Hardware-Accelerated CNN Inference

on Xilinx Zynq SoC

Depthwise Separable Convolution Accelerator

Design and Implementation

Bharat AI-SoC Student Challenge 2026

Project Report

Submitted by

Jesswin George
Mathew James

Indian Institute of Technology Guwahati

February 2026

Contents

1 Introduction

Modern artificial intelligence and computer vision applications increasingly demand deployment on edge devices with stringent constraints on power consumption, memory bandwidth, and computational resources. Convolutional Neural Networks (CNNs) have emerged as the dominant paradigm for visual recognition tasks, yet their inherent computational requirements pose significant challenges for embedded systems.

1.1 Computational Challenge

Standard convolution operations are responsible for over 90% of the total inference compute cost in typical CNN architectures, making them the critical path for optimization. While Graphics Processing Units (GPUs) offer exceptional throughput, they consume significant power and exhibit non-deterministic latency characteristics unsuitable for low-power embedded platforms.

1.2 FPGA and SoC Advantages

Field-Programmable Gate Arrays (FPGAs), particularly heterogeneous System-on-Chip (SoC) platforms such as the Xilinx Zynq family, provide distinctive advantages for embedded CNN inference:

- **Fine-grained Parallelism**
- **Deterministic Latency**
- **Custom Datapath Optimization**
- **Tight CPU–Accelerator Integration**
- **Energy Efficiency**

1.3 Project Objectives

The primary objectives of this work are:

1. **Architectural Design**: Create a modular, streaming CNN accelerator specifically optimized for depthwise separable convolution operations.
2. **Parameterizable Implementation**: Develop a fully parameterizable RTL design suitable for diverse MobileNet-style network configurations.
3. **Resource Analysis**: Comprehensively evaluate FPGA resource utilization, compute density, and efficiency metrics.
4. **Scalability Foundation**: Establish a well-architected baseline for scaling to full neural network implementations.
5. **Real-time Capability**: Validate the design's capability for real-time embedded object detection applications.

2 Target Platform and Development Environment

2.1 Hardware Platform: Xilinx Zynq-7020 SoC - Zedboard

The accelerator targets the Xilinx Zynq-7000 family, specifically the xc7z020clg484-1 variant. The Zynq platform represents a heterogeneous System-on-Chip combining ARM processing capabilities with FPGA programmable logic on a single die.

2.1.1 Processing System (PS) Components

- **Processor Core**: Dual-core ARM Cortex-A9 running at nominal frequency of 667 MHz
- **L1 Cache**: 32 KB per core (instruction and data)
- **L2 Cache**: 512 KB shared cache
- **Memory Interface**: DDR3 SDRAM controller with up to 1066 Mbps bandwidth
- **Peripherals**: USB, Ethernet, SPI, I2C, and UART interfaces

2.1.2 Programmable Logic (PL) Resources

The xc7z020 device provides the following FPGA resources:

Table 1: Available FPGA Resources in xc7z020clg484-1

Resource Type	Available Quantity	Utilization (Design)
Slice LUTs	53,200	65.59%
Slice Registers	106,400	19.98%
DSP48E1 Blocks	220	65.45%
Block RAM (BRAM) Tiles	140	0%

2.2 Design Tools and Development Environment

- **Synthesis Tool**: Vivado Design Suite 2025.1 by Xilinx
- **HDL Language**: Verilog RTL for hardware description
- **Interconnect Protocol**: AXI4-Stream for module-to-module communication
- **Simulation**: Vivado Behavioral Simulation

3 Convolutional Neural Network Fundamentals

3.1 Standard Convolution Operation

The standard convolution operation applies a trainable kernel of size $K \times K$ across an input feature map to produce an output feature map. Mathematically, the convolution can be expressed as:

$$Y(x, y, k) = \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} \sum_{c=0}^{C_{in}-1} W(i, j, c, k) \cdot X(x + i, y + j, c) + b(k)$$

where:

- $Y(x, y, k)$ is the output feature at spatial location (x, y) and output channel k
- $X(x, y, c)$ is the input feature at spatial location (x, y) and input channel c
- $W(i, j, c, k)$ represents the learnable convolution kernel weights
- $b(k)$ is the learnable bias for output channel k
- C_{in} is the number of input channels
- C_{out} is the number of output channels (equal to number of kernels)

3.1.1 Computational Complexity

The computational complexity of standard convolution is:

$$\text{Operations}_{standard} = H \times W \times C_{in} \times C_{out} \times K^2$$

where H and W are the spatial height and width of the input feature map. For a typical layer with dimensions $H = 224$, $W = 224$, $K = 3$, $C_{in} = 32$, $C_{out} = 64$, this yields approximately 72 million multiply-accumulate operations.

3.2 Depthwise Separable Convolution

Depthwise separable convolution is a factorization technique that decomposes a standard convolution into two sequential operations: a depthwise convolution followed by a pointwise convolution. This factorization dramatically reduces computational complexity while maintaining expressive power.

3.2.1 Depthwise Convolution

The depthwise convolution applies a separate $K \times K$ kernel to each input channel independently:

$$Y_{dw}(x, y, c) = \sum_{i=0}^{K-1} \sum_{j=0}^{K-1} W_{dw}(i, j, c) \cdot X(x + i, y + j, c) + b_{dw}(c)$$

The key distinction is that there is exactly one kernel per input channel. The computational cost is:

$$\text{Operations}_{dw} = H \times W \times C_{in} \times K^2$$

For the same parameters as above ($H = 224$, $W = 224$, $K = 3$, $C_{in} = 32$), this is approximately 4.5 million operations—a 16x reduction compared to standard convolution with 64 output channels.

3.2.2 Pointwise Convolution

The pointwise convolution applies 1×1 convolutions to mix channel information:

$$Y_{pw}(x, y, k) = \sum_{c=0}^{C_{in}-1} W_{pw}(c, k) \cdot Y_{dw}(x, y, c) + b_{pw}(k)$$

Since the kernel size is 1×1, there is no spatial filtering. The computation per output feature is simply a dot product of length C_{in}:

$$\text{Operations}_{pw} = H \times W \times C_{in} \times C_{out}$$

For our example ($H = 224$, $W = 224$, $C_{in} = 32$, $C_{out} = 64$), this is approximately 101.6 million operations.

3.2.3 Overall Complexity Reduction

The total computational cost of depthwise separable convolution is:

$$\text{Operations}_{separable} = H \times W \times (C_{in} \times K^2 + C_{in} \times C_{out})$$

Compared to standard convolution:

$$\text{Reduction Factor} = \frac{C_{in} \times K^2 + C_{in} \times C_{out}}{C_{in} \times C_{out} \times K^2} = \frac{1}{C_{out}} + \frac{1}{K^2}$$

For $C_{out} = 64$ and $K = 3$, this yields a reduction factor of approximately $1/64 + 1/9 \approx 0.027$, or roughly a 37x reduction. This dramatic reduction makes depthwise separable convolution the foundation of efficient mobile architectures like MobileNet, MobileNetV2, and similar designs.

3.3 MobileNet Architecture

MobileNet uses depthwise separable convolutions as fundamental building blocks. Each depthwise separable block consists of:

1. Depthwise convolution (3×3)
2. Batch normalization and ReLU activation
3. Pointwise convolution (1×1)
4. Batch normalization and ReLU activation

This modular approach enables building efficient networks for embedded vision tasks while maintaining competitive accuracy on standard benchmarks.

4 Hardware Architecture and Design

4.1 Top-Level System Architecture

The accelerator implements a fully streaming dataflow pipeline where data flows continuously through the system without requiring intermediate storage. The top-level architecture consists of:

This pipeline design ensures:

- **Low Latency**: Data is processed continuously without buffering entire feature maps
- **Bounded Memory**: Working set remains constant regardless of input size
- **Scalability**: Throughput can be increased through increased parallelism
- **AXI Compliance**: Standard interfaces enable integration with Xilinx infrastructure

4.2 Design Parameters and Configurations

The architecture is fully parameterizable to support various layer configurations:

Table 2: Key Design Parameters

Parameter	**Value**
Data Width (DATA_W)	8 bits
Image Width/Height (IMG_WIDTH)	224 pixels
Input Channels (CIN)	32 channels
Output Channels (COUT)	64 channels
Depthwise Kernel Size (KERNEL_SIZE)	3×3
Padding Strategy	SAME (zero-padded)
Parallel Input Channels (PAR_CIN)	16 channels
Parallel Output Channels (PAR_COUT)	16 channels

These parameters achieve a balance between resource utilization and performance. The 16-channel parallelism in both input and output dimensions is a design choice that provides reasonable compute density without exceeding device resource limits.

5 Depthwise Convolution Engine

5.1 Architectural Overview

The depthwise convolution engine implements the depthwise spatial filtering operation. Unlike standard convolution where different spatial kernels operate on different channel combinations, depthwise convolution maintains strict per-channel separation.

5.2 Core Components

The depthwise convolution engine consists of three primary functional modules:

5.2.1 Image Buffer and Sliding Window Generator

The `depthwise_image_control` module maintains a sliding 3×3 window of the input feature map. This window slides across the input spatial dimensions (H and W), advancing one pixel per clock cycle in the horizontal direction and wrapping at line boundaries.

Key responsibilities:

- Maintain a 3-row × 3-column window buffer
- Handle boundary conditions through zero-padding
- Generate valid output signals indicating when the kernel is fully within padded input
- Produce deterministic streaming output at 1 pixel per cycle throughput

5.2.2 Weight Storage and Management

The `depthwise_kernel` module stores the per-channel depthwise convolution weights. For efficiency, weights are stored in tiled format matching the parallelism of the compute unit:

$$\text{Weight Memory Size} = C_{in} \times K^2 \times \text{DATA\_W bits}$$

For $C_{in} = 32$ and $K = 3$:

$$\text{Memory Size} = 32 \times 9 \times 8 \text{ bits} = 2,304 \text{ bits} \approx 288 \text{ bytes}$$

This modest memory requirement allows efficient implementation using distributed FPGA memory resources.

5.2.3 Multiply-Accumulate Unit

The `depthwise_mac` module implements the core computation. For each spatial position, it performs:

$$\text{MAC Operations} = C_{in} \times K^2 = 32 \times 9 = 288 \text{ MACs per pixel}$$

With PAR_CIN=16, these operations are partitioned:

$$\text{Execution Cycles} = \frac{288}{16} = 18 \text{ cycles per spatial position}$$

The unit employs pipelined multipliers to maintain throughput despite the sequential nature of accumulation.

5.3 Operational Characteristics and Dataflow

- **Streaming Model**: One output feature map pixel is produced per clock cycle (after initial pipeline warm-up)
- **Channel-Level Parallelism**: PAR_CIN=16 enables parallel processing of multiple channels

- **AXI Compliance**: Full AXI-Stream protocol support with ready/valid handshaking
- **Backpressure Support**: Output stalls automatically propagate upstream to prevent buffer overflow

The fully streaming nature eliminates the need for large intermediate buffers, resulting in minimal memory overhead and deterministic latency.

6 Pointwise Convolution Engine

6.1 Operational Model

The pointwise convolution engine implements 1×1 convolutions to perform channel mixing. Unlike spatial convolutions, pointwise operations have no spatial kernel—all computation occurs along the channel dimension.

6.2 Parallel Compute Architecture

The design employs 2D tiling of the channel and output dimensions:

Table 3: Pointwise Convolution Parallelism

Dimension	**Parallelism**	**Iterations**
Input Channels	PAR_CIN = 16	32 / 16 = 2
Output Channels	PAR_COUT = 16	64 / 16 = 4

This configuration yields:

$$\text{Parallel MACs per Cycle} = PAR_{CIN} \times PAR_{COUT} = 16 \times 16 = 256$$

Per output spatial pixel, the engine must iterate through all channel and output combinations:

- Outer loop: $NUM_{COUT\_ITER} = 4$ iterations over output channels
- Inner loop: $NUM_{CIN\_ITER} = 2$ iterations over input channels
- Each inner iteration performs MAC_LATENCY + 1 cycles

Total cycles per output pixel:

$$\text{Cycles}_{pixel} = NUM_{COUT\_ITER} \times (NUM_{CIN\_ITER} + \text{MAC\_Latency}) = 4 \times (2+6) = 32 \text{ cycles}$$

Thus the throughput bottleneck shifts from the depthwise engine (1 pixel/cycle) to the pointwise engine (1 pixel/32 cycles).

6.3 Internal Datapath Structure

6.3.1 Finite State Machine Control

The `pointwise_conv1x1_fsm_axis` module orchestrates the multi-level iteration and tiling:

1. Initialize weight registers from external memory
2. For each input spatial pixel:
   (a) For each output channel iteration:
      i. For each input channel iteration:
      ii. Load weight tile
      iii. Perform 256 MACs (16×16)
      iv. Accumulate results
3. Output final accumulated results

6.3.2 Weight Register Management

The `pointwise_weight_regs` module stores weight values in a register file with organized addressing:

$$\text{Total Weights} = C_{in} \times C_{out} = 32 \times 64 = 2,048 \text{ weights}$$

With 8-bit precision, this requires:

$$\text{Storage} = 2,048 \times 8 = 16,384 \text{ bits} \approx 2 \text{ KB}$$

6.3.3 MAC Datapath

The `pointwise_mac_datapath` implements the core 16×16 parallel MAC unit. Each MAC position performs:

$$\text{Output}[i][j] = \sum_{k=0}^{\text{LATENCY}} \text{Input}[k][i] \times \text{Weight}[k][j]$$

The pipelined design accumulates products over the latency cycles, reducing critical path delay at the cost of multi-cycle computation.

6.4 AXI-Stream Integration

The pointwise engine implements full AXI-Stream protocol compliance:

- **Input Interface** (`s_axis_*`): Accepts streaming depthwise output
- **Output Interface** (`m_axis_*`): Produces streaming results
- **Handshaking**: Valid/Ready protocol ensures data integrity
- **Backpressure**: Stalls propagate to the depthwise engine when buffers are full

7 AXI4-Stream Protocol Compliance

7.1 Protocol Overview

AXI4-Stream is a Xilinx-defined standard for streaming data transfers. All modules implement the following standard signals:

Table 4: AXI4-Stream Handshaking Signals

Signal	Description
`tvalid`	Source asserts to indicate valid data on bus
`tready`	Sink asserts to indicate readiness to accept data
`tdata`	The actual payload data (width: DATA_W bits)
`tlast`	Marks the last word of a packet

Data transfer occurs only when both `tvalid` and `tready` are asserted.

7.2 Backpressure Propagation

In a streaming pipeline, backpressure occurs when a downstream module cannot accept data. This stall must propagate upstream to prevent data loss:

```
Output Stage Stall (tready = 0)
        ↓
Pointwise Engine Stall (tready = 0)
        ↓
Depthwise Engine Stall (tready = 0)
        ↓
Input Source Stall
```

This lossless backpressure mechanism ensures no data is corrupted or lost during stalls, a critical requirement for inference accuracy.

7.3 Timing and Synchronization

All modules operate on the same clock domain with synchronous reset. This simplifies validation and eliminates cross-clock domain hazards. The complete pipeline from input to output has:

Total Latency = Depthwise Latency+Pointwise Latency+Output Delay $\approx 18+32+2 = 52$ cycles

This latency is constant and predictable, essential for real-time systems.

8 Synthesis Results and FPGA Resource Utilization

8.1 Synthesis Configuration

- **Target Device**: Xilinx Zynq-7020 (xc7z020clg484-1)

- **Synthesis Tool**: Vivado 2025.1
- **Implementation Strategy**: Default timing-driven
- **Optimization Objective**: Maximize Performance (frequency)

8.2 Slice Logic Utilization

Table 5: Slice Logic Utilization Summary

Resource Type	Used	Available	Utilization
Slice LUTs	34,896	53,200	65.59%
Slice Registers	21,256	106,400	19.98%
LUT Flip-Flop Pairs	12,544	53,200	23.57%

8.2.1 Analysis

The 65.59% LUT utilization indicates significant computational density. The logic resources are distributed across:

- **Multipliers**: $16 \times 16 = 256$ parallel multiplications per cycle
- **Accumulators**: Multiple pipelined adder trees for reduction
- **Control Logic**: FSM state machines for tiling and iteration control
- **Datapath Routing**: Interconnect for data movement and steering

The relatively lower register usage (19.98%) compared to LUT usage reflects the architecture's emphasis on combinational computation with minimal state storage.

8.3 DSP Block Utilization

Table 6: DSP48E1 Block Utilization

Resource Type	Used	Available	Utilization
DSP48E1 Blocks	144	220	65.45%

8.3.1 DSP Allocation Breakdown

The DSP block allocation reflects the computational requirements:

- **Depthwise MACs**: ≈ 16 DSP blocks (PAR_CIN=16, sequential depth)
- **Pointwise MACs**: ≈ 128 DSP blocks (16×16 parallel array)

The high DSP utilization (65.45%) validates that the design achieves its goal of compute-intensive operation utilizing dedicated arithmetic resources rather than LUT-based arithmetic.

8.4 Block RAM Utilization

Table 7: Block RAM Utilization

Resource Type	Used	Available	Utilization
Block RAM Tiles	0	140	0%

8.4.1 Analysis and Implications

The zero BRAM utilization indicates that all weight and buffer storage is implemented using distributed LUT-based memory. This design choice has tradeoffs:

Advantages:

- Flexible memory architecture without BRAM port contention
- Simpler control logic without BRAM initialization complexity
- Lower latency for weight access through combinational logic

Disadvantages:

- Higher LUT consumption for memory implementation
- Limited scalability to larger networks with more weights
- Reduced opportunity for multi-port memories

8.5 Resource Utilization Summary

The current design demonstrates balanced utilization across both logic and arithmetic resources:

Resource Category	Utilization %	Status
LUT Logic	65.59%	Well-Utilized
Registers	19.98%	Under-Utilized
DSP Blocks	65.45%	Well-Utilized
Block RAM	0.00%	Unused

The asymmetry between LUT and register utilization is expected for a design emphasizing combinational computation. The high DSP utilization confirms successful exploitation of dedicated arithmetic resources.

9 Performance Analysis and Throughput Characterization

9.1 Throughput Bottleneck Analysis

The depthwise convolution engine operates at 1 output pixel per cycle, achieving excellent throughput for spatial filtering. However, the pointwise convolution engine serializes computation across channel and output dimensions.

With the current 16×16 parallel MAC architecture:

- **Iterations per Spatial Pixel**:
  1. Output Channel Iterations: $\frac{64}{16} = 4$
  2. Input Channel Iterations: $\frac{32}{16} = 2$
  3. Cycles per Inner Loop: MAC_LATENCY + 1 = 7
- **Total Cycles per Output Pixel**: $4 \times (2 \times 7) = 56$ cycles
- **Effective Throughput**: $\frac{1 \text{ pixel}}{56 \text{ cycles}} \approx 1$ pixel/56 cycles

For a 224×224 feature map:

$$\text{Total Computation Time} = 224 \times 224 \times 56 \text{ cycles} \approx 2.8 \text{ M cycles}$$

At 100 MHz clock frequency (typical for Zynq PL):

$$\text{Inference Latency} \approx 28 \text{ ms}$$

9.2 Throughput Improvements: Future Directions

Several architectural enhancements could significantly improve pointwise convolution throughput:

1. **Increased Output Channel Parallelism**: Scaling to PAR_COUT=32 or PAR_COUT=64 would reduce outer loop iterations at the cost of increased DSP usage.
2. **Systolic Array Architecture**: Implementing a systolic array for pointwise computation would pipeline across output channels, improving throughput from 1/56 pixels/cycle to potentially 1/2 pixels/cycle.
3. **Cross-Pixel Accumulation**: Exploiting pipeline parallelism to process multiple spatial pixels concurrently would require larger working set but could achieve near-1 pixel/cycle throughput.
4. **Reduced Precision**: Operating with 4-bit or 2-bit quantized weights and activations would reduce computation requirements and DSP usage, enabling higher parallelism within the same resource constraints.
5. **BRAM-Based Weight Tiling**: Using block RAMs for weight storage and multi-banking would eliminate LUT pressure and enable more aggressive tiling strategies.

10 Future Work and Extensions

10.1 Immediate Improvements

1. **Multi-Layer Dataflow**: Extend to complete MobileNetSSD architecture with all layer types
   - Depthwise separable blocks (current)

- Standard convolutions (for project-specific layers)
- Pooling operations (max, average)
- Non-linear activations (ReLU, ReLU6)

2. **Activation Functions**: Integrate ReLU/ReLU6 post-convolution
   - Piecewise linear implementation in LUT
   - Negligible area overhead
   - Seamless integration with AXI pipeline
3. **Batch Normalization**: Fused with convolution or implemented as separate layer
   - Scale and shift operations compatible with AXI streaming
   - Can be merged with weights for zero-overhead fusion

10.2 Architecture Extensions

1. **Precision Optimization**: Support for quantized inference
   - INT8 inference (current baseline)
   - INT4 weights with INT8 activations
   - Binary/ternary weight exploration
   - Dynamic fixed-point scaling
2. **Increased Parallelism**: Scale to xc7z045 or larger device
   - 4x DSP blocks (880 vs 220)
   - 4x LUTs (212,800 vs 53,200)
   - Enable larger batch sizes or higher channel parallelism
3. **Systolic Array Redesign**: Replace tiled architecture with systolic flow
   - Improved data reuse
   - Reduced memory bandwidth
   - Higher throughput density

10.3 Software and Deployment

1. **PetaLinux Integration**: Complete embedded Linux deployment
   - Device tree configuration for FPGA peripherals
   - Interrupt handling and DMA drivers
   - Power management and clock gating
2. **Framework Integration**: Plugin architecture for popular frameworks
   - TensorFlow Lite custom operator

- ONNX Runtime execution provider
- PyTorch JIT compiler backend

3. **Real-Time Camera Pipeline**: End-to-end object detection demonstration
   - CSI camera interface
   - Real-time preprocessing
   - Result overlay and HDMI output

10.4 Research Directions

1. **Sparse Computation**: Exploit neural network sparsity
   - Structured pruning of convolution filters
   - Sparse matrix formats for weight storage
   - Dynamic computation skipping
2. **Dynamic Power Management**: Reduce energy consumption
   - Frequency and voltage scaling based on layer requirements
   - Selective activation of compute units
   - Supply gating for unused modules
3. **Multi-Model Inference**: Support multiple networks in resource-constrained environment
   - Time-multiplexed FPGA reconfiguration
   - Partial reconfiguration of specific layers
   - Model ensemble execution

11 Conclusion

This work presents a hardware-realistic implementation of a depthwise separable convolution accelerator specifically designed for the Xilinx Zynq-7020 embedded SoC. The design demonstrates effective mapping of modern efficient CNN architectures onto FPGA fabric, achieving substantial acceleration over CPU-based inference.

11.1 Key Achievements

- **Efficient Architecture**: Fully streaming, AXI-compliant pipeline with deterministic latency
- **High Compute Density**: 256 parallel MAC operations per cycle utilizing 65.45% of available DSP blocks
- **Balanced Resource Utilization**: Well-distributed usage of LUTs (65.59%) and DSP blocks (65.45%)

- **Parameterizable Design**: Flexible architecture supporting diverse layer configurations
- **Scalable Foundation**: Modular design enabling straightforward extension to complete networks

11.2 Impact and Significance

The depthwise separable convolution accelerator demonstrates:

1. **Practical Embedded AI**: Feasibility of implementing state-of-the-art efficient neural networks on resource-constrained SoCs
2. **Hardware-Software Codesign Value**: Significant performance and efficiency gains through specialized hardware
3. **Open Design Space**: Opportunities for optimization across precision, parallelism, and architecture
4. **Scalability**: Pathway to complete real-time embedded object detection systems

11.3 Final Remarks

The implementation successfully establishes a modern, well-architected accelerator for efficient neural networks. The streaming dataflow, full AXI compliance, and parameterizable design provide a solid foundation for expanding to complete application networks. Future work should focus on multi-layer integration, enhanced throughput optimization, and complete system deployment with real-time camera pipelines.

This work contributes to the growing ecosystem of specialized AI accelerators for edge computing, demonstrating that high-performance neural network inference is achievable on modest embedded platforms through thoughtful hardware-software codesign.