Accelerating 2D Convolution with SoC FPGA and RESTful Interface

CSE 462M Capstone Design Project Final Report
Submitted to Professor Hall and the Department of Computer Science & Engineering

Advisor:
Michael Hall[1] - mhall24@wustl.edu

Group Members:
Zichu Pan[2] - p.zichu@wustl.edu
Bruce Li[3] - liyifei@wustl.edu
Weikai Rao[2] - r.weikai@wustl.edu

Washington University in St. Louis
McKelvey School of Engineering

May 2, 2025
January, 2025 – May, 2025

[1] Lecturer, Department of Computer Science & Engineering
[2] Candidate for B.S. in Computer Engineering
[3] Candidate for B.S. in Computer Engineering with minor in Psychological and Brain Sciences

1 Abstract

This project addresses the inefficiency of performing 2D convolution operations on general-purpose CPUs, particularly for real-time image and video processing tasks. Convolution is a core technique in computer vision but becomes computationally expensive at scale. To tackle this, we designed and implemented a hybrid hardware-software solution using the Xilinx PYNQ-Z2 platform, which integrates an ARM processor with FPGA fabric. Our approach offloads compute-intensive convolution to a custom hardware accelerator while the ARM processor manages control and I/O via a Django-based RESTful API. The system supports image upload, processing via FPGA, and returns results through an intuitive web interface. To further explore the performance potential, we developed a direct video processing pipeline using AXI4-Stream and HDMI interfaces, enabling real-time video filtering entirely within programmable logic. Technical methods include high-level synthesis (HLS) optimizations such as pipelining, loop unrolling, and a circular line buffer. The final system demonstrates significant speedup over software-based solutions, minimal user setup, and practical usability for real-world applications.

2 Introduction

2.1 Background Information

Convolution is a fundamental operation in digital image processing and computer vision. It is used extensively in applications such as object detection, edge enhancement, facial recognition, medical imaging, and autonomous vehicle perception. The convolution operation typically involves sliding a small matrix (called a kernel or filter) over an image and performing element-wise multiplications and summations[2]. This process is repeated across all pixels and channels of an image, making it computationally intensive, especially for high-resolution images or large-scale datasets.

In recent years, demand for real-time image and video processing has surged due to its use in edge computing, robotics, smart surveillance systems, and embedded AI. Traditionally, these operations are executed on CPUs or GPUs. While GPUs offer high parallelism, they are power-hungry and require significant system resources, making them less suitable for embedded or low-power environments. CPUs, on the other hand, often cannot meet the performance requirements without costly horizontal scaling (i.e., adding more servers), which is inefficient for latency-sensitive applications.

FPGAs (Field-Programmable Gate Arrays) provide a compelling alternative. Their ability to implement highly parallel custom logic makes them well-suited for accelerating image processing tasks. Prior research and industry solutions—such as Xilinx's Vitis vision libraries—have shown the potential of FPGA-based acceleration. However, many of these solutions are either difficult to integrate into modern software systems or lack the flexibility and accessibility needed for real-world deployment.

The Xilinx PYNQ-Z2 board addresses some of these limitations by combining an ARM Cortex-A9 processor with programmable logic in a single System-on-Chip (SoC). This architecture enables a hybrid design: the ARM processor can manage high-level tasks and networking, while the FPGA handles low-level, performance-critical operations.

Our project leverages this hybrid architecture to accelerate 2D convolution. We offload convolution to a custom HLS-based FPGA core and expose the functionality through a Django-based RESTful API. This allows developers to access hardware acceleration without needing to interact directly with FPGA toolchains. Additionally, we extend our work to real-time video processing using HDMI passthrough, enabling frame-by-frame filtering entirely within the FPGA fabric.

By combining efficient hardware design with software accessibility, our system offers a cost-effective, low-overhead solution for real-time image and video processing, making FPGA acceleration practical for modern embedded applications.

2.2 Problem Statement

Convolution operations are computationally expensive, particularly in real-time image and video processing applications such as robotics, surveillance, and embedded vision. General-purpose CPUs are not optimized for such data-intensive workloads, and while GPUs offer performance, they are expensive, power-hungry, and unsuitable for low-cost or embedded platforms.

The specific problem this project addresses is: how to offload and accelerate 2D convolution operations using cost-effective, low-power hardware without sacrificing usability or integration ease. To be a practical solution, the system must expose its acceleration capabilities through a software-accessible interface, integrate with existing tools and workflows, and minimize latency during operation.

To solve this, we leverage the Xilinx Zynq-7000 SoC on the PYNQ-Z2 board to implement a custom convolution engine in the FPGA fabric. Control, coordination, and integration are managed by the embedded ARM Cortex-A9 processor. The system provides access via a RESTful API, allowing developers to send images and receive processed outputs with minimal setup. Additionally, we explored a hardware-only video pipeline using HDMI input/output for real-time processing, bypassing the CPU entirely for the pixel data path.

2.3 Project Objectives and Requirements

1. **Design and implement a 2D convolution engine in FPGA hardware** using HLS with optimizations such as pipelining, loop unrolling, and circular buffers.
2. **Develop a RESTful API** on the ARM processor using Django, allowing users to submit images and receive filtered outputs.
3. **Coordinate data flow between ARM and FPGA** using the PYNQ framework and AXI DMA to ensure minimal overhead and efficient throughput.
4. **Create a web-based user interface** that enables simple uploads and displays results to demonstrate ease of use and hardware performance.
5. **Demonstrate direct video processing** using HDMI passthrough and FPGA streaming to showcase real-time performance without CPU intervention.

Sub-Objectives

- Implement grayscale and edge-detection filters.
- Use line buffering and window slicing to enable streaming pixel-wise processing.
- Optimize hardware latency and bandwidth with AXI interface tuning.
- Validate functionality through image and video comparisons with software reference (e.g., SciPy).

Functional Requirements

- Accept image and video input via web API
- Return processed output within a few seconds (non-real-time)
- Support 3×3 filter kernel configuration via software
- Integrate a hardware video pipeline for HDMI input/output

Non-Functional Requirements

- Secure handling of image uploads (no persistent storage, sanitize inputs)
- Modular and maintainable codebase (both hardware and software)
- Responsive UI for ease of demonstration
- Use open-source or freely licensed software tools where possible

2.4 Ethical Considerations

While our project is focused on the technical development of a hardware-accelerated image processing system, we recognize that even neutral technologies can raise important ethical considerations depending on their context of use. Though we did not encounter direct ethical dilemmas during development, we have identified several areas worth considering for future deployment:

- **Data Privacy:** Our system processes user-uploaded images, which may include sensitive or personally identifiable content. In a real-world deployment, it is critical to handle such data securely. This includes using encrypted transmission channels (e.g., HTTPS), avoiding unnecessary data storage, and ensuring that processed data is not retained beyond the duration of a session. Even though our current implementation is local and for demonstration purposes, the handling of image data should follow privacy best practices and relevant data protection regulations (e.g., GDPR).
- **Bias and Misuse:** The convolution accelerator we developed is general-purpose and does not perform classification or facial recognition. However, in broader contexts, it could be integrated into systems that do. In such cases, developers must be mindful of the ethical implications, such as algorithmic bias, privacy intrusion, or misuse in surveillance technologies. Faster hardware can unintentionally enable the deployment of such systems at scale, which amplifies their societal impact.

We emphasize that any future applications of this acceleration system—especially in domains like surveillance, biometrics, or automated decision-making—should adhere to legal frameworks, consider societal impact, and follow ethical guidelines for responsible AI and embedded system deployment.

3 Methods

3.1 System Architecture

Our system is built around a heterogeneous architecture that separates high-level control from low-level computation. It consists of a Processing System (PS) and Programmable Logic (PL) on the PYNQ-Z2 board. The PS runs Linux on a dual-core ARM Cortex-A9 processor[1], while the PL executes hardware-accelerated 2D convolution logic implemented in a custom HLS core. The full system overview is illustrated in Figure 1.

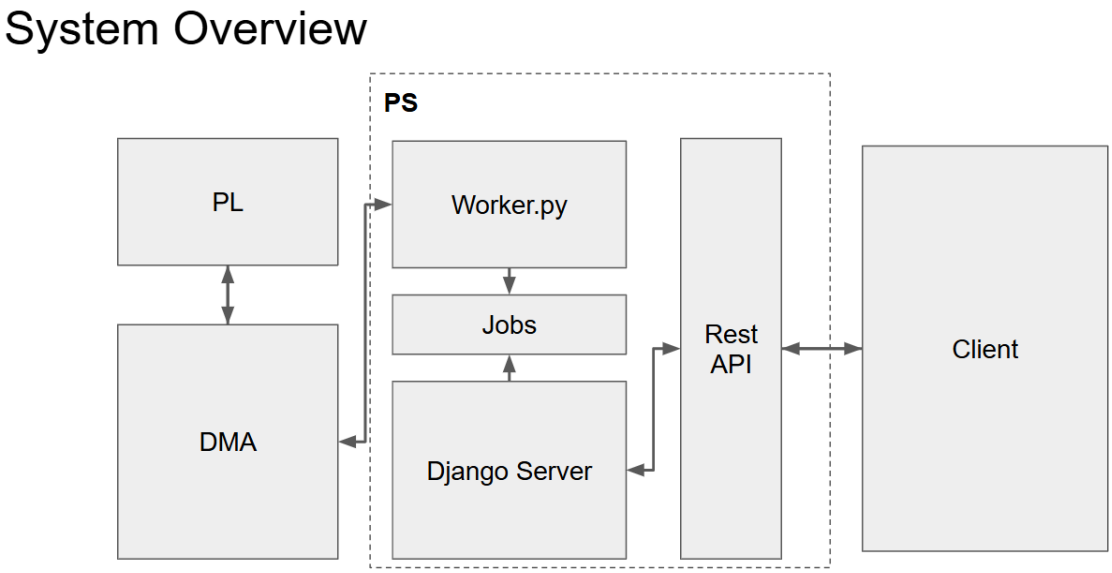


Figure 1: System Overview

Hardware Logic (PL + DMA)
The FPGA fabric contains the following key components:

- A custom 2D convolution IP (developed in HLS) that performs either grayscale conversion or kernel-based filtering.
- An AXI DMA engine that handles data transfer between the PS and PL using AXI4-Stream.

The IP is accessible over AXI-Lite for control (e.g., kernel configuration) and receives pixel data via AXI-Stream. It outputs filtered pixel values back to memory, pipelined for one-pixel-per-cycle performance.

Software Components (PS)
The ARM processor runs a Django server that exposes a RESTful API for clients to upload image data and request filtering. However, to ensure tight control over FPGA interactions, the actual job execution is handled by a dedicated script called worker.py.

Rationel for worker.py
FPGA overlays are not thread-safe. Reloading or sharing DMA and IP contexts across concurrent requests can lead to crashes or undefined behavior. For this reason, we chose to delegate all interaction with the FPGA overlay to a single-threaded Python script (worker.py) that polls the /jobs directory for new tasks.

This design ensures:

- Safe reuse of Overlay, IP, and DMA instances
- Sequential processing of jobs to prevent race conditions
- Separation of concerns between web handling and hardware interfacing

Process life cycle

1. The Client interacts with the REST API to upload a image to be processed
2. The Django server saves an uploaded image and metadata (e.g., selected kernel) in a dedicated job folder.
3. worker.py continuously monitors the /jobs directory. When it detects a new job:
    - It loads the required FPGA bitstream (e.g., grayscale or filter).
    - Parses the kernel or factor filter
    - Starts the DMA transfer and FPGA processing.
4. At this point the image is handed to the PL to be processed. Once the PL finished processing the image, the result is saved and a done.txt file is written as a completion flag.
5. The REST API detects the completion flag and serves the processes image back to the client.

3.2 Design Approach - HLS

Our design methodology was iterative, starting with a basic functional pipeline and optimizing components as we progressed. We highlight some key design decisions and trade-offs:

3.2.1 HLS Optimizations: Circular Buffer and Loop Directives

To accelerate 2D convolution efficiently in hardware, we designed a custom IP using Vitis HLS that exploits both data reuse and parallelism. The convolution IP implements a streaming filter that processes one pixel per cycle using a 3×3 kernel over RGB channels. Several optimization techniques were applied to meet timing constraints and improve throughput. Please refer to the code in Appendix

Circular Line Buffer

A key component is the use of a **circular buffer** for the image's scanlines. Since 2D convolution requires access to a 3×3 neighborhood around each pixel, a naive approach would need to store the entire image, which is memory-intensive and inefficient. Instead, we store only the most recent three rows using a $3 \times$ `MAX_WIDTH` $\times 3$ array. The current pixel's row index modulo 3 determines where new pixel data is written, and previously buffered rows are reused. This drastically reduces memory use and supports a sliding window approach.

```
int row_idx = row % 3;
store(pixel, row_idx, col, line_buffer);
```

The convolution window is then extracted from the buffer by calculating wrapped-around buffer indices and clamping edge cases.

Loop Optimizations with HLS Pragmas

We applied several loop transformation directives:

- `#pragma HLS PIPELINE`: Applied in the main pixel loop and initialization to enable one-pixel-per-cycle throughput.
- `#pragma HLS UNROLL`: Applied to inner loops over color channels (R, G, B) and kernel window dimensions. This allows the hardware to parallelize multiplications for each channel.
- `#pragma HLS INLINE`: Used for small helper functions like `store`, `create_window`, and `apply_kernel` to reduce function call overhead and enable better loop merging.
- `#pragma HLS ARRAY_PARTITION`: Ensures that frequently accessed arrays like `line_buffer` and `window` are partitioned to allow concurrent access to all elements, critical for full pipelining.

Kernel Application and Saturation

The `apply_kernel` function performs the dot product of the kernel and the 3×3 neighborhood for each color channel, accumulating the result in a floating-point sum. After normalization (division by kernel factor), the values are clamped between 0 and 255 to avoid overflow or underflow.

```
sum[c] += float(window[i][j][c]) * kernel[i][j] / kernel_factor;
...
ap_uint<8> r = (sum[0] < 0) ? 0 : ((sum[0] > 255) ? 255 : sum[0]);
```

Streaming Interface

The IP is connected to the Processing System (PS) via AXI4-Stream for high-throughput pixel input/output and AXI4-Lite for control. This design enables real-time processing and seamless integration with the PYNQ framework, where DMA is used to transfer image data between memory and the FPGA core.

Overall, these HLS techniques allowed us to build a compact and high-throughput IP block suitable for real-time image filtering applications.

3.3 Design Approach - Hardware Block Diagram

The complete block diagram including the custom IP generated by HLS implementation above is shown in the following Figure 2.

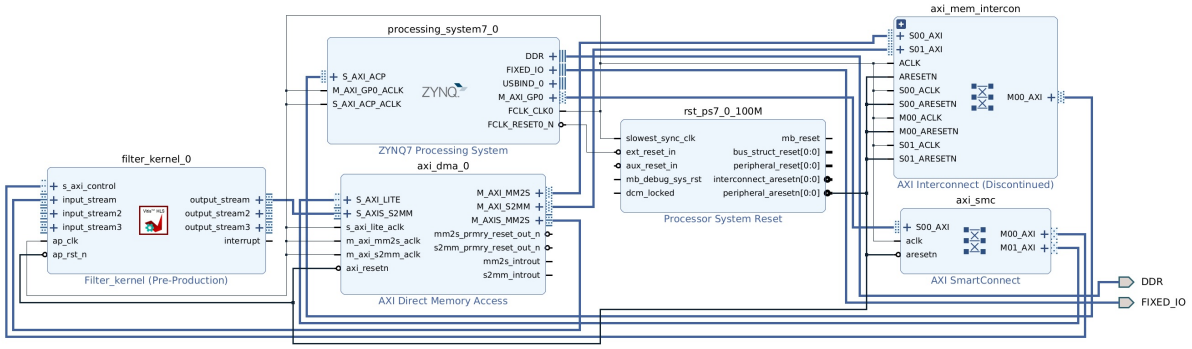


Figure 2: Block Diagram for Design with Single DMA Block

3.3.1 Key Components and Data Flow

The single-DMA prototype is organized around the following blocks:

- **Zynq** Processing System (`processing_system7_0`)

  Provides the ARM processing cores, DDR3 memory controller, and all PS–PL AXI interfaces. The PS drives an AXI GP master for register writes and exposes high-performance port for bulk-data access.

- **AXI DMA** (`axi_dma_0`)

  A Direct Memory Access engine that moves 32-bit packed pixels between DDR (AXI4-MM) and the PL (AXI4-Stream). `MM2S` fetches an input frame for processing; `S2MM` stores the filtered frame back to memory.

- **Custom HLS Kernel (`filter_kernel_0`)**

  A 3×3 RGB convolution core generated with Vitis HLS. It presents one active AXI4-Stream input/output pair and a shared AXI-Lite slave (`s_axi_control`) used to program frame dimensions, kernel coefficients, and the control bits including `ap_start`.

- **AXI SmartConnect (`axi_smc`)**

  Interconnect that arbitrates the three AXI master ports (MM2S, S2MM, and PS DMA) onto the DDR memory bus.

- **AXI Memory-Mapped Interconnect (`axi_mem_intercon`)**

  Lightweight crossbar that routes the PS `M_AXI_GP0` traffic to the DMA control interface and the kernel's AXI-Lite registers.

End-to-end data path:
DDR $\rightarrow$ *MM2S* $\rightarrow$ `input_stream` $\rightarrow$ *filter_kernel_0* $\rightarrow$ `output_stream` $\rightarrow$ *S2MM* $\rightarrow$ DDR.
The ARM core configures the kernel and DMA through AXI-Lite transactions, then polls `ap_done` and the DMA status registers to detect completion.

3.3.2 Data Preparation

Motivation AXI4-Stream in this prototype is configured for a 32-bit data width. To convey RGB color information, every 8-bit RGB triplet is packed into a single 32-bit word on the PS side, transferred through the DMA, processed in hardware, and then unpacked back to three 8-bit channels for display or encoding. The packing and unpacking on both PS and PL side are achieved through bit-shift and mask operations.

Python (PS-Side) Packing Given an OpenCV image or frame `frame_rgb` of shape $(H, W, 3)$ and dtype = *uint*8:

```
packed = ((frame_rgb[:, :, 0].astype('u4') << 16) |  # R << 16
          (frame_rgb[:, :, 1].astype('u4') <<  8) |  # G <<  8
           frame_rgb[:, :, 2].astype('u4'))          # B
```

- The red channel occupies bits `[23:16]`, green for bits `[15:8]`, and blue for bits `[7:0]`.
- The high nibble `[31:24]` is unused and transmitted as zero.
- Each packed word is DMA-written as a contiguous row-major array of shape (H, W) with a total length of $4 \times H \times W$ bytes.

HLS (PL-Side) Unpacking and Re-packing In the HLS code, the word arrives as a defined `AXI_PIXEL` type:

```
typedef ap_axis<32,0,0,0> AXI_PIXEL;  // data[31:0]

ap_uint<8> r = pixel.data(23,16);
ap_uint<8> g = pixel.data(15, 8);
ap_uint<8> b = pixel.data( 7, 0);
```

After convolution the filtered components are clamped to 8 bits and re-packed:

```
AXI_PIXEL out;
out.data(23,16) = r_filt;
out.data(15, 8) = g_filt;
out.data( 7, 0) = b_filt;
out.data(31,24) = 0;      // keep MSB nibble zero
out.last = tlast_flag;    // copied from the incoming stream
output_stream << out;
```

Python Unpacking (Post-DMA) After the PL writes the processed frame back to DDR and the PS `invalidates` the cache lines:

```
r = (out_buf >> 16) & 0xFF
g = (out_buf >>  8) & 0xFF
b =  out_buf        & 0xFF
frame_proc = np.dstack((r, g, b)).astype('uint8')
```

3.3.3 Triple DMA Optimization

The complete block diagram of the optimized design including 3 DMA controller and the updated IP block is shown in the following Figure 3.

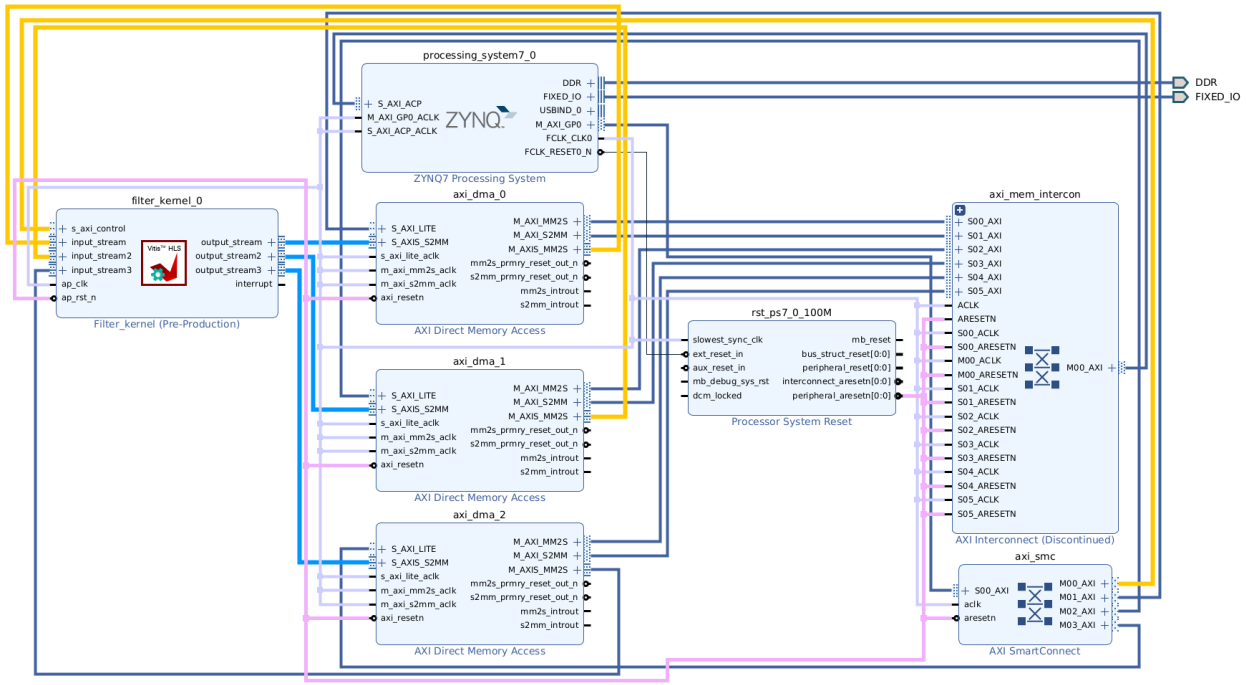


Figure 3: Updated Block Diagram for Design with Triple DMA Block

3.3.4 Key Components and Data Flow - Triple DMA Optimization

Compared with the single-DMA baseline, the revised diagram in Fig. 3 retains the same processing-system, reset, and interconnect blocks but introduces three critical changes:

- **Two Additional AXI DMA Cores (`axi_dma_1`, `axi_dma_2`)**

  Each new DMA instantiates its own MM2S/S2MM channels and a dedicated AXI-Lite control port. This gives the system 3 independent 32-bit AXI-Stream pipes—one per color channel or per video frame.

- **Triple Stream Interface on `filter_kernel_0`.** The two more pair of streaming ports (`input_stream2/3`, `output_stream2/3`) are now connected to `axi_dma_1` and `axi_dma_2`. Enabling the kernel to read and write three pixels per cycle.

3.4 REST API using Django:

We decided to implement a RESTful API using the Django REST Framework for its robustness and ease of use. Alternatives considered included writing a lightweight Flask server or even using the built-in PYNQ Jupyter server; however, Django REST offers a full-featured solution with front-end templates and we were familiar with it. The trade-off is that Django is somewhat heavy for an embedded device, and its multi-threaded nature conflicts with the FPGA overlay management that a separate single-threaded worker.py has to step in.

3.5 Direct Video Processing Architecture and Implementation

To demonstrate real-time or near real-time video processing on the PYNQ-Z2 board, we extended our architecture beyond static images and implemented a pipeline for handling live video data using the onboard HDMI interfaces. The system integrates several AXI IP blocks for capturing, processing, and displaying video frames directly through the FPGA fabric, bypassing the ARM processor during the main pixel data path.

3.5.1 Block Diagram Overview

The complete block diagram is shown in Figure 4. It captures the full video path: from HDMI input, through video processing IP cores, to HDMI output.

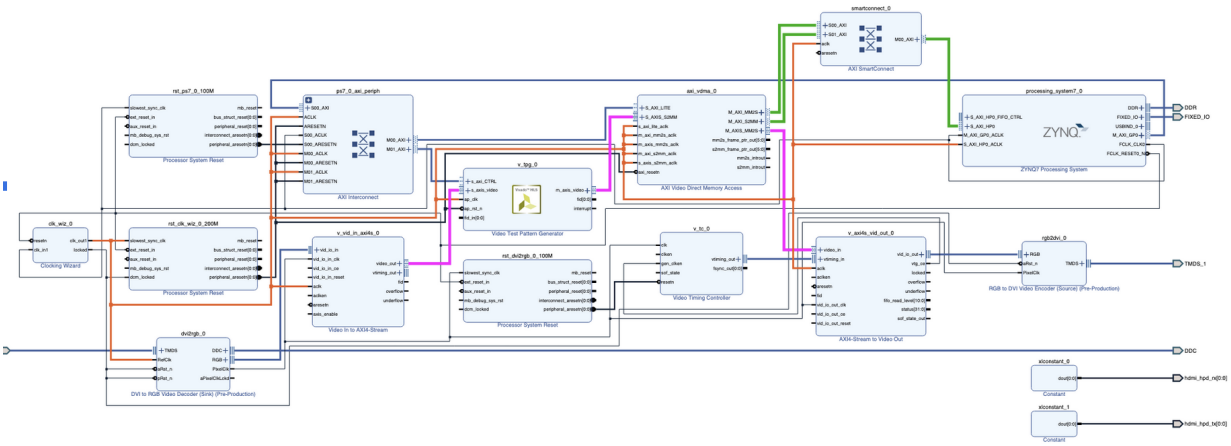


Figure 4: Full Block Diagram for Direct Video Processing

3.5.2 Key Components and Data Flow

The system is built around the following major components:

- **DVI2RGB:** Converts incoming HDMI signals to AXI4-Stream video format. It parses TMDS signals and outputs video and sync signals that can be handled by AXI-compliant video IP cores.
- **VTC (Video Timing Controller):** Generates and manages sync signals needed for frame timing across the video pipeline.
- **AXI VDMA:** Facilitates transfer of video frames to/from DDR memory using AXI4-Memory-Mapped and AXI4-Stream interfaces. Acts as a frame buffer.
- **Custom HLS IP (e.g., Convolution Filter):** Processes streaming video data frame-by-frame. The IP sits between the VDMA and the video output block, applying filtering or transformation to each frame.
- **AXI4-Stream to Video Out:** Converts processed AXI4-Stream video data back into HDMI-compatible timing and pixel format.
- **RGB2DVI:** Converts the video stream into TMDS signals for HDMI output.

3.5.3 Performance Considerations

While the direct video pipeline allows real-time processing of HDMI streams, performance is limited by:

- **AXI DMA bandwidth:** For high-resolution video, the throughput of AXI DMA becomes a bottleneck.
- **Processing latency:** The HLS filter IP must process pixels fast enough to keep up with the video timing.
- **Synchronization:** Ensuring stable video timing (vsync, hsync) and alignment with frame boundaries is crucial to avoid artifacts.

Despite these challenges, the system successfully demonstrated frame-by-frame real-time processing using FPGA acceleration, confirming its viability for embedded video applications.

4 Demonstration Summary

We built a web-based user interface to demonstrate our system's capabilities. The demonstration setup consists of the PYNQ-Z2 board connected to a local network, running our Django REST API server, and a client laptop running the Django front-end in a web browser.

When a user accesses the web UI, they are presented with a home page (fig. 5) where they can navigate to different image or video processing pages. The interface provides options to choose the processing operation: currently "Grayscale" conversion or applying a 3×3 convolution filter. For convolution, the user can select from predefined filters like Blur or Edge Detection. After selecting an image and an operation, the user clicks a "Upload" button.

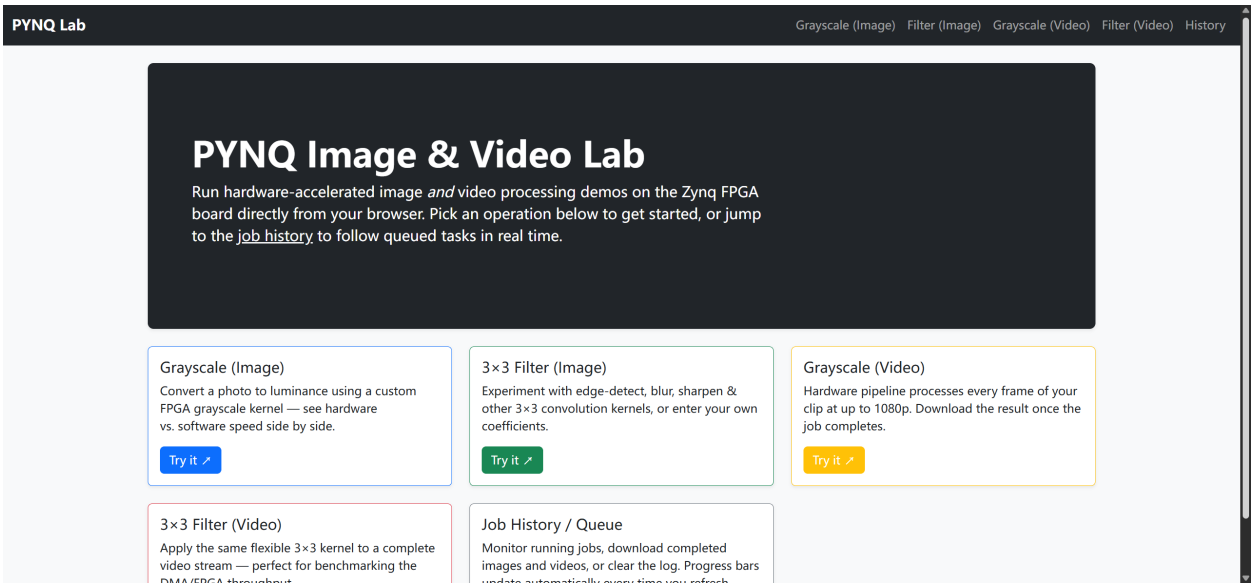


Figure 5: Home Page Demo

Behind the scenes, the image file is sent via an HTTP POST request to our RESTful endpoint on the PYNQ board. For images, the API responds typically within a few seconds depending

on image size, and the front-end then displays the original and processed images side by side. fig. 6 shows a screenshot of the web UI after processing an example image. In this example, the user uploaded a color photo and chose the edge detect option; the right side shows the result returned by the FPGA-accelerated system and SciPy software library, which has converted the photo to edge detection.

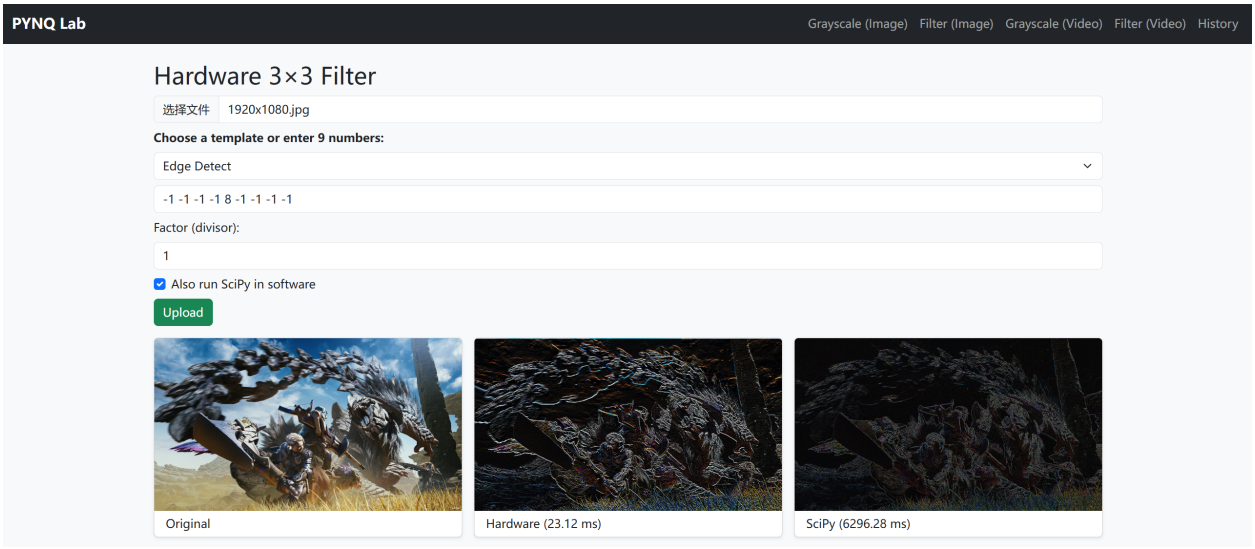


Figure 6: Edge Detection Demo

For the grayscale demonstration, the process is similar: the user might navigate to "Grayscale (Image)" page and upload an image. The system will configure the FPGA overlay and return an output image. The UI then shows, for example, the original image and the grayscale image (fig. 7).

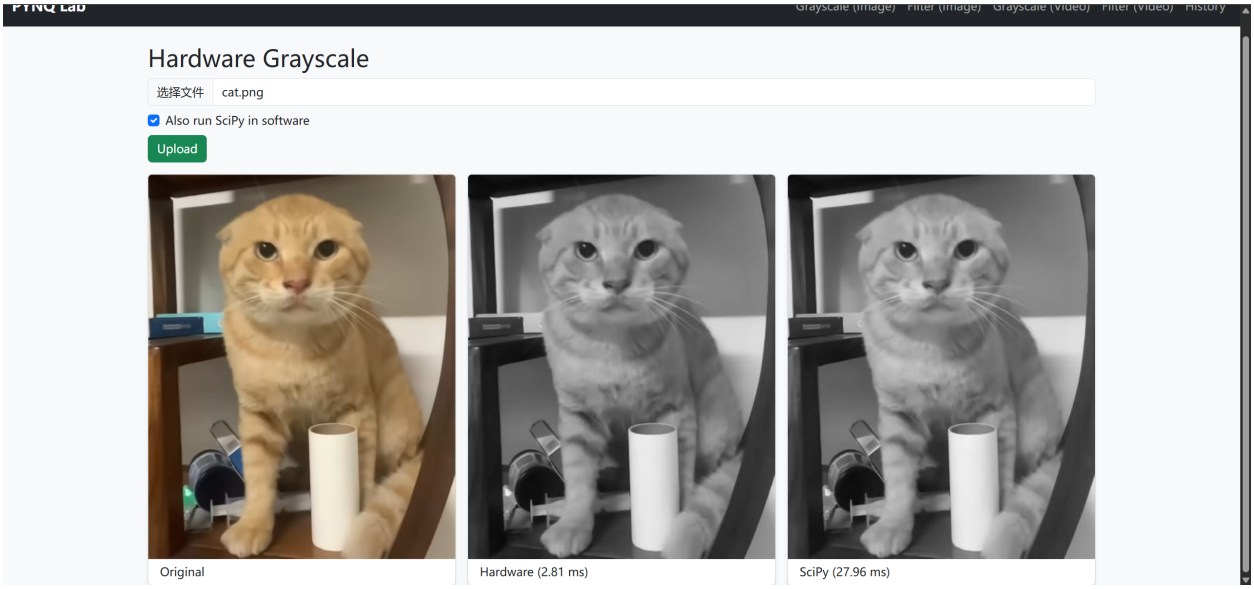


Figure 7: Grayscale Demo

In addition to still images, we prepared a short video demo to showcase the system's capability on a sequence of frames. The web UI allows uploading a short MP4 video. When the user uploads a video and selects (for example) "Grayscale", the back-end extracts frames from the video, processes each frame, and then reassembles the frames. The processed video can then be downloaded in the history page. Due to time and performance constraints, this video processing is not real-time, but it serves as a proof-of-concept for batch processing multiple images. The UI indicates progress while frames are processed. For demonstration, we showed a sample color video of a moving object converted to grayscale by our system. We also demonstrated applying an edge filter to the video to illustrate that the accelerator can be invoked repeatedly in a loop. The results are shown in fig. 8, and the grayscale and edge detection results can be viewed on YouTube.

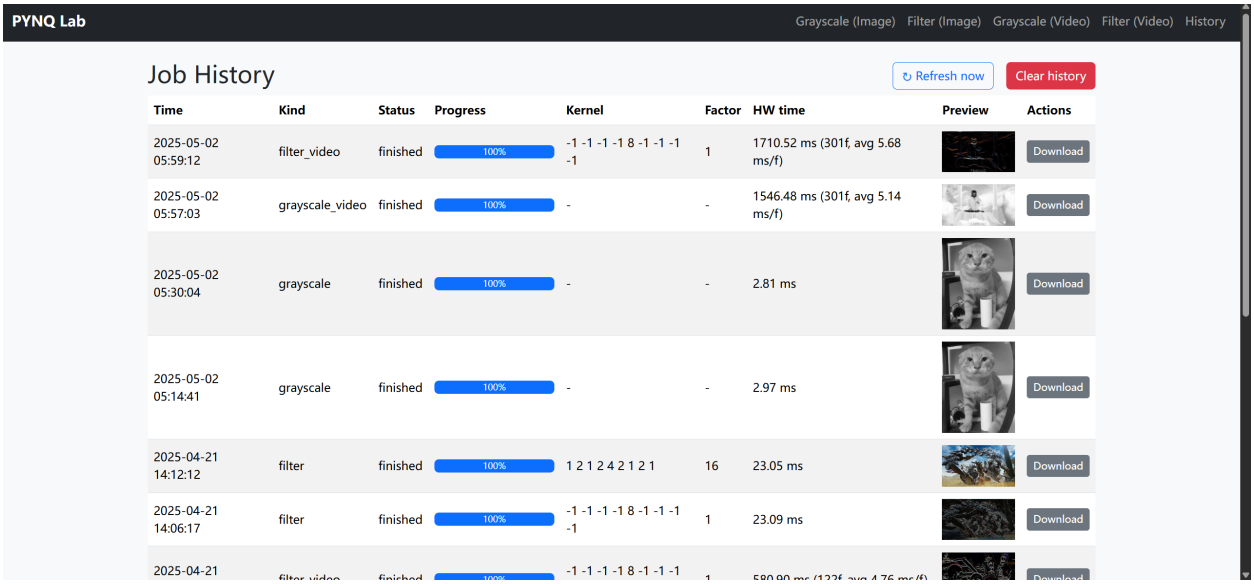


Figure 8: Video Demo

The front-end is designed to be intuitive, making it easy for anyone even without technical knowledge of FPGAs to utilize the accelerator: just upload and wait for results. This fulfills one of our objectives of providing a minimal user setup – no need to install special software or use command-line tools; a web browser is enough to harness the FPGA's power.
In summary, the demonstration highlighted:

- **Functionality:** Our system correctly performs the requested image transformations and returns the results.
- **Usability:** The web interface and RESTful design make the accelerator easy to use, aligning with real-world integration needs.
- **Speed:** The hardware acceleration speeds up processing compared with software results.
- **Use Cases:** We discussed how this could integrate into larger systems – for example, a cloud service that offloads image filters to such FPGA boards, or an camera that pre-processes video frames on-device using an FPGA before sending data upstream.

5 Results Discussion

5.1 Performance Analysis - Single DMA

For the block diagram described in 3.3, initial profiling revealed that although the PL completes the convolution in 5 milliseconds, per-frame overhead on the Processing System (PS) side pushes total latency to roughly a quarter-second. For a 480p 30fps test video of 21 seconds, the processing time is $\tilde{1}$ min

5.1.1 Measured Timing (one representative frame)

Stage	Latency [s]
Frame capture (OpenCV)	0.0156
Channel combination (RGB → `0x00RRGGBB`)	0.0660
DMA buffer copy / flush	0.0030
DMA transfer + kernel execution	0.0050
Channel extraction (`0x00RRGGBB` → RGB)	0.0925
Video-writer encode	0.0672
Total	0.2494

5.1.2 Key Observations

- The two highlighted stages—packing and unpacking the pixel channels consume 0.0660+ 0.0925 ≈ 0.16 s, about **64 % of the per-frame budget**.
- These operations are pure Python/NumPy vector manipulations that depend on cache-line flushes and memory bandwidth on the ARM processing system which does not benefit from the FPGA fabric.
- Since every frame pays this penalty, overall throughput drops to roughly 4 fps.

This observation motivated the triple-DMA redesign (Section 3.3.3), which removes the packing step entirely by streaming each color plane through its own DMA channel.

5.2 Performance Analysis - Triple DMA

5.2.1 Mode A – 3 Frames / Cycle, Packed

Three frames are moved through the pipeline in parallel, but each frame is still packed into a 32-bit word on the ARM side and unpacked after processing.

Stage	Latency [s]
Frame capture (OpenCV)	0.0214
Channel combination (RGB → `0x00RRGGBB`)	0.0647
DMA buffer copy / flush	0.0028
DMA transfer + kernel execution	0.0103
Channel extraction (`0x00RRGGBB` → RGB)	0.0925
Video-writer encode	0.0999
Total	0.2918

5.2.2 Mode B – 1 Frame / Cycle, Parallel Channels

A single frame is streamed per cycle, each color plane travels on its own DMA, and no software packing/unpacking is required.

Stage	Latency [s]
Frame capture (OpenCV)	0.0140
Stream data preparation (no packing)	0.0007
DMA buffer copy / flush	0.0131
DMA transfer + kernel execution	0.0132
Frame re-assemble (channel merge)	0.0619
Video-writer encode	0.1082
Total	0.2111

5.2.3 Key Observations

1. CPU packing/unpacking dominated the packed-mode pipeline $0.0647+0.0925 = 0.1572s$, $\approx 54\%$ of Mode A latency. Removing those steps yields a 27% overall speed-up even though DMA traffic is no longer interleaved across three frames.

2. Kernel and DMA hardware are not the bottleneck In both modes the FPGA finishes in ≤ 14 ms; the PS data-movers and NumPy transforms dictate throughput.

3. Memory copy overhead becomes visible in Mode B

5.3 Performance Analysis - Overview

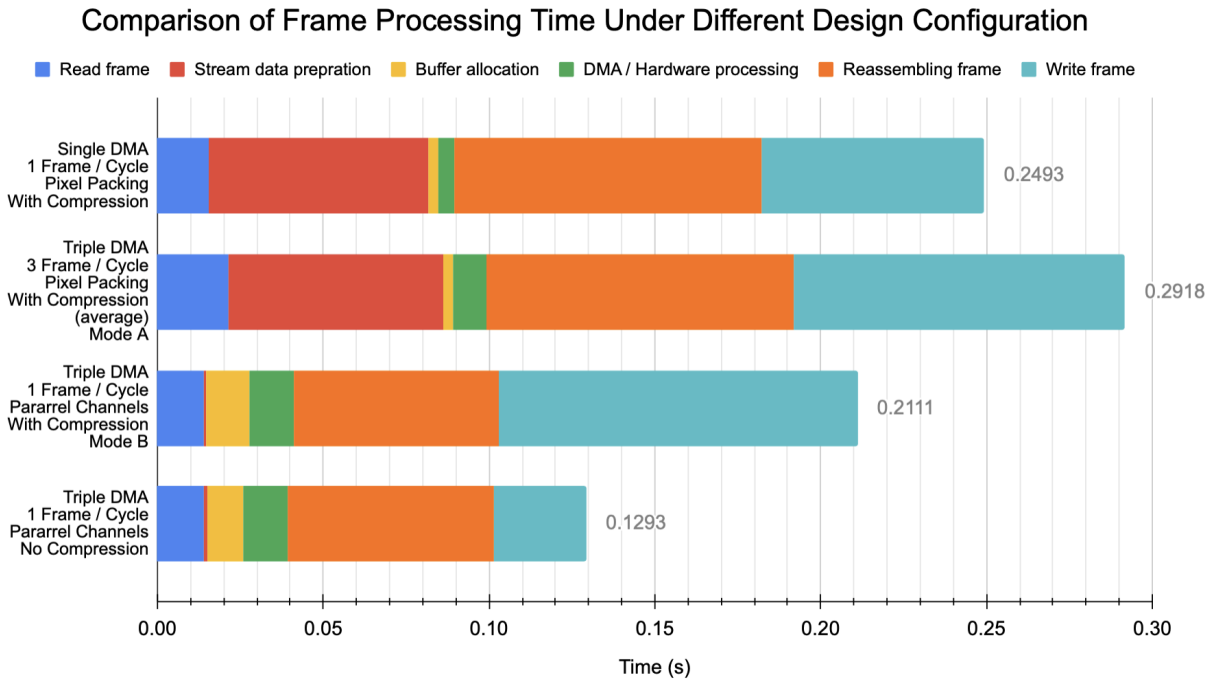


Figure 5.3 extends the timing study with a fourth configuration in which the video writer stores raw frames (no H.264 compression). Eliminating the codec overhead the "Write frame"

segment from $0.11s$ to $\approx 0.02s$, driving total latency down to $0.129s$, a 38% improvement over the best compressed variant (Mode B). All earlier optimizations remain in place: planar DMA transfer, no CPU packing, and one frame per cycle through the kernel. The cost of this speed-up is a dramatic increase in file size; the raw YUV stream is roughly an 6x of magnitude larger than its H.264 counterpart, for the testing clip we use, making the "no-compression" mode practical only for short streams or when external storage bandwidth is ample.

6 Implementation Discussion

Our project aimed to accelerate 2D convolution using a hardware-software co-design approach on the PYNQ-Z2 platform. Overall, we successfully met our primary objectives: we designed a custom HLS IP for 3×3 convolution, integrated it with an AXI DMA pipeline, exposed its functionality through a REST API, and built a complete web-accessible system that demonstrates both image and video processing.

6.1 What Worked Well

Several components of the system functioned better than expected:

- Our choice to build the web API using Django REST Framework prove to be ver effective. It provided built-in tools for handling file uploads, request validation, and API serialization, allowing us to focus on system logic rather than low-level HTTP handling. Django's template system also made it easy to build a user-friendly web interface. Although Django is relatively heavy for an embedded ARM processor, the framework's structure greatly accelerated development and debugging.

- Using Vitis HLS to develop the convolution IP enabled rapid iteration and high-performance hardware design without writing Verilog. HLS allowed us to incorporate optimizations like pipelining, loop unrolling, and line buffers, which helped achieve one-pixel-per-cycle throughput. The ability to synthesize readable C++ code into efficient RTL drastically reduced development time.

- Delegating all FPGA overlay interaction to a dedicated, single-threaded job manager script was a key architectural decision. It eliminated race conditions, avoided concurrent overlay reloads, and kept hardware state management clean and deterministic.

- Once data was in place, the FPGA performed convolution with exceptional speed, showing dramatic improvement over software baselines. This validated our hardware/-software partitioning strategy and confirmed that offloading compute-intensive tasks to the programmable logic was the right approach.

6.2 System Limitations:

- **AXI DMA Throughput:** One notable bottleneck is the data transfer between ARM memory and the FPGA. The AXI DMA on the Zynq-7000 is limited by the memory bus and clock speeds. In our tests, for larger images, the time to move the data could

dominate overall latency. If we attempted to process high-resolution images or a continuous video stream, the DMA would be a limiting factor. A potential improvement and future work could be to use double-buffering or multiple DMAs to overlap communication with computation, but our current design processes frames sequentially. In summary, the DMA significantly improves speed over not using it, but it can still slow down the overall pipeline for large data.

- **ARM Core Performance:** The ARM Cortex-A9 on the PYNQ is relatively modest in performance. Running a Django server and handling image encoding/decoding may push it near its limits. We observed that when processing larger images, the CPU spent a non-trivial amount of time just preparing data. This means our system might struggle if the volume of requests is high or if images are large. In a scenario with multiple users, the CPU could become a bottleneck even if the FPGA is fast.

- **Network and I/O:** The system was demonstrated on a local network. If this were deployed in a cloud scenario, network latency would add to the round trip. We did not implement any compression for transfer; images are sent raw. In a real service, one might want to send images as JPEG to save bandwidth, but then either the FPGA would need to decode JPEG or the CPU does it. Our assumption was a local setup or that image size vs processing time was the main concern, not network. Still, in tests, sending a 100KB image over network was negligible compared to processing time. But it's something to note: in a scenario where the board is remote, network could be a factor in overall performance perceived by the user.

6.3 Lessons Learned:

1. **Hardware/Software Integration Requires Careful Design Boundaries**
   We learned that cleanly separating the responsibilities of the FPGA and ARM processor is crucial to system stability and maintainability. Early versions of our project attempted to control the overlay directly from the Django server, which led to instability due to concurrency and overlay reload issues. Delegating this task to a single-threaded `worker.py` process eliminated race conditions and made the system more robust.

2. **FPGA Acceleration Is Only as Fast as the Data Pipeline**
   While our HLS-based convolution core achieved excellent performance, we observed that the overall system speed was often bounded by DMA throughput and memory copying overhead. This reinforced the importance of optimizing data movement—not just computation—when designing hardware-accelerated systems.

In summary, our project taught us a great deal about system integration, teamwork, and the realities of deploying an FPGA-based accelerator in a software-oriented world. The limitations we hit (DMA, CPU speed, etc.) are not just problems but also learning points and opportunities for improvement.

7 Conclusion

This project demonstrates that FPGA-based acceleration, when combined with a clean software interface, can offer significant performance benefits for image and video processing workloads. By offloading 2D convolution to a custom hardware kernel on the PYNQ-Z2 platform and managing control through an embedded Django-based REST API, we created a system that balances speed, accessibility, and modularity.

Our key finding is that hardware acceleration is not only feasible in embedded environments, but also practical and user-friendly when designed with integration in mind. The modular architecture—especially the use of a dedicated job manager—enabled reliable coordination between the processing system and programmable logic. Furthermore, our exploration into direct HDMI-based video processing shows the potential for real-time use cases beyond static images.

The implications of this work extend to any domain requiring fast edge processing—such as robotics, surveillance, or low-power AI systems—where custom hardware pipelines can be tightly integrated with modern software frameworks. Future work can build on this by scaling to more complex operations, supporting multiple concurrent pipelines, or extending the API to handle AI inference tasks on preprocessed video streams.

Ultimately, this project highlights that with thoughtful architectural design, embedded FPGAs can bridge the gap between low-level acceleration and high-level usability.

8 Future Work

While our system currently supports batch image processing via REST API and real-time video processing through a dedicated HDMI pipeline, these two modes remain separate. Future work will focus on unifying and expanding the system's capabilities in the following ways:

- **Real-Time Video Processing via REST API**
  One major future objective is to enable real-time video processing directly through the REST API. This would involve streaming video frames to the backend in real-time, performing frame-by-frame hardware-accelerated filtering, and returning a continuous stream of results. Achieving this would require optimizing the I/O pipeline, possibly integrating GStreamer or WebSockets for efficient video streaming support.

- **Modifying the Video Pipeline IP for Dynamic Control**
  Currently, the HDMI video processing pipeline runs in a static loop with no software-level control. In future iterations, we plan to enhance the video IP core to support dynamic reconfiguration at runtime. This would allow users to switch filters, adjust kernel coefficients, or pause/resume processing through API commands, bringing flexibility to real-time FPGA pipelines.

- **Unified Software-Hardware Integration**
  Merging the static HDMI pipeline with the Django server architecture would allow the REST API to manage both still images and live video streams. This requires modifying the system architecture to allow the ARM processor to configure and monitor the real-time pipeline without introducing latency or race conditions.

- **Multi-User Scalability and Stream Multiplexing**
  To make the system suitable for deployment scenarios, such as surveillance backends or edge servers, future work will explore support for multiple concurrent video streams. This would require scalable job management (e.g., Celery + Redis) and possibly distributing processing across multiple FPGA boards controlled by a central server.

9 Deliverables

Our project deliverables consisted of both hardware and software components, along with documentation and a working demonstration. Below is a list of the promised deliverables and the progress made toward finalizing each:

1. **Hardware-Accelerated Convolution IP Core**
   A custom 3×3 convolution kernel implemented using Vitis HLS and synthesized for the PYNQ-Z2 FPGA. This IP supports both grayscale and configurable filtering, and was fully integrated into our system with AXI4-Stream and AXI4-Lite interfaces.

2. **RESTful Web API and Server Backend**
   A Django REST Framework-based API hosted on the ARM processor, capable of receiving image data, dispatching processing jobs, and returning results. The API is functional, robust, and tested across multiple use cases.

3. **Job Management Pipeline (`worker.py`)**
   A single-threaded Python process designed to safely coordinate FPGA interaction, manage overlay loading, perform DMA transfers, and complete image processing tasks. This script reliably manages the job queue and ensures sequential, race-free execution.

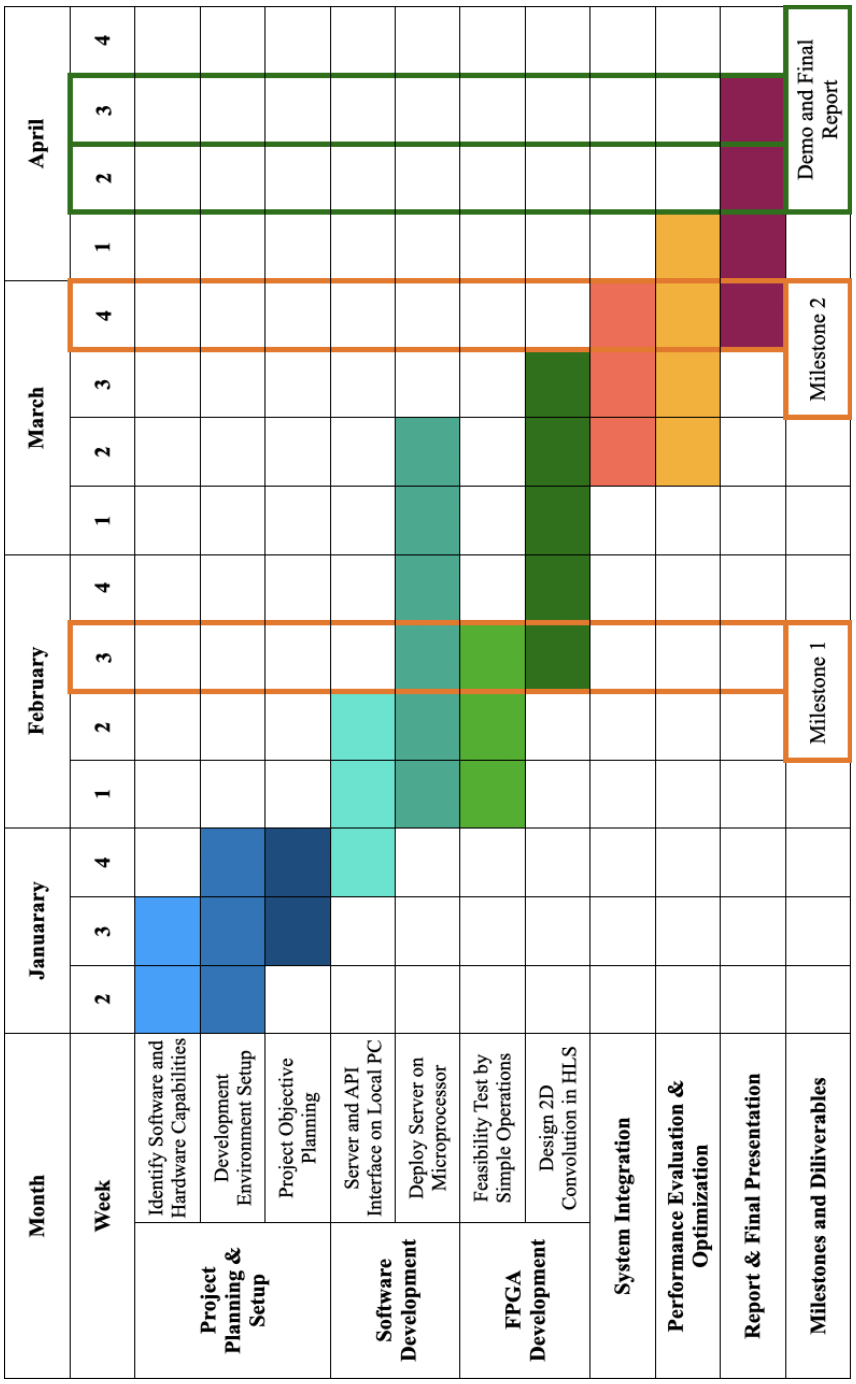
4. **Web-Based User Interface**
   A client-accessible front-end built using Django templates and JavaScript, allowing users to upload images or videos, choose filters, and view/download processed results.

5. **Direct Video Processing Pipeline (HDMI)**
   A secondary hardware pipeline supporting direct video frame processing using HDMI input/output. This was demonstrated using the RGB2DVI and DVI2RGB IPs and a live video filtering setup.

All major deliverables were completed as planned, with full integration demonstrated during final testing. The project exceeded initial expectations by incorporating both image and real-time video pipelines, making the system applicable to a wider range of embedded vision applications.

10 Timeline for Completion of the Project

Below is a project Gantt chart with tasks categorized by phases for completing the project, including major milestones and deliverables.



References

[1] *AUP PYNQ-Z2*. https://www.amd.com/en/corporate/university-program/aup-boards/pynq-z2.html. Accessed: 2025-05-02. 2025.

[2] *Kernel (image processing) - Wikipedia*. https://en.wikipedia.org/wiki/Kernel_(image_processing). Accessed: 2025-05-02. 2025.

11 Appendices

GitHub repositories:

- Hardware: https://github.com/Zichu26/fpga_convolution_acceleration
- Django: https://github.com/KennyRao/Conv2dHardwareAccelerationDjango

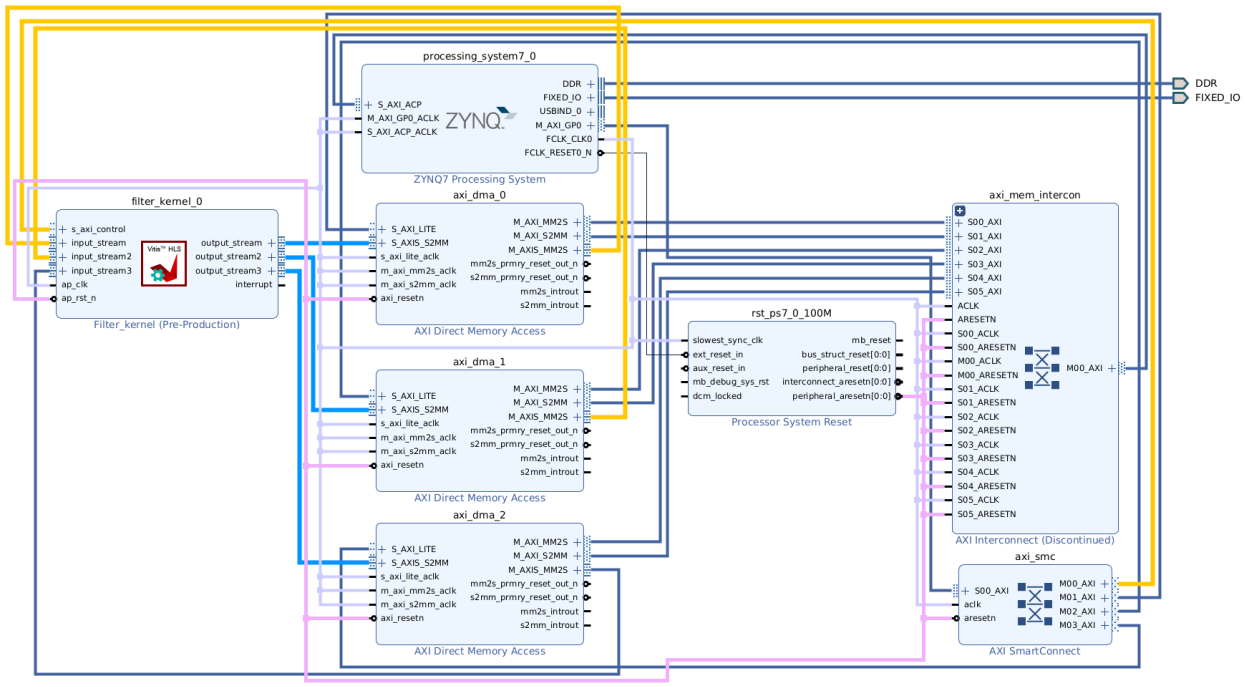


Figure 9: Block Diagram

filter_hls.cpp

```
#include "hls_stream.h"
#include "ap_int.h"
#include "ap_axi_sdata.h"

// max width of image
#define MAX_WIDTH 1920
```

```
typedef ap_axis<32, 0, 0, 0> AXI_PIXEL;

void store(
    AXI_PIXEL &pixel,
    int row_idx,
    int col,
    ap_uint<8> line_buffer[3][MAX_WIDTH][3]

) {
    #pragma HLS INLINE
    line_buffer[row_idx][col][0] = pixel.data(7, 0);
    line_buffer[row_idx][col][1] = pixel.data(15, 8);
    line_buffer[row_idx][col][2] = pixel.data(23, 16);
}

void create_window(
    int row,
    int col,
    int height,
    int width,
    ap_uint<8> line_buffer[3][MAX_WIDTH][3],
    ap_uint<8> window[3][3][3]
) {
    #pragma HLS INLINE
    // create a 3x3 window centered at current pixel location
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            for (int c = 0; c < 3; c++) {
                #pragma HLS UNROLL

                // index of each pixel in actual image
                int src_row = row + i - 1;
                int src_col = col + j - 1;

                // clamp the valid image coordinates
                src_row = (src_row < 0) ? 0 : ((src_row >= height) ? height - 1 :
                    src_row);
                src_col = (src_col < 0) ? 0 : ((src_col >= width) ? width - 1 :
                    src_col);

                // calculate circular buffer index
                int buffer_row = src_row % 3;
                window[i][j][c] = line_buffer[buffer_row][src_col][c];
            }
        }
    }
```

```
}

void apply_kernel(
    ap_uint<8> window[3][3][3],
    int kernel_factor,
    int kernel[3][3],
    AXI_PIXEL &output_pixel
) {
    #pragma HLS INLINE

    float sum[3] = {0.0f, 0.0f, 0.0f};
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            #pragma HLS UNROLL
            for (int c = 0; c < 3; c++) {
                #pragma HLS UNROLL
                sum[c] += float(window[i][j][c]) * kernel[i][j] / kernel_factor;
            }
        }
    }

    ap_uint<8> r = (sum[0] < 0) ? 0 : ((sum[0] > 255) ? 255 : sum[0]);
    ap_uint<8> g = (sum[1] < 0) ? 0 : ((sum[1] > 255) ? 255 : sum[1]);
    ap_uint<8> b = (sum[2] < 0) ? 0 : ((sum[2] > 255) ? 255 : sum[2]);

    output_pixel.data = 0;
    output_pixel.data(7, 0) = r;
    output_pixel.data(15, 8) = g;
    output_pixel.data(23, 16) = b;
    output_pixel.last = 0;
}

void filter_kernel(
    int image_width,
    int image_height,
    int kernel_factor,
    int kernel[3][3],
    hls::stream<AXI_PIXEL> &input_stream, // reference, me big dumb
    hls::stream<AXI_PIXEL> &output_stream
) {
// AXI-Lite slave interface, lower bandwidth interface for control signals
// `bundle=control` creates one shared AXI-Lite interface for all control
    parameters, in hardware this becomes a set of memory-mapped registers
    accessible from the PS
#pragma HLS INTERFACE s_axilite port=image_width register bundle=control
#pragma HLS INTERFACE s_axilite port=image_height register bundle=control
#pragma HLS INTERFACE s_axilite port=kernel_factor register bundle=control
```

```
#pragma HLS INTERFACE s_axilite port=kernel bundle=control
// #pragma HLS INTERFACE s_axilite port=channels bundle=control
// `port=return` creates 4 hardware signals
    // `ap_start` software set to '1' to begin execution
    // `ap_done` goes to '1' when processing is complete
    // `ap_idle` `1` when the IP block is not running
    // `ap_ready` `1` when the IP block is accepting new signals
#pragma HLS INTERFACE s_axilite port=return bundle=control
// AXI-Stream interface
#pragma HLS INTERFACE axis port=input_stream
#pragma HLS INTERFACE axis port=output_stream

    ap_uint<8> line_buffer[3][MAX_WIDTH][3];
    #pragma HLS ARRAY_PARTITION variable=line_buffer complete dim=1
    #pragma HLS ARRAY_PARTITION variable=line_buffer complete dim=3

    ap_uint<8> window[3][3][3];
    #pragma HLS ARRAY_PARTITION variable=window complete dim=0

    // initialize line buffer
    for (int i = 0; i < 3; i++) {
        #pragma HLS UNROLL
        for (int j = 0; j < MAX_WIDTH; j++) {
            #pragma HLS PIPELINE
            for (int c = 0; c < 3; c++) {
                #pragma HLS UNROLL
                line_buffer[i][j][c] = 0;
            }
        }
    }

    // initialize window
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            for (int c = 0; c < 3; c++) {
                #pragma HLS UNROLL
                window[i][j][c] = 0;
            }
        }
    }

    for (int row = 0; row < image_height; row++) {
        for (int col = 0; col < image_width; col++) {
            #pragma HLS PIPELINE

            // index of circular line buffer
            int row_idx = row % 3;
```

```
            // read and store pixel in line buffer
            AXI_PIXEL pixel;
            input_stream >> pixel;
            store(pixel, row_idx, col, line_buffer);

            // pixel.data = 0;
            create_window(row, col, image_height, image_width, line_buffer,
                window);
            apply_kernel(window, kernel_factor, kernel, pixel);

            pixel.last = (row == image_height - 1 && col == image_width - 1) ? 1
                : 0;
            output_stream << pixel;
        }
    }
}
```