2025 FPGA 智慧運算與終端節點創意應用競賽
A3D3 Special Track 決賽報告書
隊名: FPGenius　組員: 陳冠晰、李昀達

---

1. Abstract and Motivation

1.1. Background

Semantic segmentation is a critical component of computer vision applications, as a class label is assigned to each pixel, enabling a fine-grained understanding of the scene. This makes it useful for applications such as autonomous driving, robotics, and urban scene analysis.

The Cityscapes dataset is utilized for training and evaluation in this project, as it comprises high-resolution images of urban environments across multiple cities, along with pixel-level annotations for 19 object classes. This makes it an ideal dataset for evaluating model performance in real-world driving conditions.

The primary focus of this project is to design, train, and deploy a lightweight segmentation model on an FPGA platform using hls4ml, with the goals of achieving a high segmentation mean Intersection over Union (mIoU), while maintaining a minimum throughput of 60 FPS (≤16.67ms per frame) on the target FPGA. This metric ensures that the model balances accuracy of segmentation with computation efficiency, showing the practicality of deployment for real-time segmentation applications such as autonomous driving.

1.2. Proposed Approach

To meet these requirements, we opted for a U-Net style encoder-decoder architecture with two encoder/decoder pairs and a bottleneck layer. Each layer contains two convolution layers with residual connections with the addition of strided convolutions for downsampling during encoding to preserve spatial context. Data augmentation techniques such as upscaling, class-aware cropping, and flipping were applied to further improve generalization. The model was quantized using Quantization-Aware Training (QAT) via QKeras, which simulates quantization effects during training to maintain model accuracy under low-precision constraints.

The model was then exported into an HLS-compatible configuration using hls4ml (version 1.1.0). This configuration was synthesized and deployed to FPGA hardware using the Xilinx Vitis HLS 2023.2 toolchain, targeting the Xilinx Alveo U55C board. This end-to-end flow enabled a fully quantized, hardware-optimized neural network that maintained competitive accuracy while achieving high efficiency in logic and memory utilization.

1.3. Result Overview

This project successfully implemented a fully quantized U-Net variation model for Cityscape image segmentation using QKeras and hls4ml. The quantized model achieved 0.35179 mIoU on the testing set, demonstrating competitive performance despite its reduced parameter count. Fine-tuned reuse factors and optimized FIFO

depths allowed us to balance latency and memory usage, passing resource requirements. We achieved a utilization of 30.38% LUTs, 20.52% FFs, 66.49% BRAMs 8.75% DSPs, with a frame processing rate of 113.9 FPS for c-synthesis and 66.2 FPS for RTL-simulation. These results highlight the effectiveness of combining architectural efficiency, quantization, and hardware-aware optimizations for real-time semantic segmentation.

2. Model Design and Training Techniques

2.1. Model Architecture

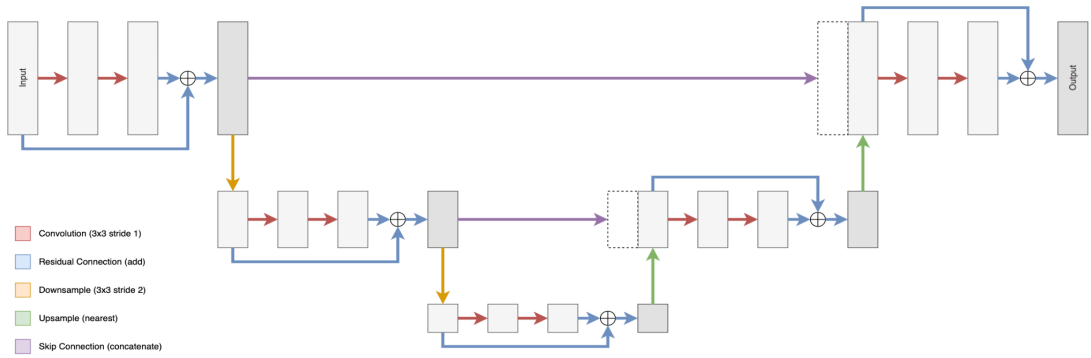

Fig. 1: Model Architecture

The backbone of our solution is a lightweight U-Net style encoder-decoder architecture with residual connections. The encoder consists of three stages. Each stage is built around a residual block that applies two convolutional layers with batch normalization and nonlinearity. A projection shortcut is used to ensure proper dimensionality alignment. The residual blocks help improve gradient flow and stabilize training. Downsampling is achieved using strided convolution rather than pooling, which preserves more spatial context while still reducing resolution. The three stages use 36, 48, and 60 filters, respectively, to capture high-level representations.

The decoder is a mirror of the encoding process with two upsampling stages. Each upsampling layer is followed by concatenating the corresponding encoder feature maps, forming the classic U-Net skip connections. The decoder stages gradually reduce the number of filters from 48 back to 36 in the final stage.

2.2. Data Augmentation

To improve model performance for mIoU and address the class imbalance in the dataset, we applied three main augmentation techniques: upscaling, class-aware crop, and horizontal flip. This enabled us to maintain the IoU for high classes while increasing the likelihood that small structures and classes remain distinguishable during training.

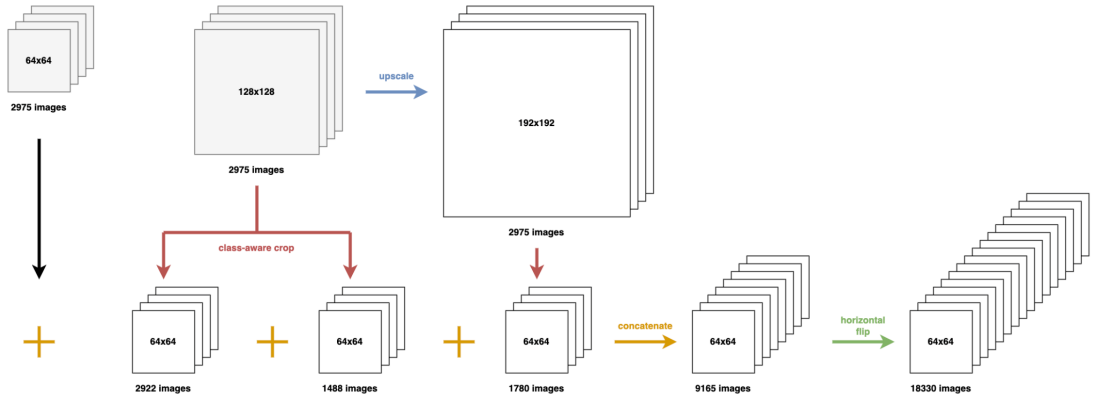

Fig. 2: Data augmentation flow

Upscale:
The default dataset includes images of 64x64 resolution, which causes small or rare classes (e.g., street signs, light poles) to shrink to only a few pixels wide, making them easily neglected and hard to learn. To mitigate this information loss, we generated an additional dataset by upscaling from 128x128 to 192x192 using bicubic resizing. The 128x128 images preserve finer detail for rare classes, while the 192x192 images further add to the resolution diversity.

Class-aware crop:
To improving underperforming classes, which disproportionately affected the mIoU metric, we utilized a class-aware crop. The cropping regions are selected based on the IoU of classes from trial runs on a baseline model. Three categories were defined:

- High IoU (1-0.5)
- Mid IoU (0.5-0.15)
- Low IoU (0.15-0)

For crop generation, we tuned two parameters:

- Number of crops per image: Mid classes receive 1 crop, since they typically occupy larger regions already, while low and extreme classes allow up to 2 crops to increase variety.
- Pixel threshold: For rare classes, a higher threshold is applied to ensure the cropped region contains a sufficient area of the class.

We also fine-tuned these parameters to keep each augmented dataset relatively similar in size so as not to dominate the default 64x64 images.


Fig. 3: Mid-class crop from 128x128 dataset

Fig. 4: Low-class crop from 128x128 dataset

Fig. 5: Low-class crop from 192x192 upscaled dataset

Horizontal flip:
Finally, a simple horizontal flip is applied after concatenating all augmented datasets. This doubles the data samples, adding invariance to orientation. Vertical flip was not utilized as the cityscape dataset has a default top-down orientation.

2.3. Quantization

The weights and biases in all layers were quantized using the format quantized_bits(8, 2, alpha=1) while quantized_relu(6) is used for activation. Reducing precision to 8-bit fixed-point with 2 integer bits and trainable scaling alpha=1 provides a good trade-off between numerical precision and hardware efficiency.

While converting the trained model to HLS using hls4ml, the default setting fixed<32, 16> was kept for generating the configuration file. However, this default value was rarely applied as the QKeras quantization specifications take precedence, ensuring consistency through training and hardware development.

2.4. Model Training

The model was trained using the AdamW optimizer with an initial learning rate of 1e-3 and a weight decay of 1e-4, as it combines the fast convergence of Adam with enhancements to weight regularization. This helps reduce overfitting and improves generalization.

To optimize segmentation performance, a combined loss was adopted, consisting of 70% Dice Loss and 30% Cross-Entropy loss. This helps achieve boundary precision and class-level segmentation as Dice loss puts emphasis on spatial overlaps while Cross-Entropy ensures per-pixel class balance across all 20 categories in Cityscape.

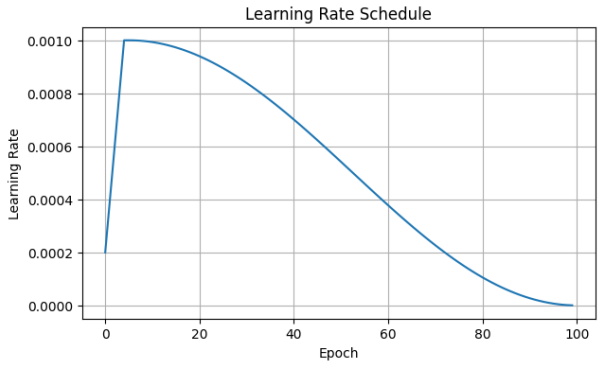

Fig. 6: Learning rate scheduler with cosine decay and 5% warmup

For improving training stability and convergence, we utilized a warmup with a cosine decay learning rate scheduler. The warmup spans the first 5% of the total epochs, gradually increasing the learning rate from zero to the target value, preventing unstable updates at the early stages of training. Cosine decay reduces the learning rate smoothly from 1e-3 to 1e-7, helping the model escape local minima early in the training while allowing fine-grain convergence in the later stages.

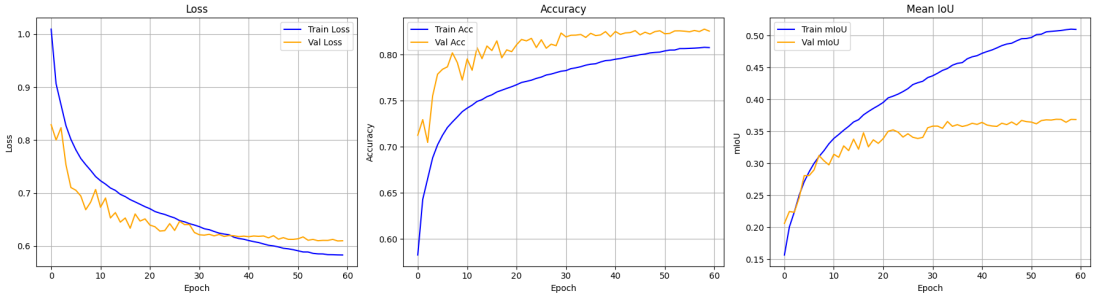

Fig. 7: Training and validation loss / accuracy / mIoU for baseline model

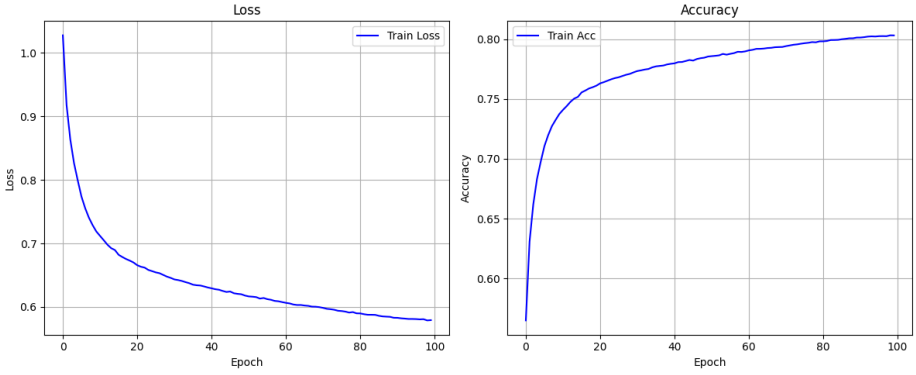

Fig. 8: Training loss / accuracy for final model

An initial 0.8/0.2 train-test split of the dataset was used. This provides a benchmark for estimating model performance, guiding design choices from validation results. Once the architecture was confirmed, the model was retrained on the full dataset,

making full use of the available image information, maximizing performance before deployment on the FPGA

3. HLS4ML Conversion

3.1. Conversion Strategy

For HLS4ML conversion, we configure the strategy to "Resource", where the convolution layers are mapped to a corresponding hardware module as follows:

```
template <class data_T, class res_T, typename CONFIG_T>
void conv_2d_cl(
    data_T data[CONFIG_T::in_height * CONFIG_T::in_width * CONFIG_T::n_chan],
    res_T res[CONFIG_T::out_height * CONFIG_T::out_width * CONFIG_T::n_filt],
    typename CONFIG_T::weight_t weights[CONFIG_T::filt_height * CONFIG_T::filt_width * CONFIG_T::n_chan * CONFIG_T::n_filt],
    typename CONFIG_T::bias_t biases[CONFIG_T::n_filt]) {
    // Inlining helps reduce latency, but may also cause timing issues in some cases, use carefully.
    //#pragma HLS INLINE recursive

    if (CONFIG_T::strategy == nnet::latency) {
        conv_2d_latency_cl<data_T, res_T, CONFIG_T>(data, res, weights, biases);
    } else {
        conv_2d_resource_cl<data_T, res_T, CONFIG_T>(data, res, weights, biases);
    }
}
```

Fig. 9: “Resource” for HLS4ML conversion strategy

This allows us to access the following module for each layer, which allows for further optimizations to the HLS4ML process.

```
template <class data_T, class res_T, typename CONFIG_T>
void conv_2d_resource_cl(
    data_T data[CONFIG_T::in_height * CONFIG_T::in_width * CONFIG_T::n_chan],
    res_T res[CONFIG_T::out_height * CONFIG_T::out_width * CONFIG_T::n_filt],
    typename CONFIG_T::weight_t weights[CONFIG_T::filt_height * CONFIG_T::filt_width * CONFIG_T::n_chan * CONFIG_T::n_filt],
    typename CONFIG_T::bias_t biases[CONFIG_T::n_filt]) {
    constexpr unsigned mult_n_in = CONFIG_T::filt_height * CONFIG_T::filt_width * CONFIG_T::n_chan;
    constexpr unsigned mult_n_out = CONFIG_T::n_filt;
    constexpr unsigned block_factor = DIV_ROUNDUP(mult_n_in * mult_n_out, CONFIG_T::reuse_factor);

    constexpr unsigned multiplier_limit = DIV_ROUNDUP(mult_n_in * mult_n_out, CONFIG_T::reuse_factor);
    constexpr unsigned multscale = multiplier_limit / mult_n_out;
```

Fig. 10: conv_2d_resource_cl module

In this configuration, the conv_2d_resource_cl layer exposes parameters such as the “reuse factor”, which enables the control of computation resources, balancing the trade-off between resource utilization and latency.

```
ReuseLoop:
    for (unsigned i_rf = 0; i_rf < CONFIG_T::reuse_factor; i_rf++) {
        #pragma HLS PIPELINE II=1 rewind
        unsigned i_in = i_rf;
        unsigned i_out = 0;
        unsigned i_acc = 0;
    MultLoop:
        for (unsigned i_blk = 0; i_blk < block_factor; i_blk++) {
            #pragma HLS UNROLL
        PixelMultLoop:
            for (unsigned i_pxl = 0; i_pxl < CONFIG_T::n_pixels; i_pxl++) {
                #pragma HLS UNROLL

                acc[i_pxl][i_out] += static_cast<typename CONFIG_T::accum_t>(
                    CONFIG_T::mult_config::template product<data_T, typename CONFIG_T::mult_config::weight_t>::product(
                        data_buf[i_pxl][i_in], weights_2d[i_blk][i_rf]));
            }
            // Increment i_in
            i_in += CONFIG_T::reuse_factor;
            if (i_in >= mult_n_in) {
                i_in = i_rf;
            }
            // Increment i_out
            if (i_acc + 1 >= multscale) {
                i_acc = 0;
                i_out++;
            } else {
                i_acc++;
            }
        }
    }
```

Fig. 11: Reuse loop in conversion based on reuse factor

3.2. MACs Calculation

To establish a solid baseline for tuning reuse factor and other HLS conversion parameters, we calculated the MACs (Multiply and Accumulate Operations) for each layer to estimate the default throughput.

$$MACs = C_h \times C_w \times C_{out} \times (K_h \times K_w \times C_{in})$$

By dividing the estimated MACs of each layer by its corresponding "reuse factor" we obtain the effective throughput of each layer under computation parallelization. Aligning the throughput across layers ensures a smooth dataflow, minimizing additional resource usage.

This calculation, however, only provides an initial baseline. Further optimization is required to meet the project constraints of maintaining resource utilization below 75% while achieving a frame rate greater than 60 FPS.

3.3. Reuse Factor

The "reuse factor" parameter in HLS4ML determines the number of times each multiplier is used to compute the values of a layer. This can be confirmed by looking at the multiplier limits defined in the configuration setting in file: parameters.h.

```
multiplier_limit = DIV_ROUNDUP(n_in * n_out, reuse_factor) - n_zeros / reuse_factor;
#define DIV_ROUNDUP(n, d) ((n + d - 1) / d)
```

Fig. 12: Multiplier count based on reuse factor settings

As demonstrated in the figure below, a low reuse factor would distribute computation, allowing each multiplier to perform fewer multiplications for the layer's output, vice versa for high reuse factors. Thus, a low reuse factor achieves the lowest latency and highest throughput at the cost of high resource usage, while the opposite applies for a high reuse factor.

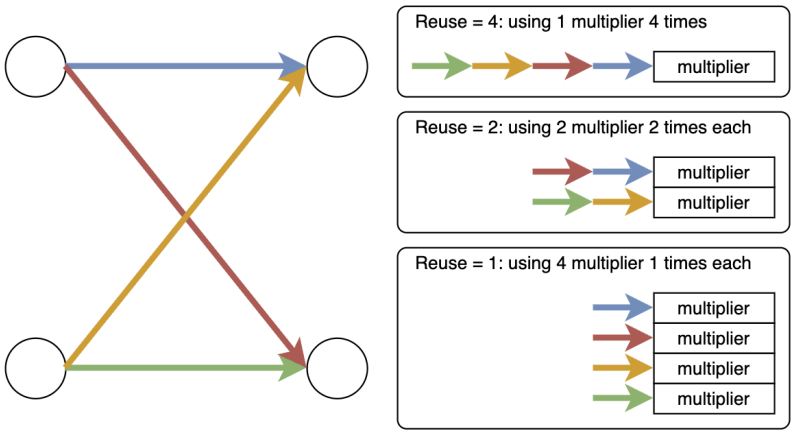

Fig. 13: Example of multiplier usage based on reuse factor

After the initial run under the configuration based on MAC calculation and throughput alignment, further tuning of the layer-specific reuse factors is required. A frame rate far below the threshold of 60 FPS stems from high reuse factors in deep

layers utilizing a long latency. Lowering the reuse factor for that specific layer calls for the use of more multipliers, increasing resource usage, but effectively lowering latency. For cases where resource utilization far exceeds the limit of 75%, increasing reuse factors of all layers to enable more shared resources, lowering utilization with the cost of increasing computation time. By balancing the tradeoff between resource utilization and latency, we can more easily reach the desired constraints.

3.4. FIFO Configuration

Tuning the reuse factor allowed us to satisfy most resource and latency metrics, except for BRAM usage. Our initial vivado_synth.rpt revealed that FIFOs were consuming nearly all the available BRAM resources. If optimizations were based solely on reuse factor adjustments, the resulting design would likely push the frame rates below limits, thereby penalizing mIoU performance.

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	2374	0	0	2016	117.76
RAMB36/FIFO*	2350	0	0	2016	116.57
RAMB36E2 only	2350				
RAMB18	48	0	0	4032	1.19
RAMB18E2 only	48				
URAM	481	0	0	960	50.10

Fig. 14: Initial model BRAM utilization

From the FIFO information reported in myproject_csynth.rpt, we see that FIFO size is determined by Depth*Bits, which will then be synthesized using RAMs, resulting in the high utilization of BRAMs:

* FIFO:

Name	BRAM_18K	FF	LUT	URAM	Depth	Bits	Size:D*B
layer10_out_U	29	2084	0	-	4096	1584	6488064
layer11_out_U	29	2084	0	-	1024	2232	2285568
layer13_out_U	8	543	0	-	1024	216	221184
layer14_out_U	29	2084	0	-	1024	1152	1179648
layer16_out_U	15	1056	0	-	1024	288	294912
layer17_out_U	29	2084	0	-	1024	1152	1179648
layer19_out_U	29	2084	0	-	1024	1008	1032192
layer21_out_U	15	1056	0	-	1024	288	294912
layer22_out_U	29	2084	0	-	1024	1056	1081344
layer23_out_U	29	2084	0	-	256	1920	491520
layer25_out_U	15	1053	0	-	256	288	73728
layer26_out_U	29	2084	0	-	256	1440	368640
layer28_out_U	15	1053	0	-	256	360	92160
layer29_out_U	29	2084	0	-	256	1500	384000

Fig. 15: FIFO size

```
INFO: [RTMG 210-285] Implementing FIFO 'layer2_out_U(myproject_fifo_w1656_d2048_A)' using Vivado Default RAMs.
INFO: [RTMG 210-285] Implementing FIFO 'layer4_out_U(myproject_fifo_w216_d2048_A)' using Vivado Default RAMs.
INFO: [RTMG 210-285] Implementing FIFO 'layer67_out_U(myproject_fifo_w216_d2178_A)' using Vivado Default RAMs.
INFO: [RTMG 210-285] Implementing FIFO 'layer5_out_U(myproject_fifo_w864_d2048_A)' using Vivado Default RAMs.
INFO: [RTMG 210-285] Implementing FIFO 'layer7_out_U(myproject_fifo_w1548_d2048_A)' using Vivado Default RAMs.
INFO: [RTMG 210-285] Implementing FIFO 'layer9_out_U(myproject_fifo_w216_d2048_A)' using Vivado Default RAMs.
INFO: [RTMG 210-285] Implementing FIFO 'layer10_out_U(myproject_fifo_w1584_d2048_A)' using Vivado Default RAMs.
```

Fig. 16: Synthesis of FIFOs using RAMs

To resolve this problem, we looked into modifying the FIFO depth directly after HLS conversion. As seen from the examples below, the default FIFO depth is usually defined by: $depth = Height \times Width\ (\times Channels)$

- Input layer of size 64 × 64 = 4096 FIFO length:

```
hls::stream<input_t> layer59_cpy1("layer59_cpy1");
#pragma HLS STREAM variable=layer59_cpy1 depth=4096
hls::stream<input_t> layer59_cpy2("layer59_cpy2");
#pragma HLS STREAM variable=layer59_cpy2 depth=4096
nnet::clone_stream<input_t, input_t, 12288>(input_1, layer59_cpy1, layer59_cpy2); // clone_input_1
```

- Zero padding and batch normalization (64 + 2) × (64 + 2) = 4356 FIFO length:

```
hls::stream<layer66_t> layer66_out("layer66_out");
#pragma HLS STREAM variable=layer66_out depth=4356
nnet::zeropad2d_cl<input_t, layer66_t, config66>(layer59_cpy1, layer66_out); // zp2d_q_conv2d_batchnorm
```

The default FIFO depth is a conservative setting, chosen to guarantee no stalls under any circumstance. However, in many pipeline scenarios, data is consumed at a faster rate than it is produced, meaning the allocated maximum FIFO depth is often never fully utilized.

In practice, the minimum required FIFO depth is determined by the latency difference between the producing and consuming layer. For example, if Layer A produces N items while Layer B is processing the current batch, the FIFO must be deep enough to buffer those N items to avoid overflow and stall.

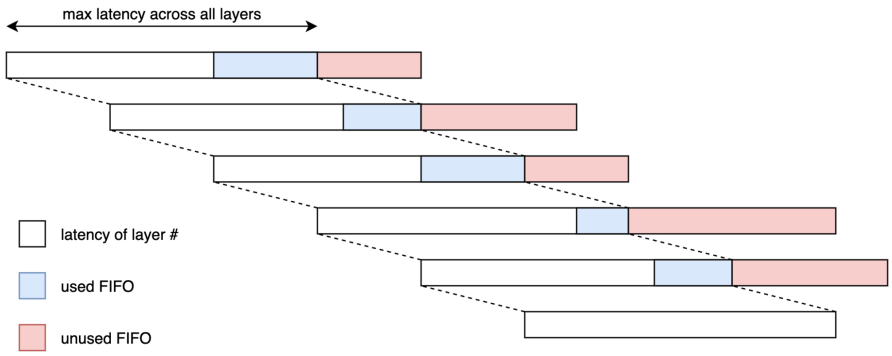

Fig. 17: FIFO usage in multi-layer pipelines

We adjust the FIFO size of each layer using the directive:

#pragma HLS STREAM variable=[layer_out] depth=[value]

By tuning FIFO depth according to producer-consumer latency, we achieve marginal savings in BRAM usage, allowing the design to remain below the 75% BRAM utilization threshold.

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	1340.5	0	0	2016	66.49
RAMB36/FIFO*	1317	0	0	2016	65.33
RAMB36E2 only	1317				
RAMB18	47	0	0	4032	1.17
RAMB18E2 only	47				
URAM	500	0	0	960	52.08

Fig. 18: Final BRAM usage after FIFO modification

To ensure this modification aligned with the dataflow, we verified our design by successfully passing both C/RTL co-simulation & validation:

```
////////////////////////////////////////////////////////////////////////////////
// Inter-Transaction Progress: Completed Transaction / Total Transaction
// Intra-Transaction Progress: Measured Latency / Latency Estimation * 100%
//
// RTL Simulation : "Inter-Transaction Progress" ["Intra-Transaction Progress"] @ "Simulation Time"
////////////////////////////////////////////////////////////////////////////////
// RTL Simulation : 0 / 5 [0.00%] @ "113000"
find kernel block.
// RTL Simulation : 1 / 5 [117.58%] @ "11468833000"
// RTL Simulation : 2 / 5 [143.42%] @ "19707003000"
// RTL Simulation : 3 / 5 [156.48%] @ "27945173000"
// RTL Simulation : 4 / 5 [169.54%] @ "36183343000"
// RTL Simulation : 5 / 5 [100.00%] @ "44421513000"
////////////////////////////////////////////////////////////////////////////////
$finish called at time : 44421542500 ps : File "/users/student/mr113/khchen24/fpga2025/final/daniel_model_08
ect.autotb.v" Line 247
run: Time (s): cpu = 04:26:58 ; elapsed = 31:49:05 . Memory (MB): peak = 4324.715 ; gain = 0.000 ; free phys
## quit
INFO: xsimkernel Simulation Memory Usage: 8740784 KB (Peak: 10216392 KB), Simulation CPU Usage: 114533120 ms
INFO: [Common 17-206] Exiting xsim at Tue Aug 26 09:15:56 2025...
INFO: [COSIM 212-316] Starting C post checking ...
```

Fig. 19: Passed C/RTL co-simulation

```
***** C/RTL VALIDATION *****
INFO: Test PASSED
INFO: [HLS 200-112] Total CPU user time: 810.13 seconds. Total CPU system time: 2.18 seconds. Total elapsed time: 8
ed memory: 776.172 MB.
```

Fig. 20: Passed C/RTL validation

4. Experimental Results and Discussion

4.1. Model Size

The final quantized model contains a total of 262,292 parameters, of which 261,200 are trainable and 1,092 are non-trainable (primarily from batch normalization layers). This is a relatively small U-Net variant by balancing depth and per-layer filter size, making it suitable for FPGA deployment while still achieving a good mIoU performance. This model configuration provides a solid starting point for HLS4ML conversion. However, the impact on hardware is better represented through post-synthesis FPGA resource utilization.

4.2. FPGA Resource Utilization

LUT & FF:

Site Type	Used	Fixed	Prohibited	Available	Util%
CLB LUTs*	396076	0	0	1303680	30.38
LUT as Logic	326259	0	0	1303680	25.03
LUT as Memory	69817	0	0	600960	11.62
LUT as Distributed RAM	0	0			
LUT as Shift Register	69817	0			
CLB Registers	535040	0	0	2607360	20.52
Register as Flip Flop	535040	0	0	2607360	20.52
Register as Latch	0	0	0	2607360	0.00
CARRY8	14539	0	0	162960	8.92
F7 Muxes	29120	0	0	651840	4.47
F8 Muxes	12752	0	0	325920	3.91
F9 Muxes	0	0	0	162960	0.00

BRAM:

Site Type	Used	Fixed	Prohibited	Available	Util%
Block RAM Tile	1340.5	0	0	2016	66.49
RAMB36/FIFO*	1317	0	0	2016	65.33
RAMB36E2 only	1317				
RAMB18	47	0	0	4032	1.17
RAMB18E2 only	47				
URAM	500	0	0	960	52.08

DSP:

Site Type	Used	Fixed	Prohibited	Available	Util%
DSPs	790	0	0	9024	8.75
DSP48E2 only	790				

Our final implementation successfully synthesized and deployed on the target FPGA platform, while maintaining all resource usage well below the competition thresholds. The resource utilization for the current configuration is as follows:

- LUTs: 30.38%
- Flip-Flops (FFs): 20.52%
- Block RAM (BRAM): 66.49%
- DSPs: 8.75%

Specifically, BRAM usage was controlled to stay under the desired 75% threshold by adjusting the FIFO size added between layers. Since we modified HLS after HLS4ML conversion, we still ran C/RTL co-simulation and validation to validate our changes.

4.3. Latency Analysis (FPS)

```
================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    |  Clock |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  |  5.00 ns|  3.935 ns|     1.35 ns|
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +---------+---------+----------+----------+---------+---------+----------+
    |  Latency (cycles) |  Latency (absolute) |      Interval     | Pipeline |
    |   min   |   max   |    min   |    max   |   min   |   max   |   Type   |
    +---------+---------+----------+----------+---------+---------+----------+
    |  1755377|  1755775|  8.777 ms|  8.779 ms|  1396738|  1755470|  dataflow|
    +---------+---------+----------+----------+---------+---------+----------+
```

Fig. 21: Performance estimates for Timing and Latency

```
$AVER_LATENCY = "3838213"
```

Fig. 22: RTL-simulation average cycle count

Based on the final round algorithm, the frame rate calculation is as follows:

$$1 / Max\ latency = 1 / (8.779 \times 10^{-6}) \approx 113.9\ FPS$$

Since RTL simulation was also conducted for verification, frame rate can also be calculated using the qualifying round algorithm:

$$1 / Csynth\ estimated\ clock\ period * RTL\text{-}sim\ average\ cycle\ count = 1 / (3838213 \times 3.935 \times 10^{-9}) \approx 66.2\ FPS$$

We can see that under both calculation methods, our model performance safely passes the frame rate requirement ≥ 60 FPS, making it suitable for real-time segmentation applications.

4.4. Performance Metrics

Baseline model (0.8 / 0.2 train validation split + 60 epochs):

- Training accuracy: 0.8069
- Training final mIoU (sklearn): 0.5073
- Validation accuracy: 0.8256
- Validation mIoU (sklearn): 0.3687
- Kaggle (post-HLS) mIoU: 0.34143

Final model (no validation set + 100 epochs):

- Training accuracy: 0.8138
- Training final mIoU (sklearn): 0.5481
- Kaggle (post-HLS) mIoU: 0.35179

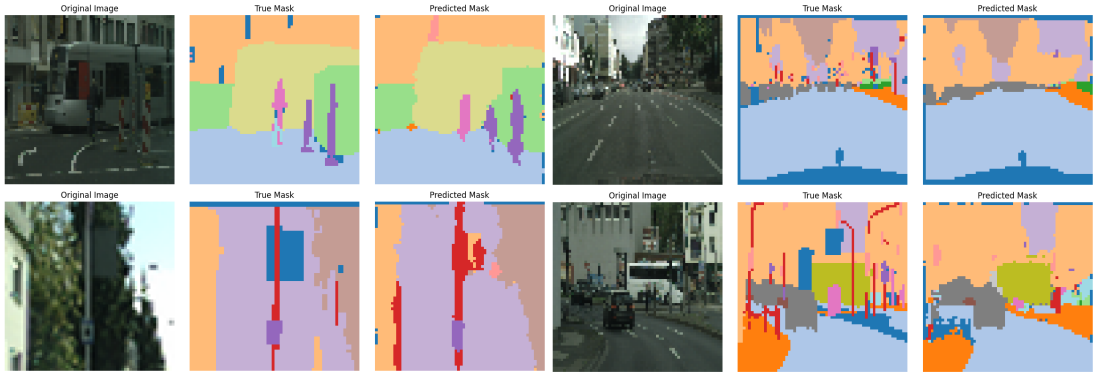

Fig. 23: Example training outputs

Since we used all the data for one last finalized run to increase the learning information, mIoU for the validation set no longer existed; thus, we utilized the training mIoU to capture and save the best model parameters. There also appears to be a slight drop in mIoU at the post-HLS stage, as intense data augmentation techniques were used to better capture features for rare classes. It could also be the result of additional fixed-point rounding and quantization effects introduced during hardware mapping.

4.5. Discussion

Throughout the development process, we explored various variations of the U-Net architecture to balance mIoU, parameter count, and utilization. We found that a model size of around 300k parameters was the limit to meet FPGA resource requirements. Changes in the model, like adding residual layers and downsampling with convolution yielded a 5~7% gain in mIoU as it greatly preserved the flow of more subtle patterns in the dataset.

The process of increasing the mIoU primarily involves data augmentation. Since the original 64x64 images are low in resolution, rare classes are likely lost during the process of training. Thus, the method of using the 128x128 dataset and upscaled 192x192 dataset was to increase the number of rare class images through class-aware crop. This method yielded an improvement in mIoU of around 3~5%. As a result, we improved the mIoU from around 25% by running the default model for 100 epochs to our current 35% post-HLS mIoU.

Despite BRAM utilization being an issue in the early stages, the FIFO method effectively resolved this problem by halving the utilization. The MAC estimation for reuse factor per layer also ensured that our project would be synthesized smoothly without stalling. It provided us with a good starting point for each synthesis run for balancing resource usage and FPS.

An unfortunate fact is that we did not acquire access to acceleration hardware for training, limiting the number of tests we could conduct. Each 100-epoch run takes around 30+ hours to complete with HLS4ML conversion and synthesis taking additional time to compile and finish. This greatly constrained our progress on implementing different model architectures or data augmentation techniques, making each run and decision crucial. For our future work, we plan to extend training and conduct broader experiments with advanced data augmentation techniques for more complex models, leveraging more powerful hardware setups to fully explore the performance-resource trade-off.