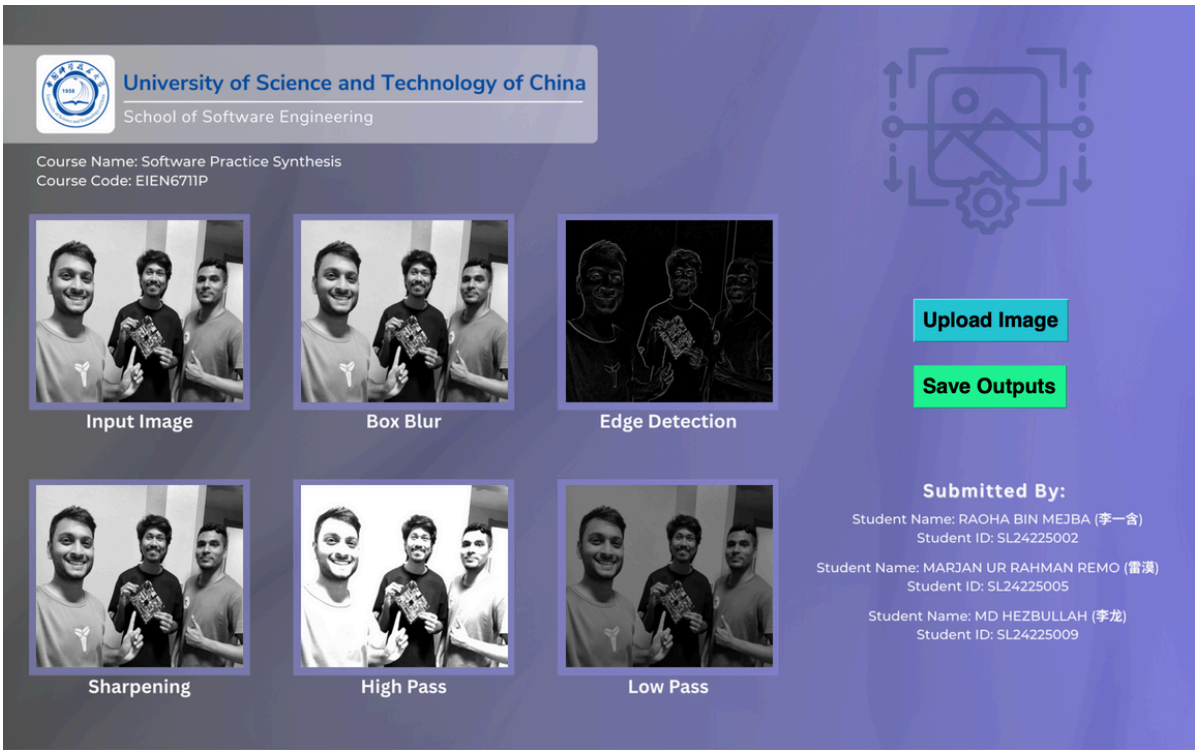
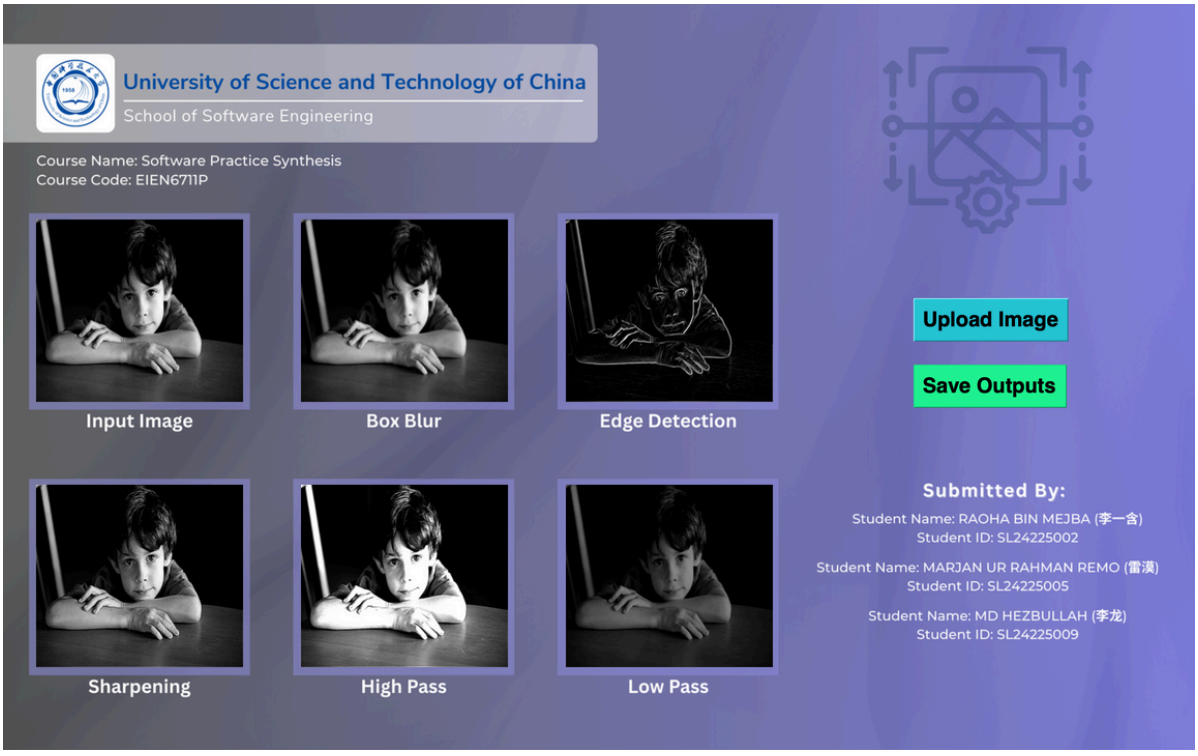


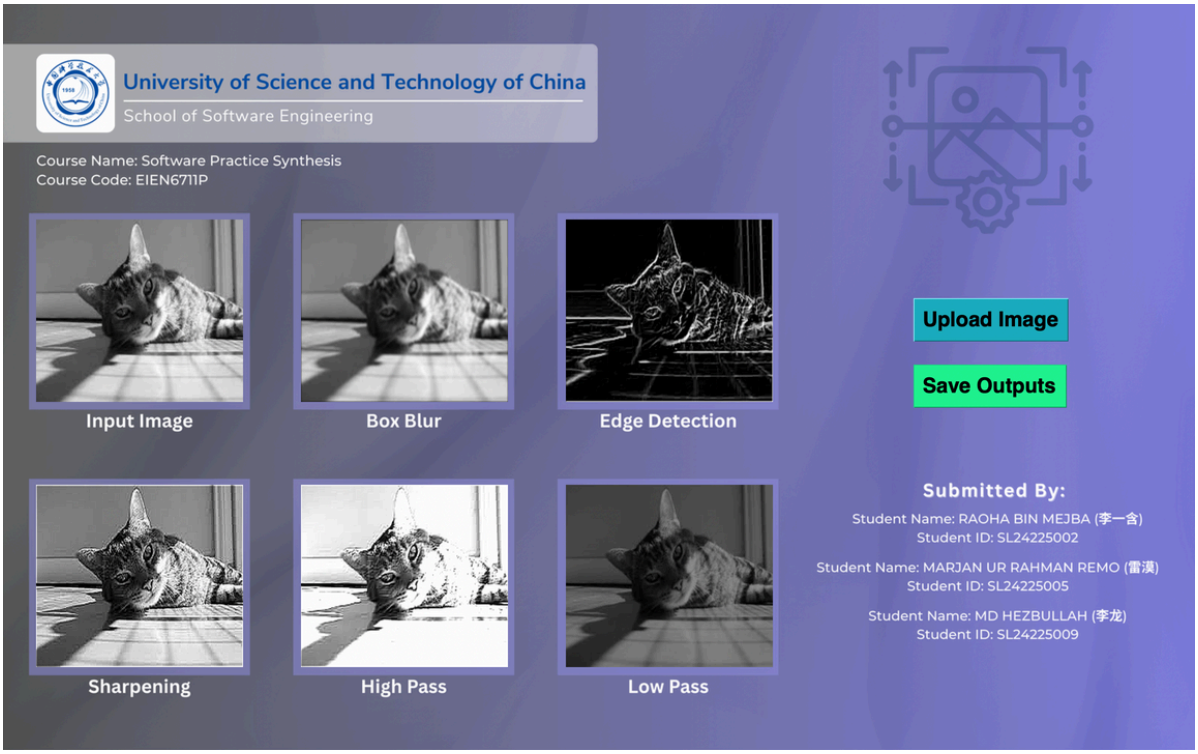
Fig 17: ZedBoard running the image filtering system (powered on, connected to PC). This confirmed that the board was correctly executing the image processing pipeline and communicating with the PC.

7.6 Project Screenshots




University of Science and Technology of China
School of Software Engineering
Course Name: Software Practice Synthesis
Course Code: EIEN6711P
Input Image
Box Blur
Edge Detection
Upload Image
Save Outputs
Sharpening
High Pass
Low Pass
Submitted By:
Student Name: RAOHA BIN MEJBA (李一含)
Student ID: SL24225002
Student Name: MARJAN UR RAHMAN REMO (雷漫)
Student ID: SL24225005
Student Name: MD HEZBULLAH (李龙)
Student ID: SL24225009


University of Science and Technology of China
School of Software Engineering
Course Name: Software Practice Synthesis
Course Code: EIEN6711P
Input Image
Box Blur
Edge Detection
Upload Image
Save Outputs
Sharpening
High Pass
Low Pass
Submitted By:
Student Name: RAOHA BIN MEJBA (李一含)
Student ID: SL24225002
Student Name: MARJAN UR RAHMAN REMO (雷漫)
Student ID: SL24225005
Student Name: MD HEZBULLAH (李龙)
Student ID: SL24225009



7.7 Output Verification Summary

Test Image	Output	Expected Behavior	Observed Behavior
Input_Image_1	Edge Map	Sharp outlines	Matched expectation
Input_Image_5	Box Blur	Soft, low contrast	Blurred successfully
Input_Image_4	Unsharp	Enhanced details	Clear sharpening seen
Input_Image_2	Low Pass	Smooth background	Reduced sharpness
Input_Image_3	High Pass	Highlighted edges	Bright outlines visible

All filters passed visual and data-level verification for correctness and consistency.

The validation process confirmed that the FPGA-based image processing system works as intended. The UART communication was reliable, DMA transfers were correct, and the IP core produced accurate filtered results. The Python GUI performed well during testing, making it easy to upload, visualize, and save the results. Simulation waveforms and board-level testing added confidence to the system's correctness and efficiency.

This comprehensive testing confirms that the hardware-software co-design approach successfully achieved the goal of real-time, interactive image filtering using FPGA acceleration.

CONCLUSION

In this project, we successfully designed and implemented a **real-time image processing system using FPGA** hardware and a user-friendly software interface. The main goal was to apply five different filters; **Box Blur, Edge Detection, Unsharp Masking, High Pass, and Low Pass**; to grayscale images using the **ZedBoard Zynq-7000** platform. The system was built with a clear hardware-software co-design methodology, which combined the power of parallel processing in FPGA with the flexibility of a Python-based graphical user interface.

On the hardware side, we developed a unified HLS IP core named **multi_filter** using Vitis HLS. This IP was integrated into a Vivado block design alongside the ZYNQ Processing System, AXI SmartConnect, and AXI DMA. The design was synthesized, implemented, and programmed onto the FPGA using **Vivado 2024.2**. The filtering IP processed input images using AXI4-Stream interfaces and communicated with the PS through interrupt signaling.

On the software side, we built two main applications. The first was a bare-metal C program running on the Zynq Processing System. It handled UART communication with the PC, managed DMA transfers, and controlled the filtering IP. The second was a Python GUI application developed using Tkinter, Pillow, and PySerial libraries. It allowed the user to upload an image, send it to the FPGA, receive the five filtered results, display them in a clean grid layout, and save them to the local device.

We validated the system through functional testing, UART terminal logs, waveform simulation, and visual output comparison. All five filters performed correctly and consistently across multiple test images. The GUI was responsive and easy to use. The hardware was efficiently utilized and successfully demonstrated real-time processing capability.

This project highlights the advantages of FPGA-based image processing:

- Speed: The filters were implemented in hardware, enabling fast, parallel execution.
- Efficiency: A single IP core handled multiple filters without redundant resource usage.
- Modularity: The design can be extended to support new filters or image sizes.
- Ease of Use: The Python GUI made the system accessible to non-expert users.

Overall, this project demonstrates a practical and scalable solution for implementing real-time image filtering using FPGA technology.

ACKNOWLEDGEMENT

We would like to express our sincere gratitude to **Dr. Ding Qing** for his continuous guidance, encouragement, and support throughout the development of this project. His valuable feedback and technical insights were instrumental in helping us understand the practical aspects of FPGA-based system design and hardware-software co-integration.

Through this project, we gained hands-on experience in several key areas of embedded and digital systems engineering. These include designing and synthesizing custom IPs using Vitis HLS, configuring hardware platforms in Vivado, and managing memory transfers with AXI DMA. We also learned to develop bare-metal embedded applications in Vitis, implement UART-based communication protocols, and design user-friendly graphical interfaces in Python.

Most importantly, this project taught us how to build a complete, functional system from hardware description to user-level interaction. We experienced the importance of modular design, debugging across multiple layers, and validating results both visually and through simulation.

We are truly thankful for the opportunity to work on this project under **Dr. Ding Qing**'s supervision, which significantly enhanced our technical skills and deepened our understanding of real-time image processing systems on FPGA.

BIBLIOGRAPHY

[1] Anthony Edward Nelson, Implementation of Image Processing Algorithms on FPGA Hardware, M.S. Thesis, 2000.

[2] D. Crookes, K. Benkrid, A. Bouridane, K. Alotaibi, and A. Benkrid, "Design and implementation of a high-level programming environment for FPGA-based image processing," 2000.

[3] R. Harinarayan, R. Pannerselvam, M. M. Ali, and D. K. Tripathi, "Feature extraction of digital aerial images by FPGA-based implementation of edge detection algorithms," in Proc. Int. Conf. on Emerging Trends in Electrical and Computer Technology, pp. 631–635, 2011.

[4] T. A. Abbasi and M. U. Abbasi, "A novel FPGA-based architecture for Sobel edge detection operator," International Journal of Electronics, vol. 94, no. 9, pp. 889–896, Sept. 2007.

[5] O. R. Vincent and O. Folorunso, "A descriptive algorithm for Sobel image edge detection," in Proc. Informing Science & IT Education Conference, pp. 97–107, 2009.

[6] Chen Lunhai, Huang Junkai, Yang Fan, and Tang Xu Li, "Real-time edge detection system based on FPGA," Liquid Crystal Display, no. 2, pp. 200–204, 2011.

[7] Liu Ziyan, "Design and implementation of real-time image edge detection," Electronic Technology, no. 12, pp. 1–3+6, 2011.

[8] Lin Yuansheng, Image Edge Detection System Design Based on FPGA [D]. Xi'an Electronic and Science University, 2014.

[9] Chen Hu, Ling Chao Dong, Zhang Hao, Yang Xiao, and Tang Wei, "Real-time color image edge detection algorithm based on FPGA," Liquid Crystal Display, no. 1, pp. 143–150, 2015.

[10] R. J. George, S. Charaan, S. P. Joy Vasantha Rani, and S. R. Swathi, "Design of an IP core for motion blur detection in fundus images using an FPGA-based accelerator," in Proc. 9th Int. Conf. on Biosignals, Images and Instrumentation (ICBSII), 2023, pp. 1–6, doi: 10.1109/ICBSII58188.2023.10181073.

[11] B. K. Upadhyaya and D. Chakraborty, "FPGA implementation of gradient based edge detection algorithms for real-time image," in Proc. 2nd Int. Conf. on Trends in Electronics and Informatics (ICOEI), 2018, pp. 1227–1233, doi: 10.1109/ICOEI.2018.8553840.