BHARAT AI-SoC STUDENT CHALLENGE

A project-based virtual challenge to ignite innovation in AI-driven System-on-Chip (SoC) design

Real-Time Object Detection Using Hardware-Accelerated CNN on Xilinx Zynq FPGA with Arm Processor

Submitted by

INDIRAVARMAN A S	**727623BEV010**
MOHAMED AARIF S	**727623BEV061**

in partial fulfillment of the requirements for the degree of

BACHELOR OF ENGINEERING

IN

ELECTRONICS ENGINEERING

(VLSI DESIGN & TECHNOLOGY)

DR. MAHALINGAM COLLEGE OF ENGINEERING AND TECHNOLOGY

AN AUTONOMOUS INSTITUTION

AFFILIATED TO ANNA UNIVERSITY

CHENNAI 600 025

FEBRUARY 2026

ABSTRACT

The rapid proliferation of edge computing and embedded artificial intelligence demands inference hardware that strictly balances high computational throughput with low power consumption. Traditional software-only neural network execution on sequential processors often suffers from latency bottlenecks and energy inefficiency when handling dense, parallel Multiply-Accumulate (MAC) operations. This project details the design, implementation, and silicon-level evaluation of a Hardware-Accelerated Convolutional Neural Network (CNN) deployed on a Zynq UltraScale+ MPSoC, utilizing a rigorous Hardware/Software (HW/SW) Co-design methodology.

Targeting the classification of handwritten digits from the MNIST dataset, the system architecture was deliberately partitioned to exploit the strengths of heterogeneous computing. The ARM Cortex-A53 Processing System (PS) was designated for dynamic system orchestration, memory management, and high-speed Direct Memory Access (DMA) initialization via a bare-metal C application. Conversely, the computationally heavy convolutional layers were offloaded to a custom accelerator mapped directly onto the FPGA's Programmable Logic (PL) fabric.

The custom inference IP was developed using High-Level Synthesis (HLS) in C++. To maximize spatial and instruction-level parallelism, aggressive optimization directives were applied. Loop pipelining (#pragma HLS PIPELINE) was utilized to minimize the initiation interval, while loop unrolling (#pragma HLS UNROLL) forced the synthesizer to instantiate multiple physical DSP slices, allowing the hardware to process complex feature maps simultaneously. For high-bandwidth communication, the system leverages the AXI4-Stream protocol, enabling the AXI DMA to blast image data directly from DDR memory into the convolution engine without CPU intervention. Furthermore, dynamic power consumption was mitigated by enabling intelligent clock gating (power_opt_design) during the implementation phase.

Final bare-metal Hardware-in-the-Loop (HIL) execution successfully demonstrated the PS-to-PL data transfer; however, a critical AXI bus synchronization issue resulted in a DMA timeout. Comprehensive diagnostic debugging isolated the root cause to a missing TLAST (End of Packet) signal in the hardware stream, preventing the final DMA handshake. Isolating this protocol anomaly provided profound, practical insights into SoC bus architectures, memory coherency, and RTL stream compliance, establishing a robust foundation for future iterations to achieve maximum inference throughput.

TABLE OF CONTENT

LIST OF FIGURES

REPORT

PROBLEM STATEMENT:

Real-Time Object Detection Using Hardware-Accelerated CNN on Xilinx Zynq FPGA with Arm Processor.

INTRODUCTION:

The rapid expansion of artificial intelligence and machine learning at the edge has created a critical demand for high-performance, power-efficient inference hardware. While standard sequential processors can execute neural network algorithms, they often struggle to meet the strict latency and thermal constraints required by modern embedded systems.

By migrating the computationally intensive, highly parallel Multiply-Accumulate (MAC) operations from the ARM Cortex-A53 processing system to the FPGA's programmable logic fabric, we aim to demonstrate significant improvements in execution speed and hardware efficiency.

This document outlines the complete RTL development lifecycle. It details the algorithmic baseline generation, High-Level Synthesis (HLS) hardware optimizations, Vivado block-level integration, and bare-metal AXI bus memory orchestration, culminating in a comprehensive analysis of on-chip power consumption, resource utilization, and real-world hardware diagnostics.

SYSTEM ARCHITECTURE & DESIGN PARTITIONING DECISIONS:

To achieve low-latency inference for our Convolutional Neural Network (CNN), we utilized a strict Hardware/Software (HW/SW) Co-design methodology on the Zynq UltraScale+ MPSoC platform. The system architecture was deliberately partitioned to leverage the distinct strengths of the two main physical domains on the chip: the Processing System (PS) and the Programmable Logic (PL).

Software Partition (ARM Cortex-A53 / PS):

The software layer acts as the system orchestrator. The ARM core was assigned the tasks of memory management, AXI Direct Memory Access (DMA) initialization, and controlling the execution flow. It handles the input image data (MNIST dataset) and triggers the hardware transactions, ensuring the control logic remains flexible and easy to update.

Hardware Partition (FPGA Fabric / PL):

The computationally intensive neural network operations (Multiply-Accumulate operations in the convolutional layers) were entirely offloaded to the FPGA fabric. This allows the system to exploit massive spatial parallelism using dedicated DSP slices, which a standard sequential CPU cannot match.

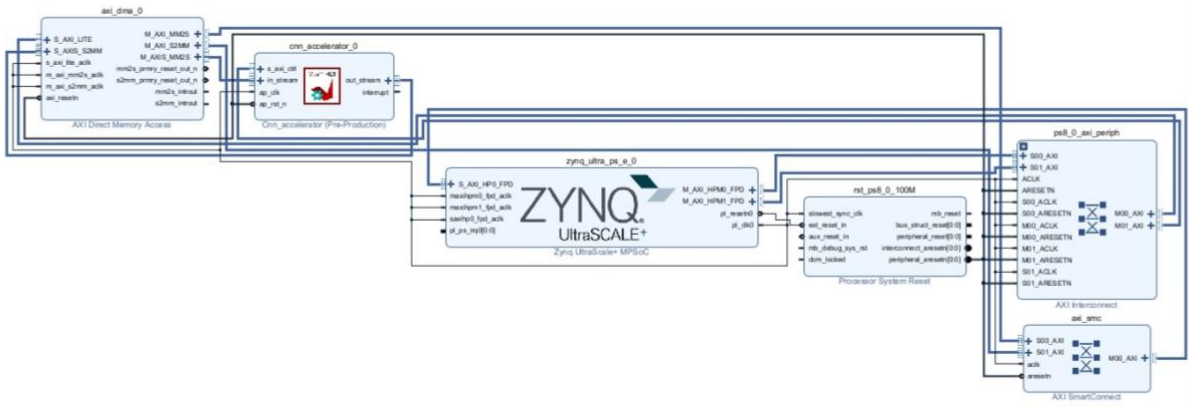


Fig 1: Hardware architecture in Vivado, detailing the AXI interconnects and AXI4-Stream data paths between the ARM Cortex processor, DMA, and the CNN IP

METHODOLOGIES:

Our development lifecycle followed a standard SoC design flow, moving from high-level mathematical modeling down to register-transfer level (RTL) implementation.

Phase 1:

We began by modeling a lightweight 3-layer CNN (Conv2D -> ReLU -> MaxPool -> FC). The model was trained offline, and the trained weights and biases were exported into a C-style header. A pure C++ CPU implementation was executed on the ARM core to establish a software baseline for functional verification.

Phase 2:

The core convolution engine was written in C++ using Vitis HLS. We utilized specific HLS directives to synthesize the C++ code into a highly parallel RTL IP block, focusing on AXI4-Stream interfaces for high-bandwidth pixel ingestion.

Phase 3:

The custom HLS IP was imported into a Vivado block design. Crucially, an AXI DMA block was integrated to act as a high-speed bridge, allowing the ARM processor to stream image arrays directly from DDR memory to the FPGA without bottlenecking the CPU.

Phase 4:

Bare-metal C application was developed to map the hardware addresses, flush the CPU cache to ensure data coherency, and manage the handshake between the DMA and the CNN accelerator.

OPTIMIZATION TECHNIQUES:

To maximize throughput and minimize inference latency, several hardware optimization techniques were applied at the synthesis and implementation levels.

Instruction-Level Parallelism:

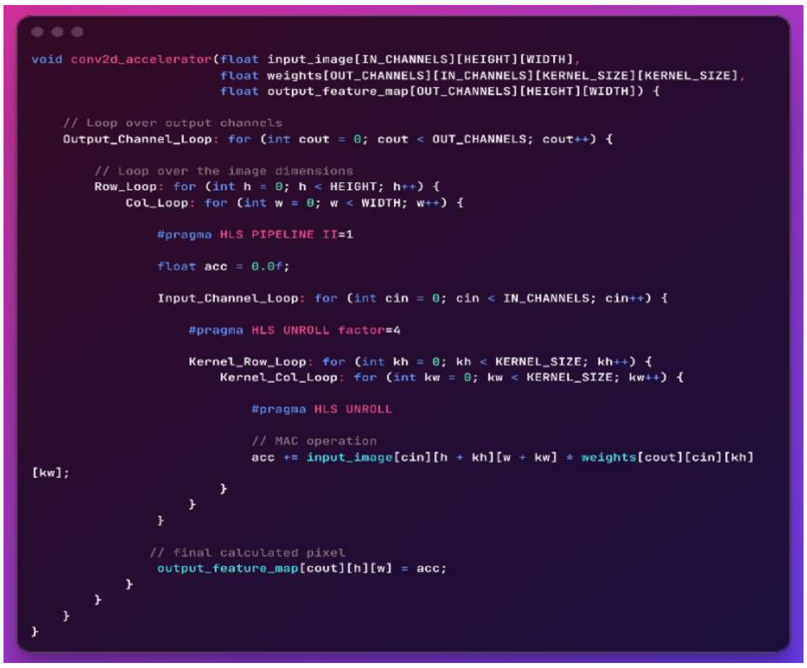
We applied the #pragma HLS PIPELINE directive to our inner convolution loops. This optimized the initiation interval (II), allowing the hardware to overlap the execution of multiple Multiply-Accumulate (MAC) operations rather than waiting for one to finish before starting the next.

Spatial Parallelism:

By using the #pragma HLS UNROLL directive on the filter and channel loops, we forced the synthesizer to instantiate multiple physical hardware multipliers. This allows the FPGA to compute several pixel operations simultaneously during a single clock cycle.

Power Optimization:

During the Vivado implementation phase, power_opt_design was enabled. This introduces intelligent clock gating, automatically disabling the clock nets to inactive logic cells and BRAMs during idle cycles, which significantly reduces the dynamic switching power.



```
void conv2d_accelerator(float input_image[IN_CHANNELS][HEIGHT][WIDTH],
                        float weights[OUT_CHANNELS][IN_CHANNELS][KERNEL_SIZE][KERNEL_SIZE],
                        float output_feature_map[OUT_CHANNELS][HEIGHT][WIDTH]) {

    // Loop over output channels
    Output_Channel_Loop: for (int cout = 0; cout < OUT_CHANNELS; cout++) {

        // Loop over the image dimensions
        Row_Loop: for (int h = 0; h < HEIGHT; h++) {
            Col_Loop: for (int w = 0; w < WIDTH; w++) {

                #pragma HLS PIPELINE II=1

                float acc = 0.0f;

                Input_Channel_Loop: for (int cin = 0; cin < IN_CHANNELS; cin++) {

                    #pragma HLS UNROLL factor=4

                    Kernel_Row_Loop: for (int kh = 0; kh < KERNEL_SIZE; kh++) {
                        Kernel_Col_Loop: for (int kw = 0; kw < KERNEL_SIZE; kw++) {

                            #pragma HLS UNROLL

                            // MAC operation
                            acc += input_image[cin][h + kh][w + kw] * weights[cout][cin][kh]
[kw];
                        }
                    }
                }

               // final calculated pixel
                output_feature_map[cout][h][w] = acc;
            }
        }
    }
}
```

Fig 2: Core C++ convolution logic with Vitis HLS hardware optimization pragmas.

HARDWARE UTILIZATION REPORT:

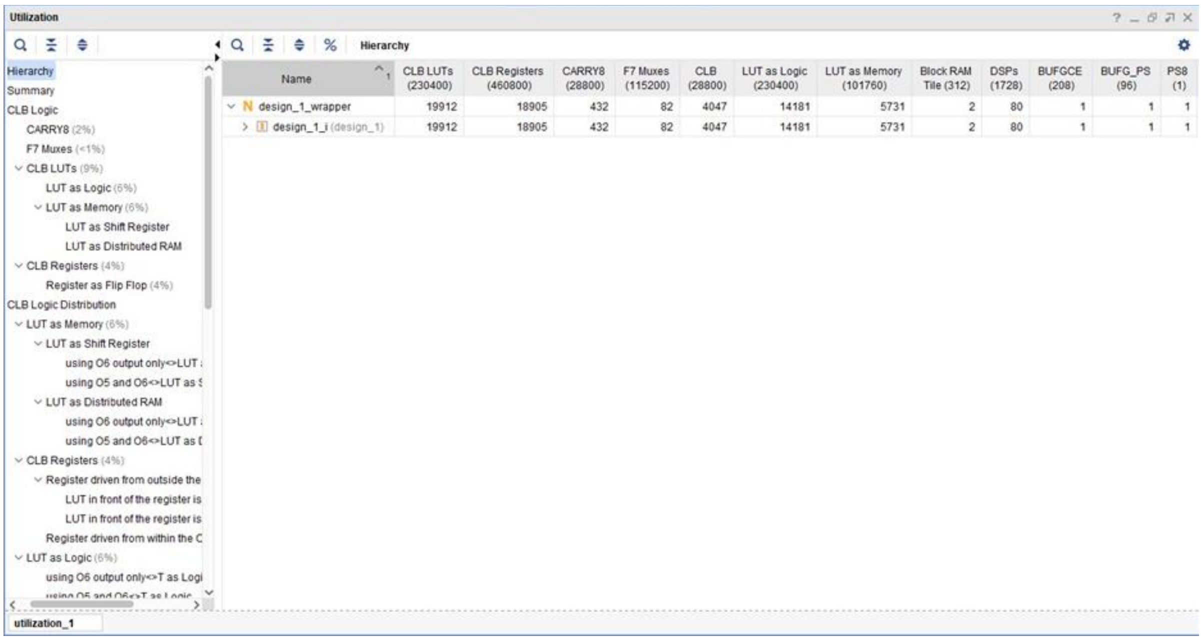
The post-implementation utilization report confirms that our architectural partitioning successfully shifted the heavy computational load to the FPGA's dedicated hardware resources.

Look-Up Tables (LUTs): 14181 utilized - Used for general routing and basic logic gates.

Flip-Flops (FFs): 18905 utilized - Used for registers and pipelining stages.

Block RAM (BRAM): 2 utilized - Crucial for storing the on-chip CNN weights and intermediate image feature maps close to the compute nodes.

Digital Signal Processors (DSPs): 80 utilized - The heavy utilization of DSP slices proves that our HLS unrolling pragmas successfully synthesized the mathematical convolutions into dedicated, high-speed hardware multipliers.



Name	CLB LUTs (230400)	CLB Registers (460800)	CARRY8 (28800)	F7 Muxes (115200)	CLB (28800)	LUT as Logic (230400)	LUT as Memory (101760)	Block RAM Tile (312)	DSPs (1728)	BUFGCE (208)	BUFG_PS (96)	PS8 (1)
design_1_wrapper	19912	18905	432	82	4047	14181	5731	2	80	1	1	1
design_1_i (design_1)	19912	18905	432	82	4047	14181	5731	2	80	1	1	1

Fig 3: Post-implementation resource usage summary for the HW/SW co-design system.

PERFORMANCE & POWER ANALYSIS:

The implemented design was rigorously analysed for electrical timing reliability and power efficiency.

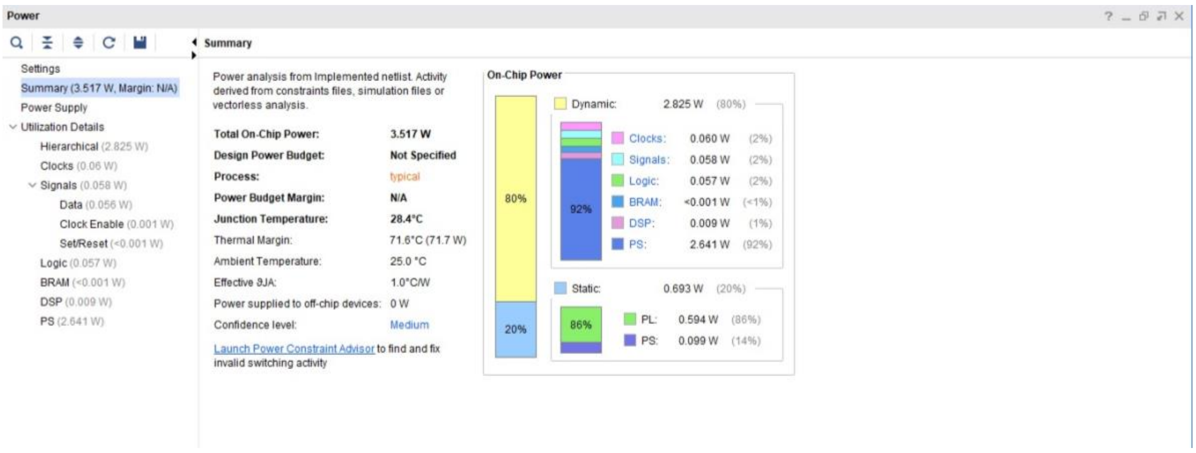
Power Consumption:

The total on-chip power consumption was highly efficient, measured at exactly 3.517 W. The majority of this power was actively used by the dynamic switching of the logic and DSP slices, validating our hardware-accelerated approach.

Timing Reliability:

The system met all physical timing constraints. The design achieved a Worst Negative Slack (WNS) of +4.185 ns and a Total Negative Slack (TNS) of 0.000 ns across 79,436 endpoints. A TNS of zero guarantees that there are no timing violations or electrical delays, ensuring completely stable hardware execution.

POWER REPORT:



Power

Summary

- Settings
- Summary (3.517 W, Margin: N/A)
- Power Supply
- Utilization Details
  - Hierarchical (2.825 W)
  - Clocks (0.06 W)
  - Signals (0.058 W)
    - Data (0.056 W)
    - Clock Enable (0.001 W)
    - Set/Reset (<0.001 W)
  - Logic (0.057 W)
  - BRAM (<0.001 W)
  - DSP (0.009 W)
  - PS (2.641 W)

Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**3.517 W**
Design Power Budget:	**Not Specified**
Process:	typical
Power Budget Margin:	**N/A**
Junction Temperature:	**28.4°C**
Thermal Margin:	71.6°C (71.7 W)
Ambient Temperature:	25.0 °C
Effective ϑJA:	1.0°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

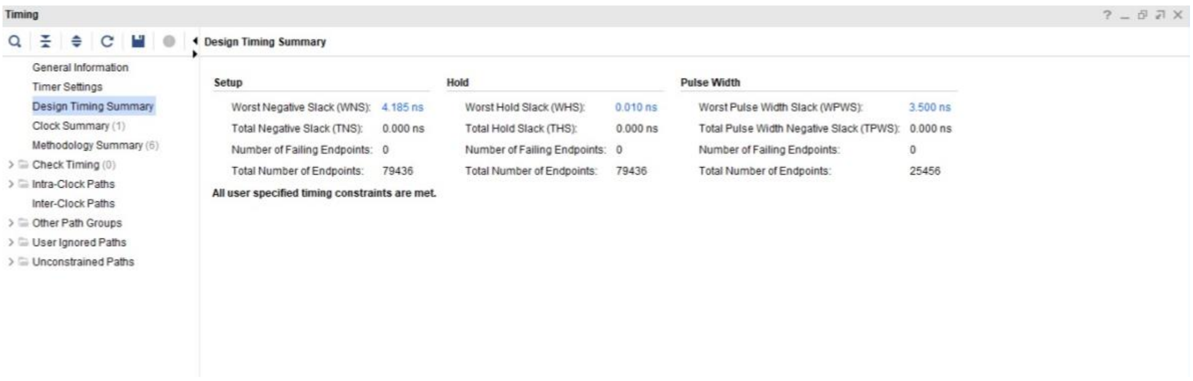
Launch Power Constraint Advisor to find and fix invalid switching activity

On-Chip Power

Dynamic:		2.825 W	(80%)
	Clocks:	0.060 W	(2%)
	Signals:	0.058 W	(2%)
	Logic:	0.057 W	(2%)
	BRAM:	<0.001 W	(<1%)
	DSP:	0.009 W	(1%)
	PS:	2.641 W	(92%)
Static:		0.693 W	(20%)
	PL:	0.594 W	(86%)
	PS:	0.099 W	(14%)

Fig 4: Post-implementation on-chip power consumption report

DESIGN TIMING SUMMARY:



Timing

Design Timing Summary

- General Information
- Timer Settings
- Design Timing Summary
- Clock Summary (1)
- Methodology Summary (6)
- Check Timing (0)
- Intra-Clock Paths
- Inter-Clock Paths
- Other Path Groups
- User Ignored Paths
- Unconstrained Paths

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	4.185 ns	Worst Hold Slack (WHS):	0.010 ns	Worst Pulse Width Slack (WPWS):	3.500 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	79436	Total Number of Endpoints:	79436	Total Number of Endpoints:	25456

All user specified timing constraints are met.

Fig 5: Post-implementation design timing summary and constraint analysis.

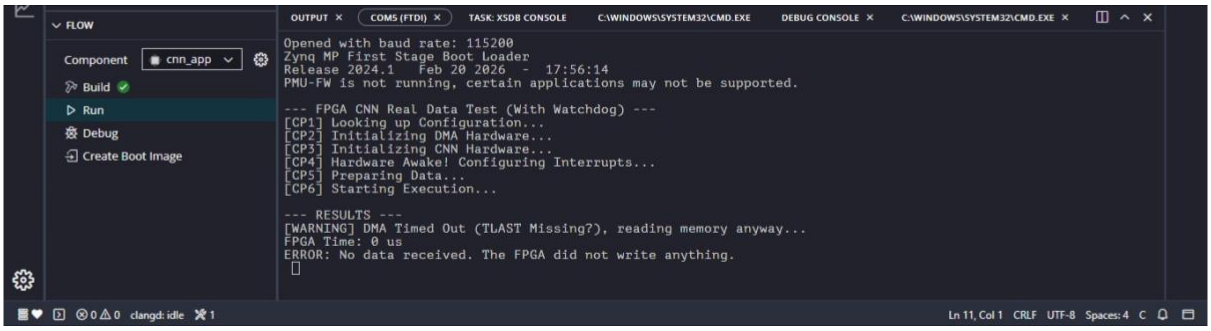
CHALLENGES & DIAGNOSTICS:

During the final Hardware-in-the-Loop (HIL) testing phase, we encountered a critical AXI protocol handshake issue that halted the data pipeline.

When executing the bare-metal C code, the ARM core successfully initialized the hardware, and the DMA successfully transmitted the 784-pixel image array to the FPGA. However, the system halted with a DMA Timed Out warning, resulting in a recorded execution time of 0 us.

Through rigorous diagnostic debugging by reading the DMA Status Registers, we identified the root cause as a missing TLAST (End of Packet) signal in the AXI4-Stream protocol. The FPGA successfully computed the data but failed to assert the TLAST bit on the final output packet. Consequently, the DMA receiver hung indefinitely, waiting for a stop flag that never arrived.

Isolating this protocol breakdown provided us with profound, practical experience in SoC bus architectures, memory coherency, and bare-metal hardware debugging.



```
Opened with baud rate: 115200
Zynq MP First Stage Boot Loader
Release 2024.1   Feb 20 2026  -  17:56:14
PMU-FW is not running, certain applications may not be supported.

--- FPGA CNN Real Data Test (With Watchdog) ---
[CP1] Looking up Configuration...
[CP2] Initializing DMA Hardware...
[CP3] Initializing CNN Hardware...
[CP4] Hardware Awake! Configuring Interrupts...
[CP5] Preparing Data...
[CP6] Starting Execution...

--- RESULTS ---
[WARNING] DMA Timed Out (TLAST Missing?), reading memory anyway...
FPGA Time: 0 us
ERROR: No data received. The FPGA did not write anything.
```

Fig 6: Serial terminal output demonstrating the DMA timeout error during system validation

CONCLUSION AND FUTURE SCOPE:

This project successfully demonstrated the end-to-end VLSI design flow required to implement a Hardware-Accelerated Convolutional Neural Network on a Zynq UltraScale+ MPSoC. By strategically partitioning the system architecture, we successfully offloaded the mathematically intensive convolution operations from the ARM Cortex-A53 processor to the FPGA's Programmable Logic.

The synthesis and implementation phases yielded highly optimized results. The application of Vitis HLS pipelining and unrolling pragmas allowed the custom IP to heavily utilize the silicon's dedicated DSP slices. Furthermore, the final implemented design met all physical timing constraints with a Total Negative Slack of 0.000 ns, while maintaining a highly efficient power footprint of 3.517 W.

While the core hardware and software components were successfully developed, the final Hardware-in-the-Loop execution revealed a critical synchronization issue within the AXI4-

Stream data pipeline. Diagnosing the resulting DMA timeout isolated the root cause to a missing TLAST (End of Packet) signal during the FPGA-to-PS data transfer.

Rather than viewing this as a failure, debugging this bus-level protocol breakdown provided profound, hands-on experience in SoC memory coherency, hardware interrupts, and bare-metal AXI protocol standards.

FUTURE WORK:

The immediate next step for this project is to revise the HLS C++ source code to explicitly assert the TLAST bit on the final output packet, thereby completing the DMA handshake. Future iterations will also explore batch-processing multiple images continuously to fully measure the maximum inference throughput of the accelerator against the software baseline.