Project Work Report
Real-Time Object Detection Using Hardware-Accelerated CNN on Xilinx Zynq FPGA

Team Leader: Parth Sangal
Team Member: Hariom Meena

1 Introduction

This project aimed to design and implement a hardware-accelerated Convolutional Neural Network (CNN) inference system on a Xilinx Zynq SoC platform. The system architecture partitions computation between the ARM processor and FPGA fabric to enable real-time object detection.

Although the complete hardware deployment could not be finalized due to toolchain and framework integration issues (particularly during FINN conversion in WSL), significant progress was achieved in software modeling, quantized training, system planning, and hardware mapping design.

This document details the work plan, implementation methodology, technical challenges, system-level decisions, and lessons learned.

2 Overall Work Plan

2.1 Dataset Selection and Preprocessing

We selected 7 object classes from the Pascal VOC 2017 dataset for object detection. The selected dataset required extensive preprocessing because:

- Pascal VOC provides annotations in XML format.
- YOLO requires normalized bounding box format (class, x_center, y_center, width, height).

A custom PyTorch script was written to:

- Parse XML annotations.
- Convert bounding boxes into YOLO-compatible format.

- Generate class-index mappings.
- Perform dataset restructuring for training.

Dataset Challenges

- Severe class imbalance:
  - Person class: 3500+ samples
  - Other classes: 300–400 samples
- Anchor box mismatch initially caused unstable training.
- YOLO loss instability due to improper anchor scaling.
- Gradient explosion during early epochs.

Class imbalance significantly biased predictions toward the "person" category, requiring careful reweighting and dataset balancing strategies.

2.2 Model Design and Training

We implemented a custom Tiny YOLOv2 architecture in PyTorch incorporating:

- Lightweight convolution blocks
- Reduced feature depth for FPGA compatibility
- Quantized layers using Brevitas

Quantization Strategy

We used Brevitas to implement:

- INT8 fixed-point quantization
- Quantized activations
- Quantized convolution weights

The motivation behind INT8 quantization:

- FPGA-friendly fixed-point arithmetic
- Reduced DSP utilization
- Lower memory bandwidth
- Simplified hardware mapping

YOLO Loss Challenges and Training Strategy

The YOLO (You Only Look Once) loss function jointly optimizes bounding box localization, objectness confidence, and multi-class classification within a single regression framework. While powerful, this multi-component structure introduces several practical training challenges.

Key Challenges

1. **Loss Component Imbalance:** The confidence loss, especially from no-object grid cells, often dominates early training, leading to unstable convergence and slower localization learning.
2. **Anchor Box Mismatch:** Improperly chosen anchor boxes increase bounding box regression error, causing large localization loss and poor IoU alignment.
3. **Class Imbalance:** Significant disparity in class sample counts (e.g., dominant "person" class) biases the classification component of the loss.
4. **Gradient Instability:** Responsibility assignment among multiple anchors per grid cell can cause sudden loss spikes and noisy gradient updates.
5. **Quantization Sensitivity:** INT8 quantization increases sensitivity to scaling errors, especially in bounding box width–height prediction.

Adopted Strategies

1. Recalculated anchor boxes using clustering on dataset bounding boxes.
2. Tuned localization (λ_{coord}) and no-object (λ_{noobj}) scaling factors.
3. Used Adam optimizer for adaptive gradient updates and improved stability.
4. Applied controlled sampling and augmentation to reduce class bias.
5. Implemented gradual learning rate decay to prevent early divergence.

CPU Baseline Performance

The inference latency on CPU-only configuration was measured between:

$$200 \text{ ms to } 400 \text{ ms}$$

This corresponds to approximately 2.5–5 FPS, which is not suitable for real-time embedded deployment.

This justified the need for hardware acceleration.

3 Initial Hardware Acceleration Plan

3.1 Vitis AI Consideration

Initially, we planned to use Vitis AI. However:

- Vitis AI is optimized for DPU (Deep Processing Unit).
- DPU overlays are mainly targeted for Zynq UltraScale+ MPSoC.
- Our target board was PYNQ-Z2 (Zynq-7000 series).

The PYNQ-Z2 does not efficiently support standard DPU overlays without significant customization.

Therefore, we shifted strategy.

3.2 Why PYNQ-Z2 and DMA Approach

We selected PYNQ-Z2 due to:

- Python-based development environment
- Jupyter Notebook support
- Easier AXI-DMA configuration
- Faster prototyping capability

Role of DMA

DMA (Direct Memory Access):

- Transfers data directly between DDR memory and FPGA fabric.
- Bypasses ARM CPU for large tensor transfers.
- Reduces CPU bottleneck.
- Minimizes latency.

Expected latency reduction mechanism:

- Convolution operations parallelized in FPGA.
- Batch tensor streaming via AXI-DMA.
- Overlapping compute and transfer cycles.

Theoretically, latency could reduce from 200–400 ms to sub-50 ms range depending on parallelism.

4 HLS4ML Attempt and Issues

We initially explored HLS4ML because:

- Automatic conversion from Keras/PyTorch models to HLS.
- Structured hardware-friendly generation.

However, issues encountered:

- Limited support for YOLO-style detection heads.
- Poor support for complex tensor reshaping.
- Anchor grid computation incompatibility.
- Large memory footprint after conversion.

Due to architectural mismatch with detection-based CNNs, we discontinued HLS4ML.

5 Adoption of FINN Framework

FINN is an open-source framework developed by Xilinx for quantized neural network deployment on FPGA.

5.1 Why FINN?

- Native support for Brevitas quantized models.
- ONNX-based intermediate representation.
- Generates hardware dataflow architecture.
- Produces synthesizable HLS and RTL code.

This made FINN ideal for our INT8 quantized Tiny YOLOv2 model.

6 Complete Planned Flow

Detailed Implementation Flow

```
PyTorch + Brevitas (INT8 Training)
             ↓
Export Quantized Model to ONNX
             ↓
```

```
Verify Graph using Netron
            ↓
FINN Import and Streamlining
            ↓
Generate HLS + RTL
            ↓
Vivado IP Block Design
            ↓
Add AXI-DMA (if not auto-generated)
            ↓
Synthesize and Generate design.bit
            ↓
Deploy on PYNQ-Z2
            ↓
Python Jupyter DMA Overlay Control
            ↓
Real-Time Inference
```

7 Netron-Based Graph Debugging

Netron was used to visualize ONNX graphs.

We discovered hardware-incompatible layers such as:

- Dropout (not used during inference)
- Batch Normalization layers not folded properly

Important clarification:

- Dropout is disabled during inference and must be removed.
- BatchNorm must be fused into convolution weights.

Unfused BatchNorm creates additional arithmetic blocks, increasing hardware complexity.

Therefore:

- We modified training pipeline.
- Removed unnecessary layers.
- Retrained model entirely.

This optimization improved hardware compatibility and reduced node complexity.

Novelty of Our Implementation

- Custom YOLOv2 detection head tailored for FPGA.
- Full INT8 training using Brevitas.
- Manual dataset restructuring.
- Hardware-aware architectural pruning.

8 Major Challenges Faced

1. WSL instability during FINN dependency builds.
2. Docker permission conflicts.
3. FINN-experimental build failures.
4. ONNX export incompatibilities.
5. Resource estimation mismatches.

Primary Blocking Issue

The major blocking issue occurred during:

- Editable build of FINN-experimental dependencies.
- Python package compilation failures in WSL.

  Possible causes:

- Incompatible Python versions.
- Conda environment conflicts.
- Docker-WSL file permission mismatch.
- Missing board definition files.

9 Future Improvements and Learnings

- Begin hardware environment setup before model training.
- Lock Python and dependency versions early.
- Test minimal CNN conversion before full YOLO deployment.
- Maintain Linux-native installation instead of WSL.
- Plan incremental validation stages.

10 Conclusion

Although the hardware deployment was not finalized, the project successfully demonstrated:

- Quantized CNN training pipeline.
- Dataset engineering for detection tasks.
- Hardware-aware neural network design.
- End-to-end deployment planning for FPGA acceleration.

The experience provided deep insights into hardware-software co-design, quantization-aware training, FPGA deployment workflows, and toolchain dependencies.

This foundation will significantly strengthen future embedded AI acceleration projects.