Bharat AI-SoC Student Challenge

Problem Statement 5 — Technical Project Report

Real-Time Object Detection Using Hardware-Accelerated CNN on AMD Xilinx Kria KV260 (Zynq UltraScale+ MPSoC)

Hardware/Software Co-Design for Edge AI Inference — Full Technical Report

Submitted in partial fulfillment of the competition requirements

Platform: AMD Xilinx Kria KV260 Vision AI Starter Kit
SoC: Zynq UltraScale+ MPSoC (K26 SOM)
Tools: Vitis HLS 2024.2 | Vivado 2024.2 | PYNQ 3.0
FEB-2026

Team Details

Team Name
TESTBENCHERS

Team Leader
VIGNESH S
3rd Year | B.E. Electronics & Communication Engineering

Team Members
N A AKILAN
2nd Year | B.E. Electronics & Communication Engineering

DEVARAM A
2nd Year | B.E. Electronics & Communication Engineering

Faculty Mentor
Dr. AATHILAKSHMI S
Associate Professor, Dept. of Electronics & Communication Engineering

Chennai Institute of Technology, Chennai

FPGA-Accelerated Edge AI | CNN Inference | AXI DMA | Vitis HLS | Zynq UltraScale+

Abstract

This report presents the design, implementation, optimization, and performance evaluation of a hardware-accelerated Convolutional Neural Network (CNN) inference engine deployed on the AMD Xilinx Kria KV260 Vision AI Starter Kit. The underlying system-on-chip is the Zynq UltraScale+ MPSoC (K26 SOM), which integrates a quad-core ARM Cortex-A53 Processing System (PS) and a reconfigurable FPGA Programmable Logic (PL) fabric on a single die, providing an optimal platform for hardware/software co-design targeting edge artificial intelligence workloads.

The proposed system implements a custom lightweight CNN trained for single-class person detection. The model employs quantization-aware training techniques, mapping 32-bit floating-point weights to a 16-bit fixed-point representation parameterized as ap_fixed<16,6>, thereby reducing data bandwidth, simplifying hardware arithmetic, and enabling efficient deployment on FPGA fabric with constrained resources. The hardware accelerator is described in synthesizable C++ using Vitis High-Level Synthesis (HLS), leveraging AXI4-Stream interfaces for streaming data flow and AXI4-Lite for control-plane communication. Data transfers between DDR memory and the accelerator are managed by the AXI Direct Memory Access (DMA) IP core, avoiding CPU intervention in the data path.

Extensive hardware optimization directives — including loop pipelining at initiation interval II = 1, array partitioning, DATAFLOW region pragma, and a line-buffer architecture to reduce BRAM utilization by approximately 70% — were applied to achieve efficient resource usage and timing closure. Integration was performed in Vivado 2022.1, followed by bitstream deployment and Python-based inference orchestration via the PYNQ framework.

Performance benchmarking demonstrates that the PS + PL co-designed implementation achieves an inference latency of approximately 215 ms per frame, compared to 630 ms for the CPU-only baseline executing the same model on the ARM Cortex-A53. This corresponds to a measured speedup of approximately 2.93x, exceeding the minimum 2x speedup requirement mandated by the competition problem statement. The report provides a comprehensive technical account of the architecture, engineering challenges encountered and resolved, resource utilization, power efficiency considerations, and design trade-offs.

— Table of Contents —

1. Introduction 3
2. Problem Statement Description 4
3. Background on Edge AI and FPGA Acceleration 4
4. System Architecture: PS–PL Co-Design 5
5. Hardware Design Using Vitis HLS 6
6. AXI DMA and Dataflow Architecture 8
7. Vivado Block Design Integration 9
8. Quantization Strategy and Fixed-Point Mapping 10
9. Training Methodology 11
10. Debugging and Engineering Challenges 12
11. Performance Analysis 14
12. Resource Utilization 15
13. Power and Efficiency Considerations 16
14. Comparison with CPU-Only Implementation 16
15. Design Trade-offs 17
16. Future Improvements 17
17. Final System Outcomes & Achievements 18
18. Conclusion 20
19. References 21

1. Introduction

The proliferation of intelligent surveillance, autonomous robotics, and industrial inspection systems has created an urgent demand for low-latency, energy-efficient computer vision at the network edge. Traditional cloud-based inference pipelines suffer from network-induced latency, bandwidth bottlenecks, and privacy concerns that are fundamentally incompatible with real-time safety-critical applications. Deploying trained deep learning models directly on edge hardware — in particular, on heterogeneous System-on-Chip (SoC) platforms that combine programmable processor cores with reconfigurable logic fabric — represents one of the most promising approaches to address these constraints.

The AMD Xilinx Kria KV260 Vision AI Starter Kit, built on the Zynq UltraScale+ MPSoC architecture, embodies this heterogeneous computing paradigm. The K26 SOM integrates a quad-core ARM Cortex-A53 cluster (the Processing System, or PS) with a substantial FPGA fabric (the Programmable Logic, or PL) on a single silicon die. Communication between the two domains occurs through high-bandwidth, low-latency AXI interconnects, enabling a hardware/software co-design methodology in which computationally intensive operations are offloaded to dedicated hardware accelerators while higher-level control and orchestration logic remains in software.

This project targets Person Detection as a representative real-time vision task. A custom lightweight CNN was designed, trained with quantization-aware techniques, and deployed as an FPGA-resident convolution accelerator described in Vitis HLS synthesizable C++. The overarching objective is to demonstrate that a carefully designed hardware accelerator, properly integrated via AXI DMA and controlled through the PYNQ Python framework, can deliver inference latency improvements of at least 2x relative to a pure ARM CPU baseline — and to document every engineering decision, optimization applied, and obstacle overcome in achieving that result.

The remainder of this report is structured as follows: Section 2 characterizes the problem statement in detail; Section 3 provides foundational background on edge AI and FPGA acceleration; Sections 4 through 9 describe the system architecture, hardware design, data movement architecture, Vivado integration, quantization strategy, and training methodology respectively; Section 10 chronicles the engineering challenges encountered; Sections 11 through 16 present performance analysis, resource utilization, power considerations, comparison with CPU, trade-offs, and future directions; Sections 17 and 18 conclude the report and list references.

2. Problem Statement Description

Competition Problem Statement 5 of the Bharat AI-SoC Student Challenge mandates the implementation of a real-time object detection system on an AMD Xilinx FPGA SoC platform. The specific requirements are:

- The system must be deployed on a Zynq-family or Kria-series SoC that exposes both a programmable processor and FPGA fabric.
- A convolutional neural network (CNN) must be used for the detection task. The network may be custom-designed or derived from a published architecture, but must be adapted for FPGA deployment.
- The FPGA fabric must be actively utilized for CNN acceleration, not merely as a pass-through; the hardware accelerator must perform measurable arithmetic computation.

- The measured inference speedup of the PS+PL co-designed system over a CPU-only baseline executing the same model must be at least 2x.
- All hardware descriptions must be synthesizable; the accelerator must be demonstrated on real hardware with a generated bitstream.

The target application selected is single-class Person Detection on static frames. Person detection is a foundational task in video surveillance, smart city systems, and autonomous vehicles, and provides a realistic workload for evaluating the throughput characteristics of a custom CNN accelerator. The system must process frames at sufficient throughput to support practical deployment, with the 2x speedup threshold serving as the minimum bar for hardware acceleration to be considered meaningful.

3. Background on Edge AI and FPGA Acceleration

3.1 Edge Artificial Intelligence

Edge AI refers to the execution of machine learning inference workloads on devices located physically close to the data source, rather than in centralized cloud data centers. The motivations are multifold: latency reduction (eliminating round-trip network delays), bandwidth conservation (avoiding transmission of raw video streams), privacy preservation (keeping sensitive data on-device), and resilience (continued operation during network outages). The primary constraint at the edge is the strict power budget — typically tens of watts for embedded platforms — which precludes deployment of large, computation-intensive models without significant architectural adaptation.

3.2 Convolutional Neural Networks for Vision

Convolutional Neural Networks have become the canonical architecture for visual recognition tasks since the seminal work of LeCun et al. [1] and the ImageNet breakthrough of Krizhevsky et al. [2]. A CNN processes input images through successive layers of learned linear convolution kernels followed by nonlinear activations (ReLU being most prevalent), spatial downsampling via max-pooling or strided convolution, and ultimately fully connected layers or global pooling for classification. The fundamental operation — computing output feature map O[n][oc][oh][ow] from input feature map I[n][ic][ih][iw] and filter weight W[oc][ic][kh][kw] — is given by the discrete convolution sum:

```
O[n][oc][oh][ow] = SUM_{ic} SUM_{kh} SUM_{kw} W[oc][ic][kh][kw] *
                   I[n][ic][oh*S+kh][ow*S+kw] + b[oc]
```

where S is the convolution stride and b[oc] is the per-channel bias. This computation is characterized by a high degree of data reuse (the same weight kernel is applied to every spatial position), making it amenable to deep pipeline execution in hardware.

3.3 FPGA Acceleration of CNN Inference

Field-Programmable Gate Arrays offer a distinctive combination of attributes that makes them particularly well-suited for CNN inference: fine-grained spatial parallelism (multiple multiply-accumulate units executing in parallel), configurable data paths (fixed-point arithmetic can be sized to exactly the precision required), deep pipelining (new inputs can enter the pipeline every clock cycle once it is filled), and deterministic real-time behavior (no operating system scheduling jitter). Unlike GPU acceleration, FPGA

acceleration operates at comparatively low power (often 5–15W) and achieves deterministic latency, critical for real-time systems.

The key performance metric for pipelined hardware loops is the Initiation Interval (II), defined as the minimum number of clock cycles that must elapse between successive initiations of a pipelined loop iteration or function invocation. An II = 1 indicates that the pipeline can accept a new input every clock cycle, delivering maximum theoretical throughput. Achieving II = 1 requires that no loop-carried data dependency or resource conflict exists that would force a stall. Techniques such as loop restructuring, array partitioning, and register retiming are applied to remove such dependencies.

```
Throughput = f_clk / II      [operations per second]
```

The Zynq UltraScale+ MPSoC employed in this project combines a hardened ARM processor cluster with a state-of-the-art FPGA fabric featuring UltraRAM (URAM), Block RAM (BRAM), Digital Signal Processing (DSP48E2) slices, and configurable logic blocks (CLBs), all interconnected via a high-bandwidth AXI network-on-chip. This architecture has been validated for production-grade edge AI deployment in automotive, industrial, and medical imaging domains.

4. System Architecture: PS–PL Co-Design

4.1 Architectural Overview

The system adopts a canonical hardware/software co-design partitioning strategy. The ARM Cortex-A53 Processing System handles all high-level control operations: frame acquisition, pre-processing, DMA transaction scheduling, post-processing (bounding box decoding, non-maximum suppression), and result rendering. The FPGA Programmable Logic implements the time-critical inner loops of the CNN inference pipeline: the convolution operation, activation, and pooling — the operations that collectively account for over 95% of inference compute time in a typical CNN.

The PS and PL communicate exclusively through the AXI interconnect fabric. Input feature maps and weight tensors are stored in DDR4 SDRAM, accessible by both domains. The AXI DMA IP core orchestrates bulk data transfer from DDR to the accelerator's AXI4-Stream input port and from the accelerator's output port back to DDR, without CPU involvement in the data movement itself. Once a DMA transaction is initiated by the PS, the entire data transfer and processing pipeline executes autonomously in hardware.

4.2 PS Responsibilities

The Processing System, running the PYNQ Linux distribution, executes a Python inference script that performs the following operations in sequence: (1) load input frame from storage or camera interface and perform normalization and resizing to the expected input tensor dimensions; (2) allocate contiguous PYNQ DMA buffer objects aligned to cache-line boundaries; (3) copy pre-processed tensor data into the MM2S (Memory-to-Stream) DMA source buffer; (4) perform cache flush (dcache_flush) to ensure DDR coherency before DMA transaction; (5) initiate MM2S and S2MM (Stream-to-Memory-to-Stream) DMA channels; (6) poll or interrupt-wait for DMA completion; (7) perform cache invalidate (dcache_invalidate) on the destination buffer; (8) read the output tensor from the S2MM destination buffer; (9) apply sigmoid activation and decode detection bounding boxes.

4.3 PL Responsibilities

The Programmable Logic hosts the custom Convolutional Accelerator IP core synthesized from Vitis HLS. Upon receiving data from the AXI4-Stream input (MM2S channel), the accelerator: (1) buffers incoming pixel rows into a three-row line buffer resident in Block RAM; (2) constructs 3x3 input patches on demand from the line buffer; (3) performs fixed-point multiply-accumulate operations using DSP48E2 slices; (4) applies ReLU activation; (5) computes max-pooling over 2x2 windows; (6) streams the resulting output feature map values to the AXI4-Stream output (S2MM channel), asserting TLAST on the final transfer beat.

4.4 Memory Map and Interconnect

The Zynq UltraScale+ address map assigns the DDR SDRAM to the range 0x0000_0000 to 0x7FFF_FFFF. The AXI DMA IP is mapped at a fixed AXI4-Lite base address accessible to the ARM CPU for register-level control (SA_START_ADDRESS, DA_START_ADDRESS, LENGTH fields). The accelerator control interface is mapped at a separate AXI4-Lite address range that exposes the ap_start, ap_done, ap_idle, and ap_ready handshake signals, as well as configurable parameters (output channels, feature map dimensions, stride).

Block	Domain	Interface	Function
ARM Cortex-A53	PS	AXI HP / AXI GP	Inference orchestration
DDR4 SDRAM	Shared	AXI HP0/HP1	Tensor storage
AXI DMA IP	PL	AXI4-Stream + AXI Lite	Data movement
CNN Accelerator	PL	AXI4-Stream + AXI Lite	Convolution compute
AXI Interconnect	PL	AXI4	Bus fabric routing
Clock Wizard	PL	Clocking	PL clock generation

Table 1: System block summary with domain and interface type

5. Hardware Design Using Vitis HLS

5.1 Design Philosophy and Tool Choice

Vitis High-Level Synthesis (Vitis HLS) enables hardware description at the algorithmic C++ level, automatically generating synthesizable RTL (VHDL or Verilog) from annotated source code. The adoption of HLS for this project is motivated by several considerations: the ability to rapidly iterate on architectural variations without manual RTL re-coding; the availability of optimization directives (pragmas) that map directly to established hardware micro-architecture patterns (pipelines, loop unrolling, array partitioning); and the seamless integration of HLS-generated IP cores into the Vivado IP Integrator block design flow.

5.2 Accelerator Function Signature

The top-level accelerator function is declared with AXI4-Stream ports for data ingress and egress, and an AXI4-Lite port for control and configuration. The simplified function signature is as follows:

```
void conv_accelerator(
    hls::stream<ap_axis<64,0,0,0>> &in_stream,
    hls::stream<ap_axis<64,0,0,0>> &out_stream,
    ap_uint<8> out_channels,
    ap_uint<8> feature_h,
    ap_uint<8> feature_w,
    ap_fixed<16,6> weights[MAX_OC][MAX_IC][KH][KW],
    ap_fixed<16,6> bias[MAX_OC]
);
```

The ap_axis<64,0,0,0> template type wraps 64-bit data with AXI4-Stream sideband signals TLAST, TKEEP, and TSTRB. TLAST is asserted on the final beat of each frame or tensor transfer, signaling end-of-frame to both the AXI DMA and any downstream consumers.

5.3 AXI Interface Pragma Annotations

Three HLS interface pragmas govern the port binding of the top-level function to AXI protocols:

```
#pragma HLS INTERFACE axis port=in_stream
#pragma HLS INTERFACE axis port=out_stream
#pragma HLS INTERFACE s_axilite port=return bundle=CTRL
#pragma HLS INTERFACE s_axilite port=out_channels bundle=CTRL
#pragma HLS INTERFACE s_axilite port=feature_h bundle=CTRL
#pragma HLS INTERFACE s_axilite port=feature_w bundle=CTRL
```

The s_axilite pragma on the return port generates the standard ap_start/ap_done/ap_idle/ap_ready handshake registers, allowing the PS to start and poll the accelerator through memory-mapped register accesses.

5.4 Line-Buffer Architecture

A central optimization in the accelerator is the three-row line-buffer architecture. Naive implementation of a 3x3 convolution would require storing the entire input feature map in BRAM and performing random-access reads to construct each 3x3 input patch. For a 32x32 input feature map with 8-bit representation, this would require 32 * 32 * 3 = 3,072 bytes per input channel, and BRAM consumption scales linearly with the number of input channels.

The line-buffer approach exploits the sequential, raster-scan order in which convolution patches are accessed. Three shift registers, each one row wide (equal to the feature map width), store the three most recently received rows of the input tensor. As new pixel values arrive via the AXI4-Stream input, they are shifted into the line buffer, and a 3x3 sliding window is constructed from the three rows and a three-element column shift register. The BRAM requirement reduces from storing the entire feature map to storing only three rows:

BRAM_line_buffer = 3 * W * C * sizeof(ap_fixed<16,6>) vs BRAM_full = H * W * C * sizeof(ap_fixed<16,6>)

For a 32x32x8 input (width 32, 8 channels, 16-bit elements), the line buffer requires 3 * 32 * 8 * 2 = 1,536 bytes vs 32 * 32 * 8 * 2 = 16,384 bytes for full storage — a reduction factor of approximately 10.7x. Measurements from the Vivado implementation reports confirm approximately 70% BRAM reduction relative to the naive array-based baseline design.

5.5 Loop Pipelining and Initiation Interval

The innermost computation loop — iterating over output channels, kernel height, and kernel width — is annotated with the PIPELINE directive requesting II = 1:

```
COMPUTE_OC: for(int oc = 0; oc < out_channels; oc++) {
  COMPUTE_KH: for(int kh = 0; kh < 3; kh++) {
    COMPUTE_KW: for(int kw = 0; kw < 3; kw++) {
      #pragma HLS PIPELINE II=1
      acc[oc] += window[kh][kw] * weights[oc][ic][kh][kw];
    }
  }
}
```

Achieving II = 1 on the innermost loop requires that no loop-carried dependency exists across loop iterations and that the memory banking allows concurrent access to multiple elements. This is enabled by array partitioning of the weights array across the last two dimensions (kh, kw), ensuring that the weight values for all kernel positions are simultaneously accessible without port conflicts.

5.6 Array Partitioning

The weight array is partitioned using the ARRAY_PARTITION directive, which replicates the BRAM storage into multiple banks, each with an independent read/write port:

```
#pragma HLS ARRAY_PARTITION variable=weights dim=3 complete
#pragma HLS ARRAY_PARTITION variable=weights dim=4 complete
#pragma HLS ARRAY_PARTITION variable=window complete
```

Complete partitioning on dimensions 3 and 4 (kernel height and width) creates 3 * 3 = 9 independent weight registers, enabling all nine multiply-accumulate operations of the 3x3 convolution kernel to execute in a single clock cycle, contributing to the II = 1 achievement.

5.7 DATAFLOW Optimization

The DATAFLOW pragma is applied at the top level of the accelerator function to enable task-level pipelining between the input ingestion stage, the convolution compute stage, and the output emission stage. Under DATAFLOW, these three stages execute as concurrent, communicating processes rather than sequentially:

```
#pragma HLS DATAFLOW
```

Data flows between stages via HLS streams (hls::stream<T>) or ping-pong buffers, allowing the output emission of processed feature map data to overlap with the ingestion of subsequent input rows. This reduces the overall latency and improves sustained throughput by hiding data transfer overhead behind computation.

6. AXI DMA and Dataflow Architecture

6.1 AXI Protocol Background

The AXI (Advanced eXtensible Interface) protocol family, part of the ARM AMBA specification, is the standard interconnect fabric for Zynq and UltraScale+ devices. Three AXI protocol variants are relevant to this design:

- AXI4 (Full AXI): Used for high-bandwidth, burst-capable memory-mapped transactions between processing masters (CPU, DMA) and DDR memory slaves.
- AXI4-Lite: Used for low-bandwidth, single-beat register access — appropriate for control-plane operations such as configuring the DMA base address, length, and the accelerator's ap_start register.
- AXI4-Stream: A unidirectional streaming protocol with no address phase; data flows continuously from master to slave, qualified by TVALID/TREADY handshake and terminated by TLAST. Ideal for streaming input feature map data to the accelerator and receiving output feature maps.

6.2 AXI DMA IP Core Configuration

The Xilinx AXI DMA IP core is configured in Scatter-Gather mode disabled (simple DMA mode) with both Memory-to-Stream (MM2S) and Stream-to-Memory (S2MM) channels enabled. The data width is set to 64 bits (matching the accelerator's AXI4-Stream port width) with a maximum burst length of 256 beats, balancing DDR bandwidth efficiency with FIFO buffer size. The DMA is connected to the ARM HP (High Performance) AXI slave ports of the Zynq PS, which provide direct DDR access without routing through the General-Purpose AXI port.

A DMA transfer is initiated by writing the source buffer physical address to the MM2S SA register, the destination buffer physical address to the S2MM DA register, and the transfer byte count to the MM2S LENGTH and S2MM LENGTH registers. The PS then writes to the MM2S DMACR.RS and S2MM DMACR.RS bits to start both channels simultaneously. Transfer completion is indicated either by polling the DMASR.IOC_Irq bit or by an interrupt assertion.

6.3 TLAST and End-of-Frame Signaling

A critical requirement for correct AXI DMA operation is the correct assertion of the TLAST signal on the AXI4-Stream output of the accelerator. TLAST serves as the end-of-packet indicator, telling the S2MM channel to commit the received data to DDR and assert the transfer complete interrupt. Failure to assert TLAST at the correct position — specifically, exactly on the final data beat corresponding to the last element of the output feature map — results in the DMA waiting indefinitely for a TLAST that never arrives, causing an observable hang with no error indication.

The accelerator maintains an output beat counter that increments on each valid output transfer. TLAST is asserted when the counter value equals the expected total output size (out_channels * out_height * out_width) minus one:

```
out_beat.last = (output_count == (OC * OH * OW - 1)) ? 1 : 0;
```

This was a significant source of difficulty during integration testing, as described in Section 10. The resolution required careful accounting of the exact output tensor dimensionality including padding and pooling effects.

6.4 Cache Coherency Management

The ARM Cortex-A53 caches (L1 and L2) are not automatically coherent with DMA operations that bypass the CPU cache hierarchy and access DDR directly. Without explicit cache management, the DMA may read stale data from DDR (data that the CPU has modified but not yet written back from cache), or the CPU may read stale data from cache (data that the DMA has written to DDR but the CPU cache has not invalidated).

The PYNQ framework's xlnk.CmaArray and pynq.allocate() allocators return physically contiguous, cache-line-aligned DMA buffers. Before initiating a MM2S transfer, the PS flushes the source buffer's cache lines to DDR. After the S2MM transfer completes, the PS invalidates the destination buffer's cache lines, forcing subsequent CPU reads to reload from DDR where the DMA has written the fresh results:

```
src_buf.flush()   # dcache flush before MM2S
dma.sendchannel.transfer(src_buf)
dma.recvchannel.transfer(dst_buf)
dma.sendchannel.wait()
dma.recvchannel.wait()
dst_buf.invalidate()  # dcache invalidate after S2MM
```

7. Vivado Block Design Integration

7.1 IP Integrator Methodology

Vivado IP Integrator (IPI) provides a graphical block design environment for assembling complex SoC designs from IP cores, configuring their parameters, and routing their interfaces through AXI interconnects. The block design for this project was assembled using the following methodology: (1) instantiate the Zynq UltraScale+ MPSoC IP and configure the PS for the Kria K26 SOM board preset, enabling AXI HP0 and HP1 interfaces; (2) add the AXI DMA IP; (3) add the Convolutional Accelerator IP (exported from Vitis HLS); (4) add an AXI Interconnect or SmartConnect to arbitrate PS master access to the DMA and accelerator AXI4-Lite slave ports; (5) connect AXI4-Stream ports between DMA and accelerator; (6) run connection automation; (7) add and configure the Clocking Wizard IP.

7.2 Clock Domain Configuration

The Programmable Logic is clocked by a user-defined PL fabric clock generated by the Clocking Wizard IP from the PS's 99.999 MHz reference output (pl_clk0). For this design, the PL fabric clock was configured at 150 MHz, providing a 6.67 ns clock period. This target frequency was chosen to balance pipeline performance against timing closure feasibility: higher frequencies demand tighter timing on critical paths through DSP slices and BRAM read ports, while lower frequencies reduce achievable throughput. All AXI interfaces (DMA, accelerator AXI4-Stream, AXI4-Lite) are clocked from this single domain, eliminating the need for AXI clock crossing logic.

7.3 Address Assignment and Validation

After connectivity is established, the Address Editor in Vivado IPI was used to assign memory-mapped addresses to the AXI4-Lite slave interfaces. The AXI DMA base address was set to 0xA000_0000 with a 64KB address range, and the accelerator control register base was set to 0xA001_0000. These addresses correspond to the ranges used in the PYNQ Python overlay control script. Vivado performs DRC (Design Rule Check) validation to verify address map consistency prior to synthesis.

7.4 Synthesis, Implementation, and Bitstream Generation

Following block design validation, the design was synthesized using Vivado synthesis with the strategy set to 'Flow_PerfOptimized_high' to maximize timing performance. After synthesis, place-and-route implementation was run with the 'Performance_ExplorePostRoutePhysOpt' strategy, enabling aggressive post-route physical optimization passes. Timing closure was achieved at the 150 MHz target after

addressing several timing violations that arose from the initial default implementation strategy. The generated bitstream (.bit) and hardware handoff file (.hwh) were deployed to the KV260 board via the PYNQ filesystem.

8. Quantization Strategy and Fixed-Point Mapping

8.1 Motivation for Quantization

Neural network models are conventionally trained using 32-bit single-precision floating-point arithmetic (FP32), which provides sufficient numerical dynamic range for the gradient-based optimization process. However, FP32 arithmetic is expensive to implement in FPGA fabric: a single-precision floating-point multiplier requires significant LUT and DSP resources and introduces longer critical paths compared to fixed-point arithmetic. For inference (as opposed to training), the full dynamic range of FP32 is rarely necessary, and the precision can be reduced substantially with minimal impact on accuracy if the quantization is handled correctly.

8.2 The ap_fixed<16,6> Format

The Vitis HLS arbitrary-precision fixed-point type ap_fixed<W,I> represents a signed fixed-point number with W total bits and I integer bits (including sign), yielding W-I fractional bits. For ap_fixed<16,6>, this gives 6 integer bits and 10 fractional bits:

```
ap_fixed<16,6>  =>  1 sign bit + 5 integer bits + 10 fractional bits
Representable range: [-32.0, +31.9990234375]
Resolution (LSB): 2^(-10) = 0.0009765625
```

This format is well-matched to the typical distribution of trained CNN weights, which after batch normalization tend to cluster in the range [-4.0, +4.0] with high density near zero. The 10 fractional bits provide sufficient precision to preserve inter-class discrimination in the weight distribution, while the 16-bit total word length enables two weights to be packed into a single 32-bit word for efficient memory bandwidth utilization.

8.3 Quantization-Aware Training

Quantization-Aware Training (QAT) is a training-time methodology in which the effects of post-training quantization are simulated during the forward pass, allowing the optimizer to adapt the model weights to minimize accuracy degradation under quantized arithmetic. In this project, fake quantization nodes are inserted after every convolutional layer's parameter update: the FP32 weight values are rounded and clipped to the grid of ap_fixed<16,6> representable values during the forward pass, but gradients flow through in FP32 during backpropagation (the straight-through estimator approach [3]).

The fake quantization function for a scalar weight value w is defined as:

```
w_q = clip(round(w / delta) * delta,  w_min,  w_max)
```

where delta = 2^(-10) is the quantization step size, w_min = -32.0, and w_max = 31.999. After training converges, the final weights are exported as NumPy .npy arrays and loaded by the Vitis HLS accelerator simulation and hardware deployment scripts.

8.4 Quantization Error Analysis

The maximum quantization error for any weight value is bounded by half the LSB:

```
max_quant_error = delta / 2 = 2^(-11) = 0.000488
```

For a convolution layer with K = 3x3 = 9 kernel elements and C input channels, the accumulated quantization error in the output accumulator is bounded by:

```
max_accum_error <= K * C * max_quant_error * max_input_value
```

For K=9, C=8, max_input_value ≈ 1.0 (after normalization), the accumulated error is approximately 9 * 8 * 0.000488 ≈ 0.035, which is well below the ReLU activation threshold for any meaningfully activated neuron and does not materially affect detection accuracy for the person detection task.

9. Training Methodology

9.1 Dataset

The model was trained on a curated subset of the PASCAL VOC 2012 dataset [4], filtered to retain only images containing the 'person' class annotation. Bounding box annotations were converted to binary spatial ground-truth maps for the detection objective. The subset consists of approximately 3,500 training images and 800 validation images after filtering. Images were resized to 128x128 pixels during preprocessing and normalized to zero mean and unit variance using ImageNet statistics.

9.2 Network Architecture

The custom lightweight CNN architecture was designed with inference efficiency on constrained FPGA resources as the primary constraint. The architecture consists of three convolutional stages:

Layer	Type	Input Shape	Output Shape	Parameters
Conv1	Conv2D 3x3 + ReLU	128x128x3	64x64x8	8*(3*3*3+1) = 224
MaxPool1	MaxPool 2x2	64x64x8	32x32x8	0
Conv2	Conv2D 3x3 + ReLU	32x32x8	32x32x16	16*(3*3*8+1) = 1,168
MaxPool2	MaxPool 2x2	32x32x16	16x16x16	0
Conv3	Conv2D 3x3 + ReLU	16x16x16	16x16x16	16*(3*3*16+1) = 2,320
GlobalAvgPool	GAP	16x16x16	1x1x16	0
Dense	FC + Sigmoid	16	1	17
Total				3,729 parameters

Table 2: Custom lightweight CNN architecture for person detection

The total parameter count of 3,729 is intentionally small, ensuring that all weight tensors fit within the on-chip BRAM of the FPGA fabric without requiring DDR weight streaming, which would introduce latency and bandwidth overhead.

9.3 Training Configuration

Training was conducted in Google Colab using TensorFlow 2.x / Keras with the following hyperparameter configuration: batch size 32; optimizer Adam with initial learning rate 1e-3 and decay to 1e-4 at epoch 20; binary cross-entropy loss; training for 40 epochs total. The fake quantization callbacks were applied from epoch 5 onward, allowing the optimizer to first converge in full FP32 precision before adapting to the quantization constraints. Final training accuracy reached 94.2% and validation accuracy 91.8% on the binary detection task.

9.4 Weight Export

After training, the model weights for all convolutional and dense layers were extracted using layer.get_weights() and saved as NumPy .npy files. A post-export verification step confirmed that rounding each weight value to the nearest ap_fixed<16,6> representable value preserved the validation accuracy to within 0.5 percentage points, validating the quantization strategy.

10. Debugging and Engineering Challenges

This section provides a detailed account of the significant engineering challenges encountered during the development of the system, the diagnostic methodology applied, and the solutions implemented. These challenges represent a realistic and instructive account of the difficulties inherent in FPGA-based CNN deployment.

10.1 Colab RAM Crash During Dataset Caching

During dataset preparation in Google Colab, the TensorFlow tf.data pipeline was configured to cache the preprocessed dataset to memory. For the full PASCAL VOC image set, the in-memory cache exceeded the 12 GB Colab RAM allocation, causing the kernel to crash without saving progress, resulting in loss of preprocessed data and partial training state.

Resolution: The cache() call was removed from the data pipeline. Additionally, the dataset loading was restructured to use lazy loading with map() and prefetch() only, without any caching. Memory profiling confirmed that peak RAM usage dropped below 4 GB with this configuration.

10.2 Weight Export Failure

After initial training, a script error in the weight extraction code caused the output .npy files to be written with incorrect shape metadata — specifically, transposing the output channel and input channel dimensions of the weight tensor. This mismatch was not immediately apparent as the files were non-empty and appeared valid under a cursory inspection.

Resolution: The training and export were merged into a single monolithic script, eliminating the state transfer between separate notebook cells that had introduced the shape error. A shape assertion was added immediately after export to verify that each weight array matched the expected dimensions.

10.3 FP32 vs Fixed-Point Mismatch

The initial deployment workflow exported FP32 weights without applying quantization, and the Vitis HLS simulation was run with ap_fixed<16,6> weight arrays initialized by direct assignment from FP32 values.

Because C++ integer truncation is the default behavior for ap_fixed initialization (not rounding), values near quantization boundaries were systematically rounded toward zero, introducing a directional bias in the weight representation not present during QAT.

Resolution: Fake quantization was applied during training using custom Keras callbacks that simulate rounding to the nearest ap_fixed<16,6> grid point (round-half-to-even) using the formula: w_q = round(w * 1024) / 1024. Weight export was updated to apply this rounding prior to saving, ensuring the .npy values exactly matched the ap_fixed<16,6> representable set.

10.4 DMA Hanging Due to TLAST Issues

During hardware-in-the-loop testing, the S2MM DMA channel would enter a hung state — DMASR.Halted bit asserted, no transfer complete interrupt — after every transfer. The MM2S channel completed normally. PYNQ DMA wait() calls blocked indefinitely.

Root Cause Analysis: The accelerator was not asserting TLAST on the AXI4-Stream output at the correct position. Specifically, the output beat counter was initialized incorrectly (counting from 1 rather than 0), causing TLAST to be asserted one beat early. The S2MM DMA received TLAST before the final data beat, committed an incomplete transfer to DDR, and then received one additional spurious beat without TLAST, causing it to hang waiting for the next TLAST.

Resolution: The output beat counter was rebaselined to count from 0. TLAST was asserted when the counter value equals (total_output_beats - 1), matching the exact final beat. This was verified in HLS simulation before re-synthesizing.

10.5 AXI Stream Deadlock

After correcting the TLAST issue, a new problem emerged: the system would occasionally deadlock with both DMA channels reporting 'Running' state but no data transfer occurring. Chipscope ILA (Integrated Logic Analyzer) probing of the AXI4-Stream handshake signals revealed that the MM2S channel was asserting TVALID (data available) but the accelerator's TREADY was de-asserted, stalling the input side. Simultaneously, the accelerator's output TVALID was de-asserted, preventing the S2MM channel from consuming data and draining the output FIFO.

Root Cause Analysis: The accelerator's internal processing pipeline had a latency dependency: it could not assert output TREADY until it had received enough input beats to fill the 3-row line buffer (i.e., 3 * feature_width beats). However, the AXI DMA MM2S channel had a short FIFO that could stall if TREADY was not asserted promptly, and the stall caused the DMA to assert backpressure upstream, which in turn caused the AXI interconnect to stall all AXI transactions including the PS's AXI4-Lite write to the S2MM DA register — creating a circular dependency.

Resolution: An AXI4-Stream data FIFO IP with depth 1024 was inserted between the MM2S DMA output and the accelerator input, providing sufficient buffering to decouple the DMA's burst transfers from the accelerator's line-buffer fill latency.

10.6 Initiation Interval Greater Than 1

The initial Vitis HLS synthesis reported II = 3 for the innermost convolution loop, despite the PIPELINE II=1 pragma. The HLS synthesis log cited a loop-carried dependency on the accumulator array acc[], which was read and written in the same loop iteration.

Resolution: The accumulator was restructured as a local scalar variable inside the output channel loop, removing the array memory access from the critical path. Separately, the weight array was fully partitioned on the kernel spatial dimensions, eliminating BRAM read-port conflicts. After these changes, synthesis reported II = 1.

10.7 Timing Violations in Vivado Implementation

Post-route timing analysis reported several setup-time violations (negative slack) on paths through the ap_fixed multiply-accumulate chains at the initial 200 MHz target clock frequency. The worst negative slack was -0.73 ns on a path through a DSP48E2 cascade.

Resolution: The target clock frequency was reduced from 200 MHz to 150 MHz, providing additional timing margin. Additionally, the Vivado implementation strategy was changed to 'Performance_ExplorePostRoutePhysOpt', which enables post-route physical optimization. After these adjustments, timing closure was achieved with all setup and hold slack values positive (worst setup slack: +0.18 ns at 150 MHz).

10.8 DDR Cache Coherency Issue

During initial PYNQ deployment, inference results read from the S2MM destination buffer contained a mix of correct and stale (zeroed) values. The issue was non-deterministic, sometimes manifesting on the first transfer and sometimes only after 10–20 successive transfers.

Resolution: This was diagnosed as a cache coherency issue. The PYNQ pynq.allocate() call was verified to return a CmaArray with the correct physical address, but explicit cache management calls (flush before MM2S, invalidate after S2MM) were not being applied. Inserting src_buf.flush() before DMA start and dst_buf.invalidate() after DMA completion resolved the non-deterministic behavior entirely.

10.9 No Detection Due to Architecture–Weight Mismatch

After achieving functional DMA operation, the inference output was all-zeros regardless of the input image. This indicated that the forward pass was producing near-zero activations throughout the network.

Root Cause Analysis: The weight files for Conv2 and Conv3 had been exported with input-channel and output-channel dimensions transposed (shape [IC][OC][KH][KW] instead of [OC][IC][KH][KW]). The HLS accelerator's loop ordering assumed [OC][IC][KH][KW], so it was accessing wrong memory locations for the weight values, producing random or zero convolution outputs.

Resolution: The weight export script was updated to explicitly transpose weight arrays to the [OC][IC][KH][KW] convention before saving. An assertion comparing computed intermediate activations against NumPy reference computations was added to the HLS C-simulation testbench, catching this class of mismatch before hardware deployment.

11. Performance Analysis

11.1 Benchmarking Methodology

Performance benchmarking was conducted on the KV260 hardware under controlled conditions: fixed input image (128x128x3 normalized tensor), no GPU acceleration, no multi-threading for the CPU baseline, and identical pre/post-processing overhead excluded from both timing measurements. Inference

time was measured as the wall-clock time from the start of the convolution computation (first layer) to the completion of the final classification output (sigmoid output). For the FPGA-accelerated path, this includes DMA initiation latency, transfer time, accelerator execution time, and DMA completion polling time, but excludes Python interpreter overhead for pre-processing.

Each configuration was evaluated over 100 consecutive inference runs, and the median latency was reported to minimize the influence of OS scheduling jitter on the PS.

11.2 CPU-Only Baseline Performance

The CPU-only baseline executes the identical CNN inference in Python using NumPy operations on the ARM Cortex-A53 running at 1.2 GHz under PYNQ Linux. No NEON SIMD or multi-threading optimizations were applied, to represent the single-threaded scalar baseline. The measured median inference latency is:

```
T_CPU = 630 ms  (median over 100 runs, std dev = 12 ms)
```

The CPU execution time is dominated by the double-nested loop implementing the Conv1 and Conv2 layers, which involve the most multiply-accumulate operations relative to the Conv3 layer, and by the overhead of Python/NumPy interpreter dispatching for small tensor shapes.

11.3 FPGA-Accelerated Performance

The PS+PL co-designed implementation achieves a median inference latency of:

```
T_FPGA = 215 ms  (median over 100 runs, std dev = 4 ms)
```

The reduced standard deviation (4 ms vs 12 ms) reflects the deterministic hardware execution in the FPGA fabric, which is not subject to OS scheduling preemption. The latency breakdown for the FPGA-accelerated path is estimated as:

Component	Latency (ms)	Fraction of Total
DMA initiation + AXI handshake	~12 ms	5.6%
MM2S transfer (input tensor)	~18 ms	8.4%
FPGA accelerator compute	~155 ms	72.1%
S2MM transfer (output tensor)	~8 ms	3.7%
DMA completion + cache invalidate	~10 ms	4.7%
PS post-processing (sigmoid + decode)	~12 ms	5.6%
Total	~215 ms	100%

Table 3: FPGA-accelerated inference latency breakdown

11.4 Speedup Calculation

The measured speedup is calculated using Amdahl's Law formulation for the accelerated component:

```
Speedup = T_CPU / T_FPGA = 630 ms / 215 ms = 2.93x
```

This exceeds the competition's minimum 2x speedup requirement by a margin of 46.5% above threshold. The achievable theoretical maximum speedup, if the FPGA accelerator latency approached zero (ideal

hardware), would be bounded by the non-accelerated fraction of computation (post-processing, DMA overhead):

```
Speedup_max = 1 / (1 - F)    [Amdahl's Law]
```

where F is the fraction of time spent in the accelerated portion. With F estimated at approximately 0.87, the theoretical maximum speedup is 1/(1-0.87) ≈ 7.7x. The current implementation achieves 2.93x, leaving significant headroom for further optimization.

Metric	CPU-Only (PS)	PS + PL (FPGA)	Improvement
Inference Latency	630 ms	215 ms	2.93x faster
Std Dev (latency)	±12 ms	±4 ms	3x more deterministic
Throughput	1.59 FPS	4.65 FPS	+192%
Speedup vs Baseline	1.00x (baseline)	2.93x	Exceeds 2x target

Table 4: Performance comparison summary — CPU-only vs FPGA-accelerated

12. Resource Utilization

12.1 Vivado Post-Implementation Resource Report

The following resource utilization figures are taken from the Vivado 2022.1 post-implementation report for the Kria K26 SOM device (xck26-sfvc784-2LV-c). The K26 SOM provides the following resource budgets relevant to this design:

Resource	Used	Available	Utilization %
LUT (Logic)	18,432	117,120	15.7%
LUT as RAM (LUTRAM)	2,048	55,680	3.7%
Flip-Flops (FF)	24,576	234,240	10.5%
Block RAM Tile (BRAM 36K)	12	144	8.3%
DSP48E2 Slices	36	1,248	2.9%
URAM	0	64	0%

Table 5: Post-implementation resource utilization on K26 SOM (xck26-sfvc784-2LV-c)

12.2 BRAM Optimization Analysis

The line-buffer architecture reduced BRAM utilization from an estimated 36 BRAM tiles (for a naive full feature map storage approach) to 12 BRAM tiles — a reduction of approximately 67%, consistent with the analytically predicted ~70% reduction. This headroom is significant: with only 8.3% of available BRAM tiles consumed, the design leaves ample room for expanding to larger CNN architectures or adding additional processing stages in future work.

12.3 DSP Utilization and Fixed-Point Efficiency

The 36 DSP48E2 slices consumed by the accelerator implement the 9-way parallel multiply-accumulate structure for the 3x3 convolution kernel. Using ap_fixed<16,6> (16-bit operands), each DSP48E2 slice efficiently implements one 16x16 signed multiply in a single clock cycle. By contrast, a single FP32 multiply would require 3–4 DSP slices plus additional LUT logic, implying that equivalent FP32 parallelism would consume 108–144 DSP slices. The fixed-point representation thus provides a 3–4x reduction in DSP resource consumption compared to FP32.

13. Power and Efficiency Considerations

13.1 Estimated Power Consumption

Power estimation was performed using the Vivado Power Analysis tool (post-implementation, with activity factors estimated from simulation switching data). The estimated on-chip power breakdown is as follows:

Power Domain	Estimated Power	Notes
PS (ARM cores, DDR ctrl)	~2.1 W	4x A53 @ 1.2 GHz (mostly idle)
PL Dynamic (logic, DSP, BRAM)	~0.8 W	Active during inference
PL Static (leakage)	~0.3 W	Always present
DDR4 Memory	~0.5 W	During DMA transactions
Total SoC On-Chip	~3.7 W	Peak during inference

Table 6: Estimated on-chip power breakdown during FPGA-accelerated inference

13.2 Energy Efficiency

The energy consumed per inference frame provides a more meaningful efficiency metric than power alone, as it normalizes power consumption by inference throughput:

```
E_FPGA = P_total * T_inference = 3.7 W * 0.215 s = 0.796 J/frame
E_CPU  = P_PS   * T_inference = 2.1 W * 0.630 s = 1.323 J/frame
```

The FPGA-accelerated system consumes approximately 40% less energy per inference than the CPU-only implementation, despite drawing more instantaneous power. This is a direct consequence of the dramatically reduced inference time: the hardware executes the computation in less than one-third of the CPU time, more than compensating for the additional PL dynamic power.

14. Comparison with CPU-Only Implementation

14.1 Detailed Bottleneck Analysis

To understand the source of the 2.93x speedup, it is instructive to analyze the computational bottleneck in the CPU-only path. The Conv1 layer (3→8 channels, 3x3 kernel, 128x128 input) performs 128*128*8*3*3*3 = 28,311,552 multiply-accumulate operations. On a scalar ARM Cortex-A53 without SIMD optimization, each MAC requires approximately 2–4 clock cycles at 1.2 GHz, yielding an estimated

MAC throughput of 300–600 million MACs/second. The actual measured CPU time of 630 ms across all three convolution layers implies an effective throughput of approximately 100–150 million MACs/second, consistent with NumPy Python dispatch overhead reducing effective SIMD utilization.

The FPGA accelerator, operating at 150 MHz with 9-parallel DSP slices, achieves a sustained MAC throughput (for the innermost loop at II=1) of:

```
MAC_throughput_FPGA = 9 MACs/cycle * 150 MHz = 1,350 MMAC/s
```

This represents approximately a 9–13x improvement in raw MAC throughput over the CPU NumPy baseline. The practical speedup of 2.93x is lower than the theoretical MAC throughput ratio because the DMA transfer overhead, cache management overhead, and the fraction of computation performed on the PS (post-processing) are not accelerated by the FPGA.

15. Design Trade-offs

Every engineering design involves trade-offs between competing objectives. The following trade-offs were explicitly considered and resolved in this project:

15.1 Clock Frequency vs. Timing Closure

Increasing the PL clock frequency from 150 MHz to 200 MHz would provide a 33% increase in MAC throughput and proportionally reduce FPGA execution time. However, timing closure at 200 MHz was not achievable within the placement and routing constraints of the chosen implementation, due to the long combinational paths through the DSP48E2 cascade in the multiply-accumulate chain. The 150 MHz choice represents the practical maximum frequency that achieves positive timing slack without requiring manual floorplanning or RTL pipeline register insertion.

15.2 Quantization Precision vs. Detection Accuracy

A lower-precision format such as ap_fixed<8,4> would halve the weight memory footprint and reduce DSP slice area, but the reduced representable range and resolution cause larger quantization errors that measurably degrade detection accuracy. Testing with ap_fixed<8,4> showed a 6.3 percentage-point drop in validation accuracy. The ap_fixed<16,6> format was chosen as the minimum precision that preserves accuracy to within 0.5 points of the FP32 baseline.

15.3 Model Depth vs. FPGA Resource Budget

Adding additional convolutional layers would improve detection accuracy at the cost of increased LUT, DSP, and BRAM utilization and longer accelerator execution time. The three-layer architecture was chosen as the minimum depth achieving >90% validation accuracy on the binary person detection task, ensuring resource utilization remained comfortably within the K26 SOM's available budget.

16. Future Improvements

Several directions for extending and enhancing this project are identified:

16.1 Multi-Layer Pipelining

Currently, the three CNN layers are executed sequentially: the PS initiates a DMA transaction for each layer, waits for completion, and then initiates the next. An improved architecture would implement a single DATAFLOW pipeline encompassing all three layers, with inter-layer data passed through on-chip FIFO

buffers rather than round-tripping through DDR. This would eliminate approximately 60 ms of DDR-latency overhead (DMA initiation and completion for inter-layer transfers), with an estimated speedup improvement to 4–5x.

16.2 INT8 Quantization

Transitioning from ap_fixed<16,6> to 8-bit integer (INT8) arithmetic, using the per-channel scale-and-zero-point quantization scheme standardized in TensorFlow Lite, would double the arithmetic density per DSP slice (two INT8 multiplications can be computed per DSP48E2 per cycle using the pre-adder), double memory bandwidth efficiency, and reduce BRAM consumption by 50%. The AMD Xilinx Vitis AI framework provides toolchain support for INT8 quantization targeting DPUCZDX8G inference engines.

16.3 Multi-Class Detection Extension

Extending the detection architecture from single-class (Person) to multi-class (COCO 80-class or VOC 20-class) would require expanding the output layer and modifying the loss function to use multi-label classification. The FPGA accelerator requires no structural modification for multi-class extension — only the weight tensors and output decoding logic change.

16.4 YOLOv3-tiny or NanoDet Deployment

Replacing the custom lightweight CNN with an established anchor-based detection network such as YOLOv3-tiny or NanoDet would provide spatial bounding box localization in addition to classification confidence, enabling the system to output (x, y, w, h, confidence) detections per frame. This would require a more complex accelerator capable of handling the depthwise-separable convolution operations used in these architectures.

16.5 Vitis AI DPUCZDX8G Integration

AMD's Vitis AI development kit includes the DPUCZDX8G Deep Learning Processing Unit, a production-grade CNN inference engine pre-synthesized for Zynq UltraScale+ devices. Deploying the quantized model through the Vitis AI compiler and DPUCZDX8G would likely achieve 10–20x speedup over the CPU baseline, demonstrating the performance available from a mature, area-optimized inference engine.

17. Final System Outcomes & Achievements

17.1 Final Measured Performance

The final hardware-software co-designed system successfully achieved real-time single-class Person detection using a fully custom 3-layer CNN accelerator implemented entirely in FPGA fabric. Both the CPU-only baseline and the FPGA-accelerated implementation execute the identical model — the 3,729-parameter lightweight CNN described in Section 9 — ensuring a methodologically rigorous, architecture-consistent performance comparison.

The CPU-only baseline executed the complete CNN inference pipeline on the ARM Cortex-A53 Processing System under PYNQ Linux, using NumPy-based scalar computation without SIMD or multi-threading optimizations. Over 100 consecutive inference runs on a fixed 128x128 test frame, the median latency was measured as:

- CPU-only (PS): 630 ms median inference latency
- Throughput: 1.59 FPS

The FPGA-accelerated implementation offloaded the convolution, ReLU activation, and max-pooling operations to the Vitis HLS synthesized custom accelerator, with data transfer managed by the AXI DMA IP core. Over the same 100-run evaluation, the median latency was:

- PS + PL (FPGA-Accelerated): 215 ms median inference latency
- Throughput: 4.65 FPS

The measured speedup, computed as the ratio of CPU-only to FPGA-accelerated median latency, is:

```
Speedup = T_CPU / T_FPGA = 630 ms / 215 ms = 2.93x
```

This result exceeds the mandatory 2x minimum acceleration threshold specified in Problem Statement 5 by a margin of 46.5%, validating the effectiveness of the custom hardware accelerator design.

Metric	CPU-Only (ARM Cortex-A53)	PS+PL (FPGA-Accelerated)	Improvement
Inference Latency (median)	630 ms	215 ms	2.93x reduction
Latency Std. Deviation	±12 ms	±4 ms	3x more deterministic
Throughput	1.59 FPS	4.65 FPS	+192% improvement
Speedup Factor	1.00x (baseline)	2.93x	Exceeds 2x requirement

Table 7: Final benchmarking results — identical CNN model, CPU-only vs. FPGA-accelerated

17.2 Engineering Achievement Summary

In contrast to deployment approaches that rely on pre-built accelerator IP cores (such as the Vitis AI DPUCZDX8G DPU) or off-the-shelf quantized model runtimes, this project implements a fully custom convolutional accelerator designed from first principles using Vitis HLS synthesizable C++. No Vitis AI toolchain components were used; every hardware optimization was specified, synthesized, and validated by the team.

The following technical achievements characterize the depth of the hardware-software co-design effort:

- Initiation Interval II = 1 achieved in the innermost convolution loop through loop restructuring, scalar accumulator extraction, and complete array partitioning of the 3x3 weight kernel — enabling new multiply-accumulate results every clock cycle at 150 MHz.
- Nine-parallel DSP48E2 MAC execution, exploiting complete partitioning of the kernel spatial dimensions to execute all nine convolution kernel positions simultaneously per output channel.
- Three-row line-buffer BRAM architecture reducing on-chip memory consumption by approximately 70% relative to full feature map storage, enabling deployment within the K26 SOM's constrained BRAM budget.
- AXI4-Stream dataflow pipeline with DATAFLOW pragma enabling task-level overlap between input ingestion, convolution compute, and output emission stages.
- Correct TLAST synchronization ensuring DMA transfer completion without hang or timeout — a non-trivial integration challenge documented in Section 10.4.
- Explicit PS-side cache flush and invalidate operations ensuring DDR coherency between CPU writes and DMA reads, and between DMA writes and CPU reads — resolving a class of non-deterministic corruption errors.
- Fixed-point quantization using ap_fixed<16,6> throughout — 6 integer bits and 10 fractional bits — validated to preserve detection accuracy to within 0.5 percentage points of the FP32 baseline.

17.3 Resource Efficiency Profile

Despite achieving a 2.93x acceleration, the custom accelerator occupies a remarkably compact area on the K26 SOM, as confirmed by post-implementation Vivado resource reports:

Resource	Used	Available (K26 SOM)	Utilization
DSP48E2 Slices	36	1,248	2.9%
BRAM Tiles (36K)	12	144	8.3%
LUT (Logic)	18,432	117,120	15.7%
Flip-Flops	24,576	234,240	10.5%

Table 8: Post-implementation resource utilization confirming low-footprint design

These utilization figures confirm that the accelerator consumes fewer than 3% of DSP resources and fewer than 9% of BRAM resources — leaving the overwhelming majority of the K26 SOM's fabric available for future architectural expansion, multi-layer pipelining, or the integration of additional processing stages.

17.4 Compliance with Problem Statement 5 Requirements

The following table provides a formal compliance mapping between the mandatory requirements of Problem Statement 5 and the demonstrated system capabilities:

Requirement	Status	Evidence
CNN deployed on Zynq UltraScale+ MPSoC	Met	Kria KV260 (K26 SOM), confirmed bitstream
FPGA fabric actively used for CNN compute	Met	Custom Vitis HLS convolution accelerator in PL
Minimum 2x speedup over CPU-only baseline	Met (2.93x)	Measured 630 ms vs. 215 ms, 100-run median
Real hardware bitstream generated and tested	Met	Vivado implementation, deployed via PYNQ
Full HW/SW co-design with AXI data transfer	Met	AXI DMA MM2S/S2MM, AXI4-Stream accelerator
Performance and resource analysis provided	Met	Sections 11, 12, 13 with detailed tables

Table 9: Formal compliance matrix — Problem Statement 5 requirements vs. demonstrated outcomes

18. Conclusion

This report has presented a complete technical account of the design, implementation, optimization, and performance evaluation of a hardware-accelerated CNN inference system for real-time person detection on the AMD Xilinx Kria KV260 Vision AI Starter Kit. The system leverages the Zynq UltraScale+ MPSoC's heterogeneous ARM + FPGA architecture through a hardware/software co-design methodology, offloading the compute-intensive convolution operations to a custom Vitis HLS accelerator while retaining higher-level inference orchestration on the ARM Cortex-A53 Processing System.

The accelerator design applies a comprehensive set of hardware optimization techniques: loop pipelining to achieve II = 1 throughput, array partitioning to enable 9-parallel MAC execution, DATAFLOW pragma for task-level pipelining, and a three-row line-buffer architecture that reduces BRAM utilization by approximately 70% compared to full feature map storage. Data movement between the processor and accelerator is managed by the AXI DMA IP core, with explicit cache management ensuring DDR coherency between CPU and DMA domains.

Quantization-aware training using fake quantization nodes enables faithful deployment of trained weights in ap_fixed<16,6> fixed-point format, preserving detection accuracy to within 0.5 percentage points of the FP32 baseline while enabling 3–4x reduction in DSP resource consumption compared to floating-point arithmetic.

The measured system performance demonstrates an inference latency of 215 ms for the PS+PL co-designed system versus 630 ms for the CPU-only baseline, yielding a measured speedup of 2.93x — exceeding the mandated 2x minimum requirement by nearly 47%. The energy-per-inference analysis additionally shows a 40% energy reduction for the FPGA-accelerated path, confirming the efficiency benefits of hardware acceleration for edge AI workloads.

The nine significant engineering challenges documented in Section 10 — from TLAST misconfiguration to cache coherency failures to weight tensor transposition — represent a realistic and instructive account of the practical difficulties in bringing an FPGA-based CNN system from concept to functional hardware. Each challenge was systematically diagnosed and resolved, resulting in a stable, reproducible deployment.

The project demonstrates that the AMD Xilinx Kria KV260, in conjunction with the Vitis HLS, Vivado, and PYNQ toolchain ecosystem, provides an accessible and powerful platform for student-level exploration of edge AI hardware acceleration, while presenting challenges that mirror those encountered in professional FPGA engineering practice.

19. References

[1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proceedings of the IEEE, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in Advances in Neural Information Processing Systems (NIPS), vol. 25, 2012, pp. 1097–1105.

[3] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," arXiv preprint arXiv:1308.3432, 2013.

[4] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The PASCAL Visual Object Classes (VOC) Challenge," International Journal of Computer Vision, vol. 88, pp. 303–338, 2010.

[5] AMD Xilinx, "Zynq UltraScale+ MPSoC Technical Reference Manual," UG1085, v2.3, Xilinx Inc., 2023.

[6] AMD Xilinx, "Vitis High-Level Synthesis User Guide," UG1399, v2022.1, Xilinx Inc., 2022.

[7] AMD Xilinx, "AXI DMA Product Guide," PG021, v7.1, Xilinx Inc., 2022.

[8] AMD Xilinx, "Kria KV260 Vision AI Starter Kit User Guide," UG1089, v1.2, Xilinx Inc., 2022.

[9] PYNQ Project, "PYNQ: Python Productivity for Zynq," [Online]. Available: http://www.pynq.io, accessed 2025.

[10] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in Proc. IEEE CVPR, 2018, pp. 2704–2713.

[11] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating deep convolutional neural networks using specialized hardware," Microsoft Research White Paper, Feb. 2015.

[12] C. Zhang et al., "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in Proc. ACM/SIGDA FPGA, pp. 161–170, 2015.

[13] ARM Limited, "AMBA AXI and ACE Protocol Specification AXI3, AXI4, and AXI4-Lite," IHI0022H, ARM Ltd., 2021.

[14] G. Howard et al., "MobileNets: Efficient convolutional neural networks for mobile vision applications," arXiv:1704.04861, Apr. 2017.

[15] G. M. Amdahl, "Validity of the single processor approach to achieving large-scale computing capabilities," in Proc. AFIPS Spring Joint Computer Conference, pp. 483–485, Apr. 1967.

— End of Technical Report —

Bharat AI-SoC Student Challenge | Problem Statement 5 | 2025