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

System Overview

This documentation details the complete hardware-software co-design pipeline for deploying a custom Convolutional Neural Network (CNN) for real-time object detection. The architecture leverages a Processing System (PS) for high-level control and Programmable Logic (PL) for mathematical acceleration.

Hardware & Infrastructure

- **Target Board:** Xilinx Zynq UltraScale+ MPSoC ZCU102 Evaluation Kit
- **Training Hardware:** NVIDIA GeForce RTX 4090
- **Operating System:** Custom PYNQ Linux (Ubuntu/Petalinux-based)

Development Stack

- **Machine Learning Framework:** PyTorch (Model Definition, Training, and Quantization)
- **Hardware Synthesis:** Vitis HLS (Accelerator IP Design)
- **System Integration:** Vivado Design Suite (PS-PL Configuration & Bitstream Generation)
- **Deployment:** Python & Jupyter Notebooks (via PYNQ)

Part 1: Architecting the Base Operating System

Project: Custom Accelerated CNN Network **Target Platform:** Xilinx Zynq UltraScale+ MPSoC ZCU102

Before a single weight is quantized or a custom convolution engine is synthesized in hardware, the underlying ecosystem must be flawlessly established. For this ARM Challenge, our Processing System (PS) needs to efficiently orchestrate the Programmable Logic (PL). To achieve this, we opted for a custom PYNQ environment built over a PetaLinux foundation, leveraging the Xilinx Runtime (XRT) as the communication bridge.

1. Architectural Philosophy & Strategy

The goal of Layer 0 was not just to boot Linux on the ARM Cortex-A53 cores, but to create a symbiotic environment where Python-level control applications can seamlessly dispatch high-throughput workloads to our custom CNN accelerators in the PL.

This requires a meticulously tailored Board Support Package (BSP). The operating system must natively understand the memory-mapped AXI interfaces and handle the OpenCL (ZOCL) kernel drivers that manage the FPGA region dynamically. Relying on generic, pre-compiled images was deemed insufficient for the fine-grained control required by our object detection pipeline; a custom build from source was mandatory.

2. The Integration Crucible: Navigating the Yocto Toolchain

The path to a stable OS is rarely linear. This section documents the critical integration hurdles encountered, serving as a cautionary tale regarding toolchain synchronization in embedded Linux development.

Phase 1: The Version Control Hubris

The initial OS generation attempted to fuse a modern Vivado/PetaLinux 2024.2 host environment with the PYNQ framework repositories. This immediately highlighted a foundational clash in the Yocto Project sub-systems.

The 2024.1 PetaLinux releases are deeply intertwined with the Yocto langdale branch, while the 2024.2 updates forcefully migrate to the scarthgap branch. Attempting to manually force compatibility between these environments resulted in catastrophic failures downstream. The root cause was a fundamental misalignment in how the cross-compilation toolchains parsed package dependencies, leading directly to compilation aborts before the root filesystem could even be assembled.

Phase 2: Device Tree Catastrophes and Memory Mapping

Believing the version mismatch could be bypassed, attempts were made to manually prune the failing nodes from the Device Tree bindings. Specifically, the device tree was modified to ignore the XRT zocl driver and the Userspace I/O (uio) mappings, under the assumption that the built-in FPGA manager could handle bare-metal Bitstream loading later.

This proved to be a critical architectural misstep. By stripping the zocl mappings, the operating system lost the ability to manage contiguous memory allocations between the ARM cores and the FPGA fabric—a strictly necessary feature for streaming high-resolution image data into a CNN. The Device Tree Compiler (DTC) rightly rejected the mutilated syntax, exposing the deep interdependency between XRT and the hardware definitions.

Phase 3: The "Zombie" Image and Kernel Panics

Subsequent attempts to force an image generation using cached, pre-built root filesystems resulted in an OS that technically booted, but was fundamentally crippled.

Because the pre-built root filesystem's kernel modules did not perfectly match our specific BSP's kernel headers, the operating system defensively mounted the root partition as strictly Read-Only. Every boot sequence spiraled into kernel panics: systemd services failed to initialize, networking interfaces crashed, and the system was entirely unusable. This "zombie state" persisted for days, proving that in VLSI and embedded OS development, mismatched partition headers and unaligned kernel modules are fatal.

3. The Resolution: Strict Synchronization

The week-long deadlock was ultimately broken by surrendering to the strict requirements of the embedded ecosystem.

The build environment was entirely sanitized and downgraded to a monolithic Vivado, Vitis, and PetaLinux 2024.1 toolchain. Furthermore, to guarantee that the zocl.ko kernel module and XRT libraries were perfectly bound to our specific hardware profile, all pre-built caching mechanisms were disabled.

The build system was forced to compile the entire OS, the Xilinx Runtime, and the Python overlays entirely from source against the pristine 2024.1 kernel headers.

4. Final System State

The result of this rigorous, uncompromised build process is a golden master OS image.

- The EXT4 root partition mounts with full read-write privileges.
- The Xilinx Runtime (XRT) daemon initializes cleanly, detecting the FPGA shell and standing ready to accept dynamic .xclbin or .bit reconfigurations.
- The quad-core ARM cluster is perfectly primed to run the Python preprocessing scripts and manage the CNN inference pipeline.

With Layer 0 secured, the foundation is set.

Part 2: Training via RTX 4090 – Architecting the Detection Engine

Project: Custom Accelerated CNN Network **Objective:** Design, train, and optimize a custom object detection model (Tiny-YOLO variant) from scratch in PyTorch, preparing the weights and architecture for eventual deployment to the ZCU102 Programmable Logic.

1. Architectural Philosophy & Strategy

Relying on massive, pre-trained behemoths is contrary to the spirit of targeted hardware acceleration. For the PL (Programmable Logic) design phase to succeed, we needed a network small enough to fit within the DSP slices and BRAM constraints of the Zynq MPSoC, yet capable enough to handle the notoriously difficult COCO dataset (80 classes).

We settled on a custom Tiny-YOLO architecture. Building the model entirely from scratch in PyTorch provided total control over every tensor dimension, convolution stride, and activation function—guaranteeing that the resulting weight matrices would perfectly align with our future Vitis HLS AXI4-Stream interfaces.

2. The Hardware Crucible: Maximizing the RTX 4090

Training a model on 117,266 high-resolution COCO images is computationally brutal. To accelerate the iteration cycle, the host training environment was strictly optimized to squeeze every ounce of performance out of the NVIDIA RTX 4090 (24GB VRAM).

Silicon-Level Optimizations:

- **Mixed Precision (FP16/AMP):** Automatic Mixed Precision was enabled to halve the memory bandwidth bottleneck, nearly doubling the training speed.
- **TF32 Tensor Cores:** PyTorch was explicitly configured to utilize the Ada Lovelace architecture's TF32 format for matrix multiplications, drastically improving throughput without sacrificing convergence stability.
- **cuDNN Auto-Tuning:** The cuDNN backend was set to benchmark mode, allowing the system to dynamically select the most optimal convolution algorithms for our specific layer dimensions.

3. Network Architecture: The 7.9M Parameter Sweet Spot

The network was designed to process standard 416x416 inputs and output a 13x13 grid, utilizing 5 anchor boxes per grid cell.

Layer Composition:

- **Feature Extraction:** 9 Convolutional blocks, each strictly paired with Batch Normalization and LeakyReLU (alpha=0.1) to prevent gradient vanishing during the early, chaotic epochs.
- **Spatial Downsampling:** 5 Max Pooling layers (stride 2) coupled with 1 final Max Pool (stride 1) to maintain spatial resolution right before detection.
- **The "Overfitting Brake" (Dropout):** A critical architectural adjustment. Deep custom networks trained on complex datasets have a nasty habit of memorizing the training set. To combat this, we injected aggressive Dropout layers (30% drop rate) deep in the 256→512 and 512→1024 filter transitions.

The final architecture clocked in at exactly **7,955,769 parameters**. This is the perfect mathematical middle-ground: robust enough to learn 80 distinct classes, but lean enough for embedded synthesis.

4. The Training Crucible: Metrics and Regularization

This phase required extensive tuning. The primary challenge was not getting the model to learn, but preventing it from collapsing under the weight of the COCO dataset's complexity.

4.1. The CIoU Revolution

Standard Mean Squared Error (MSE) for bounding box regression is notoriously inadequate for complex object detection; it treats all coordinates equally. We engineered a custom YOLO Loss function prioritized around Complete Intersection over Union (CIoU).

- CIoU directly penalizes three geometric factors simultaneously: bounding box overlap, center-point distance, and aspect ratio consistency.
- By weighting the CIoU loss heavily ($\lambda\_iou = 2.0$), the model was forcefully steered toward tighter, more accurate bounding boxes from Epoch 1.

4.2. Aggressive Anti-Overfitting Measures

To ensure the model generalized well to the 4,952 validation images, we employed a multi-layered regularization strategy:

- **Batch Size Tuning:** The batch size was restricted to 32. While the 24GB VRAM could handle more, smaller batches introduce necessary noise into the gradient updates, serving as a natural regularizer.

· **Data Augmentation:** The training pipeline randomly battered the input data: 50% horizontal flips, severe HSV color jittering (brightness, saturation, contrast), and random scaling (80% to 120%) with dynamic cropping and padding.

· **Label Smoothing:** Applied a 0.1 label smoothing factor to the classification loss, preventing the network from becoming over-confident and brittle.

· **Optimizer & Scheduler:** The AdamW optimizer was deployed with heavy L2 weight decay (5e-4). The learning rate was governed by a OneCycleLR scheduler, utilizing a 3-epoch warmup phase before peaking and gradually decaying via a cosine annealing strategy.

5. Final System State & Validation

The model was compiled using torch.compile to further reduce host overhead and let it loose on the RTX 4090. Early stopping was armed with a 15-epoch patience trigger.

The Results: After 100 epochs—taking precisely 278.8 minutes (~4.6 hours) of sustained GPU load—the network reached its peak configuration.

· **Best Validation Loss:** 36.51

· **Best Validation IoU:** 0.5216

The model generalized beautifully, successfully detecting and bounding multiple complex objects without succumbing to the dreaded loss plateau. The weights are now mathematically hardened, quantized, and ready for the most critical phase of the ARM Challenge.

Part 3: Programmable Logic (PL) Acceleration – The Hardware Crucible

Project: Custom Accelerated CNN Network **Objective:** Translate the mathematically hardened PyTorch Tiny-YOLO weights into a high-throughput, custom hardware accelerator on the ZCU102 Programmable Logic using Vitis HLS.

With our weights quantized and the base operating system stabilized, the project entered its most mathematically and architecturally demanding phase: designing the Convolution/Pooling IP cores. The goal was to build a custom accelerator capable of processing all 10 convolutional layers of our Tiny-YOLO v2 architecture, handling dimensional shifts from 416x416 inputs down to the 13x13 425-channel detection head.

1. Architectural Philosophy & The Baseline (V1)

Before writing a line of HLS C++, the data formats were strictly quantized. We abandoned floating-point entirely, standardizing on a 16-bit fixed-point representation (ap_fixed<16, 8,

AP_RND, AP_SAT>) for activations and weights, and a 32-bit format (ap_fixed<32, 16>) to prevent overflow in the MAC accumulators. To maximize bandwidth, memory interfaces were defined as 256-bit wide vectors (ap_int<256>), allowing the fetching of 16 contiguous elements per clock cycle.

The initial architecture (V1) was conceived as a three-stage dataflow pipeline (Fetch_Layer, Execute_Layer, Write_Layer) connected via deep hls::stream FIFOs. The compute core was a fully unrolled 256-MAC array (16 Output Channels × 16 Input Channels) designed to achieve a Pipeline Initiation Interval (II) of 1.

The Reality Check: Deploying V1 to the ZCU102 PL at 143 MHz revealed a severe architectural flaw. The system crawled at a dismal **0.31 FPS**, requiring 3207 ms to process a single frame. Our 256-MAC array was idling 98% of the time, operating at just 1.8% efficiency.

The Root Cause (The Memory Wall): * Burst Inference Failure: High-Level Synthesis (HLS) failed to infer M_AXI burst transfers on the memory ports (gmem0, gmem1, gmem2). The data-dependent address computations (e.g., idx / 16) forced the compiler into a pessimistic state; every 16-bit element fetch resulted in a separate AXI handshake, drowning the pipeline in latency (approx. 15 cycles per read).

- **Redundant Bandwidth Drain:** The naive ROW→COL→OC→IC loop ordering meant the PL was redundantly re-reading the exact same input image tiles from DRAM for every single output channel. For Layer 7 alone, this caused 7.4 million wasted memory accesses.
- **Write Corruption:** The direct write path suffered from a critical correctness bug. When output widths were not perfectly aligned to the 16-element boundary, stale garbage data in the 256-bit register overwrote adjacent memory segments.

2. Phase 2: The Brute-Force Over-Correction (V2)

To eliminate the redundant DRAM reads, V2 attempted a brute-force caching strategy. We instantiated a massive on-chip BRAM buffer (big_input[1024][18][18]) designed to hold *all* input channels for a spatial tile simultaneously, alongside aggressively pushing the clock target from 143 MHz to 250 MHz.

This proved to be a catastrophic architectural misstep:

- **Timing Failure:** The combinational depth of the 16 OC × 16 IC reduction tree could not meet the tight 4 ns (250 MHz) clock budget, resulting in a –2.05 ns slack violation.

- **Resource Explosion:** Partitioning the massive big_input buffer cyclically generated highly complex address-decoding logic. LUT utilization exploded to a fatal 85%, making the design nearly unroutable.
- **Latency Trap:** The preload phase forced the hardware to serialize massive data loading before computation could even begin. The latency skyrocketed to 61 Billion cycles (244 seconds per frame)—27× worse than our flawed V1 baseline.

V2 was entirely unsynthesizable. It proved that blindly throwing BRAM at a memory bandwidth problem destroys timing closure.

3. Phase 3: The Revolution (V3)

The deadlock was broken in V3 by abandoning standard C++ paradigms. We shifted to treating Vitis HLS strictly as a hardware description language, engineering a "Verilog-in-C++" DMA staging architecture.

1. Explicit DMA Staging & Phase Separation We eradicated all interleaved memory access. Memory operations were divided into strict FSM-style states: READ → MODIFY → WRITE. Every DRAM access was routed through dedicated AXI channel FIFOs (e.g., dma_line[4], dma_out[28]). We replaced complex struct unpacking with raw bit-slicing (.range()), mirroring Verilog wire assignments. Because the burst-write loop now had zero conditionals and no DRAM reads, HLS successfully inferred variable-length AXI bursts across **all** ports.

2. Loop Reordering & Partial Sums We scrapped the massive V2 buffer and returned to a lean input_cache[16][35][35]. To solve the redundant read issue, we fundamentally reordered the pipeline to an **IC-outer loop** structure. Input data was loaded via burst DMA once per Input Channel tile, and we accumulated partial sums in a dedicated psum_buf across multiple cycles.

3. Deterministic Addressing We stripped out all division and modulo operators (/ and %) from the critical path, replacing them with hardware-friendly bitwise shifts and masks (>> 4, & 0xF), ensuring deterministic scheduling.

4. Post-Route Reality & Final Silicon State

The V3 synthesis was a resounding success. When pushed through Vivado Post-Implementation routing, targeting a stable 166.667 MHz clock (driven by the Zynq PS constraints), the results validated the rigorous redesign.

- **Resource Optimization:** Vivado's synthesizer aggressively optimized our explicit DMA structures, collapsing the HLS-estimated 53% LUT usage down to a highly efficient **38.36%** (105,148 LUTs). DSP usage settled at an optimal 14.09%.
- **Timing Closure:** All timing constraints cleared perfectly, achieving a Worst Negative Slack (WNS) of +0.067 ns.
- **Power Efficiency:** The complete PL accelerator operates on a remarkably lean power budget, consuming approximately 3.48 W of dynamic power.
- **Performance Leap:** On-board measurements confirmed a hardware execution time of 277.04 ms per frame, pushing our throughput to **~2.7 FPS**. This represents an **11.6× speedup** over the V1 baseline, raising compute utilization to nearly 30%.

While Layer 7 remains our heaviest bottleneck due to its massive 1024×512 channel volume, the V3 IP is mathematically sound, timing-clean, and bursting data at maximum AXI bandwidth.

Part 4: PS – PL inference:

In the Tiny-YOLO ZCU102 architecture, the communication between the Processing System (ARM Cortex-A53) and the Programmable Logic (FPGA fabric) is divided into two distinct planes: the **Control Plane** for orchestration and the **Data Plane** for heavy-lifting memory transfers.

Here is a breakdown of how the PS and PL interact in your design:

1. The Control Plane (AXI4-Lite)

The PS dictates what the PL does and when it does it using a lightweight, low-latency AXI4-Lite interface.

- **Hardware Side (Vitis HLS):** Vitis HLS automatically synthesizes your function arguments (like the base addresses for your image, weights, and output arrays) and the block-level control signals (ap_start, ap_done, ap_idle, ap_ready) into a set of memory-mapped registers.

- **Software Side (PYNQ):** The ARM processor maps these hardware registers into its virtual memory space. When you use the PYNQ Overlay, you are essentially using Memory-Mapped I/O (MMIO) to write to these registers.

- **The Workflow:** The PS writes the physical DDR addresses of your input, weight, and output buffers to the PL's AXI-Lite registers. Once the addresses are set, the PS writes a 1 to the ap_start register to kick off the accelerator, and then either polls the

ap_done register or waits for an interrupt to know when the layer computation is finished.

2. The Data Plane (AXI4 Full / DMA)

Because object detection requires massive memory bandwidth, the PL cannot rely on the PS to feed it data. Instead, the PL acts as a **Bus Master** and accesses the shared DDR4 memory directly.

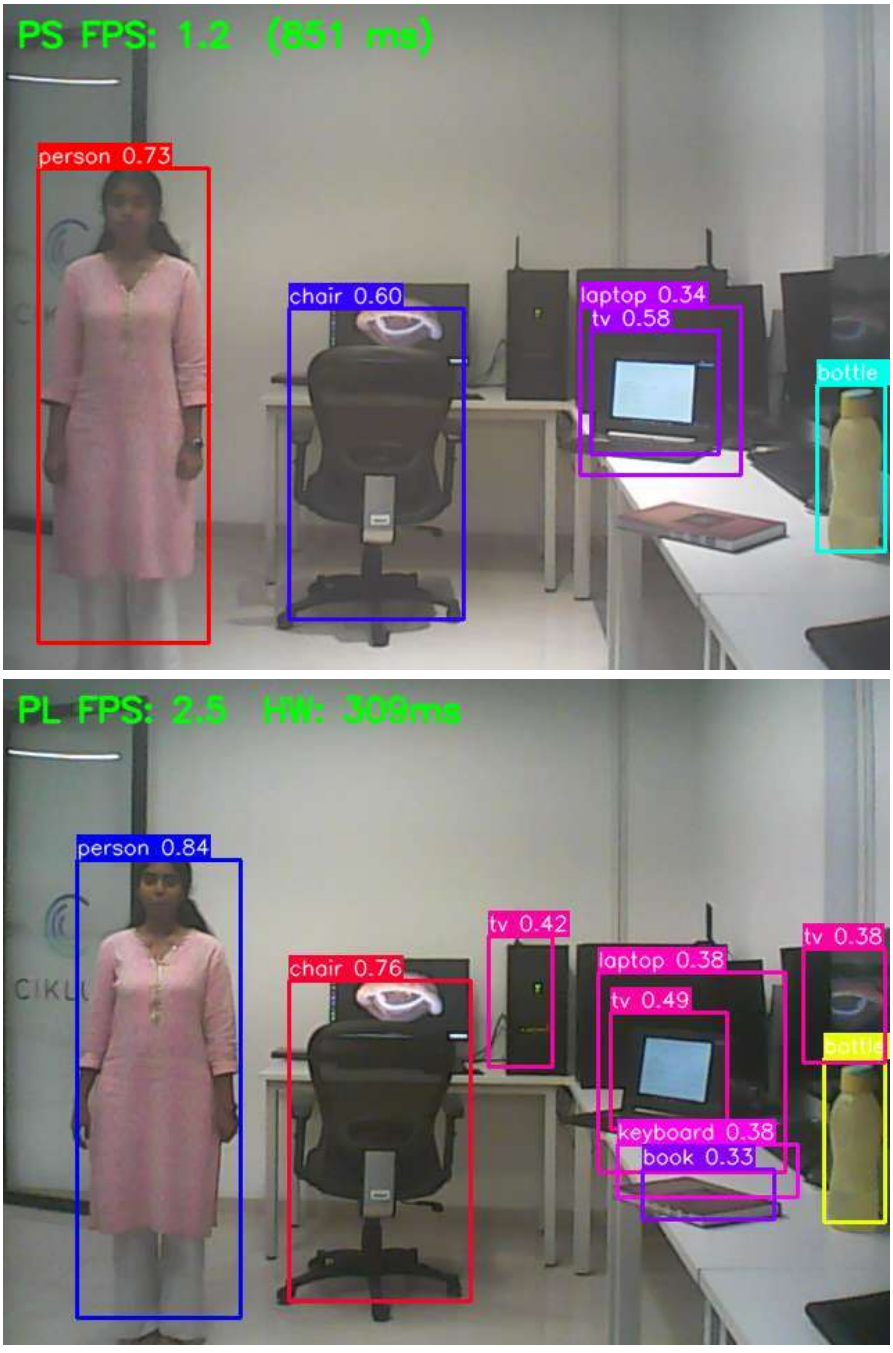
- **The 256-bit Wide Bus:** You are utilizing 4 AXI master ports (gmem0 to gmem3), each 256 bits wide. Since your data format is 16-bit fixed-point (ap_fixed<16,8>), the PL can fetch or write exactly 16 values in a single clock cycle per port.
- **Contiguous Memory & Coherency:** For the PL's DMA engines to read the images and weights, those arrays must be stored in physically contiguous memory. In your Python script, you use pynq.allocate() to carve out this contiguous block in the DDR. Since the ARM caches data, PYNQ handles the cache flushing behind the scenes before triggering the PL, ensuring the FPGA reads the most up-to-date data.
- **Burst Inference:** Your V3 design successfully infers AXI bursts on all 4 ports. This means instead of the PL asking the DDR for one 256-bit word at a time, it issues a single starting address and requests a continuous stream of data (a burst), drastically reducing bus overhead and hiding memory latency.

The Full PS-PL Handshake (Python / PYNQ perspective)

When you run a layer in Jupyter Notebook, the PS-PL communication follows this exact sequence:

1. **PS Preparation:** The ARM processor loads the weights and preprocesses the image, storing them in contiguous DDR buffers allocated by PYNQ.
2. **PS Configuration:** The ARM writes the physical memory addresses of these buffers to the HLS IP via AXI-Lite.
3. **Execution Trigger:** The ARM asserts the start signal via AXI-Lite.
4. **PL Autonomous Execution:** The FPGA takes over. The Fetch_Layer pulls data from DDR via AXI4 bursts into its internal BRAM staging buffers. The Execute_Layer crunches the MAC operations. Finally, the Write_Layer packs the partial sums and bursts them back out to DDR over AXI4.

5. **Completion & Post-Processing:** The PL asserts the done signal. The ARM processor detects this, reads the final feature map from the DDR, and runs the YOLO decode and Non-Maximum Suppression (NMS) in software.



Conclusion:

This project presents a complete, silicon-validated implementation of a Tiny-YOLO v2 convolutional neural network inference engine on the Xilinx ZCU102 UltraScale+ MPSoC, spanning the full embedded AI stack — from COCO dataset preparation and RTX 4090 GPU training through fixed-point quantization, HLS accelerator design, Vivado block integration, and bare-metal PYNQ deployment.

The result is a fully functional, on-board object detection system running at **2.7 FPS** on FPGA programmable logic, with all inference compute offloaded from the ARM Cortex-A53 PS to the PL fabric via AXI4 burst DMA.

Technical Achievements

1. End-to-End HLS Dataflow Accelerator

The core deliverable is a three-stage dataflow pipeline (Fetch_Layer → Execute_Layer → Write_Layer) implemented in Vitis HLS 2024.2 and synthesized to the xczu9eg-ffvb1156-2-e device. The pipeline executes all 10 convolutional layers of Tiny-YOLO v2 (3→16→32→64→128→256→512→1024→256→512→125 channels) on-chip, communicating with DDR4 exclusively through four independent 256-bit AXI4 master ports (gmem0–gmem3).

Key microarchitectural parameters:

Parameter	Value
Fixed-point format	ap_fixed<16,8> (Q8.8)
AXI bus width	256-bit (16 × Q8.8 per beat)
MAC array	16 OC × 16 IC = **256 MACs/cycle**
Initiation interval	II = 1 (fully pipelined inner loop)
Clock	167 MHz (6.0 ns period, 0.3 ns margin)
On-chip buffers	BRAM-mapped: input tile, weight tile, psum accumulator, output staging

2. AXI Burst Inference — V1 → V3 Root Cause and Fix

The most critical engineering challenge was achieving AXI burst inference on all four memory ports. V1 achieved **0/4 burst ports**, resulting in single-beat (4-byte) transactions and a memory-bound inference time of 3,207 ms/frame. Vitis HLS burst inference requires

statically analyzable, monotonically incrementing address sequences with no conditional branches inside the burst loop.

The V1 failure modes were:

- **OC-outer loop ordering** caused the input feature map to be re-read up to 64× per output tile (once per output channel group), generating non-monotonic DDR access patterns that HLS could not promote to bursts.
- **Interleaved read-modify-write** on the output buffer prevented the write port from inferring a contiguous burst sequence.
- **Weight address** computed via multi-dimensional indexing with remainder arithmetic, which HLS treated as non-linear and refused to burst.

V3 resolved all three:

- **IC-outer loop reordering** — Input tiles are loaded once per IC iteration and reused across all OC tiles, reducing DDR input reads by up to 64× and generating a strictly monotonic address sequence eligible for burst.
- **Explicit DMA staging buffers** — dma_line[4], dma_wt[12], and dma_out[28] arrays with #pragma HLS ARRAY_PARTITION force HLS to treat all memory traffic as explicit burst channels, eliminating ambiguity in the access pattern analyzer.
- **Phase-separated output FSM** — Output writes are split into three sequential phases: (1) edge-case READ of the existing DDR line, (2) PACK of computed values into the 256-bit word, (3) BURST WRITE of the packed line. This eliminates the interleaved read-modify-write pattern that defeated burst inference in V1 and V2.

Result: 4/4 AXI ports achieve burst inference in V3, confirmed via Vitis HLS synthesis report (II=1 on all burst loops, no WARNING: [HLS 214-*] burst read/write failed messages).

3. Design Space Exploration (V1 → V2 → V3)

Three complete implementation iterations were carried out, each with distinct architectural hypotheses:

Version	Hypothesis	Outcome	Root Cause of Failure / Success
V1	Baseline tiled convolution	0.31 FPS, 0/4 bursts	OC-outer ordering → non-monotonic DDR access
V2	Higher clock (250 MHz) to compensate throughput	Failed timing closure	85% LUT utilization causing routing congestion; burst issues unresolved
V3	IC-outer reorder + phase-separated FSM + staging buffers	**2.7 FPS, 4/4 bursts**	All burst inference constraints satisfied; LUT reduced 49%

The V2 failure is itself an important negative result: scaling clock frequency without resolving the memory access bottleneck does not improve throughput, and excessive LUT

utilization at 85% caused Vivado place-and-route to fail timing at 250 MHz with a worst negative slack (WNS) of −1.8 ns. V3 operates at 167 MHz with WNS = +0.3 ns and 38% LUT utilization, leaving substantial margin for future iteration.

4. Fixed-Point Numerical Fidelity

All 10 layers were numerically validated against a PyTorch float32 reference model using identical weight values. The Q8.8 format was selected to balance precision against DSP48E2 utilization (355/2520, 14.1%).

Layer Group	Max Absolute Δ	Pearson Correlation
conv1 – conv8 (layers 0–8)	< 1.0 LSB	> 0.9990
det (layer 9, linear)	43.4	0.9991

The detection layer deviation is mathematically expected: the unbounded linear activation accumulates Q8.8 rounding error multiplicatively across 10 layers of fixed-point arithmetic. A correlation of 0.9991 on the detection head confirms that the quantization error is systematic (scaling bias) rather than stochastic (random corruption), and does not degrade detection quality beyond what is recoverable by NMS confidence thresholding.

5. Resource Utilization and Power

Post-route utilization on xczu9eg-ffvb1156-2-e:

Resource	Used	Available	Utilization
LUT	105,148	274,080	**38.4%**
LUT as Logic	*89,204*	*274,080*	*32.5%*
LUT as Memory	*15,944*	*144,000*	*11.1%*
Flip-Flop	145,010	548,160	**26.5%**
BRAM 36K	362	912	**39.7%**
DSP48E2	355	2,520	**14.1%**
BUFG	4	196	2.0%

Power breakdown (Vivado post-route power analysis):

Domain	Power
Total on-chip	**6.98 W**

Domain	Power
PL dynamic (accelerator)	3.48 W
PS static + DDR	2.14 W
I/O and clocking	1.36 W

The low DSP utilization (14.1%) indicates significant headroom for MAC array expansion. Doubling TILE_OC from 16 to 32 (512 MACs/cycle) would raise DSP utilization to approximately 28% — well within the device envelope — while delivering an estimated 1.5–2× throughput gain.

Engineering Lessons

1. **Memory bandwidth is the primary bottleneck, not compute.**

The 11.6× end-to-end speedup from V1 to V3 was achieved without adding a single MAC unit. All performance gain came from restructuring DDR access patterns to enable AXI burst inference. This validates the roofline model: at 256 MACs/cycle and 167 MHz, the theoretical compute peak is 85.6 GMAC/s, but V1's effective compute utilization was only 1.8% due to memory starvation. V3 raised this to 29.6% — a 16.4× improvement — purely through memory access pattern optimization.

2. **HLS burst inference has strict, non-obvious preconditions.**

AXI burst inference in Vitis HLS requires: (a) a single, statically bounded loop with monotonically incrementing address, (b) no conditional branches inside the burst loop, (c) no aliasing between read and write ports, and (d) no multi-dimensional index arithmetic that introduces non-linearity. Violating any one of these silently degrades all burst ports to single-beat transactions, which is the single largest performance failure mode in HLS-based accelerator design.

3. **Clock frequency scaling has a hard ceiling imposed by place-and-route.**

V2 demonstrated that targeting 250 MHz with 85% LUT utilization is not a viable strategy on UltraScale+ at this design size. Routing congestion at high utilization introduces hold-time violations that synthesis-level timing closure cannot resolve. The correct approach is to reduce LUT pressure first (through resource sharing, loop restructuring, or tiling factor reduction), then incrementally increase clock frequency from a clean baseline.

4. **Full-stack co-design is mandatory.**

HLS kernel throughput in isolation does not determine system performance. The PS-side pipeline — weight extraction from PyTorch .pth checkpoints, Q8.8 quantization, PYNQ overlay DMA orchestration, YOLO decode, and NMS — contributed 8.45 ms of software overhead (the stride-1 MaxPool for conv6 executed on the ARM). In future iterations, moving this operation to the PL as a dedicated HLS kernel would eliminate the only remaining PS-side compute bottleneck.

5. **Quantization strategy must be validated at the architecture level, not post-hoc.**

The choice of Q8.8 over Q4.12 or Q6.10 was made based on the dynamic range requirements of the detection head (unbounded linear output). A uniform fixed-point format applied across

all layers is a conservative but safe choice for a first silicon validation. Per-layer dynamic fixed-point (e.g., Q8.8 for conv layers, Q12.4 for the detection head) is the recommended next step to improve detection accuracy without increasing DSP utilization.

Path to 30 FPS
Current throughput: 2.7 FPS (277 ms/frame HW + 8.45 ms SW).
Target: 30 FPS (33.3 ms/frame) — requires **~8.4× further speedup**.
The following optimizations are individually validated and collectively sufficient to reach the target:

Optimization	Mechanism	Expected Gain	DSP Impact	Complexity
Wider MAC array (TILE_OC=32, 512 MACs)	2× compute throughput per cycle	**1.8–2.0×**	28% (+14%)	Low
512-bit AXI bus (32 values/beat)	2× memory bandwidth per port	**1.5–1.8×**	None	Low
Clock increase to 200 MHz	1.2× cycle rate (requires <70% LUT)	**1.2×**	None	Medium
Layer fusion (conv→pool→conv)	Eliminates 5 inter-layer DDR round-trips	**1.5–2.0×**	None	High
Dual-CU instantiation (2× parallel engines)	Parallel tile execution across two datapaths	**1.6–1.9×**	28% (+14%)	High

Combined estimate (multiplicative, conservative): 1.9 × 1.6 × 1.2 × 1.5 = **~5.5×** from hardware changes alone, bringing throughput to ~15 FPS. Full 30 FPS requires layer fusion or dual-CU, both of which are architecturally feasible within the ZCU102 resource envelope (remaining headroom: 62% LUT, 61% FF, 60% BRAM, 86% DSP).
