A Hardware-Software Co-Design for Real-Time Vision Processing with Remote Web Control

CSE 4602 Capstone Design Project Final Report

Submitted to Professor Hall and the Department of Computer Science and Engineering

Team Members:

Chung-Hao Tseng (ctseng@wustl.edu)
Baoyang Su (baoyang@wustl.edu)
Minghe Liu (lminghe@wustl.edu)
Ziling Zhu (ziling@wustl.edu)

Advisor:

Dr. Michael Hall

Date of Submission: May 2, 2026

Spring 2025

Abstract

Real-time video processing is the backbone of many interactive applications, where user experience relies heavily on immediate visual feedback. However, achieving low-latency processing while retaining the flexibility to adjust effects on the fly presents a significant engineering challenge. This project proposes the implementation of a high-performance, real-time video processing pipeline on an FPGA to address these needs. The system is designed to perform object tracking and chroma keying, specifically detecting a target color (e.g., red) within a live camera feed, calculating its spatial coordinates, and dynamically replacing the object with a custom color or graphical pattern. The architecture utilizes a hardware-software co-design approach: a custom pipelined hardware accelerator is developed to handle computationally intensive pixel operations, while an embedded CPU hosts a Python-based web server. This server exposes REST APIs and a user interface, enabling users to adjust vision parameters over the network in real-time. Development follows an incremental methodology, progressing from static image validation and offline video processing to full low-latency integration with a live video stream.

1 Introduction

1.1 Background and Motivation

Real-time video processing serves as the backbone of numerous interactive applications, ranging from augmented reality to industrial automation. In these contexts, user experience and system reliability depend heavily on immediate visual feedback. Conventionally, software-based computer vision frameworks running on standard CPUs have been utilized for these tasks; however, prior work shows these implementations frequently suffer from non-deterministic latency and frame drops due to operating system overhead and memory bottlenecks. To achieve low-latency processing while maintaining the flexibility to adjust vision parameters on the fly, specialized hardware acceleration is required. The use of FPGAs provides a unique advantage by allowing the implementation of custom, deeply pipelined hardware logic that can handle high-throughput video data with deterministic latency, effectively bridging the gap between performance and adaptability.

1.2 Problem Statement

While software-based image processing is flexible, it often struggles to meet the stringent timing requirements of high-frame-rate, real-time video streams, especially on embedded systems with limited CPU resources. This project addresses the challenge of implementing a high-performance, real-time video processing pipeline that can isolate specific objects based on color and perform dynamic pixel replacement. To be viable, the design must meet strict constraints: it must process video with an end-to-end latency of less than 100ms at a sustained frame rate of 30-60 FPS. Furthermore, the system must not only execute this processing entirely in hardware to meet these performance goals but also remain accessible for runtime parameter adjustments via a software-controlled interface without interrupting the video stream.

1.3 Project Objectives and Requirements

The primary objective of this design is to create a fully functional, hardware-accelerated video processing pipeline with software-controllable parameters.

- **Functional Requirements:**
  - **Develop Custom Hardware IP:** Implement a multi-stage hardware accelerator in the FPGA Programmable Logic comprising distinct custom IP cores:
    - **Color Processing IP:** Performs sequential RGB-to-HSV and HSV-to-RGB conversions for robust color isolation.
    - **Pixel Replacement IP:** A dedicated hardware module that executes dynamic pixel substitution in real-time based on the parameters passed from the user interface.
  - **Control Interface:** Create a Python-based web server on the embedded CPU to expose REST APIs and user interface, allowing users to adjust detection thresholds and vision parameters over the network at runtime.

- **Stream Architecture:** Utilize AXI4-Stream for efficient video data transfer and AXI4-Lite for control parameter management between the processor and programmable logic.

- **Non Functional Requirements:**
  - **Performance:** Maintain a target frame rate of 30-60 FPS.
  - **Latency:** Ensure that total end-to-end processing latency remains below 100 ms.

2 System Design & Implementation

2.1 System Overview

The system is a real-time, hardware-accelerated color replacement pipeline built on the Xilinx Zynq-7020 SoC. It captures live video from an OV5640 camera, isolates pixels falling within a user-defined HSV range, replaces their hue and saturation with new values, and streams the result to an HDMI display. Remote operators configure the target color range and replacement values over Ethernet through a REST API, without rebuilding the bitstream or restarting the pipeline.

The design follows a hardware–software co-design partitioning: high-throughput per-pixel processing runs in the Programmable Logic (PL), while management, UI, networking, and run-time configuration run on the Processing System (PS) under PYNQ Linux. The two domains communicate through the AXI4-Lite control bus and a pair of AXI VDMA engines that move frames between the PL pipeline and DDR memory.

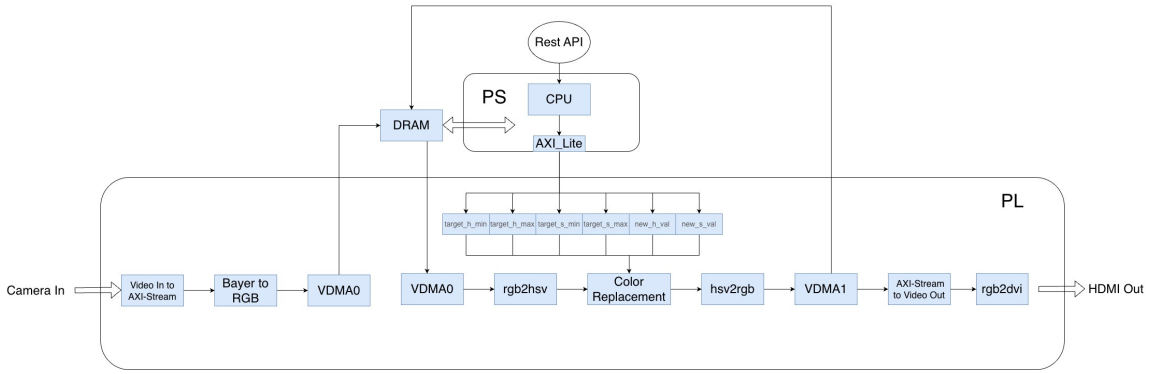

Figure 1: High-level system architecture showing the PL video pipeline, the PS control path, and the AXI-Lite register bank exposed to the REST API.

Data Flow: Pixels enter from the OV5640 camera and are converted to AXI4-Stream by the `Video In to AXI-Stream` block. A `Bayer to RGB` core demosaics the sensor pattern into 24-bit RGB. `VDMA0` writes frames into DRAM and reads them back into the image processing chain, decoupling capture from processing rates. The frame is then converted to HSV (`rgb2hsv`), passed through the `Color Replacement` core, and converted back to RGB (`hsv2rgb`). `VDMA1` buffers the output frame, which is serialized by `AXI-Stream to Video Out` and driven onto HDMI by `rgb2dvi`.

Control Flow: A remote client issues HTTP requests to a REST API hosted on the ARM CPU. The server validates the parameters and writes them to a bank of memory-mapped registers (`target_h_min`, `target_h_max`, `target_s_min`, `target_s_max`, `new_h_val`, `new_s_val`) over AXI4-Lite. These registers feed the `Color Replacement` core directly, so threshold updates take effect on the next frame with no pipeline stall.

2.2 Design Approach

Key Design Decisions

- **HW/SW partitioning.** Per-pixel work (color-space conversion, thresholding, replacement) is fixed-function and bandwidth-heavy at 1280×720 @ 30 FPS as a streaming dataflow pipeline. Configuration, networking, and UI are mapped to the PS, where they benefit from the software stack.
- **HSV color space for thresholding.** Thresholding directly in RGB is sensitive to lighting changes because brightness and chromaticity are entangled across all three channels. Converting to HSV isolates hue and saturation from value (intensity), making the target-color range stable under varying illumination and reducing the threshold-tuning surface from six bounds across correlated channels to four bounds on independent ones.
- **Frame buffering through DDR via VDMA.** A round-trip through DRAM (camera write to DRAM via `VDMA0`, then `VDMA0` read into the processing chain) decouples the capture clock domain from the processing clock domain and allows the processing pipeline to be debugged or replaced without re-timing the capture front-end. `VDMA1` provides the same decoupling on the display side.
- **AXI4-Lite register bank.** Exposing the six tuning parameters as discrete AXI4-Lite registers keeps the PS-side write path trivial (`MMIO.write`) and allows individual parameters to be updated by the REST API.
- **REST over a custom protocol.** HTTP/JSON is supported by every language and tool out of the box, which makes the system easy to script, test, and integrate. The throughput cost is negligible because configuration writes are infrequent compared to the 30 FPS pixel stream.

Trade-offs and Alternatives Considered

- **Vitis HLS vs. hand-written RTL.** HLS shortens the development cycle for the vision cores (`rgb2hsv`, `color_replacement`, `hsv2rgb`) and makes architectural exploration cheap via pragmas. Hand-written Verilog could shave area and improve f_{max}, but the HLS output already meets the 30 FPS / 100 ms budget, so the productivity win outweighs the resource cost.
- **FastAPI vs. Flask.** Both are viable; FastAPI is preferred for its automatic OpenAPI documentation, which reduces the chance of malformed register writes from a misbehaving client.
- **Color Replacement vs. Chroma Keying.** An earlier design considered a chroma keyer that composites the target region with a separate background. Color

replacement was chosen instead because it requires only a single video stream end-to-end, halving the DDR bandwidth and simplifying the VDMA configuration.

- **Resolution / frame-rate target.** 1280×720 @ 60 FPS was considered but rejected: the Zynq-7020's BRAM and DDR bandwidth are tight at HD, and the project's tracking goals are met at 1280×720. Dropping to 30 FPS was also considered as a fallback if timing closure becomes a problem.

Algorithm: Color Replacement

For each incoming pixel, the `rgb2hsv` core produces (H, S, V). The `Color Replacement` core asserts a match when

$$\texttt{target\_h\_min} \le H \le \texttt{target\_h\_max} \quad \textit{and} \quad \texttt{target\_s\_min} \le S \le \texttt{target\_s\_max}.$$

Matching pixels have their hue and saturation overwritten by `new_h_val` and `new_s_val` respectively; the value (V) channel is preserved so that shading and texture survive the swap. Non-matching pixels pass through unmodified. The core is fully pipelined at one pixel per cycle, so its latency is constant regardless of the number of matches in a frame.

2.3 Implementation

Hardware Development

- **Xilinx Vivado 2024.2:** System integration, block design, floorplanning, and bitstream generation, using Xilinx IP cores (`Video In to AXI-Stream`, `Bayer to RGB`, `AXI VDMA`, `AXI-Stream to Video Out`, `rgb2dvi`).
- **Vitis HLS:** Generates RTL with AXI4-Stream interfaces from C++ for the custom `rgb2hsv`, `color_replacement`, and `hsv2rgb` cores. `PIPELINE` and `INTERFACE` pragmas drive the one-pixel-per-cycle target.
- **Interconnects:** AXI4-Stream carries pixels through the dataflow pipeline; AXI4-Lite maps the six configuration registers into the PS address space.

Software Development

- **PYNQ Framework:** Loads the bitstream as an overlay, allocates contiguous buffers via the allocator for the VDMA frame stores, and exposes the AXI4-Lite registers through the `MMIO` class.
- **REST API:** A FastAPI server running on the ARM CPU exposes endpoints to read the current configuration and write new threshold/replacement values.
- **User Interface:** We use HTML to built an interactive user interface, which allow users to preview the color replacement thresholds and target before submitting.

Hardware Platform

- **TUL PYNQ-Z2:** Zynq-7020 SoC with on-board HDMI in/out, Ethernet, and sufficient PL resources for the streaming pipeline.
- **Peripherals:** OV5640 camera as the video source and an HDMI monitor for visual verification of the processed output.

Testing and Debugging Approach

- **HLS C-simulation and co-simulation.** Each custom core (`rgb2hsv`, `color_replacement`, `hsv2rgb`) is verified in C against a Python/OpenCV reference on a set of synthetic and real frames before RTL is generated, then re-verified with C/RTL co-simulation to confirm pipeline behavior.
- **Behavioral simulation in Vivado.** The integrated streaming pipeline is exercised with an AXI4-Stream traffic generator at the input and a stream monitor at the output to confirm `TLAST`/`TUSER` framing and back-pressure handling.
- **Software-side testing.** The REST API is exercised with `curl` and Postman. A loopback test reads back each register after writing it to confirm the AXI4-Lite path. For the correctness, the tests were mainly done by visual check to compare the output with the UI preview.

OV5640 Camera Integration

The OV5640 camera serves as the primary video input source of the system. Its integration involves sensor configuration, signal interfacing, and data format conversion to match the FPGA video pipeline.

- **Sensor Configuration via I2C:** The Processing System initializes the OV5640 by writing a sequence of configuration registers through the I2C interface. These registers define the output resolution, pixel format, frame rate, and streaming control. Proper initialization is critical to ensure stable image output and correct color representation.
- **Clock and Signal Interface:** The FPGA provides the external clock (XCLK) required by the camera sensor. The camera outputs pixel data synchronized by PCLK, along with control signals including VSYNC (frame boundary) and HREF (line validity). These signals must be properly aligned and synchronized with the FPGA clock domain to ensure correct data capture.
- **DVP to AXI4-Stream Conversion:** The OV5640 outputs raw Bayer-pattern data through its DVP interface. The Video In to AXI4-Stream IP captures this parallel data stream and converts it into AXI4-Stream format, which is compatible with the downstream FPGA processing pipeline.
- **Bayer to RGB Conversion:** Since the raw sensor data follows a Bayer pattern, a demosaic module is required to reconstruct full-color pixels. The Bayer to RGB IP converts the input stream into 24-bit RGB format, enabling subsequent color-space processing and visualization.

3 Evaluation & Results

3.1 Testing and Data Collection

We tested color conversion under various colors and lighting conditions, and measured the frame rate.

- **Conversion of Different Colors:** By adjusting the Hue and Saturation parameters within the HSV color space, we tested three sets of different color conversions. The Hue and Saturation values used are shown in the table below:

	target H min/max	**target S min/max**	**new H**	**new S**
red to green	170/10	100/255	60	240
green to blue	35/85	100/255	120	240
blue to red	100/140	100/255	0	240

Table 1: Color Conversion Parameters

- **Conversion Under Different Lighting Conditions:** We tested color conversion under different lighting conditions, observing and analyzing the results.
- **Test Method:** We conducted tests under indoor lighting using an A4 sheet containing a color bar. This is how it looks like:

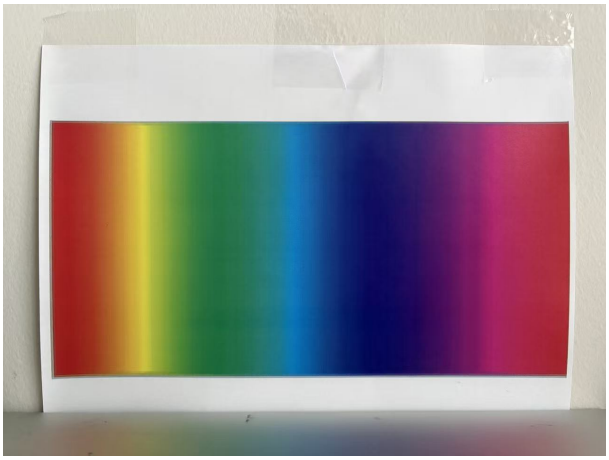

Figure 2: A Color Bar

- **Frame Rate Measurement:** We measured the frame rate of video stream processing using FPGA and compared it with processing the same video stream using PYNQ's CPU. We measured the average frames per second over a 10 seconds for 5 times.

3.2 Results

Color Conversion

- **Control Group:** This is the video output without color conversion (all values set to 0).

Figure 3: Video Output: Control Group

- **Red to Green, Green to Blue, Blue to red:** These are the video outputs for three sets of color conversions:

Figure 4: Video Output: Red To Green

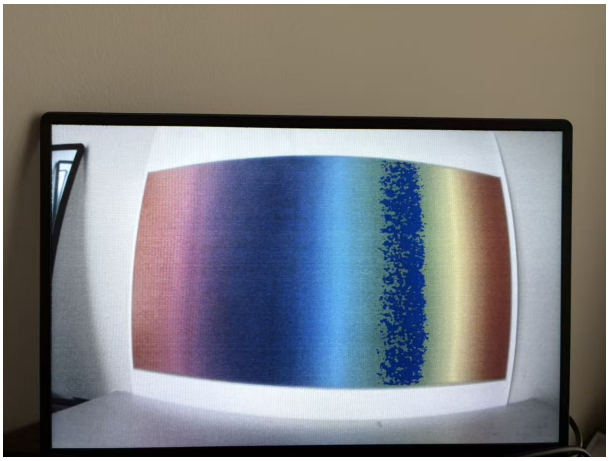

Figure 5: Video Output: Green To Blue

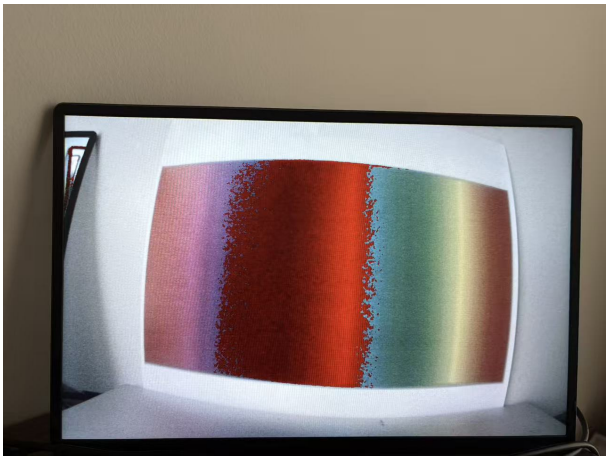

Figure 6: Video Output: Blue To Red

- **Green To Blue, At Different Light Intensities:** These are the video outputs for the green-to-blue group in brighter and darker ambient light:

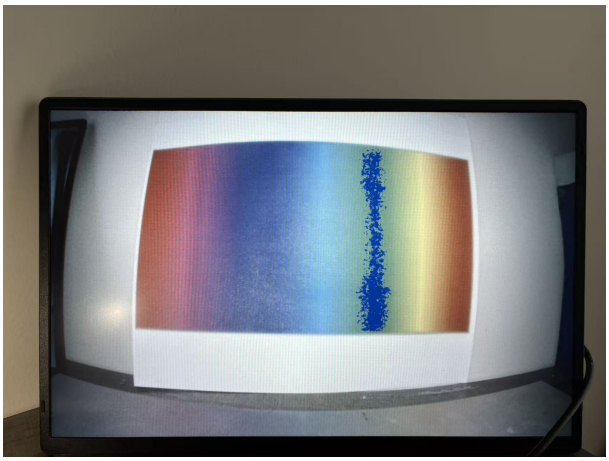

Figure 7: Video Output: Green To Blue, Brighter

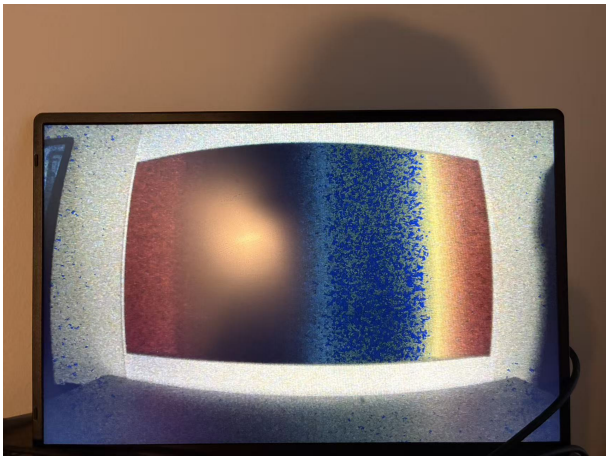

Figure 8: Video Output: Green To Blue, Darker

- **Evaluation:** For each group, the color conversion process successfully identified the

correct color regions and executed the transformation. However, the boundaries of the selected color areas appear indistinct. Furthermore, the effectiveness of the color conversion is influenced by ambient lighting conditions; variations in ambient light result in shifts in the selected color regions.

Frame Rate

By measuring the frame rate of the video stream and comparing it with the CPU's frame rate, the performance enhancement provided by the FPGA can be intuitively demonstrated. The results are shown in the table below:

	1	**2**	**3**	**4**	**5**
FPGA (FPS)	29.8	30.1	30.2	29.7	30.1
CPU (FPS)	2.5	2.5	2.5	2.5	2.5

Table 2: Frame Rate Comparison

- **Evaluation:** The FPGA frame rate fluctuates around 30 FPS, while the CPU frame rate remains stable at 2.5 FPS. The FPGA frame rate aligns with our design specifications, and our design demonstrates distinct advantages compared to processing video streams using the CPU.

3.3 Performance Metrics

- **Frame Rate:** 30 FPS.
- **Resolution:** Support for video input up to 1280x720.
- **Throughput:** Reliable AXI4-Lite data transfer between PS and PL.

4 Discussion & Reflection

4.1 Technical Discussion

This project demonstrates that a hardware–software co-design approach is effective for real-time video processing. By implementing pixel-level operations in FPGA programmable logic and using AXI4-Stream for data transfer, the system achieves stable throughput and low latency. The integration with a REST API allows runtime parameter adjustments without interrupting the video stream, improving system flexibility.

The system meets the main functional requirements, including real-time color detection, pixel replacement, and centroid tracking. However, several challenges were encountered during development. In particular, handling different image data formats led to color distortion issues that required careful debugging. In addition, camera initialization and I^2C communication were sometimes unstable, affecting system reliability. These challenges highlight the importance of consistent data formats and robust hardware interfaces in FPGA-based designs.

4.2 Ethical Considerations

Since the system processes real-time video data, it may raise privacy concerns if used for tracking or monitoring individuals. To mitigate these risks, such systems should be deployed with user consent and avoid unnecessary data storage or transmission.

4.3 Project Reflection

The modular pipeline design and incremental testing approach (from static images to real-time video) were effective in ensuring system correctness. However, hardware debugging proved more complex than expected, especially when dealing with multiple clock domains and interfaces.

If extended, the system could be improved by enhancing camera stability, adding better debugging tools, and supporting higher resolutions. Overall, this project provided valuable experience in FPGA-based system design, hardware–software integration, and real-time processing.

5 Project Management

5.1 Deliverables

At the beginning of the project, we planned to deliver a complete real-time vision processing system capable of detecting a target color in a live video stream and performing dynamic pixel replacement. The system was also expected to support runtime parameter adjustment through a software interface without interrupting the video pipeline.

The final deliverables include:

- A fully functional FPGA-based video processing pipeline running on the PYNQ-Z2 platform
- Custom HLS IP cores for RGB-to-HSV conversion, color detection, and pixel replacement
- A stable HDMI input/output pipeline supporting real-time video processing at 1280×720 resolution
- A REST API backend implemented using FastAPI for runtime configuration of detection thresholds and replacement parameters
- A complete FPGA bitstream and software stack for deployment and testing

Overall, the implemented system successfully meets the core functional requirements, including real-time processing, dynamic color replacement, and remote parameter control.

5.2 Timeline and Progress

The project followed a milestone-based development plan, with key stages including HDMI pipeline bring-up, HLS IP development, control interface integration, and system validation.

In practice, early milestones such as HDMI input/output validation and initial HLS IP development were completed on schedule. However, during system integration, additional time was required to resolve issues related to AXI-Stream interfaces, including data width mismatches, clock domain inconsistencies, and control signal alignment.

Despite these challenges, the team maintained steady progress by adopting an incremental debugging strategy. Simplified pipeline configurations (e.g., bypass designs) were used to isolate and resolve issues before full integration. While some delays occurred during the integration phase, the overall project remained aligned with the planned timeline, and all major milestones were completed before the final demonstration.

Phase/Task	**Description**	**Target Date**
HDMI Baseline	Verify HDMI I/O pass-through.	02/22
HLS Vision IP	Develop the RGB-to-HSV modules in Vitis HLS.	03/08
Control Logic	Map AXI-Lite and centroid calculation.	03/22
Milestone 2	**Demonstration of Core Functionality**	**03/23**
Optimization	Implemen Chroma Keyer, optimize the to 60 FPS.	04/05
System Validation	Conduct end-to-end latency testing ($<$ 100ms).	04/19
Final	**Final Presentation & Demo**	**04/20**

Table 3: Project Schedule and Milestones

5.3 Team Contributions

The project was divided into hardware, software, and integration components, with each team member taking primary responsibility for specific subsystems:

- **Baoyang Su** Responsible for FPGA hardware integration and system bring-up, including camera interfacing, video pipeline development, and HDMI output. Debugged clocking, I2C, and data path issues, and validated stable real-time video streaming. Also contributed to system integration and cross-module debugging.
- **Chung-Hao Tseng** Led system architecture design and hardware–software integration, including coordination between FPGA modules and the software control interface. Developed key image processing logic, including the RGB-to-HSV conversion and dynamic pixel replacement algorithms. Also contributed to REST API design and system-level debugging.
- **Minghe Liu** Focused on unit testing and debugging data path issues (e.g., color format inconsistencies—specifically the distinction between RGB and BGR), while ensuring the correct and error-free operation of the processing pipeline. Additionally, contributed to the design of REST APIs, the implementation of IP modules, and system-level debugging efforts, ensuring that all components interoperate seamlessly and meet deliverability standards.
- **Ziling Zhu** Contributed to FPGA design and vision IP development, and participated in the implementation and deployment of the REST API service for runtime control. Responsible for the on-board execution and testing of the code. Implemented the hardware connection and operational debugging of the OV5640 camera, and resolved video color mismatch issues.

In addition to individual responsibilities, all team members collaborated closely during the integration and debugging phases, particularly when resolving cross-module issues such as AXI interface mismatches and system-level synchronization problems.

6 Conclusion & Future Work

6.1 Conclusion

This project presents a hardware–software co-designed real-time vision processing system implemented on the PYNQ-Z2 platform. The design integrates an OV5640 camera with an FPGA-based video processing pipeline, where streaming data is processed through AXI4-Stream interfaces and managed via AXI4-Lite control. Computationally intensive operations, including color space conversion, centroid calculation, and pixel replacement, are implemented in programmable logic to ensure deterministic latency and high throughput. In parallel, a Python-based REST API running on the processing system enables runtime parameter configuration without interrupting the video stream.

The implemented system achieves stable real-time performance for color-based object detection and dynamic pixel manipulation, demonstrating the feasibility of combining FPGA acceleration with software-level control for interactive vision applications. The results highlight the importance of efficient dataflow design, consistent color format handling, and reliable peripheral integration in building end-to-end embedded vision systems.

6.2 Future Work

Future work will focus on improving both the visual quality and functional capability of the system. First, integrating Automatic White Balance (AWB) and Automatic Exposure (AE) control mechanisms at the camera or processing stage would enhance color accuracy and brightness consistency under varying lighting conditions.

Second, the addition of a dedicated filtering module within the FPGA pipeline could reduce image noise and improve overall visual clarity, which is particularly important for robust object detection in real-time scenarios.

Finally, extending the current color-based processing approach by incorporating edge detection techniques, such as a Sobel operator, would enable more advanced object-level manipulation. This enhancement would allow the system to identify and replace entire objects based on structural features rather than relying solely on color thresholds, significantly expanding its applicability to more complex vision tasks.

References

[1] *Vivado IP Library Reference*. `https://digilent.com/reference/vivado/library`

[2] *PS/PL Interface*. `https://pynq.readthedocs.io/en/latest/overlay_design_methodology/pspl_interface.html`

[3] textitGuide to Video Object Tracking Algorithms. `https://encord.com/blog/video-object-tracking-algorithms`

[4] *RGB to HSV Color Conversion Algorithm.* https://math.stackexchange.com/questions/556341/rgb-to-hsv-color-conversion-algorithm

[5] *ov5640 datasheet.* https://cdn.sparkfun.com/datasheets/Sensors/LightImaging/OV5640_datasheet.pdf

Appendices

Github: https://github.com/WashU-CSE4602-SP26/cse4602-team-baoyang-chunghao-minghe-ziling

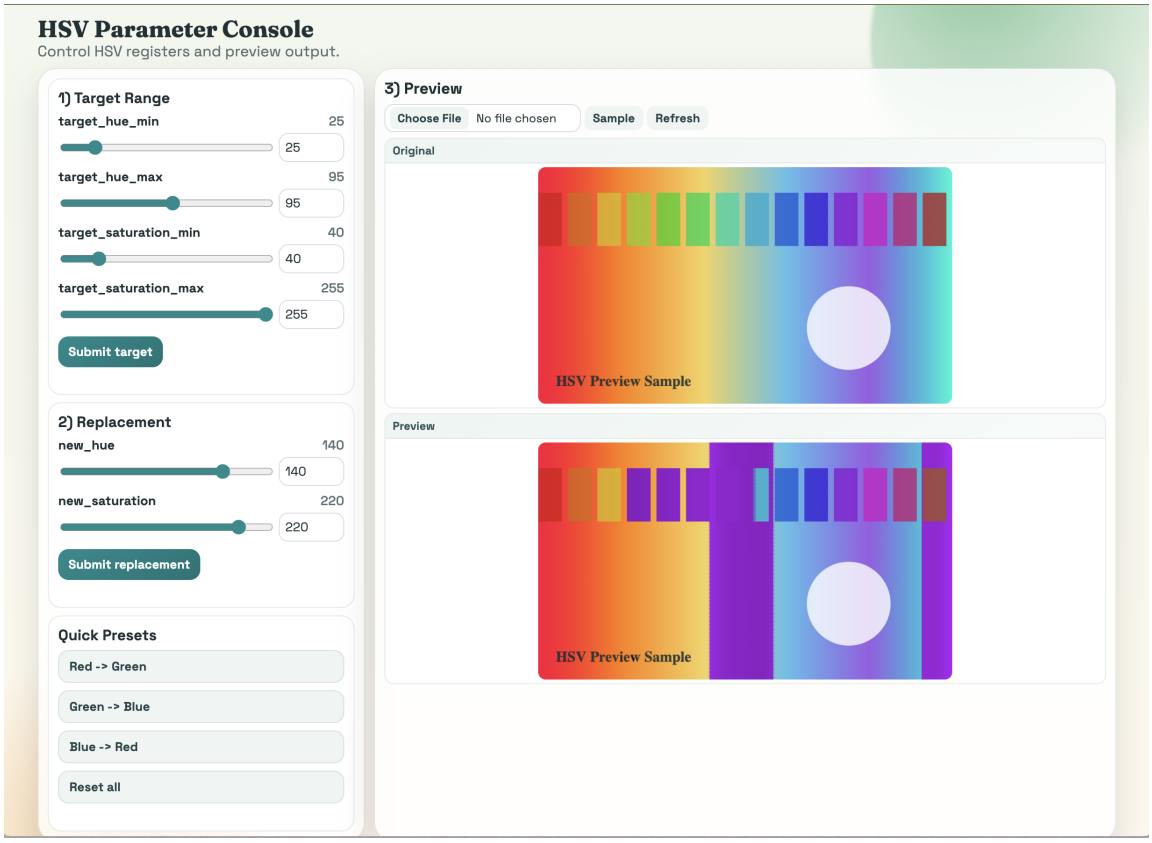

Figure 9: User Interface.