SqueezeNet v1.0 Implementation on Kria KV260 using HLS

Abdelaziz Abdelrhman, Kobe Kodachi, Jackie Pham
EE 511: Machine Learning Hardware Accelerators
University of Southern California
Fall 2025

***Abstract*—In this work, we explore an implementation of SqueezeNet v1.0, a compact convolutional neural network, on the Xilinx Kria KV260 Vision AI Starter Kit using High-Level Synthesis (HLS). We first focus on training the SqueezeNet model in full precision on the CIFAR-10 dataset, achieving a test accuracy of 81.19%. Next, we explore quantization-aware training (QAT) to optimize the model for hardware deployment into eight-bit fixed precision with four integer bits. This quantized model achieved a test accuracy of 79.62%. The final implementation utilizes reusable HLS modules for convolution, pooling, and Fire modules, with a controller-based architecture for optimal synthesis and sequential execution management.**

***Index Terms*—SqueezeNet, FPGA, High-Level Synthesis, Quantization-Aware Training, Kria KV260**

I. Introduction

Convolutional neural networks (CNNs) have revolutionized computer vision tasks by allowing efficient processing of spatial data through feature reuse in the convolutional layers. However, their deployment on resource-constrained edge devices remains challenging due to high MAC operations and a heavy memory footprint. SqueezeNet, introduced by Iandola et al. in 2016 [1], addresses these challenges by achieving AlexNet-level accuracy with 50× fewer parameters through innovative architectural design.

This project implements SqueezeNet v1.0 on the Xilinx Kria KV260 FPGA platform, leveraging High-Level Synthesis to create hardware-accelerated inference. The original model was trained on the ImageNet dataset, but we used CIFAR-10 for this project approach. Our implementation upsamples CIFAR-10's 32×32 images to match the original 224×224 dimensions. This allowed us to maintain the ImageNet-designed architecture and core Fire module structure despite the differences in input size.

The key contributions of this work include:

- Training and validation of SqueezeNet v1.0 on CIFAR-10 with comprehensive hyperparameter documentation
- Implementation of 8-bit quantization-aware training to optimize for hardware constraints
- Development of modular HLS components for efficient FPGA deployment
- Controller-based architecture design for managing computational flow in hardware

II. Related Work

The original SqueezeNet architecture utilizes two convolutional layers at the input and output, with fire modules serving as the core building block in between the outer layers. Each Fire module consists of a squeeze layer, which uses 1×1 convolutions to reduce channel dimensionality, followed by an expand layer that combines 1×1 and 3×3 convolutions, whose outputs are concatenated. This design strategy significantly reduces parameter count while maintaining representational capacity.

Quantization is a technique that reduces the numerical precision of a model, empowering numerous hardware benefits such as faster computation, smaller memory footprint, and less power per MAC operation. A major issue with quantization is that models tend to suffer accuracy drops in the conversion from FP32 to INT8, or eight-bit fixed, in our case. As a result, quantization-aware training (QAT) has emerged as a vital element of modern model deployment as it reduces the accuracy drop during conversion [3]. This technique prepares models for reduced bit-width representations during the training process by allowing quantized forward passes with full precision gradients during backward propagation. As a result, QAT typically achieves superior accuracy compared to post-training quantization methods.

Field-Programmable Gate Arrays (FPGAs) offer unique advantages for neural network inference, including customizable data paths, parallel processing capabilities, and low-latency execution. High-Level Synthesis tools like Vitis HLS enable hardware design at higher abstraction levels, making FPGA development more accessible [5]. Designers can insert pragmas that assist the synthesis tool, allowing fine-grained optimization for pipelining, loop unrolling, and memory partitioning. In this work, we utilize these capabilities to empower efficient inference, eight-bit fixed precision with four integer bits.

III. Model Training and Quantization

A. Dataset Preparation

The CIFAR-10 dataset consists of 60,000 32×32 color images across 10 classes, split into 50,000 training and 10,000 test images [2]. Since SqueezeNet v1.0 was originally designed for ImageNet's 224×224 input dimensions, we resize CIFAR-10 images to 224×224 pixels using bilinear interpolation.

Data augmentation strategies were applied during training to improve generalization:

- Random horizontal flipping with 50% probability
- AutoAugment policy specifically tuned for CIFAR-10
- Normalization using CIFAR-10 dataset statistics (mean: [0.4914, 0.4822, 0.4465], std: [0.2023, 0.1994, 0.2010])

B. Network Architecture Implementation

The SqueezeNet v1.0 architecture was implemented in PyTorch following the original specification without bypass connections. The Fire module implementation includes:

- *Squeeze layer*: 1×1 convolution reducing channel count
- *Expand layer*: Parallel 1×1 and 3×3 convolutions with outputs concatenated along channel dimension
- *Activation*: ReLU applied after squeeze and expand operations

In this work we use the (Kaiming) initialization [4] to set the weights of all convolutional and fully connected layers, which is well suited for networks that use ReLU activations. Weights are sampled from a zero-mean normal distribution and scaled according to the layer's fan-out. This helps to preserve activation variance as signals propagate through deep networks which reduces the risk of vanishing or exploding gradients. Bias terms are initialized to zero to avoid introducing unintended offsets at the start of training.

C. Training Configuration

The full-precision model was trained using the hyperparameters shown in Table I.

TABLE I
FULL-PRECISION TRAINING HYPERPARAMETERS FOR SQUEEZENET ON CIFAR-10

Parameter	Value
Optimizer	Adam
Learning Rate	1e-3
Weight Decay	1e-3
Batch Size	128
Epochs	100
Loss Function	CrossEntropyLoss (label_smoothing=0.1)
Initialization	Kaiming Normal (fan_out, ReLU)

D. Training Results

Our training process was well within the boundaries of expected results. The training loss steadily decreased from approximately 2.2 to around 1.1 over 100 epochs, indicating effective learning. Test loss decreased from approximately 1.9 to about 0.9, showing good generalization.

Training accuracy improved from approximately 10% to 75.85%, while test accuracy increased from about 10% to 81.19%. Since both test and training accuracy started at around 10% this means that the model was initialized effectively. CIFAR-10 has 10 classes, which means that the model was effectively outputting random predictions for the first few epochs. The final training accuracy was 5.34% lower than the final test accuracy, as expected, since data augmentation and label smoothing are only applied during training to facilitate better generalization. The final test accuracy of 81.19% represents a reasonable performance for this architecture on CIFAR-10, demonstrating successful adaptation of the originally ImageNet-designed architecture to this dataset.

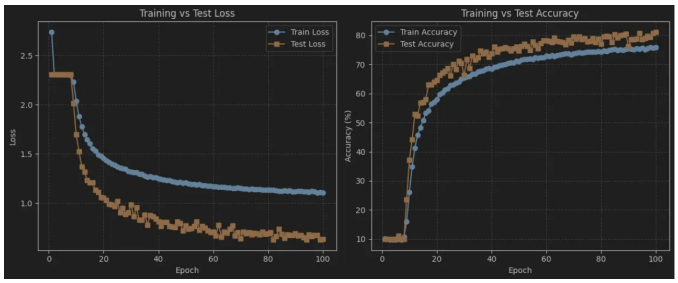

Fig. 1. Training curves for full-precision SqueezeNet over 100 epochs. Left: Training and test loss showing convergence. Right: Training and test accuracy, reaching 81.19% test accuracy.

E. Quantization-Aware Training

Following convergence of the full-precision model, quantization-aware training was performed to prepare the network for fixed-point hardware implementation. The quantization configuration employed:

- Bit-width: 8-bit fixed-point representation
- Integer bits: 4 bits
- Fractional bits: 4 bits
- Range: [-8.0, 7.9375] with 0.0625 precision
- Quantization scope: Both weights and activations

Custom quantization functions were inserted into the model architecture to simulate quantization effects during training. The forward pass includes explicit quantization and dequantization operations, allowing gradients to flow through these operations using straight-through estimators. This approach enables the network to learn weights that are robust to quantization noise.

The QAT process used the pre-trained full-precision weights as initialization, with a reduced learning rate of 1e-4 and weight decay of 1e-4 for stable fine-tuning. Training was performed for 15 epochs.

F. Quantization Results

The quantized model demonstrated minimal accuracy degradation compared to the full-precision baseline. Final test accuracy of 79.62% (compared to 81.19% full-precision) represents only a 1.57% degradation, validating the effectiveness of the quantization-aware training approach. The 8-bit representation provides substantial memory savings (4× reduction from 32-bit floating-point) and enables efficient fixed-point arithmetic in the FPGA implementation while preserving model performance.

IV. HIGH-LEVEL SYNTHESIS IMPLEMENTATION

A. Design Architecture Overview

The HLS implementation follows a modular design philosophy with controller-based execution management. Rather

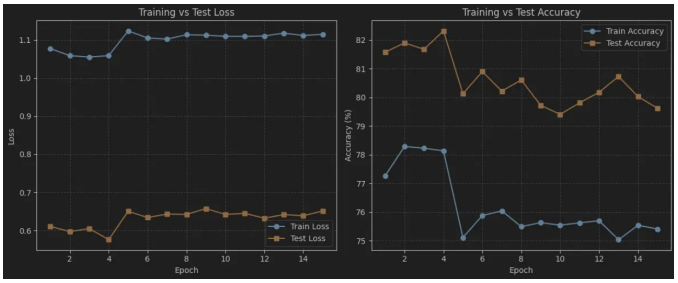

Fig. 2. QAT fine-tuning curves over 15 epochs. The model maintains ∼80% test accuracy while adapting to 8-bit fixed-point quantization.

than sequential function calls, the architecture employs enable signals for each computational module, with a dedicated controller responsible for:

- Activating and deactivating module enable signals
- Managing memory address generation for input and output buffers
- Passing appropriate arguments to each functional unit
- Maintaining correct computational ordering across Fire modules

B. Core HLS Modules

1) Convolution Modules: Two specialized convolution implementations were developed:

- *conv1*: Handles the initial 7×7 convolution (stride 2, padding 3) from 3 input channels to 96 output channels. Uses line buffering with array partitioning (factor=7) for efficient data reuse.
- *conv10*: Final 1×1 convolution reducing 512 channels to 10 class outputs.

Key optimizations include loop pipelining (II=1) for inner loops, array partitioning for parallel access to weight and activation, and partial sum accumulation using 24-bit accumulators to prevent overflow. We chose to use two convolution modules because they allowed for further loop unrolling and smaller buffers, as the input and output sizes are known at compile time.

2) Max-Pooling Module: Implements 3×3 max-pooling with stride 2 using a line buffer architecture. Three input rows are buffered simultaneously, enabling pipelined comparison operations with full loop unrolling of the 3×3 window.

3) Fire Module: Encapsulates the complete Fire module computation through three sub-functions:

- *squeeze*: 1×1 convolution with fused ReLU activation
- *expand1*: 1×1 convolution for expand path with fused ReLU
- *expand3*: 3×3 convolution (stride 1, padding 1) with line buffering and fused ReLU

Channel concatenation is achieved by writing expand1 outputs to channels [0, EC) and expand3 outputs to channels [EC, 2×EC) of the output buffer.

4) Average Pooling Module: Performs global average pooling over 14×14 spatial dimensions using a 24-bit accumulator to maintain precision during summation.

C. Controller Design

The controller implements layer-by-layer execution of the SqueezeNet pipeline through two key functions:

- *configure_layer()*: A switch-case structure that populates a LayerConfig structure with layer-specific parameters (dimensions, channel counts, kernel sizes) based on layer index (0-13).
- *execute_layer()*: Dispatches to the appropriate computational module based on layer type, passing configured parameters and selecting the correct weight buffers.

The controller manages 14 sequential layers and employs ping-pong buffering with two activation buffers (buf1, buf2) that alternate as input/output between layers. We experimented with multiple design choices for storing the weights. The first approach stores weights in static buffers and loads them on demand for each Fire module, thereby minimizing the memory footprint. This approach can be seen in the final code. Due to memory issues that will be explained in the following sections, we also explored taking the weights as input in the top function. This approach was not included in the final draft.

D. Memory Architecture

Memory organization critically impacts performance. The design employs:

- On-chip BRAM for intermediate activations
- Double-buffering for input/output overlap
- Partitioned arrays for parallel access patterns
- Optimized data layout for cache efficiency

E. Optimization Strategy

HLS pragmas were strategically applied to balance latency and resource utilization:

- `#pragma HLS PIPELINE II=1`: Applied to inner loops across all modules to achieve single-cycle initiation intervals for maximum throughput.
- `#pragma HLS UNROLL`: Full unrolling of MAC (multiply-accumulate) loops in convolution kernels to enable parallel computation of kernel window operations.
- `#pragma HLS ARRAY_PARTITION`: Applied with multiple strategies:
  - Complete partitioning for kernel weight buffers (enabling full parallel access)
  - Factor-based partitioning (factor=4 to 7) for line buffers and partial sum arrays to balance parallelism with resource usage
  - Cyclic partitioning for activation buffers in conv3d
- `#pragma HLS DEPENDENCE`: False inter-iteration dependencies declared for output arrays to enable loop pipelining without stalls.

All critical inner loops achieved the target initiation interval of II=1, as confirmed by HLS synthesis reports.

V. Results

A. Software Results

Task 1 - Full-Precision Training:

- Final Training Accuracy: ∼75%
- Final Test Accuracy: 81.19%
- Training Loss: ∼1.1 (final epoch)
- Test Loss: 0.63 (final epoch)
- Convergence: Achieved after 100 epochs

Task 2 - Quantization-Aware Training:

- Quantization Scheme: 8-bit fixed-point (4 integer bits, 4 fractional bits)
- Quantized Model Accuracy: 79.62% (1.57% degradation from full-precision)
- Model Size Reduction: 4× (from 32-bit float to 8-bit fixed-point)

B. HLS Synthesis Results

Each computational module was synthesized individually using Vitis HLS 2021.1 targeting the Kria KV260 platform. Table II summarizes the estimated resource utilization for each kernel.

TABLE II
HLS resource utilization estimates by module (Vitis HLS 2021.1, targeting Kria KV260)

Module	BRAM	DSP	FF	LUT
conv1	3	0	55,942	184,992
maxpool	3	0	8,000	12,009
fire	343	23	18,364	98,497
conv10	0	0	102	728
avgpool	0	6	691	1,136
Total	**349**	**29**	**83,099**	**297,362**

The Fire module dominates BRAM consumption (343 blocks) due to intermediate activation buffers and weight storage for three convolution operations. The conv1 layer requires substantial LUT resources to implement the 7×7 kernel with line buffering. All pipelined loops achieved II=1.

Table III presents the timing analysis for each module. All modules met timing constraints with estimated clock periods ranging from 5.669 ns to 7.300 ns, providing positive slack against the 10 ns target (100 MHz). Latency varies based on input dimensions due to parameterized loop bounds.

TABLE III
HLS timing estimates by module targeting 100 MHz (10 ns clock period)

Module	Target Clock (ns)	Estimated Clock (ns)	Timing Met?
conv1	10.00	7.274	Yes
maxpool	10.00	7.103	Yes
fire	10.00	7.300	Yes
conv10	10.00	5.741	Yes
avgpool	10.00	5.669	Yes

All pipelined loops achieved the target initiation interval (II) of 1. Functional correctness was verified through C simulation, with all modules (conv1, maxpool, fire, conv10, avgpool) matching their respective golden reference implementations (Fig. 3).

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../../../Downloads/Vitis-HLS-Convolution/dim/testbench.cpp in release mode
   Compiling ../../../../../Downloads/Vitis-HLS-Convolution/dim/maxpool.cpp in release mode
   Compiling ../../../../../Downloads/Vitis-HLS-Convolution/dim/fire.cpp in release mode
   Compiling ../../../../../Downloads/Vitis-HLS-Convolution/dim/conv3d.cpp in release mode
   Compiling ../../../../../Downloads/Vitis-HLS-Convolution/dim/conv10.cpp in release mode
   Compiling ../../../../../Downloads/Vitis-HLS-Convolution/dim/conv1.cpp in release mode
   Compiling ../../../../../Downloads/Vitis-HLS-Convolution/dim/avgpool.cpp in release mode
   Generating csim.exe
Calling conv1 kernel
Calling conv1 golden
Comparing conv1 with golden
conv1() matches golden
Calling maxpool kernel
Calling maxpool golden
Comparing maxpool with golden
maxpool() matches golden
Comparing fire with golden
fire() matches golden
Calling conv10 kernel
Calling conv10 golden
conv10() matches golden
Calling avgpool kernel
Calling avgpool golden
Comparing avgpool with golden
avgpool() matches golden
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************

```

Fig. 3. C simulation results showing functional verification of all HLS kernels. Each module (conv1, maxpool, fire, conv10, avgpool) matches its corresponding golden reference implementation, confirming correctness before synthesis.

Note: Due to design complexity, synthesis of the complete SqueezeNet design did not finish. Each kernel was synthesized individually, and the total resource estimates represent the summation of individual module resources. Further discussion is provided in Section V-C.

C. Hardware Validation Results

Hardware deployment on the Kria KV260 was not completed due to memory limitations encountered during the Vivado synthesis and implementation stages. This section documents the deployment attempts and the challenges encountered.

1) Deployment Attempts: 1) Initial HLS Synthesis (Full Model): Multiple overnight synthesis attempts were made on both a home desktop computer and university lab workstations. The full SqueezeNet design caused the home desktop to crash during synthesis, and lab synthesis sessions were interrupted by external interference. The VM on the home desktop was given as many resources as possible while all other processes were killed. These synthesis attempts were spread out and planned in advance up to a week before the final due date. However, across these multiple attempts, none were able to synthesize fully due to interruptions or crashes.

2) Alternative Approaches Attempted: Since we did not have the resources to synthesize the whole model with weights, we attempted to read all the weights from .bin files; however, the functions to read these files are not supported by synthesis. As our final effort to reduce memory footprint, we attempted to load weights from external memory at runtime rather than storing them on-chip as constants. Even after this synthesis was still too large for memory, so we used other optimization methods as detailed in the following sections.

3) Model Pruning and Successful HLS Synthesis: To address the synthesis failures, several Fire module layers were removed

to reduce design complexity. After pruning, HLS C synthesis completed successfully in Vitis HLS 2021.1, achieving:

- Estimated Fmax: 154.85 MHz
- Peak allocated memory: 1.553 GB
- All loop constraints satisfied
- Total synthesis time: ∼370 seconds

The synthesis generated valid VHDL and Verilog RTL for the reduced SqueezeNet design (see supplementary screenshots: squeezenetsynthesis1.png, squeezenetsynthesis2.png).

4) Vivado Implementation Failures: Despite successful HLS synthesis, the Vivado implementation stage failed on both available systems due to out-of-memory (OOM) conditions during bitstream generation (see supplementary screenshots: vivadomemorylimit_homecomputer, vivadomemorylimit_labcomputer):

Lab Computer:

- OOM failures occurred during the "Out-of-Context Module Runs" phase
- The squeezenet submodule synthesis failed while other IP modules (zynq_ultra_ps, xbar, rst, auto_pc) completed successfully
- Error status showed "Submodule Runs Failed" with the squeezenet module as the failing component

Home Computer:

- After lab we attempt to synthesize the pruned approach on the home desktop
- We investigated the cause since the error messages were the same as the lab computer
- The Linux OOM killer terminated the Vivado synthesis process multiple times

2) Analysis: Our attempts to reduce the memory were unsuccessful as the Vivado synthesis OOM errors persisted despite not synthesizing weights and pruning the model. The memory limitations appear to stem from the large intermediate activation buffers required for SqueezeNet's 224×224 input resolution and the high channel counts in later Fire modules. The Kria KV260's Zynq UltraScale+ device has sufficient on-chip resources for the synthesized design (as indicated by HLS resource estimates), but the Vivado synthesis tool itself requires substantial host memory to complete the place-and-route process for designs of this complexity.

The .tcl file was successfully generated, indicating that the HLS-to-Vivado export completed. The failure occurred specifically during Vivado's RTL synthesis of the SqueezeNet IP core, before reaching the implementation or bitstream generation stages.

3) Supplementary Evidence: Screenshots documenting the synthesis attempts and errors are included in the project submission:

- squeezenetsynthesis1.png, squeezenetsynthesis2.png: Successful Vitis HLS synthesis after model pruning
- vivadomemorylimit screenshots: OOM errors during Vivado synthesis on both systems

VI. Conclusion

This work presents a comprehensive implementation of SqueezeNet v1.0 for the Kria KV260 platform, spanning from initial model training through quantization and HLS design. The full-precision model successfully achieved 81.19% test accuracy on CIFAR-10, demonstrating effective adaptation of the ImageNet-designed architecture to this smaller dataset through proper preprocessing and training strategies.

The 8-bit quantization-aware training maintained model performance at 79.62% accuracy despite the precision reduction. This enables multiple benefits for deployment, such as a smaller memory footprint, faster MAC operations, and reduced power consumption. The modular HLS architecture with controller-based execution management provides a flexible framework for inference on the FPGA. All individual kernels were successfully synthesized and verified through C simulation, meeting timing constraints at 100 MHz with an estimated Fmax of 154.85 MHz.

Hardware deployment was ultimately limited by host machine memory constraints during Vivado synthesis, despite successful HLS C synthesis of the pruned model. This highlights a practical challenge in FPGA development: while HLS tools can estimate on-chip resource utilization, the host system requirements for synthesis and place-and-route of complex CNN designs can exceed available memory on typical development machines.

Future work could address these limitations by employing more aggressive model pruning, implementing layers sequentially with smaller activation buffers, or utilizing tile-based processing to reduce peak memory requirements. Additionally, access to workstations with larger memory capacity (64+ GB) would enable the successful completion of the Vivado implementation flow.

References

[1] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," arXiv preprint arXiv:1602.07360, 2016.

[2] A. Krizhevsky, "Learning multiple layers of features from tiny images," Technical Report, 2009.

[3] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in Proc. IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018.

[4] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in Proc. IEEE International Conference on Computer Vision (ICCV), 2015.

[5] Xilinx Inc., "Vitis High-Level Synthesis User Guide (UG1399)," 2023.