Hardware-Accelerated Image Filter with PYNQ Board

CSE 4602 Capstone Design Project Final Report
Submitted to Professor Hall and the Department of
Computer Science & Engineering
May 1, 2026

Client:
Computer Science & Engineering Department

Advisor:
Michael Hall - mhall24@wustl.edu

Project Duration:
Jan 2026-April 2026

Engineers:

Anthony Stewart
anthony.stewart@wustl.edu
Computer Science & Engineering Department
B.S in Computer Engineering

Angelina Viviano
a.m.viviano@wustl.edu
Computer Science & Engineering Department
B.S./M.S. in Computer Science
B.S. in Comp Engineering

David Kim
d.h.kim@wustl.edu
Computer Science & Engineering Department
B.S./M.S. in Computer Science
B.S. in Comp Engineering

Washington University in St. Louis
McKelvey School of Engineering

Abstract

This project presents the design and implementation of a hardware-accelerated image filtering system on the PYNQ FPGA platform. The system enables users to upload images via a web-based interface, select from a suite of filtering operations, including blur and sharpen, and receive the processed results through a RESTful API. Core filtering algorithms are offloaded to custom hardware modules on the FPGA, significantly reducing processing latency and increasing throughput compared to conventional CPU-based approaches. The end-to-end pipeline integrates hardware acceleration, software control, and user interaction, providing a seamless experience. System performance is rigorously evaluated in terms of latency, throughput, and resource utilization, demonstrating the advantages of FPGA-based acceleration for real-time image processing applications.

Introduction

There is an increasing demand for lower latency and higher throughput in modern image processing applications. Prior work has explored both software-based and hardware-accelerated solutions, with FPGAs and GPUs offering significant speedups for computationally intensive tasks such as filtering and enhancement. However, many existing systems are either inflexible, difficult to use, or require specialized expertise, limiting their accessibility for general users. As image and video workloads grow in size and complexity, traditional CPU-based approaches struggle to provide real-time responses or efficiently handle large data volumes, leaving a gap for accessible, high-performance solutions.

This project addresses the lack of accessible, hardware-accelerated pipelines that allow users to process images and video with minimal delay, while maintaining flexibility in filter selection and system control. The key challenge is to design a system that combines the speed of FPGA-based computation with the usability of modern web APIs, meeting constraints on latency, throughput, and user accessibility.

The primary objectives are to: (1) Enable efficient image filtering with low latency using FPGA acceleration, (2) Support dynamic filter selection and image input/output through a RESTful API, and (3) Evaluate system performance in terms of latency, throughput, and resource utilization. Functional requirements include support for multiple filter types (e.g., blur, sharpen), a user-friendly web interface, and robust API endpoints. Non-functional requirements include real-time performance, scalability, and ease of deployment.

One objective from our original project proposal that was not completed was the extension to real-time video filtering using HDMI input and output. Due to time constraints, we prioritized

refining and optimizing the image filtering pipeline, and were unable to fully implement and test the real-time video processing component.

System Design and Implementation

System Overview

The project implements a hardware-accelerated image processing pipeline using a PYNQ FPGA board, a Flask-based backend server, and a web-based user interface. The system enables users to upload images, select processing filters (e.g., blur, sharpen), and receive processed results, leveraging FPGA acceleration for efficient computation.

Major Components and Interactions

- Frontend Web UI
  - A browser-based interface allowing users to upload images, select filters and strength, and view results. Communicates with the backend via REST API calls.
- Proxy Server (upload_client.py)
  - A Flask application running on the user's laptop, serving the UI and forwarding API requests to the backend on the PYNQ board. This enables seamless access regardless of network configuration.
- Backend Server (PYNQ, server.py)
  - A Flask server running on the PYNQ board. Handles image uploads, filter selection, and manages the FPGA hardware for image processing. Stores both original and processed images, and maintains upload history.
- FPGA Hardware Accelerator
  - Custom logic implemented on the PYNQ board (using Vivado and HDL), performing image filtering operations. Communicates with the backend via DMA and MMIO registers.
- Storage
  - Images and metadata are persisted on the PYNQ board for retrieval and history tracking.

Component Interaction Flow

1. The user uploads an image and selects a filter via the web UI.
2. The proxy server forwards the request to the backend on the PYNQ board.
3. The backend stores the image, configures the FPGA, and streams image data to the hardware accelerator.
4. The FPGA processes the image and returns the result to the backend.
5. The backend stores the processed image and returns it to the user via the proxy and UI.
6. Users can view, download, or delete previous uploads and results.

To illustrate this flow, we have the below block diagram:

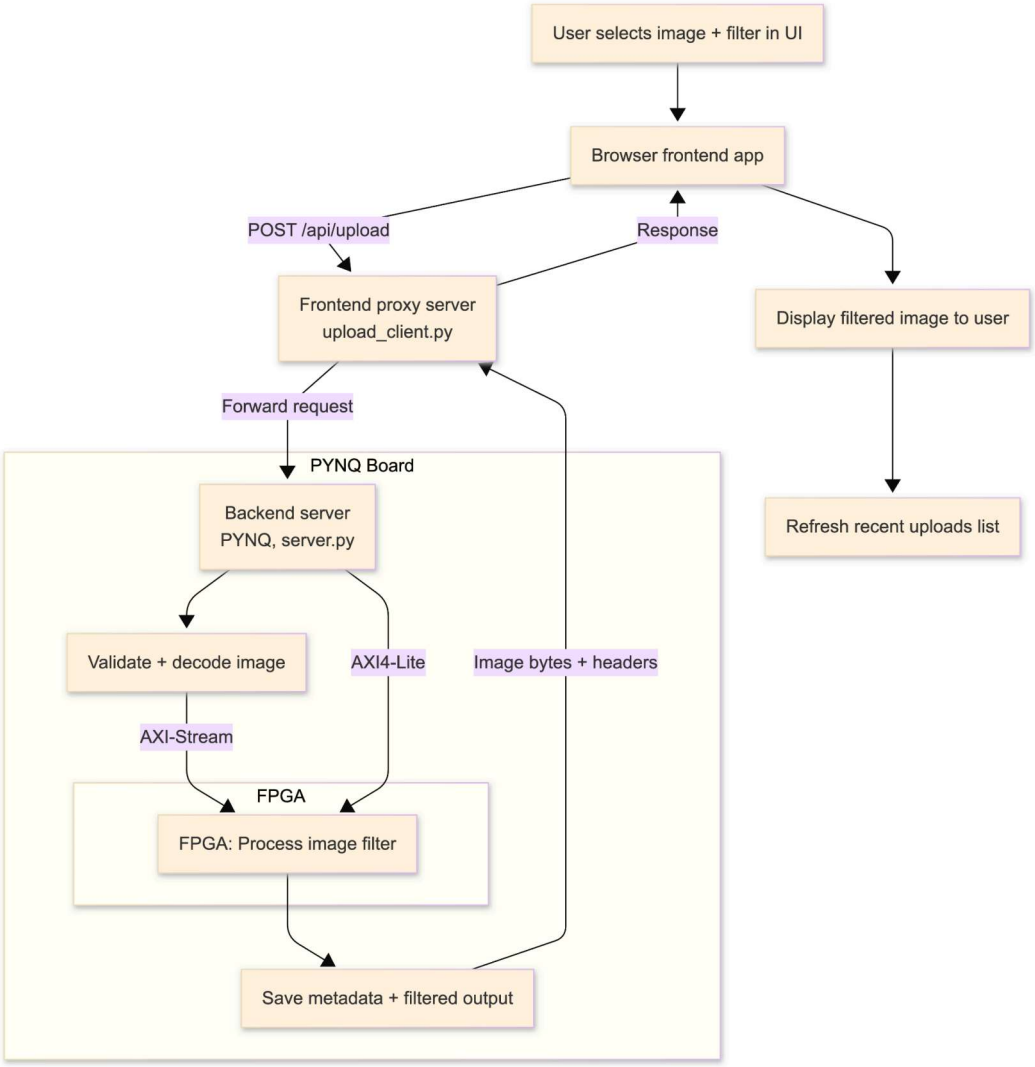


Figure 1: Component Interaction Block Diagram

Design Approach

The design of this system was driven by the goal of achieving efficient, real-time image filtering using FPGA acceleration, while maintaining flexibility and ease of use through a modern web-based interface. One of the most significant architectural choices was the decision to implement image filtering on the FPGA using a two-pass, 1D convolution approach, rather than a direct 2D convolution. This decision was motivated by both hardware resource constraints and the mathematical properties of many common image filters.

In traditional 2D convolution, each output pixel is computed as a weighted sum of its surrounding neighbors, requiring a two-dimensional kernel. However, many useful filters,such as

Gaussian blur and box blur,are separable, meaning their 2D kernel can be decomposed into two 1D kernels: one applied horizontally and one vertically. By leveraging this property, the system first applies a 1D convolution across each row of the image, producing an intermediate result, and then applies a second 1D convolution down each column. This two-pass method dramatically reduces the number of multiplications and additions required per pixel, lowering the computational load on the FPGA and simplifying the hardware design. For example, a 3×3 2D convolution would require nine multiplications per pixel, while two 1D convolutions of length three require only six. This efficiency gain allowed the design to fit within the available FPGA resources and operate at higher speeds. To clarify the process of the two-pass 1D convolution, the following diagram illustrates the data flow:

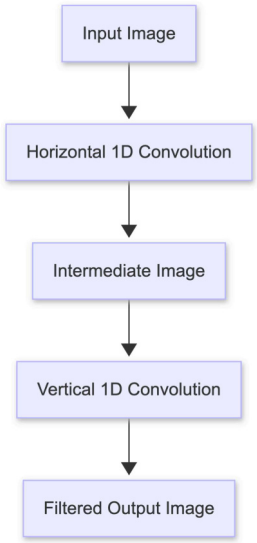


Figure 2: Two-Pass 1D Convolution Flow

Another key aspect of the design was the choice of data layout for transferring images between the backend and the FPGA. Images are packed into a planar format, where all red channel values are stored contiguously, followed by green and blue channels. This layout aligns with the memory access patterns of the hardware accelerator and enables efficient use of the FPGA's DMA engine. The backend software is responsible for converting standard image formats into this planar representation before sending data to the FPGA, and for unpacking the results after processing.

Several trade-offs were considered during the design process. While the two-pass 1D convolution approach is highly efficient for separable filters, it does not support non-separable filters, which would require a more complex and resource-intensive 2D convolution implementation. The team prioritized performance and resource efficiency for the most common use cases, accepting this limitation. Additionally, the decision to handle pre- and post-processing in software, rather than hardware, was made to maximize flexibility and simplify hardware design, at the cost of some additional data movement.

Alternative approaches, such as implementing a full 2D convolution in hardware or performing all filtering in software, were evaluated. A hardware 2D convolution would have provided greater flexibility but exceeded the available FPGA resources and increased design complexity. A software-only solution would have been easier to implement but would not meet the performance requirements for real-time or large-scale image processing.

The system's RESTful API architecture further supports modularity and extensibility, allowing the frontend, proxy, and backend to communicate cleanly and enabling future enhancements. The backend dynamically configures the FPGA by writing filter selection and strength parameters to hardware registers, allowing users to experiment with different filters and settings without reprogramming the FPGA.

In summary, the design approach balances hardware efficiency, system flexibility, and user accessibility. The use of a two-pass 1D convolution for separable filters, planar data layout, and a clear software/hardware partitioning enables high-performance image processing while keeping the system maintainable and extensible.

Implementation

The software side of the system was built to connect the user-facing interface, backend API, image storage system, and filtering pipeline into one complete end-to-end workflow. The frontend web interface allows users to upload a PNG or JPEG image, select a filter type such as blur or sharpen, adjust the filter strength, and view the processed output. This interface communicates with the backend through REST API calls, making the system easier to use without requiring the user to directly interact with the PYNQ board or hardware-level tools.

The backend was implemented using Flask. The system includes a proxy server running on the user's laptop and a backend server running on the PYNQ board. The proxy server serves the web UI and forwards requests to the PYNQ backend, while the backend handles image uploads, filter selection, metadata tracking, image storage, and communication with either the software filter path or the FPGA hardware path. Uploaded images and processed results are stored with metadata so users can view previous outputs, compare results, and delete older uploads when needed.

A major part of the software implementation was preparing images for both software and hardware processing. Standard image files had to be decoded into usable pixel data, optionally resized or downscaled, converted into the correct RGB format, and then either processed directly in software or packed for transfer to the FPGA. After processing, the result had to be converted back into a standard image format and returned to the frontend. This made the software layer responsible for the full image input/output pipeline, even when the main filtering computation was performed in hardware.

The software implementation also included a CPU-based version of the blur and sharpen filters. This software path served two purposes. First, it allowed the team to test the full upload, decode, filter, encode, and return workflow before the FPGA implementation was fully integrated. Second, it provided a baseline for comparing the hardware-accelerated version against a traditional software-only implementation. This was important because the project was not only focused on whether the FPGA filter worked, but also whether the complete system improved end-to-end performance.

Testing and debugging were done throughout development instead of only at the end. Early testing focused on verifying that images could be uploaded correctly, that the frontend sent the correct filter parameters, and that the backend returned a valid processed image. Later testing focused on timing and performance. The backend was instrumented to report timing values for different stages of the pipeline, including decode time, filter time, encode time, and total server processing time. These values were returned through HTTP response headers and collected by benchmark scripts. This made it easier to identify whether latency was coming from the actual filter computation or from surrounding steps such as image decoding, resizing, encoding, or data transfer.

The filtering system was built using SystemVerilog and is deployed on the PYNQ board. The design uses two interfaces. One is AXI4-Lite that passes in parameters and the other one is AXI4-Stream that passes in image pixel data. To filter, we use a convolution approach. A full 2D convolution requires averaging every pixel around its neighbours, which in streaming architecture would require buffering entire rows before computing. Like mentioned before, it will require significant latency and register cost. Instead, we used a smarter approach where the 2D blurring can be decomposed into a horizontal pass followed by a vertical pass which essentially produces the same result. Therefore, the only design we need to implement is the horizontal filter, where each output pixel is the weighted average of its horizontal neighbors within the same row.

Three control parameters are passed over AXI4-Lite. The first is img_width where it tracks row boundaries to prevent cross row bleeding. We do not want to have a situation where a pixel at the end of one row is averaged with the first pixel of the next row. The next is filter_strength where it selects the kernel size. We set it so that any values ≤ 25 maps to kernel size of 3, ≤ 50 maps to size of 5, ≤ 75 maps to 7, and the rest maps to 9. Lastly, we have filter_type where 0 selects blur and 1 selects sharpen.

Because data arrives one pixel per cycle, the filter must first accumulate enough numbers before it can produce a valid output. A kernel size of k requires (k-1)/2 cycles of preparation, where previous pixel values are stored in a shift register queue. Boundary pixels are handled using clamping, so the first and last pixel of each row will be duplicated to fill missing neighbors rather than padding with all zeros. One key edge case we encountered was that once the last pixel

arrives as input, the pipeline still has (k-1)/2 outputs pending in the shift register. We created an endbit counter that uses the tlast signal and counts down each time to flush out the final outputs with correctly clamped averages to the output stream.

For sharpening, the same blur pipeline is reused here. We applied a technique called unsharp masking where it amplifies the difference between the center and its neighbours by using this formula: sharpened = center + (center − blurred).

To average, we also used a technique called multiply and shift approximation. Dividing by a power of 2 in hardware can be done using a right bit shift. However, kernel sizes are always odd numbers and using regular division is computationally expensive in hardware. We can instead approximate division by finding a fraction whose denominator is a power of 2, allowing the division to be replaced by a shift register. For example, we can divide by 3 using *85 >> 8 which is 85/256 = 0.332 ≈ /3. The more shifts we use, the closer we can get with our approximation.

For testing our hardware, we verified it using testbenches, where it initiates our image_manipulation module as the DUT. In the testbench, we first configured all the parameters before streaming began. Then we drove each pixel over the AXI stream and monitored the computed output value and whether tlast is asserted. Using the testbench, we verified many things with our system. First, we made sure to skip the first (k-1)/2 outputs during preparation. We also checked whether the clamping worked on the boundary pixels. Lastly, we checked if the last outputs have been flushed out to the output and that the tlast fires only once the output is ready to be streamed out. In the testbench, we streamed multiple pixels with different image widths and kernel sizes to check whether the behavior matches with our expectations. Once we tested that it worked, we also created a python test script in our Jupyter Notebook from the PYNQ board to check if the horizontal filtering worked based on the parameter we set using AXI4-Lite.

Evaluation & Results

Testing and Data Collection

The system was tested using an end-to-end benchmarking approach. The main endpoint tested was POST /upload, which represents the complete user workflow: a client uploads an image, the server decodes it, applies the selected filter, encodes the result, and returns the processed image. This allowed testing to reflect the real behavior of the system instead of only measuring one isolated function.

The test inputs were organized into multiple image size tiers, including small, medium, large, and extra-large images. These tiers were used to evaluate how the system scaled as image size increased. This was important because image processing performance depends heavily on both file size and pixel resolution. Larger images require more decoding work, more memory movement, more filtering operations, and more encoding work.

A benchmark script was created to automate testing. Before running timed trials, the script first checked the /health endpoint to confirm that the server was active. It then sent warm-up requests followed by repeated timed requests for each test image. For every request, the script measured client-side round-trip time, which represents the time from sending the image to receiving the full processed response. The script also collected server-reported timing values from HTTP headers, including decode time, filter time, encode time, total server time, downscale information, and filter mode. These results were saved in a structured format so they could be analyzed and converted into tables or charts.

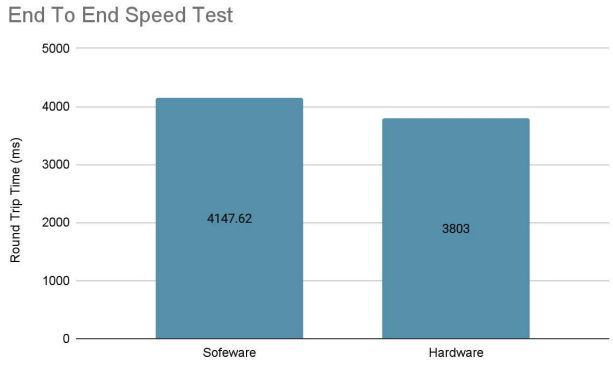
The main performance metrics were end-to-end latency, server-side processing time, filter time, decode time, encode time, and throughput/bandwidth-related behavior. Latency was used to evaluate how responsive the system felt to a user. Throughput and bandwidth were used to understand how much image data the system could move and process over time. The comparison between the software and hardware paths helped show whether FPGA acceleration improved the full system or only the filtering stage.

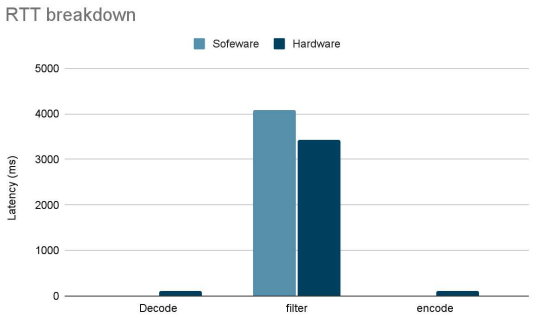
Results

The results showed that the system successfully supported the full image filtering workflow from user upload to processed output. In the final demonstration, users were able to upload an image through the web interface, select a filter and strength, send the request through the REST API, and view the filtered result returned by the system. This demonstrated that the frontend, proxy server, backend server, software processing path, hardware processing path, and storage/history features were integrated into a working pipeline.

The benchmarking results showed that image size and downscaling had a major effect on end-to-end latency. In the software path, reducing the working image size significantly improved response time. For example, using a medium image scaled to approximately 300×400 produced an average round-trip time of about 625 ms, with the filter stage taking about 560 ms. Increasing the working size to approximately 600×800 increased the average round-trip time to about 2,560 ms, with the filter stage taking around 2 seconds. This showed that the number of pixels being processed had a direct impact on system latency.

The results also showed that hardware acceleration improved the core filtering operation, but did not automatically reduce total end-to-end latency by the same amount. The FPGA filter itself was faster than the software equivalent, but the full system still included decoding, encoding, RGB conversion, planar packing and unpacking, DMA transfer, and API overhead. Because of this, the total user-facing latency was closer between the hardware and software paths than expected. This was one of the most important findings of the project because it showed that improving the main computation is not enough by itself. In a complete embedded system, the surrounding data movement and software infrastructure can become a major performance bottleneck.





Overall, the evaluation showed that the system met its core requirements: it accepted user image uploads, supported configurable filters, returned processed images, stored output history, and collected timing data for performance analysis. The main limitation was that end-to-end performance depended heavily on preprocessing and data movement, not only the speed of the FPGA filtering logic.

Discussion & Reflection

Like mentioned before, one strength of our hardware design is the two pass convolution. Applying the horizontal filter twice to achieve 2D filtering makes the hardware design much more manageable while still producing the correct results. Another strength is that we used AXI4-Lite to pass in parameters such as filter type, strength, and image width. By doing this, we can configure it in our software without reloading the bitstream every time. Lastly, the hardware convolution logic runs significantly faster than the software equivalent, but with the preprocessing overhead such as decoding/encoding, RGB conversion, planar packing, and DMA transfers, it brings the overall end to end latency to be very similar to the software path.

On the software side, one strength of our design is that the system connects the user interface, REST API, image storage, and processing pipeline into one complete workflow. The web interface makes the FPGA system easier to use by allowing users to upload an image, choose a filter, adjust the strength, and view the processed result without needing to interact directly with the hardware. Another strength is that the software pipeline collects detailed timing data for each stage, including decode time, filter time, encode time, total server time, and round-trip time. This made it possible to identify where latency was coming from instead of only measuring the final output. The software implementation also gave us a CPU baseline that could be compared against the FPGA path. From testing, we found that image size and downscaling had the largest impact on software latency, while output JPEG quality mainly affected response size and bandwidth. Overall, the software side was valuable because it made the system usable, testable, and measurable, while also showing that end-to-end performance depends on the entire pipeline, not just the filtering operation.

For ethical considerations, our current design stores uploaded images, up to 12, on our server side with metadata. To ensure privacy, maybe it is better to explicitly notify the users that the images they upload will be retained unless they delete it themselves. As for data security, we have no code for any authentication at all. This means that any user on the same network could potentially breach and access stored images. As of now, we have no mitigation to address this, so this is one of our project’s limitations. Furthermore, attackers can flood the server and FPGA by uploading repeatedly, as the system has no rate limiting on the upload endpoint. We do have file size and image resolution limits that reduce the cost of each individual request, but they do not prevent a high volume of requests that can potentially be misused to degrade performance. Last

thing we didn't consider is filtering out harmful content. Our current system performs no content screening on uploaded images. While the filtering we are using is safe, the architecture of our system can be extended to process inappropriate or harmful imagery. It will definitely be safer if we have content filtering at the upload stage, though it may be a stretch with our current project goal.

Overall, the 2 pass convolution approach worked correctly and produced valid filtered output across all tested image sizes. It allowed us to make the hardware logic very manageable without sacrificing the quality of our filtering. The AXI4-Lite also worked reliably, where based on our configuration, it did the expected behavior without any changes in our bitstream. Using the RGB layout also made the hardware logic much more straightforward since all pixels of each channel are sequential in the buffer. Lastly, the testbenches we created really worked out in catching filter issues early before deploying into the board as a bitstream.

Currently, only blur and sharpen are supported with limited fixed kernel sizes. Adding new filter types or custom kernels may be an improvement for our current design. Even better, we can make an extensible and flexible design that would allow users to manipulate kernel coefficients to be loaded through AXI4-Lite so that they can try new filters without touching hardware, though it will require significant and complicated changes to support this. Additionally, the AXI4-Stream uses 32 bit data to carry a single pixel value, where each pixel has an 8 bit value. This wastes a lot of bus bandwidth per transfer. Perhaps packing multiple pixels into a single 32 bit would significantly reduce latency, especially for larger images. Like again, this would considerably complicate the hardware logic, so there is a tradeoff between optimizing performance and the complexity of hardware design.

On the software side, the current system could be improved by reducing the amount of preprocessing and data movement required before and after filtering. The pipeline currently performs several steps, including image decoding, optional resizing, RGB conversion, planar packing and unpacking, and re-encoding. These steps make the system flexible, but they also add overhead to the end-to-end latency. A future improvement would be to streamline this process by reducing unnecessary format conversions and making the transfer between software and hardware more direct.

Another improvement would be adding more control over image size and compression settings. Our testing showed that downscaling had the largest impact on software latency, while JPEG quality mainly affected response size and bandwidth usage. Because of this, the system could let users choose between faster processing, higher image quality, or lower bandwidth usage depending on their needs. This would make the software pipeline more adaptable instead of using one fixed processing configuration.

The software system would also need more security and reliability features before being used outside of a local demo. Currently, the system does not include authentication, user-specific storage, rate limiting, or content screening. Adding these features would help prevent unauthorized access, repeated upload abuse, and accidental exposure of stored images. Overall, the main software tradeoff is between keeping the system simple and flexible for a class project versus making it optimized, secure, and production-ready.

Project Management

The original plan was to build a hardware-accelerated image processing system that could process images or video using the PYNQ FPGA platform. The early project goal included real-time video filtering through HDMI input and output, but this objective was reduced because of time constraints and hardware integration complexity. The team instead focused on delivering a complete image filtering pipeline with a working frontend, REST API, backend server, software baseline, FPGA filtering logic, and performance evaluation.

In the first phase of the project, the team focused on defining the system architecture and dividing responsibilities between frontend software, backend software, and FPGA hardware. The main deliverables during this phase were the project proposal, high-level block diagrams, interface planning, and early API design. During this stage, the team clarified how images would move from the user interface to the backend and eventually to the FPGA.

In the second phase, the team built the software pipeline and hardware filtering logic in parallel. The software side focused on image upload, filter controls, API routing, image storage, metadata tracking, and CPU-based filter testing. The hardware side focused on the SystemVerilog filtering module, AXI4-Lite control interface, AXI4-Stream image data interface, clamping logic, and blur/sharpen behavior. This phase produced the first working end-to-end software workflow and the first hardware filter simulations.

In the third phase, the team focused on integration and debugging. The software and hardware paths were connected through the PYNQ board, and the team worked through issues related to image formatting, RGB layout, planar packing, DMA transfer, and matching software expectations with hardware behavior. Benchmarking was also added during this phase so the team could collect timing data instead of only verifying visual correctness.

In the final phase, the team focused on evaluation, presentation, and documentation. The completed deliverables included a working web-based image upload interface, a Flask proxy server, a Flask backend server on the PYNQ board, software blur and sharpen filters, FPGA blur and sharpen support, image history and metadata storage, benchmark scripts, latency measurements, hardware testbenches, final presentation materials, and this final report. The main

incomplete deliverable was real-time HDMI video filtering, which was removed from the final scope so the team could finish and evaluate the image filtering system more thoroughly.

David Kim - Implemented the hardware horizontal filter in SystemVerilog, including the clamping logic, blur and sharpen, shift register queue, and AXI4-Lite control registers for configuration of filter type, strength, and image width. Developed the hardware testbenches for both AXI4-Lite register validation and full image simulation.

Angelina Viviano - Implemented the hardware software integration layer, such as RGB format conversion and planar memory packing and unpacking. Developed CPU-based blurring and sharpening to directly compare against the hardware version. Helped David with the AXI4-Stream input and output interfaces for data transfer between the processing system to the FPGA.

Anthony Stewart - Implemented the UI for image uploading and displaying filtered results. Contributed to the Flask API backend for image storage and metadata tracking. Designed the performance benchmarking infrastructure, including benchmark scripts, displaying latency for each part, and analysis of end-to-end latency and throughput across both hardware and software paths.

Conclusion & Future Work

In conclusion, we successfully delivered an image filtering pipeline using our FPGA hardware design with a web-accessible page through the REST API. The system supports different filter types, including blur and sharpen, which users can select from the browser UI. Our system also stores images, metadata, and detailed end-to-end timing data that breaks down each stage of the pipeline for analysis.

The most significant finding from the hardware side is that hardware acceleration does not automatically improve end-to-end pipeline latency. Although the FPGA filtering logic can execute the core convolution faster than a software equivalent, the preprocessing required to use the hardware path, including decoding, encoding, RGB conversion, planar packing, and DMA transfers, increases the total latency. As a result, the full hardware pipeline had latency closer to the software implementation than expected.

On the software side, we found that image size and downscaling had the largest impact on latency. Smaller working resolutions produced much faster response times, while larger images caused filter time and round-trip time to increase significantly. We also found that JPEG quality mainly affected response size and bandwidth usage rather than the main computation time. These

results demonstrate that in an integrated system pipeline, we should not only focus on improving the main computation, but also the surrounding infrastructure. This is our main takeaway and will help us approach embedded system design more effectively in the future.

For future work, we could make the hardware design more extensible by allowing users to add new filters and kernel sizes. One possible improvement would be allowing custom kernel coefficients to be loaded through AXI4-Lite, so users could experiment with new filters without changing the hardware design. Another hardware improvement would be packing multiple pixels into the 32-bit AXI stream instead of using one 32-bit transfer for a single 8-bit pixel. This would better utilize bus bandwidth and could reduce latency for larger images, although it would also make the hardware logic more complex.

On the software side, future work should focus on reducing preprocessing overhead and improving system reliability. The current pipeline performs several steps before and after filtering, including decoding, resizing, RGB conversion, packing, unpacking, and re-encoding. Reducing unnecessary conversions or streaming data more directly between software and hardware could improve end-to-end latency. The system could also provide user-selectable performance modes, such as faster processing, higher image quality, or lower bandwidth usage. To make the system more complete and production-ready, future versions should include authentication, rate limiting, stronger input validation, clearer image retention controls, and better handling of large uploads.

References

PYNQ Documentation, PYNQ Project, https://pynq.readthedocs.io/
Flask Documentation, Pallets Projects, https://flask.palletsprojects.com/
AMD/Xilinx, Vivado Design Suite User Guide and AXI Reference Documentation, https://docs.amd.com/
NumPy Documentation, https://numpy.org/doc/
Pillow Image Library Documentation, https://pillow.readthedocs.io/
"Unsharp Masking," Wikipedia, Wikimedia Foundation, 16 Mar. 2026, https://en.wikipedia.org/wiki/Unsharp_masking.

Appendices

Source code repository:
https://github.com/WashU-CSE4602-SP26/cse4602-team-angelina-anthony-david-capstone