Multi-Filter Camera

CS 152B Final Project Report

Lana Lim
105817312
lanachloe@ucla.edu

Jared Velasquez
905699244
jaredvel25@ucla.edu

Abstract

Modern imaging applications often require real-time processing to enhance visual information, extract image data or augment features. This data can be utilized in image classification, medical equipment (x-ray, MRI, etc.), or surveillance systems. Traditional image processing systems are implemented in software, introducing latency and unnecessary abstractions. To address these shortcomings, we present a multi-filter camera embedded system. This system leverages hardware for real-time image processing that is implemented with an architecture that can generalize to an arbitrary number of filters with a minimal increase in memory requirements.

1. Introduction

This project implements a multi-camera filtering system on an FPGA platform, designed to capture images from a video stream and apply zero or more filters to the output. By leveraging FPGA-based processing, the system provides a highly efficient and customizable approach for real-time image augmentation. The ability to process images directly on hardware enables lower latency and greater adaptability, making it a powerful tool for various applications.

One key application is enhanced feature extraction for AI and machine learning, where filters improve image clarity and optimize data for model training [1]. Additionally, low-light enhancement makes the system valuable for surveillance [2]. It also serves as a versatile image enhancement tool for digital photography, offering customizable filtering options.

This project must balance memory efficiency, real-time performance, power constraints, and privacy considerations. The architecture should keep memory buffers constant regardless of the number of filters, ensuring scalability. Filters must be applied instantly after frame capture to minimize latency, while power consumption must remain low for embedded applications. Privacy enhancements, such as anonymization and obfuscation, require advanced filtering techniques and potential integration with classical or deep-learning-based image recognition.

2. Related Works

Our project builds upon prior work that developed a simpler FPGA-based image processing controller capable of switching between gradient filtering and adaptive thresholding. Their implementation established the foundational architecture for processing images on an FPGA through the OV7670 camera module communicating with an I2C Serial Communication Bus while displaying the results on a VGA screen via the VGA graphics standard [3].

One of the key optimizations in previous works was pipelining the image processing stages to achieve parallelism. Instead of processing each pixel sequentially, pipelining allowed multiple pixels to be processed simultaneously in different stages of computation. By structuring their design as a sequence of operations where each step processes different parts of the image simultaneously, they were able to maximize the efficiency of FPGA hardware resources [3].

Leveraging a soft processor like MicroBlaze with the FPGA board can significantly enhance the system's adaptability and performance in image processing tasks. Another paper incorporates a soft processor in MicroBlaze that can offload high-throughput image computations such as filtering, edge detection, or transformation to dedicated hardware, while the soft processor handles system control and less computationally intensive tasks, such as data acquisition and coordination between components [4]. This combination provides flexibility in adjusting to various image processing algorithms without requiring extensive hardware redesign, making the system more versatile for different applications.

Building upon these techniques, our implementation extends the prior work by incorporating a multi-buffer system, which allows for smooth switching between raw and processed images. Additionally, our FPGA system accomplishes real-time results through a VGA driver, ensuring that users can immediately view the results of the processing pipeline. By combining these elements with an efficient memory access strategy, our project demonstrates a scalable approach to FPGA-based image preprocessing for imaging applications.

3. Implementation

3.1 Architecture

Our device consists of three modular components: (1) basys3 FPGA for integration, (2) OV7670 PMOD for real-time image capturing, and a (3) VGA display to present the augmented image. Below is a block diagram of the mentioned components interacting with the embedded system. A larger block diagram can also be found in Appendix A.

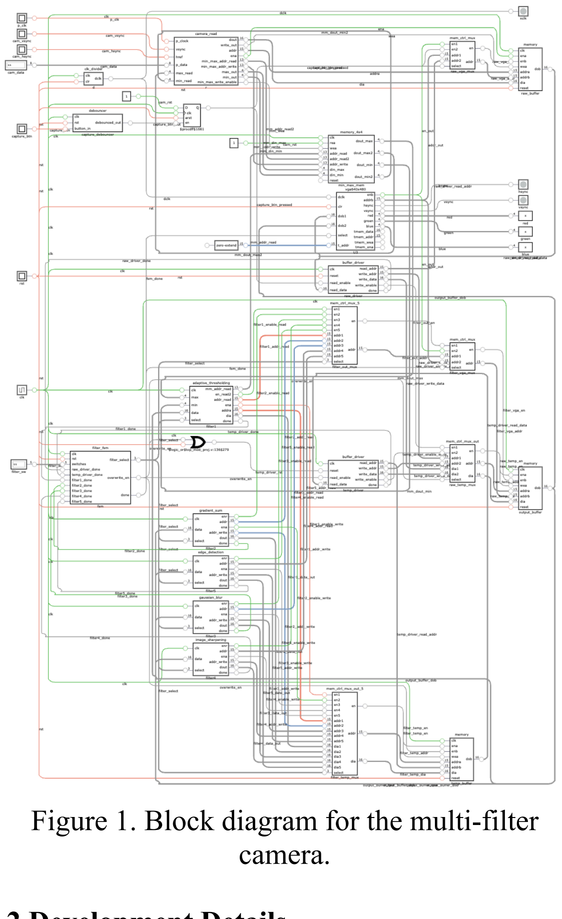


Figure 1. Block diagram for the multi-filter camera.

3.2 Development Details

The diagram in Figure 1 was generated using the DigitalJS online digital logic simulator tool. After synthesis and implementation, Vivado could translate this block diagram directly to a hardware bitstream to program the FPGA. Once programmed, the FPGA is ready to run Verilog source code and interact with the PMOD IPs.

We then wrote the main logic of the multi-filter camera in Verilog code. Although embedded C source code would abstract bare-metal concepts such as clock timing, memory, low-level control logic, there were very few resources on how to sufficiently integrate the existing OV7670 driver code with code executing on the Microblaze soft microprocessor. Moreover, one of the main concerns of developing on Microblaze was the overhead and inclusion of unnecessary resources, which would significantly impact the amount of code, restrict the types of logic performed, and limit the architecture on the embedded system. Developing in pure Verilog allowed flexibility in all of these fields without concern of memory partitioning.

Furthermore, we also extended the existing architecture in a previous project that was created to accelerate image processing on an FPGA board. The previous work built a basic filtering module that takes in a block made up of 4 consecutive pixels and applies a specific transformation to it within 1-3 clock cycles. In the ideal case, we wanted to generalize this structure to convolve each pixel with a 3x3 filter matrix via pipelining. I.e., pixels needed for the final calculation were already fetched in the previous clock cycle(s).

To test our code, we connected the basys3 FPGA with the OV7670 camera via a Serial Camera Control Bus and a VGA display by adhering to the VGA graphics standard [5].

With the embedded system programmed and properly connected to the peripheral components, the multi-filter camera is ready for use.

3.3 Development Challenges

Expanding on the previous project's structure turned out to be an unexpectedly large cost. One issue was with creating new filters, where the original design was not well-suited for more complex operations like Gaussian filtering. Additionally, we faced difficulties with the output buffer displaying black pixels, likely due to errors in the temp driver's interaction with the temp buffer and the output buffer.

The filter module in the existing project was specifically designed to fetch 4 pixels at a time, which is unconventional for image processing where image kernels are typically 3x3 matrices. The project was also intentional in choosing gradient filtering and an adaptive threshold filter, which can both be easily optimized for pipelining/parallelism. The gradient filter computes the horizontal/vertical gradients, or 'slope', between adjacent pixels, while the threshold filter essentially calculates the weighted mean of a pixel's local neighborhood. Therefore, their approach avoided excessive overhead that is sometimes necessary in a more generalized system. For instance, we wanted to create a Gaussian filter, which requires each pixel in a 3x3 neighborhood to be multiplied by a different weight. This would not fit the constraint of calculating the final pixel values for a single block within–at worst–3 clock cycles, thus violating the strict timing requirements for the entire system.

Another significant challenge we faced was the unintended behavior of the output buffer (introduced below) containing entirely black pixels when enabling any filter. This is likely due to the filter -> temp buffer -> temp driver -> output buffer pipeline. The temp driver is not correctly reading from the temp buffer and writing to the output buffer; for instance, the MUX that selects between the raw driver writing the first frame into the output buffer or the temp driver overwriting the output buffer may not be activated when the temp driver begins writing. As the filters have already been tested to read correctly from the output buffer, the issue likely lies in the FSM control signal logic being emitted to the other logic components.

4. Results

When code execution begins, the OV7670 camera will start converting image data into a bytestream that can be read by a camera driver and written into a buffer named "raw buffer." This raw buffer contains image frames unaugmented by any filter processing. By default, a VGA driver directly reads from the raw buffer to display image frames. Once the "capture button" has been pressed, the VGA will stop reading from the raw buffer, and instead a "raw driver" will read a single frame of data and write it to the "output buffer." If zero filters are switched on, the VGA driver will read from the output buffer and write that data to the display. If one or more filters are on (via using the corresponding switches on the basys3 board), a finite state machine (FSM) will begin to transmit signals to coordinate the application of filters to the frame present in the output buffer.

Filters are applied to data in the output buffer sequentially in a predetermined fashion (e.g. if filters 1, 2, and 4 are enabled, filter 1 will execute, then 2, then 4). The first enabled filter will begin reading from the output buffer when the FSM sends an enable signal for this filter. When the filter reads the pixel data, it will apply a convolution and/or a general transformation per pixel and store this data in a "temporary (temp) buffer."

Once the filter has finished writing data, it will send an interrupt-style signal to the FSM to transition the system into an overwrite state. In this state, the "temporary (temp) driver" will read from the temp buffer and overwrite the pixel data present in the output buffer. Once the temp buffer has completed the overwrite, it will send a signal to the FSM to transition into the next state. If more filters have been applied, the above process will repeat (the filter will read from the output buffer, write into the temp buffer, etc.). If all filters have completed their computations, the FSM will send a signal to the VGA driver to read from the output buffer to display the final image, augmented by one or more filters.

This system architecture was chosen because it allows for easy extension and/or modification of the filters that can be applied to the image. The system will require only two buffers for an arbitrary number of filters, thereby satisfying FPGA memory constraints.

As mentioned above, the filter computation and temp driver overwrite issues were not resolved before the end of development, leading to a partial implementation of the multi-filter camera. These waveforms demonstrate the individual correctness of each module despite the incorrect behavior of the entire embedded system.

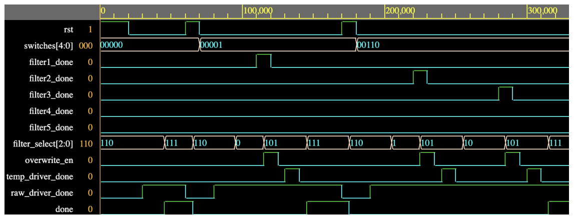


Figure 2. FSM testbench waveform.

The FSM contains 7 states. States 0-4 are the filter states, where each state indicates that the corresponding filter can start reading and writing: adaptive threshold, gradient sum, Gaussian blur, image sharpening, and edge detection. State 5 is the overwrite state, which indicates that the temp driver can start overwriting the output buffer with temp buffer's contents. State 6 is the initial wait state. State 7 is the final done state. The FSM begins transitioning from state 6 (wait) to the filters when the raw driver is done writing content into the output buffer.

With no switches turned on (switches[4:0] = 00000), once the raw driver signals it is done, the FSM will transition from state 6 (wait) to state 7 (done), as no filters have been enabled.

With adaptive threshold turned on (switches[4:0] = 00001), the FSM will transition from state 6 (wait) to state 0 (adaptive threshold). Once this filter indicates it has completed writing into temp buffer, the FSM will transition into state 5 (overwrite) and the overwrite enable signal is triggered, allowing temp driver to overwrite output buffer. Once the temp driver signals that it is done, the FSM will transition to state 7 (done), as there are no more filters left to apply; the final output can now be displayed by the VGA.

With gradient sum and Gaussian blur turned on (switches[4:0] = 00110), the FSM will transition from state 6 (wait) to state 1 (gradient sum). Once gradient sum has completed writing into temp buffer, the overwrite signal will allow temp driver to overwrite output buffer. Once the driver is finished overwriting, the FSM will transition to state 2 (Gaussian blur) to start filtering with this module. Once this filter has finished writing into temp buffer, the overwrite signal allows temp driver to overwrite output buffer. Once done, the FSM will transition into state 7 (done) and the final output is displayed by the VGA.

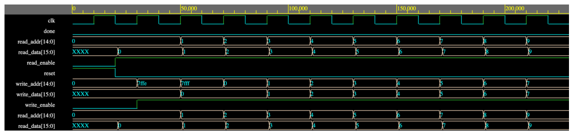


Figure 3. Buffer driver testbench waveform.

The buffer driver essentially reads memory from one buffer and writes that memory into another buffer at the same address, essentially copying the information of one buffer to another.

Due to register timing issues, a memory address offset had to be introduced to ensure that the data prepared in the write data register would sync up with the same address that this data from the original buffer was read. In the above waveform, once the buffer driver has been initialized, the address and data to be written syncs up with the address and data that the buffer read from.

5. Conclusion

In conclusion, while we successfully implemented key aspects of the multi-filter camera system, such as frame capturing, finite state machine (FSM) logic, and high-level design, the project remains a partial implementation due to issues in filter computation and output buffer management. The existing memory structures and read/write logic presented significant obstacles in performing convolutions efficiently, and difficulties with filter overwrite logic further limited the system's ability to handle dynamic filter selections. With further refinement of the timing and buffer overwrite mechanisms, the full implementation of a multi-filter camera system is still possible.

A potential solution for multi-filtering would be the relaxed timing constraints of the FPGA controller from our original design (assuming it was functioning correctly). The controller is designed to wait for each filter to complete before moving to the next one, allowing each filter to execute in an arbitrary number of clock cycles. This would enable the system to properly fetch the required pixels within the 3x3 neighborhood for more complex filters, such as the Gaussian filter, without violating timing constraints. By prioritizing accuracy over throughput for specific filters, the controller could provide greater flexibility in handling different types of image processing tasks.

Taking inspiration from related works, we recognize that simplifying the control logic by migrating non-filter-related tasks to the MicroBlaze processor holds significant potential for future improvements [4]. This approach could streamline operations, reduce complexity, and allow for more efficient resource allocation. However, one of the major hurdles encountered during the implementation was the lack of sufficient documentation for linking communication between the soft processor and the PMOD OV7670 camera. To address this, creating a custom IP for the MicroBlaze processor could serve as an effective solution.

References

[1] Krig S. Image Pre-Processing. In: Computer Vision Metrics. 2016.

[2] Appiah O, Hayfron-Acquah J. B, Asante M. Real-Time Motion Detection and Surveillance using Approximation of Image Pre-processing Algorithms

[3] FPGA Acceleration of Image Preprocessing.

[4] Wang C, Zhu S. A Design of FPGA-Based System for Image Processing. 2015.

[5] Upadhyay R. VGA Monitor Interfacing. 2022.

A. Appendix A

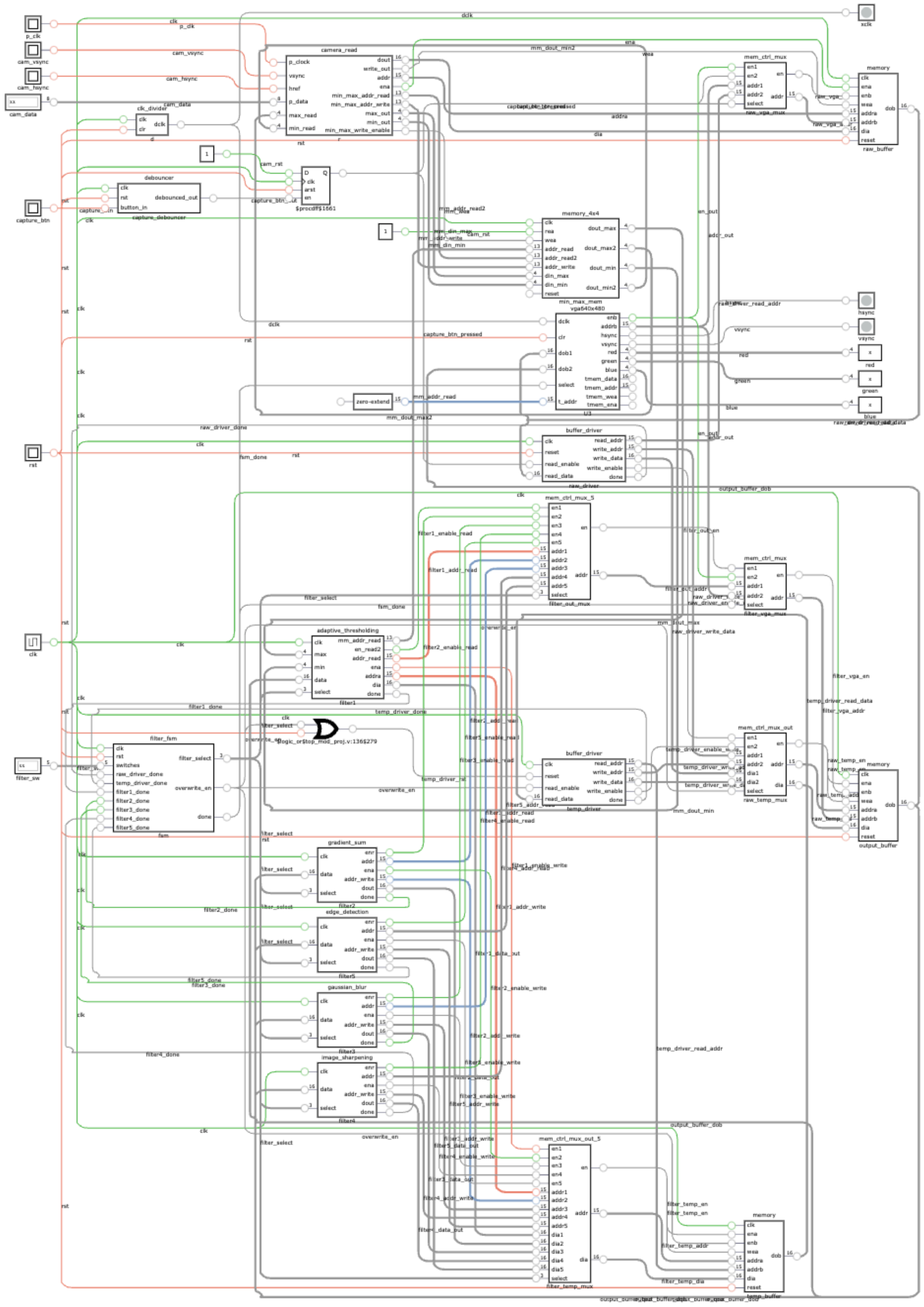


Figure 1. Block diagram for the multi-filter camera.

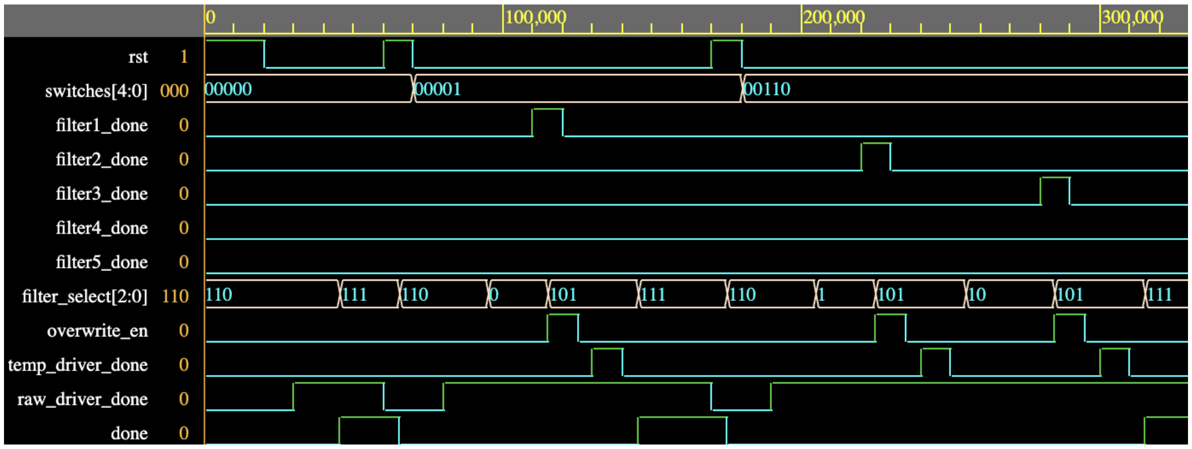
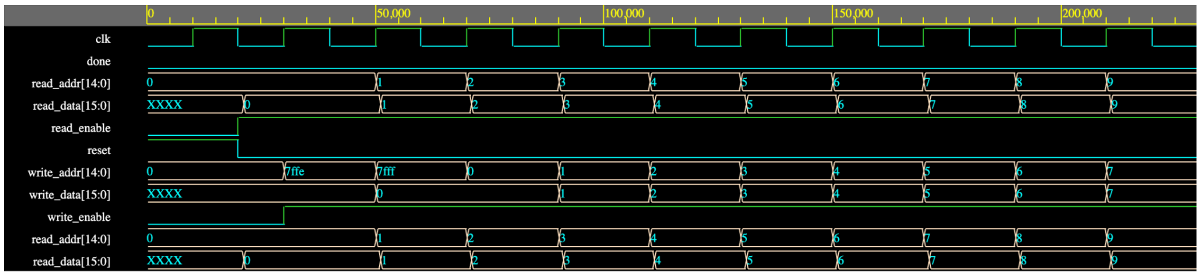


Figure 2. FSM testbench waveform



.

Figure 3. Buffer driver testbench waveform.