CSE 462M Capstone Design Project Final Report

FPGA-Based Acceleration of Convolutional Neural Networks for Facial Recognition

January 2025 - April 2025

Authors

Ben Ko, B.S. Computer Science, B.S Computer Engineering

Dept. of Computer Science & Engineering, ben.k@wustl.edu

Chiagozie Okoye, B.S. Computer Engineering

Dept. of Computer Science & Engineering, c.i.okoye@wustl.edu

Sandro Xiao, B.S. Computer Science

Dept. of Computer Science & Engineering, sandro@wustl.edu

Washington University in St. Louis

Project Supervisor

Michael Hall, Ph.D Computer Engineering

Dept. of Computer Science & Engineering, mhall24@wustl.edu

Submitted to Professor Michael Hall and

the Department of Computer Science & Engineering

2 May 2025

Contents

List of Figures

Abstract

FPGA-Based Acceleration of Convolutional Neural Networks for Facial Recognition is the capstone project of Ben Ko, Chiagozie Okoye, and Sandro Xiao, supervised by Dr. Michael Hall. This project recognizes the ever-developing field of machine learning, focusing specifically on facial recognition via a convolutional neural network (CNN). The computations used in CNN's to identify images and faces are time-expensive. This project aimed to accelerate these calculations using a dedicated hardware design implemented on a field-programmable gate array (FPGA). Using SystemVerilog to describe a pipelined convolution engine and pooling unit, the design implements part of a CNN's feature extraction phase on a PYNQ-Z2. With a 480×480 pixel input, the FPGA design averages a runtime of 2.595 milliseconds. Compared to 5.057 milliseconds average runtime using the same input to a software model, the hardware design provides a $1.94\times$ speedup, a roughly 48.6% performance boost.

1 Introduction

Facial recognition has become an integral technology across industries such as security, consumer electronics, and healthcare. Its rapid growth can be attributed to three primary factors: (1) introduction of graphics processing units (GPUs) in deep learning [1]; (2) rapid growth of publicly available facial datasets from social media [2]; (3) development of convolutional neural networks (CNNs), which excel at processing image data [3]. However, convolution operations within CNNs are computationally intensive, often forming a major bottleneck in real-world applications [4].

The rise of hardware accelerators using field programmable gate arrays (FPGA) provides a promising approach in resolving the computational bottleneck. FPGAs offer a slate of computing resources that can be reconfigured to implement a computer design. They are well-suited for developing dedicated designs to accelerate tasks that may be trouble for general purpose processors. Traditional facial recognition implementations relying solely on CPUs or GPUs still encounter limitations in performance, scalability, and cost—particularly for embedded or decentralized applications. Therefore, this project focuses on designing and implementing a system that offloads CNN feature extraction operations to a dedicated FPGA design. By accelerating these operations on an FPGA and supervising data transfer with an AXI-Stream interface, the system can reduce the computational burden on general-purpose processors to create a faster solution for facial recognition.

Background

A typical CNN architecture is composed of multiple layers with distinct stages: convolution extracts features by applying kernel filters; activation functions introduce non-linearity, enabling the network to capture complex patterns; pooling reduces the spatial resolution of feature maps, lowering computational cost for future layers. These stages are often stacked into modular blocks and followed by fully connected layers that perform the final classification based on the high-level features extracted earlier.

CNNs are well-suited for facial recognition due to their ability to detect hierarchical features—starting from low-level features like edges and textures, and progressing to high-level representations like facial landmarks. However, while their performance is powerful, CNNs are computationally demanding, particularly during the convolution operations. For real-time or resource-constrained applications, such as embedded systems, the cost of repeatedly performing convolutions on large image datasets becomes a significant bottleneck.

Project Objectives

Based on the problem statement, our initial project objectives were to: (1) accelerate a convolutional neural network by offloading the feature extraction operation to a dedicated FPGA design on a PYNQ-Z2 board; (2) establish a multi-channel transfer pipeline between the board's programmable software (PS) and programmable logic (PL) using an AXI-Stream interface; (3) implement a fully connected layer and classification logic in Python; (4) validate the architecture design using public datasets and facial image dataset of the project's group members.

A complete feature extraction system would look like Fig. 1. By nature, the number of convolutions grows exponentially with each subsequent layer. Figure 1 shows six convolutional layers, where each square represents a convolution operation with a unique kernel and dimensions.

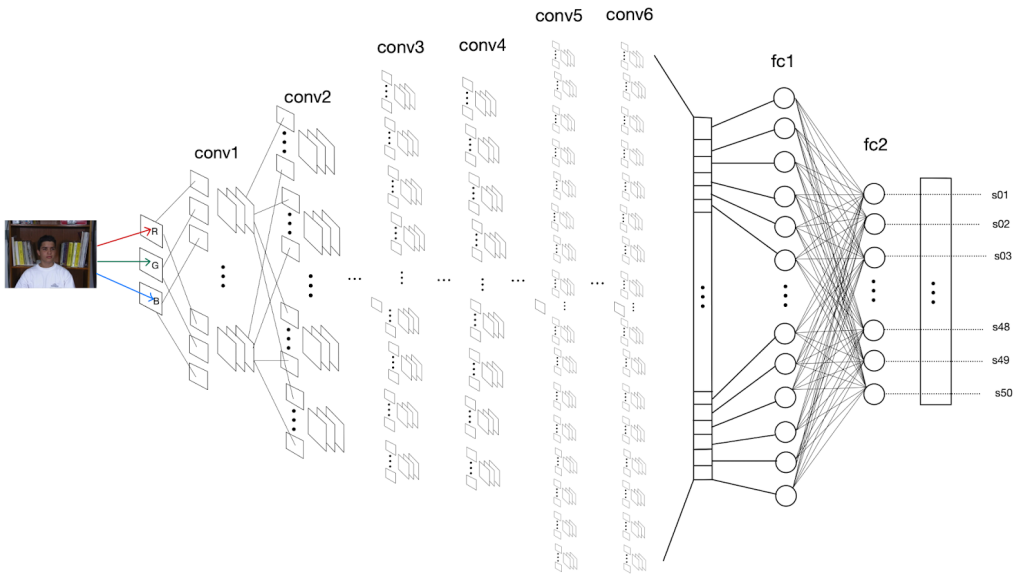

Figure 1: Proposed Architecture for Facial Classification

Initial steps were taken to attempt this form; however, upon implementing a three-layer, single-channel feature extraction—as seen in Fig. 2—we realized that feature extraction on the PYNQ-Z2 would not be feasible. The greatest constraint on attempting to implement the full feature extraction was resource availability on the PYNQ-Z2. Figure 3 shows how many resources were consumed by just a simple design.

Even if resource utilization did not grow linearly, this spelled trouble for our initial objectives. Figure 3 shows that the three-layer implementation uses 15% of available lookup tables (LUTs) on the FPGA. With 864 expected convolution operations in a complete design, it would not be a feasible architecture. In addition to resource constraints, there was difficulty with instantiating multiple AXI accelerator adaptors and AXI direct memory access (DMA)

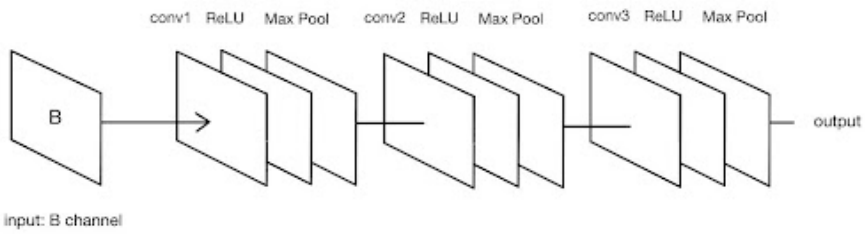

Figure 2: Three Layer Single-Channel Feature Extraction Pipeline

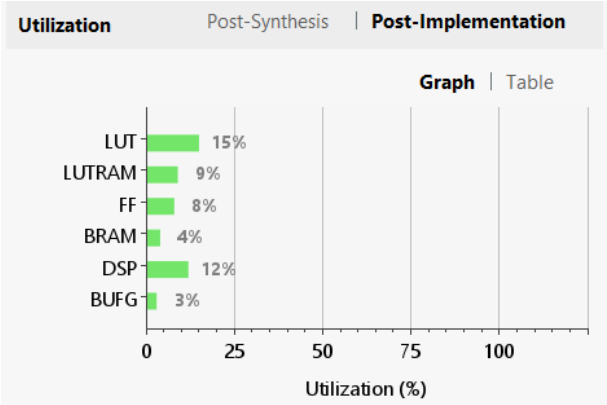

Figure 3: Resource Utilization of Three, Single-Channel Feature Extractions on a PYNQ-Z2

controllers, meaning separated RGB color channels could not be streamed in parallel.

The project's scope had to shift to account for these setbacks and the deadline of a complete project by the Spring 2025 semester's end. The project's objectives were updated to the following: (1) accelerate a single-channel, three-layer feature extraction by offloading work to a dedicated FPGA design; (2) establish a single-channel data transfer between the programmable software (PS) and programmable logic (PL) using AXI-Stream; (3) validate the design's output using a software model.

With revised objectives, the functional requirements of the project were identifiable. First, the project's product must provide an interface for users to upload images, suggesting a server. The image must be pre-processed by the server into a hardware-readable format, which includes separating the image into singular color channels, selecting one for transfer to the accelerator. Communication between the PS and PL requires an AXI-Stream interface. Finally, the accelerator's output must return to the server, the server appropriately handling the response.

The project also presented non-functional requirements. These included minimizing the end-to-end delay between image upload and result display, maintain clear separation between software and hardware for scalability and testing, fit the design within the FPGA's limited resources, ensure consistent operation and valid output across repeated image submissions, and

provide a simple and intuitive web interface for users to interact with the system.

Ethical Considerations

While our final design does not carry significant immediate ethical risks due to its restricted capabilities, broader deployment of similar systems could introduce substantial societal concerns. Scaled-up and integrated into public camera networks or surveillance systems, decentralized and accessible facial recognition accelerators could raise serious issues surrounding privacy, consent, algorithmic bias, and misuse [5]. Facial recognition systems have historically demonstrated disparities in accuracy across different demographic groups, potentially leading to unfair treatment or systemic bias [6]. Recognizing these risks, it is essential that future development and deployment of hardware-accelerated facial recognition systems include careful consideration of ethical safeguards, transparency measures, and regulatory oversight to ensure responsible use.

2 Methods

Our primary objective was a performance boost in the convolution computation, the most resource-intensive operation in convolutional neural networks and a key bottleneck in real-time image processing systems. The approach in designing required a three paths of task execution: (1) write the to-be-implemented FPGA design in hardware to validate outputs; (2) construct a hardware design in SystemVerilog to accelerate the CNN computation; (3) provide an application programming interface (API) with which inputs and outputs could be sent to and from the accelerator.

System Overview

Figure 4 illustrates the high-level architecture for this project. This system integrates a web-based user interface with a hardware-accelerated feature extraction pipeline. Beginning with the front-end website, a user uploads a facial image. The image is sent to a Flask-based server that transmits a request to the PYNQ-Z2's programmable logic (PL) using an AXI direct memory access (DMA) controller and AXI accelerator adapter. The AXI communication forwards the data to the accelerated convolution design. Upon completion, the results take the reverse path back to the Flask server. The server handles the response and re-renders the user-facing web page.

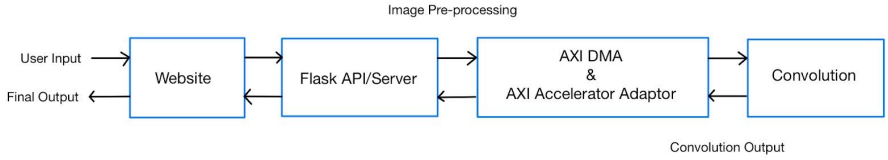

Figure 4: High-Level Overview of Project Architecture

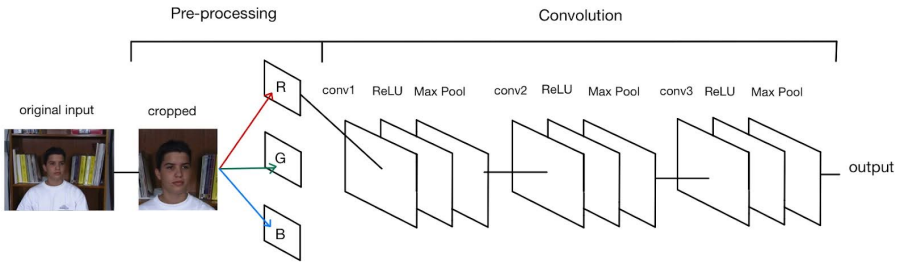

Figure 5: Overview of Pre-Processing & Single-Channel Feature Extraction

The Application Programming Interface

The Flask application provides a web interface for image upload and pre-processing. Upon uploading an image, the server temporarily saves the file and detects faces using OpenCV's Haar cascade classifier [7]. The detected face is cropped to remove surroundings and resized to a fixed resolution of 480×480 pixels. The resulting image is then separated into its red, green, and blue (RGB) channels. We arbitrarily selected the red channel to be transmitted through a AXI DMA controller module that instantiates a stream to the FPGA for convolution processing. To facilitate display in the web interface, the pre-processed RGB channels are base64-encoded and routed to a separate page for visualization.

Feature Extraction Design

As shown in Fig. 5, after pre-processing an image, each of the RGB channels can be sent independently through the hardware-accelerated CNN pipeline. The pipeline consists of three stages, each performing a 2D convolution, an activation function by means of a rectified linear unit (ReLU), and max pooling. This modular approach simplifies hardware implementation while preserving the core structure of the original software model.

Within the PL, the channel data passes through a pipelined feature extraction consisting of three layers. Each layer features a convolution, activation function, and pooling operation. Figure 6 magnifies the internal structure of each feature extraction layer. Input feature maps are first padded with zeros to preserve spatial dimensions during convolution. A 3×3 kernel passes across the input, performing multiply-accumulate operations to generate the convolved

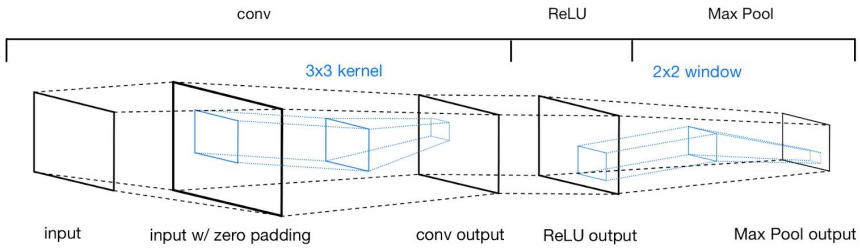

Figure 6: Overview of Each Feature Extraction Layer

output. The result is passed through a ReLU to introduce non-linearity and prevent the vanishing gradient problem, followed by a 2×2 max pooling stage that downsamples the feature map. This structure is replicated across the three stages of the hardware pipeline.

The Hardware Accelerator

The approach to building the accelerator began with writing SystemVerilog modules for the individual components of a CNN's feature extraction stage. The design for a zero padder and rectified linear unit were relatively straightforward to implement, but the convolution engine and pooling unit would require more complex logic to ensure speedup via pipelining. Fortunately, this project is not the first of its kind in exploring CNN acceleration on an FPGA. Publishing under the pseudonym "Batman," an engineering blogger had designed a pipelined model of the two modules [8]. This project builds off Batman's work, making appropriate modifications for its objectives.

Numerical Representations

Typical facial recognition kernel weights are fractional values. This necessitates a fractional representation for the hardware design. It was clear that 8 integer bits are needed for the activation map's RGB pixel values; additionally, a most significant bit to indicate sign. A fixed point Q8.11 format was chosen: a sign bit, 8 integer bits, 11 fractional bits totaling 20 bits. When compared to floating-point values used in software, the chosen fixed-point representation had a maximum error of 2.19×10^{-4} and a mean error of 9.9×10^{-5}.

Timing violations occurred when attempting to perform the convolution's 20-bit multiplication and addition in one cycle on the PYNQ-Z2. This was resolved by latching the multiplication's product in registers one one clock cycle and performing the addition on the next clock cycle. Attempts to expand the format to 32 bits once again raised timing violations. A similar strategy as before was attempted, however, it did not resolve the violation. Further, even disregarding the violation, the wider bit width ballooned resource utilization, particularly for digital signal processing (DSP) blocks, seen in Fig. 7. With the project objective to implement only

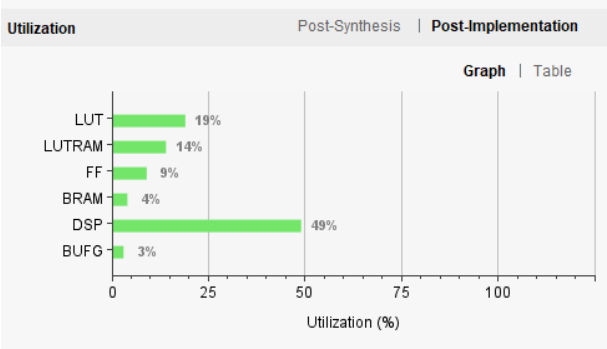

Figure 7: Resource Utilization with a 32-bit Fixed Point Representation

part of feature extraction, establishing a foundation for future work was considered important. This meant a 3-layer baseline design could not overuse resources.

Zero Padding

Zero padding takes an input matrix and surrounds its perimeter with zeros. As such, a zero padder module would lie between the input into the computation accelerator and the convolution engine. It was known at development that input activation values would be streamed into the accelerator only once. With an $N \times N$ matrix, the zero padder offsets the first $N + 2P$ entries, where P is the added layers of padding. This requires the zero padder to store activation values while it outputs zeros or previously stored values. The simplest approach is to use a First In, First Out (FIFO) queue. With each valid input to the padder, the module enqueues the value onto the FIFO, and either outputs a dequeued value from the front of the FIFO—an earlier input—or zero. For a simple matrix, Fig. 8 provides snapshots of the zero padder's functionality.

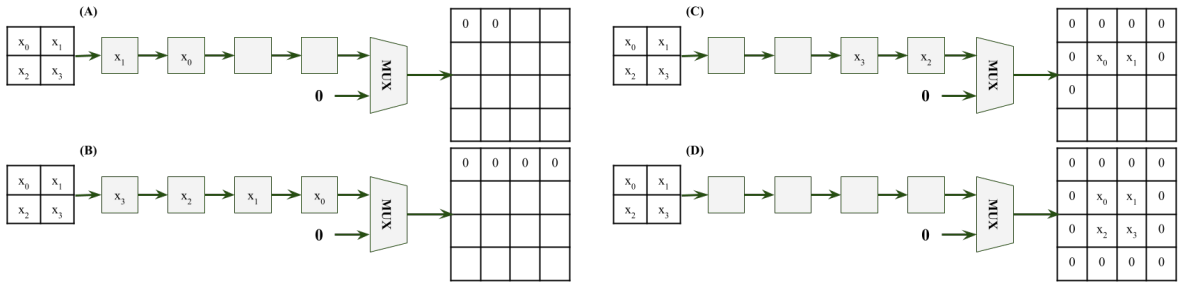

Figure 8: The Zero Padder's Behavior with a 2×2 Input and 4×4 Output

Figure 8 shows the padding module's behavior. At Step (A), the module has received and stored two values into the FIFO. Step (B) is when the FIFO is full, holding $N + 2P$ values. The FIFO has only dequeued two values in Step (C) even though there are five new output values between (B) and (C). The FIFO dequeues when the module is not at the output matrix's border. This leads to Step (D), when the FIFO is empty and the output matrix is constructed.

Convolution

Each valid output of the zero padding module is input to the convolution engine. A traditional approach would perform entry-wise multiplication between the kernel weights and the activation map's target window, sum the products, and store to the output matrix. Instead, the engine multiplies one streamed input against all kernel weights in a single cycle, adds it with the results from the previous cycle, and stores the sum in a temporary register to be available for the next cycle.

Kernel weights are known at synthesis for this project, so they are defined in the design as constants to simplify I/O. The kernel weights are module parameters—a SystemVerilog construct—allowing the kernel weights to be different when instantiating multiple modules, used later when implementing a multi-layer network. Each set of weights in the 3×3 filter was extracted from a pre-trained model available online [9].

For a sample 4×4 activation map and 2×2 kernel, the engine's architecture appears as in Fig. 9.

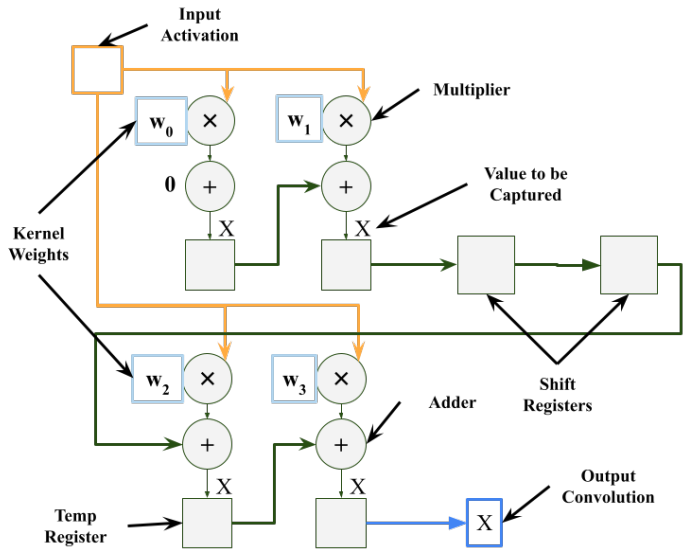

Figure 9: Convolution Engine Architecture With Components Labeled

For a general design with an $N \times N$ activation map and $K \times K$ kernel matrix, there will be $K \times K$ multiplier-adders and a $N - K$-deep shift register. The engine can be described as computing the next value while it computes the current. This is best deconstructed visually, which has been done so in this video: Convolution Engine Walkthrough (see Appendix A for the full URL).

Activation Function

The rectified linear unit (ReLU) takes output from the convolution engine and maps negative values to zero while letting others pass unchanged. Thus, the module features a multiplexer whose select bit is the most significant bit of the ReLU's input, i.e., the sign bit. When the sign bit is set, the value is negative and the multiplexer selects the constant zero; the input value is output otherwise.

Max Pooling

In a standard pooling approach, all values are compared to each other within a single pooling window to determine the maximum value. Much as with the convolution engine, a streamlined hardware module can be constructed to perform max-pooling. The pooling unit works by comparing an input to a previously determined maximum value stored in a register, which leads to the design pictured in Fig. 10.

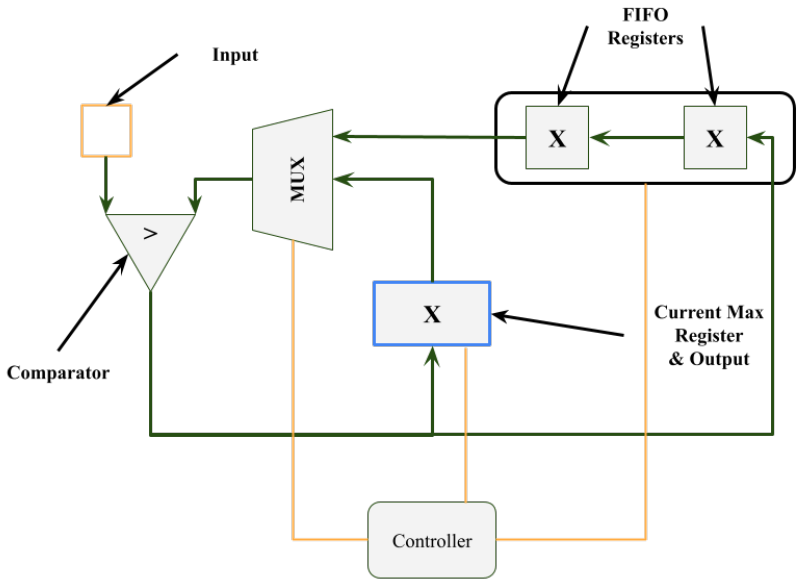

Figure 10: Pooling Unit Architecture With Components Labeled

With this design, for each non-overlapping pooling window of dimension p in the convolved matrix row, the maximum value is computed and stored to a FIFO. When the matrix wraps around to the next row, the unit compares new values within the same pooling window to the corresponding value stored to the FIFO, updating the maximum value seen and storing accordingly. This process is deconstructed visually in the following video: Pooling Unit Walkthrough (see Appendix A for the full URL).

Interconnecting Stages & Layers

Connecting the described modules involved in feature extraction necessitates a higher level design that controls when the modules are enabled to capture inputs, update internal states, and compute outputs.

The zero padder is enabled whenever there is a valid input to the accelerator. Its output is then the input to the convolution engine. The engine produces a valid convolution aperiodically, each value accompanied by a signal marking it as such. For the pooling unit's enable, the aforementioned signal is logically `AND`ed with the engine's enable. The unit must consider the engine's enable signal because when the engine is not enabled, it preserves all current values, which might be a HIGH valid convolution signal. With this organization, the convolution engine and pooling unit may have overlapping spans of activity, sidestepping the need to store all valid convolutions. Output from the pooling unit is the output of the feature extraction layer.

Multiple extraction layers can be instantiated using another top level design. The final output from one layer is simply the input to the following layer. As with the convolution engine, the pooling unit—and thus a layer as a whole—generates a signal marking valid outputs. This signal propagates to the next layer to serve as an enable.

For logical control units to function, each layer must know the dimensions of its input matrix—i.e., the output dimensions of the previous layer. This can be accomplished in SystemVerilog with parameter values evaluated at synthesis. In general, the output dimensions of a layer are dependent on the input dimension N, the padding dimension P, the kernel dimension K, and the pooling window dimension p. When the convolution kernel has a stride of 1, the expression for the output dimension is $(N + 2P - K + 1)/p$.

Prototyping & Validating Hardware with Software Models

Developing software models at each design stage was a crucial part of our project. Software models allowed us to test ideas before hardware implementations. These models could also be used to validate the hardware design.

A software model for a complete CNN was developed in PyTorch [10]. It features six sequential convolutional layers each followed by batch normalization, ReLU activation, and max-pooling, as shown in Fig. 1. The output was flattened and fed into a two-layer fully connected classifier, simulating the end-to-end processing pipeline targeted for FPGA acceleration. The proposed model attained a promising 92% accuracy using an open source face database from Center for Signal and Image Processing at Georgia Institute of Technology (Georgia Tech) [11].

Following the success of the initial software model, we started implementing three, single-channel convolution modules, which then showed that the fully accelerated model of the proposed mock software architecture would not be feasible in our case due to computational re-

straints as shown earlier.

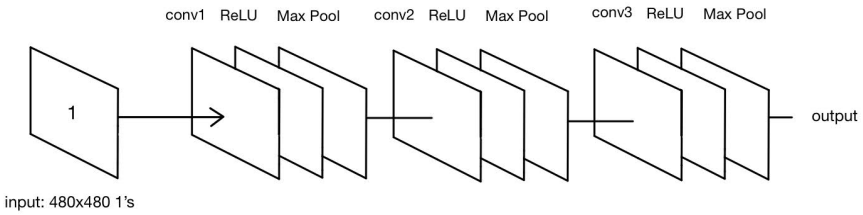

Figure 11: Software Model Replicating the FPGA Feature Extraction Pipeline for Design Verification

With our revised project objectives, another software model shown in Fig. 11 was developed using PyTorch to mirror the simplified three-stage, single-channel convolution pipeline implemented in hardware. This verification model consisted of three sequential convolution layers, each followed by ReLU activation and max pooling, matching the exact structure of the FPGA design. To facilitate debugging and validation, we used a simple test input: a 480×480 image where all pixel values were set to one. This predictable input made it easier to isolate and identify discrepancies between the software and hardware outputs. By assigning fixed kernel weights and ensuring identical configurations across both implementations, we were able to perform side-by-side comparisons and confirm the functional correctness of the hardware pipeline at each stage.

Runtime Evaluation

The key high-level objective of this project was to accelerate the stages of feature extraction. After implementing the design, a test was run to determine the time it takes for the design to complete three layers of feature extraction. Because the PYNQ-Z2 performs other tasks—rendering the Flask web page, pre-processing images, overlaying the design onto the FPGA—it was important to isolate and time only the steps that involved the FPGA-deployed design.

This isolated path was considered to be the time taken for the AXI DMA controller to complete all transactions between the PS and PL. After transferring the buffers viewable by the PS to the PL, the timer starts when the accelerator adapter is issued an execute command. This command tells the accelerated design to begin processing inputs from the adapter, and outputs are returned to the adapter. When the adapter sees all valid outputs from the design, it notifies the DMA controller, prompting the end of the transaction. This was executed and recorded 100 times.

Similarly, to gauge a purely software-based implementation, PyTorch was used on a general-purpose CPU. The input was a single-channel 480×480 tensor mirroring the test conditions

of the hardware design, as described previously and depicted in Fig. 11. The software model was run in evaluation mode to simulate inference conditions and eliminate gradient tracking overhead. For each run, the start and end times were recorded using Python's `time` module, and the duration of each forward pass was appended to a list. After 100 runs, the average runtime was computed and written to a file for comparison with the FPGA execution runtime.

3 Demonstration Summary

A live demonstration of the project was performed during the final presentation. It featured the Flask application running on the PYNQ-Z2. Using a split window screen on a connected computer, one window showed the rendered Flask web page while the other showed the terminal running Flask. The demonstration involved uploading an image from the Georgia Institute of Technology's database to the web page. On the terminal window, statements were printed as the code processed the input image and sent it to the accelerator. A pre-recorded video may be found at Demonstration (see Appendix A for the full URL). The output of the accelerator is written to a file in the Flask application's working directory. This last step showcases a successful completion of feature extraction's principle stages. Additionally, the in-time prints as the input image is processed shows the accelerator's speed as the prints occur quickly.

4 Results

Evaluation of the accelerated design based on time follows the processes previously discussed. For three feature extraction layers executed on both software and hardware, the results are visually represented in Fig. 12 and 13.

The software implementation achieved an average runtime of 5.057 milliseconds while the FPGA implementation completed the same operation in 2.595 milliseconds. This corresponds to a 1.94$\times$ speedup, or approximately a 48.6% performance improvement in favor of the hardware design.

5 Discussion

The results previously covered support the project's primary objective: accelerate part of the most computationally intensive component of a CNN. By offloading this operation to an FPGA, we successfully reduced computation time. Further, this project demonstrates the feasibility of hardware acceleration for CNN feature extraction.

As mentioned before, the accelerated computation performs well considering execution

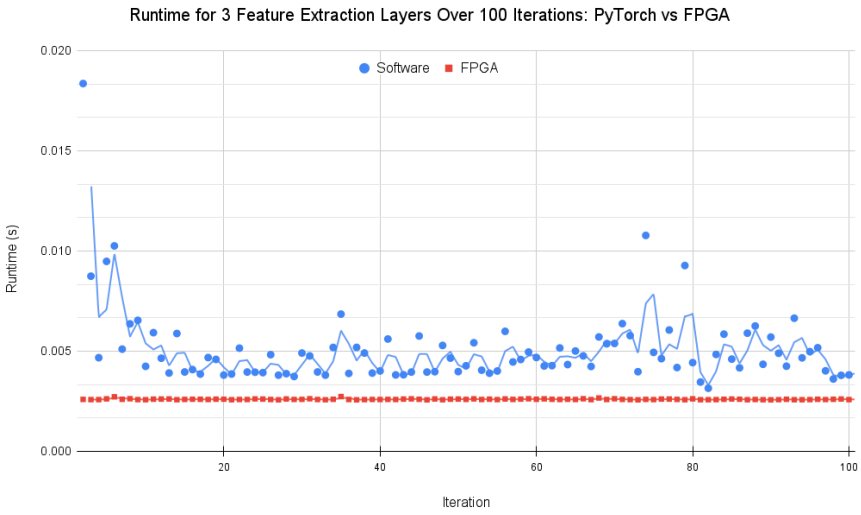

Figure 12: Runtime Iterations for Three Layers in Hardware Compared to Software

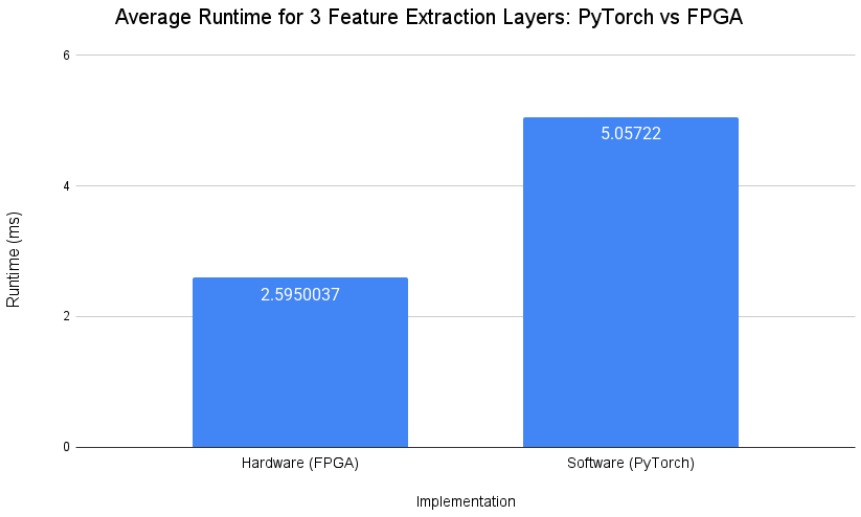

Figure 13: Average Runtime for Three Layers in Hardware Compared to Software

time. There is a drawback with the hardware implementation, though, that stems from the choice of value representation. The values returned by the hardware are not as accurate as those computed in the software model. This is believed to be caused by values' fixed point representation, which cannot match the precision of software's floating point arithmetic. Further, the Q8.11 format constrains the precision, preliminary attempts to expand this being unsuccessful. With the current design, a boost in time comes at the cost of accuracy. In situations where greater accuracy is desired, the hardware design may be unfit.

The hardware design features fixed kernel weights as parameters to SystemVerilog modules. This works well for this project's purposes as it was assumed a facial recognition kernel would not change for certain inputs. However, we note that other designs include kernel weights as a dynamic input to the design during runtime [8]. If kernel weights are, as we assumed, updated irregularly, then the cost of re-synthesizing the design with new kernel values is negligible.

An unforeseen issue that did arise in this project's development involved the communication between the AXI DMA controller and the accelerated design. The design was wrapped in a SystemVerilog module provided by the project's advisor, Dr. Michael Hall. Modifying the provided source code worked for our purposes, but when running the synthesized design on the PYNQ-Z2, communication between the PS and PL would intermittently fail. This failure manifested as "stalls" or "hangs" in the communication, where the AXI DMA controller would report as busy from the perspective of the PS with a transaction that should have been completed. The stalls indefinitely halted code execution. After the initial transfer on a freshly overlaid design, stalled transfers to the PL would occur more frequently. With time dwindling to complete the project, a temporary but working remedy was chosen: re-program the FPGA with the synthesized on each new request to the FPGA.

A natural next step for this project would be to extend the current convolution-only implementation into a fully functional facial recognition system. This would involve integrating feature extraction and classification stages following the convolution pipeline. Future work could also explore implementing a full multi-channel convolution architecture on the FPGA to more closely match the original CNN model. Finally, research into fairness, robustness, and privacy in hardware-based facial recognition would be essential to ensure responsible and ethical adoption of such systems in real-world application.

Reflecting on the project as a whole, our group finds great satisfaction with the opportunity to explore facial recognition, a subset of the now ever-prevalent artificial intelligence world. Though the project may not have achieved all of its initial objectives, it still provided invaluable experience in the engineering process of drafting, designing, implementing, and testing a project idea.

One of the most successful aspects of the project was our iterative use of software models to prototype and validate hardware implementations. The PyTorch-based models enabled rapid

testing, debugging, and verification of feature extraction stages before committing to a hardware design. Additionally, the modular pipeline design in SystemVerilog allowed us to isolate and test individual components—whether it be the convolution engine or zero padder—which helped with identifying where exactly in the pipeline things might be going wrong.

However, several challenges emerged during hardware integration. The most persistent issue involved communication between the PS and PL via AXI DMA. While our design was logically sound in simulation and post-synthesis, real-world execution revealed intermittent stalls during runtime AXI transactions. This significantly impacted system stability and forced us to adopt a temporary workaround of reprogramming the FPGA before each request, which is clearly not sustainable in a production setting.

Other project groups in this semester's cohort had significant success with high-level synthesis (HLS) using tools such as Vitis IDE. Groups using HLS commented on its ease of use, particuarly because it provided sample code for constructs such as the AXI interface.

In hindsight, using HLS could have provided improvements in this project. HLS had not been chosen at this project's onset because SystemVerilog was a language more familiar to group members. Looking back, however, it may have been beneficial to dedicate time to learning HLS. The teams that did use HLS reported high success with AXI integration. With HLS, we could have leveraged example templates to generate IP blocks with reliable AXI-Stream interfaces, potentially avoiding the PS-PL communication stalls that plague our design.

Furthermore, another key lesson was the importance of allocating sufficient time for system integration and testing on real hardware. While our functional modules worked well in simulation, real-world behavior introduced challenges that could not be predicted without earlier hardware trials.

Finally, we learned that a modular, testable approach was invaluable. It is important to begin with a software model, and piecemeal translate the design to hardware. Using software to verify every hardware change simplifies debugging. When a new change is made and output becomes incorrect, the source of that incorrect behavior can be identified. Future projects would benefit from maintaining this incremental development style.

6 Deliverables

Per our revised project objectives, the final project required a user interface for image uploads, an accelerator design that produced the results of a partially-implemented CNN, and a communication path between the two.

To deliver on the first requirement, a server API was built using Flask. Flask provides a simple yet responsive web page that supports image uploading. The server interacts with the AXI interconnection elements from the PS-side to send work to the accelerated design. The

parts of the CNN that were targeted for acceleration correctly did so, returning output to the server quickly. A comparison to the hardware could be made by constructing an equivalent software model to time and generate values with which the hardware could be verified. Finally, the Flask package provided means to render a display to the user when the computation was complete, writing results to appropriate output files.

All in all, though limited, a complete project was successfully submitted by the deadline. This project demonstrates all the aspects described and was functional in a live demonstration. The source files—including Python and SystemVerilog code—for this project can be found in a public GitHub repository (URL in Appendix A).

7 Project Schedule & Adherence

At the onset of this project, the group proposed the discussed capstone project to the project advisor. Included in that written proposal was the Gantt chart presented in Fig. 14. Admittedly, our initial goals were ambitious: implement an entire feature extraction system in hardware with a classification stage in software. As weeks were dedicated to the principle hardware modules, it became clear that adherence to this schedule would be impossible with the known project deadline. The difficulty in trying to meet the deadline lay predominantly in debugging the SystemVerilog modules to ensure they generated correct results, which had to be module-by-module. At the same time, there were setbacks with AXI communication between the PS and PL. By Milestone 2, the group had come close to implementing the simple design that would ultimately become our project, but it was then that the issue of resource utilization killed any further belief that implementing a complete, 6-layer feature extraction on the PYNQ-Z2 was possible.

The deadlines for deliverables would not move because of these setbacks. Therefore, the group pivoted plans to ensure a complete project could be submitted. All efforts were put into refining the single-channel, three-stage pipeline with the goal to demonstrate the FPGA receiving input from a functional front-end and returning valid results. This pivot was wisely taken, and the complete end-to-end system was completed by the final presentation date, a complete and working capstone project.

8 Conclusion

This project demonstrated the feasibility and impact of accelerating convolution operations—the most computationally demanding component of CNNs—using FPGA-based hardware. By offloading three convolution stages to a pipelined single-channel design on the PYNQ-Z2 board,

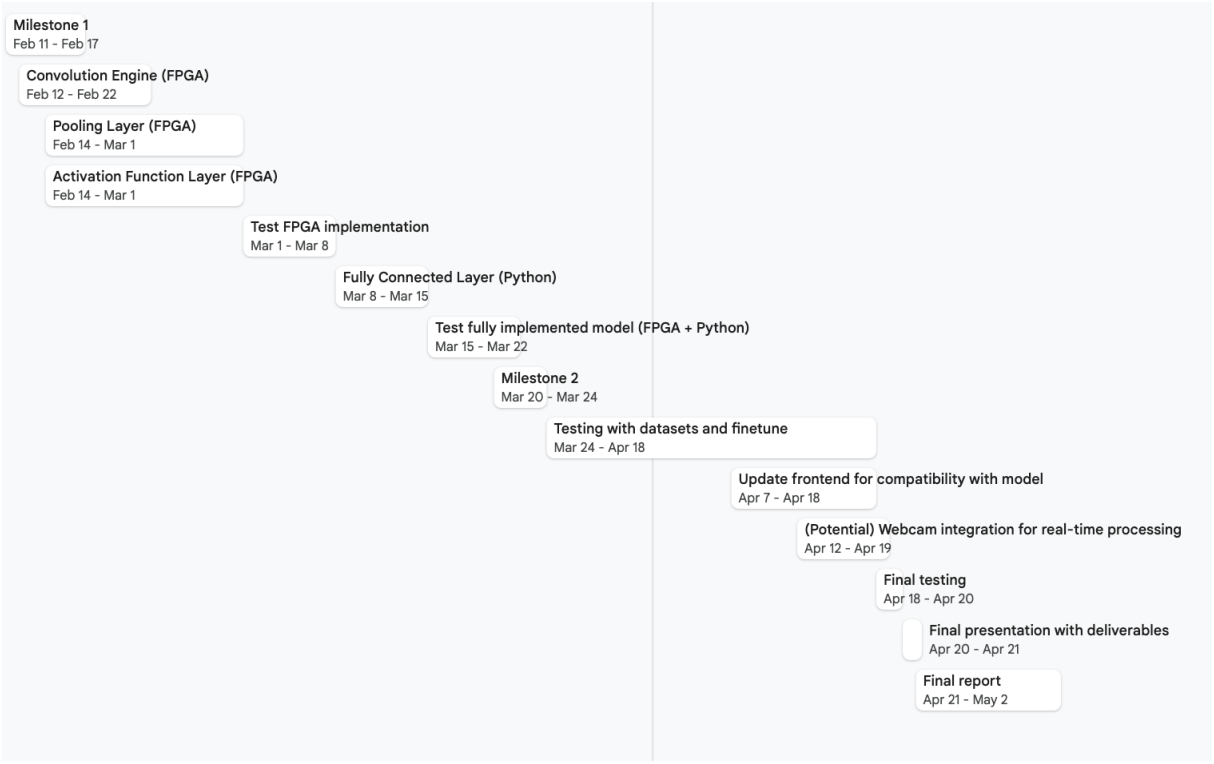

Figure 14: Initial Proposed Project Timeline

we achieved a 1.94× speedup over software execution in PyTorch, validating both our architecture and performance objectives.

While resource limitations prevented full implementation of a multi-layer, multi-channel CNN in hardware, our iterative prototyping approach using software models enabled rigorous testing and validation at each stage. The results support the potential for decentralized, low-power facial recognition systems built on lightweight, task-specific accelerators.

Looking ahead, the successful integration of hardware acceleration with a modular web-based interface paves the way for further development of complete facial recognition pipelines on FPGA platforms which will be scalable for edge applications and responsive to the growing demand for efficient machine learning inference in real-world settings.

References

[1] M. Pandey, M. Fernandez, F. Gentile, O. Isayev, A. Tropsha, A. C. Stern, and A. Cherkasov, "The transformational role of gpu computing and deep learning in drug discovery," *Nature Machine Intelligence*, vol. 4, p. 211–221, 03 2022. [Online]. Available: https://www.nature.com/articles/s42256-022-00463-x

[2] "Papers with code - machine learning datasets," Paperswithcode.com, 2022. [Online]. Available: https://paperswithcode.com/datasets?task=face-recognition

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, pp. 84–90, 05 2012.

[4] L. Moreira De Albuquerque and E. Teodoro Da Silva Junior, "Evaluating resources cost of a convolutional neural network aiming an embedded system," in *2018 VIII Brazilian Symposium on Computing Systems Engineering (SBESC)*, 2018, pp. 172–177.

[5] X. Wang, Y. C. Wu, M. Zhou, and H. Fu, "Beyond surveillance: Privacy, ethics, and regulations in face recognition technology," *Frontiers in Big Data*, vol. 7, 07 2024. [Online]. Available: https://www.frontiersin.org/journals/big-data/articles/10.3389/fdata.2024.1337465/full

[6] R. Fergus, "Biased technology: The automated discrimination of facial recognition — aclu of minnesota," www.aclu-mn.org, 02 2024. [Online]. Available: https://www.aclu-mn.org/en/news/biased-technology-automated-discrimination-facial-recognition

[7] OpenCV, "Opencv library," Opencv.org, 2019. [Online]. Available: https://opencv.org/

[8] Batman, "Fpga based acceleration of machine learning algorithms involving convolutional neural networks - thedatabus.in," Thedatabus.in, 07 2020. [Online]. Available: https://thedatabus.in/introduction

[9] Q. Cao, L. Shen, W. Xie, O. M. Parkhi, and A. Zisserman, "Vggface2: A dataset for recognising faces across pose and age," 2018. [Online]. Available: https://arxiv.org/abs/1710.08092

[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," arXiv.org, 2019. [Online]. Available: https://arxiv.org/abs/1912.01703

[11] A. V. Nefian, “Georgia tech face database,” Academic Torrents, 2025. [Online]. Available: https://academictorrents.com/details/0848b2c9b40e49041eff85ac4a2da71ae13a3e4f

Appendix A: Hyperlinks to Additional Materials

1. Convolution Engine: https://wustl.box.com/s/1igz0zunj2wp9qdgtuzrrww2p6k6nqgq
2. Pooling Unit: https://wustl.box.com/s/p179sfcazo9kz4bvpj10cs3k5eei2nbf
3. Demonstration: https://wustl.box.com/s/qykass2lv8bgjwsyacovjpialq34xtrt
4. Code Repository: https://github.com/Panagris/cse462M-capstone-final