Convolutional Layer Implementations in High-Level Synthesis for FPGAs

Kelvin Lin

A thesis
submitted in partial fulfillment of the
requirements for the degree of

Master of Science in Electrical and Computer Engineering

University of Washington
2021

Committee:
Scott Hauck
Shih-Chieh Hsu

Program Authorized to Offer Degree:
Department of Electrical and Computer Engineering

Abstract

Field programmable gate arrays (FPGAs) offer a flexible hardware platform on which machine learning algorithms can be efficiently implemented. However, developing these algorithms on FPGAs can be prohibitive due to complex implementation details. We use the HLS4ML (High-Level Synthesis for Machine Learning) framework to translate models trained using traditional machine learning libraries into C++ which can then be translated into FPGAs firmware using High-Level Synthesis (HLS). We propose an alternative approach for convolutional neural networks within the HLS4ML framework. Using the new approach on benchmark convolutional neural network (CNN) models, we show a potential reduction of FPGA critical resource consumption by up to 30% and latency by up to 12%. Lastly, we describe the process in which we integrate the proposed approach in the HLS4ML framework.

Contents

1 Introduction

In the last decade, Convolution Neural Networks (CNNs) have become a standard for high-quality in machine vision related tasks [1]. Leveraging the availability of big data, CNNs have grown bigger, more accurate and more computationally intensive, thus requiring dedicated hardware for deployment. Field Programmable Gate Arrays (FPGAs) offer a flexible hardware platform to efficiently implement CNNs but require specialized expertise. High Level Synthesis for Machine Learning (HLS4ML) is an open-source framework that seeks to bridge the expertise gap [2]. By providing a user-friendly interface and toolset, HLS4ML facilitates the deployment of machine learning models on FPGAs [2]. In this thesis, we propose an alternative approach for convolutional layers within the HLS4ML framework. We demonstrate its performance on an example CNN architecture optimized for both resource utilization and latency.

This thesis is organized as follows: Section 2 provides an overview of neural networks and an introduction to CNNs. Section 3 introduces HLS4ML and establishes the necessity for an HLS-based approach to FPGA development. Section 4 discusses two convolution implementations for FPGAs. Section 5 introduces the benchmark models and examines performance metrics for both implementations. Section 6 describes the integration process for supporting multiple convolution implementations. Conclusions and further steps are given in Section 7 and 8, respectively.

2 Neural Networks

2.1 Overview

Neural networks are inspired by the neural connections in the human brain. They are a network of nodes, each emulating a neuron, that receives, processes and passes data to neighboring nodes. In a neural network, these connections are represented as edges, as seen in Figure 1, and each edge is assigned a weight that reflects the degree of attention given to the input. The network is made up of multiple layers of neurons wherein each of the neurons outputs a non-linear function of the input connections.

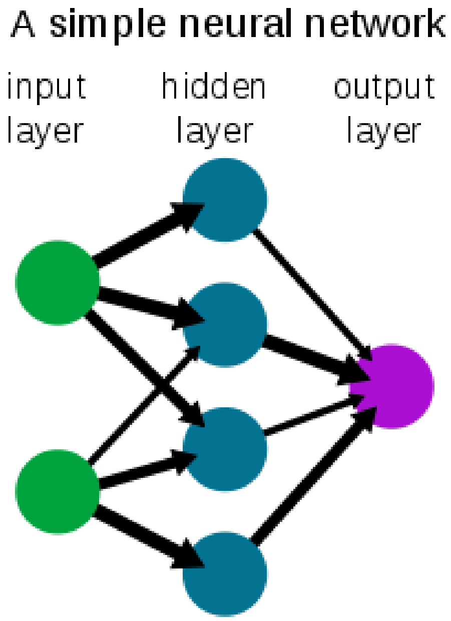


Figure 1. Simplified view of a feedforward neural network. Each circle represents an individual neuron. Edges between nodes are indicated by black arrows. The weight for each input is shown as the boldness of the arrow. [3]

2.2 Basic Components

The simplest architecture for neural networks is the multi-layer perceptron (MLP) [4]. This class of neural networks is organized in a similar fashion to Figure 1. MLPs consists of at least three layers of nodes: an input layer, hidden layer, and output layer. The hidden layer can be composed of one or more layers and are used to identify features in the input. The nodes compute a non-linear function of the sum of weighted inputs.

$$y = f\left(\sum_i w_i x_i \; + b\right) = f(\mathbf{Wx} \; + \; \mathbf{b}) \qquad (1)$$

Equation (1) describes the output computation for a MLP node. We multiply the weight matrix **W** with the input vector **x** and add a bias term **b**. This result is then input into a non-linear function, called the activation function, **f** and output to connected nodes.

Activation functions are added on the output of MLP nodes to introduce non-linearity into the network. This non-linearity assists the network in learning complex patterns in the data. Without an activation function, the nodes can at best learn linear boundaries as the computation of **Wx + b**.

These activation functions have a few properties that are desirable.

1. <u>Differentiable</u>. In order to perform backpropagation (the algorithm used in training neural networks), each operation performed within the network needs to be differentiable.
2. <u>Low computational cost</u>. Every node will use an activation function, so it should be efficient to compute.
3. <u>Non-saturating gradients</u>. In neural networks, we use the gradient to determine the best direction to adjust the parameters of the network in order to minimize prediction errors. In networks with many layers, the gradient with respect to the output will tend to disappear in early layers. This phenomenon is known as vanishing gradients and is caused by the activation function driving the gradient towards zero. When the gradient is close to zero, the network is essentially unable to learn because the weight updates during backpropagation become negligible.

Some commonly used activation function includes the sigmoid, tanh, rectified linear unit (ReLU), and softmax functions. Of these functions, sigmoid and tanh can lead to vanishing gradients and are not commonly used in convolutional neural networks.

2.3 Training and Inference

Neural networks are deployed in two stages: training and inference. During the training phase, the network is fed an annotated dataset with expected outputs for each input. These data samples are used to compare the model predictions against the expected values and to penalize the model via a loss function when updating the model's weights. After each round of predictions, we perform an operation called backpropagation where the network parameters are iteratively updated based on the loss function penalty [5]. This process of propagating the errors is essential for improving the predictive power of the neural network. The rate at which a neural network learns is defined by the learning rate. The learning rate is an important parameter that describes the degree that the model should adjust based on the error. Training with a high learning rate can change network weights significantly and cause convergence issues, while training with a low learning rate may result in long training times.

In the second stage, inference, the trained network makes predictions on new unseen data samples. While training is conducted offline, inference can be either offline or deployed online.

As such, the inference latency or throughput becomes an important factor when choosing the deployment hardware. This thesis focuses on the algorithmic design for convolutional layers to accelerate the inference phase on FPGAs.

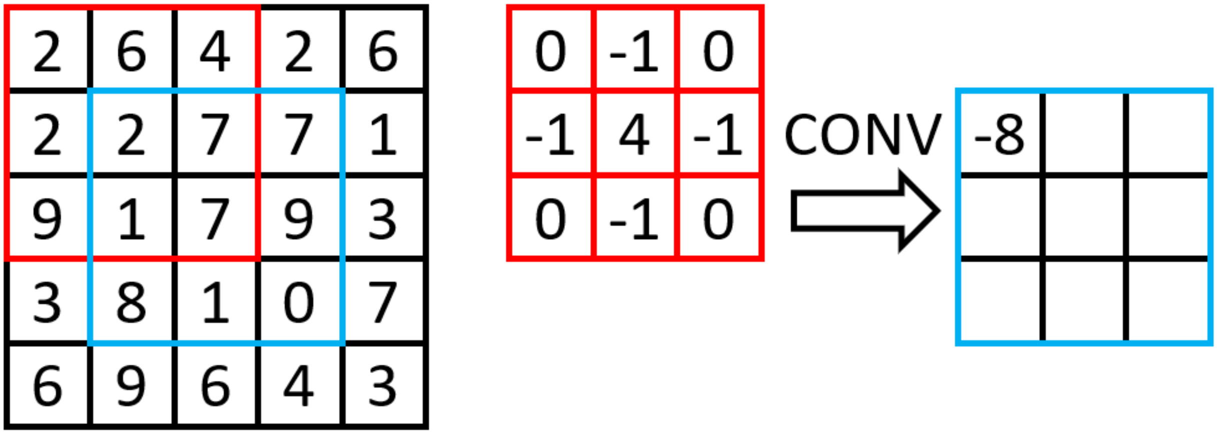
2.4 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a class of neural networks that are often used to analyze images. They are composed of a pipeline of convolutional, pooling, and fully connected layers. Each layer computes a feature map from the input feature map that contains progressively higher-level features. For example, if we train a CNN for the task of face identification, the low-level features might be edges, the mid-level features might be parts of a face, and the high-level features might be multiple parts of a face.

Typically, in CNN architectures, the number of feature maps (output channels) will increase, and the input size will decrease as we go deeper in the network. Each successive stage in the network will condense the features maps from the previous stage.

2.4.1 Convolutional Layer

Convolutional layers are the core component of CNNs. They perform the feature extraction process via a convolution operation. For a given input image of height **H**, width **W**, input channels **C** and output feature maps **N**, we shift a **J** x **K** window, called the kernel, across the image. At every position where the window completely overlaps the input feature map, we compute the Frobenius inner product of the convolutional kernel weights and the input region. This product computes a single output which is the summation of the elementwise product of the two matrices as shown in Figure 2. This process repeats for every input channel **C** and output channel **N** pair.



2	6	4	2	6
2	2	7	7	1
9	1	7	9	3
3	8	1	0	7
6	9	6	4	3

0	-1	0
-1	4	-1
0	-1	0

-8		

Figure 2. An example convolution operation on a 5x5 image with a 3x3 weight kernel (outlined in red). The output (outlined in blue) has size 3x3 because there are only 9 positions where the kernel can fully overlap the input image. The first output is computed as $-1 \times (6 + 2 + 7 + 1) + 4 \times 2 = -8$.

In the example shown in Figure 2, the output image is smaller than the input image due to the overlapping kernel constraint. In many cases, this is undesirable as it gives less attention to edge pixels, so it is common to add additional zero-padding to the image to allow the kernel to fully scan the entire image equally. When we pad the image, we add (**J – 1) / 2** pixels to the left and right side of the image and **(K – 1) / 2** pixels to the top and bottom of the image. This ensures that the convolutional output is the same size as the input.

Additionally, when computing the convolution, we typically only slide the kernel one pixel at a time. The number of pixels that the kernel is shifted for each convolution is called the *stride*. Increasing the stride has the effect of downsampling the image. For example, instead of using a default stride of (1, 1), we can elect to use a stride of (2, 2). This will cause the kernel to slide two pixels right for every horizontal shift and two pixels down for every vertical shift. The output image will then be half as wide and half as tall as the original image.

2.4.2 Pooling Layer

Pooling layers are periodically inserted in between successive convolutional layers. They are primarily used to reduce the spatial size of the feature maps and are useful for extracting dominant features. Like convolution, we slide a window across the image and compute the pooling operation when the window fully overlaps the input feature map. The stride is typically set to be the same as the pooling window size such that no pixel is pooled more than once.

There are two different types of pooling that are commonly used: max pooling and average pooling. Figure 3 illustrates an example of both types pooling operations on a 4x4 image. Max pooling takes the maximum value within the window while average pooling averages all values within the window.

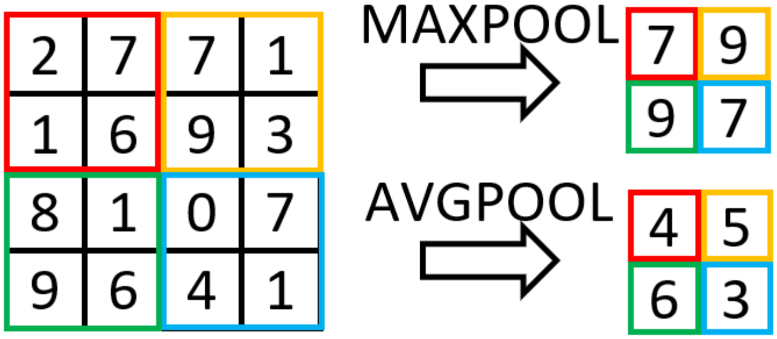


Figure 3. Comparison of MaxPooling and AveragePooling in two-dimensions with a (2, 2) pooling window. The different pooling windows are indicated by color.

Pooling layers also utilize padding if the feature map size is not a multiple of the pooling window size.

2.4.3 Fully Connected Layer

Fully connected layers are inserted at the end of the CNN pipeline. These layers operate identically to the multi-layer perceptron, but instead of using the input directly, they use the learned feature maps from the convolutional layers.

2.4.4 Other Layers

After each convolutional layer, we insert an activation layer. This activation works the same as the activations used in the multi-layer perceptron networks and adds non-linearity to the pipeline. Recent CNN architectures use the Rectified Linear Unit (ReLU) function which has reduced training time and computational complexity when compared to tanh or sigmoid activations [6]. However, the ReLU function has a gradient of 0 when the input is less than 0 and can cause neurons to become inactive and only output 0 for any input. Alternative variants of the ReLU activation have been proposed such as the parameterized ReLU (PReLU) or leaky ReLU where the negative values are multiplied by a small scalar [7].

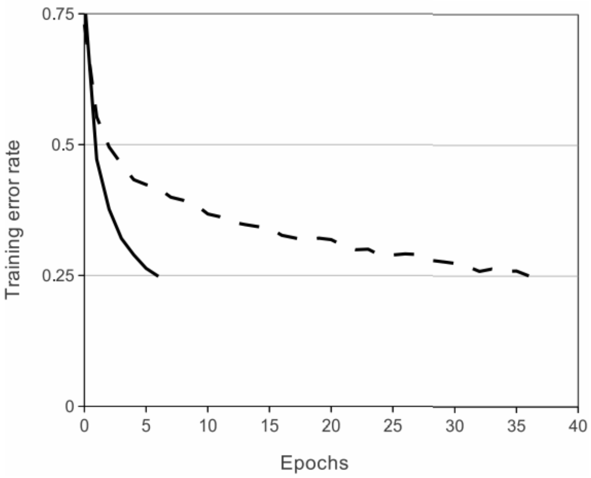


Figure 4. Plot from Krizhevsky et al. showing a 6x faster convergence rates of ReLU (solid) compared to tanh (dashed) activations in training [6].

Batch normalization layers were also introduced to accelerate training by (1) normalizing inputs to zero mean and unit variance and (2) rescaling and offsetting the normalized values [8]. During training, for each mini batch (or group) of inputs, we compute the mean μ_B and variance σ_B^2 across the mini batch and normalize the inputs. Then the network learns two additional parameters: a scaling parameter γ and bias parameter β for each output feature map. These additional parameters help to preserve model expressivity by allowing training to determine if normalization is needed.

Equation (2) describes the batch normalization computation during inference. During inference, the averaged mini batch mean μ_{Bk} and variance σ_{Bk}^2 are used to normalize the inputs and the learned parameters γ and β rescale and shift the normalized result.

$$\hat{x}_i \longleftarrow \frac{x_i - \mu_{Bk}}{\sqrt{\sigma_{Bk}^2 + \epsilon}}$$

$$y_i \longleftarrow \gamma\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \tag{2}$$

Batch normalization layers regularize training by normalizing inputs based on the mini batch statistics [8]. Since a given training sample is unlikely to be grouped with the exact same minibatch, the network sees unique inputs every training loop. Other research has shown that batch normalization has the effect of smoothing the optimization landscape [9]. By smoothing the gradients, we can use larger learning rates as the gradient will remain an accurate estimate even over larger steps [9].

2.5 Neural Networks on Hardware

Neural networks come with a large computational cost, and thus often require dedicated hardware to run efficiently. The most widely used hardware platform to deploy neural networks are Graphical Processing Units (GPUs) since their parallelized architecture and high-bandwidth memory make them suited for the repeated vector and matrix operations required by neural networks [10].

However, Field Programmable Gate Arrays (FPGAs) have shown superior performance compared to GPUs in terms of energy efficiency (performance per watt). FPGAs also have a highly configurable architecture that helps them adapt to different algorithms. Recent trends towards sparser networks and more compact data types favor FPGA devices as they can handle the irregular parallelism and custom data types [10].

However, designing algorithms for FPGAs can be difficult due to the lower level of abstraction of hardware description languages (HDL) like Verilog or VHDL. Additional attention must be given to implementation details that go beyond the design of the algorithm. In the next section, we will discuss tools that can assist in developing algorithms on FPGAs, namely, high level synthesis.

3 High-Level Synthesis for Machine Learning

3.1 High-Level Synthesis

High-level synthesis (HLS) refers to the process in which a design, written in a language like SystemC or C++, is translated to a register transfer level (RTL) implementation in a hardware description language (HDL) [11]. The higher level of abstraction of HLS enables users to develop algorithms without needing to write the HDL manually.

During the HLS process, the high-level code is analyzed and converted in an HDL language, like Verilog or VHDL, that accurately replicates the algorithmic behavior. The HLS tools handle the microarchitecture and timing to generate a cycle-by-cycle timed hardware implementation. The generated RTL can then be used in standard synthesis flows to create gate-level implementations that go onto the FPGA.

3.2 Motivation for HLS4ML

High-Level Synthesis for Machine Learning, also referred to as HLS4ML, is an open-source Python framework for implementing machine learning algorithms in firmware [2]. This framework simplifies the HLS process, providing both command line and Python interfaces to convert, build, and compile models into an RTL implementation. It enables users to rapidly prototype machine learning models on FPGAs while balancing performance, utilization, and latency requirements.

HLS4ML utilizes a YAML configuration file that can either be edited locally or through the Python API to configure various hardware implementation parameters. These parameters control aspects such as the degree of parallelism, compression, computation precision, and pipelining in the generated RTL. Additionally, each parameter can also be specified on a global, layer type, or by-layer level, thus providing the users with tools to fine tune their implementations to suit their applications.

The HLS4ML framework supports a variety of popular machine learning libraries:

- Keras/TensorFlow/QKeras
- PyTorch
- ONNX

Currently, the following machine learning models are either supported or in development by HLS4ML community:

- fully connected neural networks (multi-layer perceptrons)
- boosted decision trees (BDT)
- convolutional neural networks (CNN)
- recurrent neural networks (RNN).

3.3 HLS4ML Framework

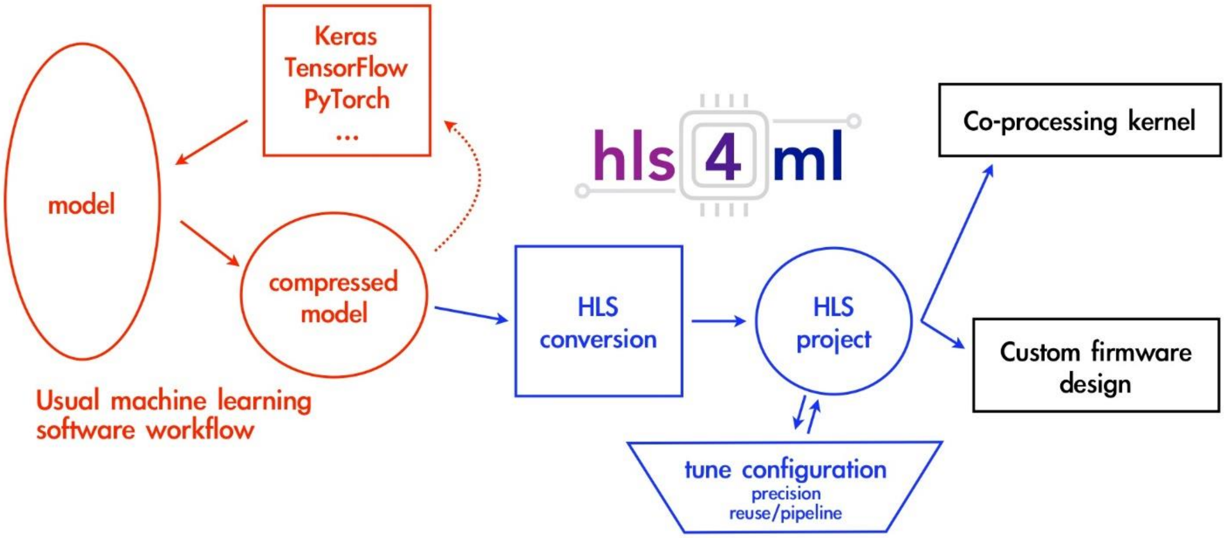
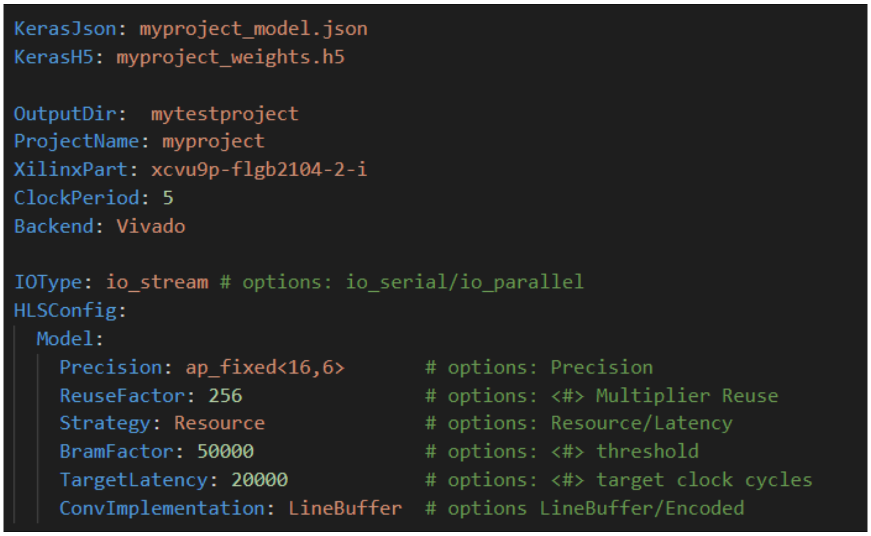


Figure 5. Diagram of the hls4ml workflow. Blocks outlined in orange are part of the usual ML workflow. Blocks in blue are involved in the HLS4ML workflow. Blocks in black are part of the hardware synthesis workflows. [2]

Figure 5 illustrates the role of HLS4ML as an intermediate step bridging the gap between software and hardware workflows. Specifically, it takes a trained machine learning model and outputs a hardware IP block which can then be integrated into more complex designs or into a kernel for CPU co-processing.



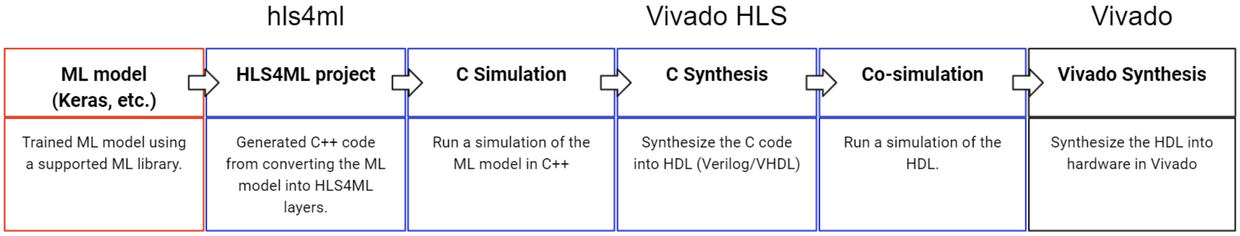
```
KerasJson: myproject_model.json
KerasH5: myproject_weights.h5

OutputDir:  mytestproject
ProjectName: myproject
XilinxPart: xcvu9p-flgb2104-2-i
ClockPeriod: 5
Backend: Vivado

IOType: io_stream # options: io_serial/io_parallel
HLSConfig:
  Model:
    Precision: ap_fixed<16,6>       # options: Precision
    ReuseFactor: 256                # options: <#> Multiplier Reuse
    Strategy: Resource              # options: Resource/Latency
    BramFactor: 50000               # options: <#> threshold
    TargetLatency: 20000            # options: <#> target clock cycles
    ConvImplementation: LineBuffer  # options LineBuffer/Encoded
```

Figure 6. An example YAML configuration file. This configuration generates a project for a pre-trained model called "myproject". It targets a Xilinx Virtex UltraScale+ FPGA using a clock speed of 5 nanoseconds. The HLSConfig category specifies HLS-related configuration parameters.

HLS4ML takes a YAML file, like the one shown in Figure 6, as an input to configure the generation of the project directory. This YAML file specifies parameters such as the model, project name, target Xilinx part, clock speed and backend. It is also used to specify hardware or HLS-related parameters like IO type, precision, strategy, reuse factor, etc. Tuning these parameters are vital for controlling timing and utilization.

3.4 Workflow

Within the HLS4ML workflow, there are several additional steps as shown in Figure 7.



ML model (Keras, etc.): Trained ML model using a supported ML library.

HLS4ML project: Generated C++ code from converting the ML model into HLS4ML layers.

C Simulation: Run a simulation of the ML model in C++

C Synthesis: Synthesize the C code into HDL (Verilog/VHDL)

Co-simulation: Run a simulation of the HDL.

Vivado Synthesis: Synthesize the HDL into hardware in Vivado

Figure 7. Intermediate steps in the HLS4ML workflow. The blue steps are part of the HLS process but are streamlined by using the HLS4ML framework.

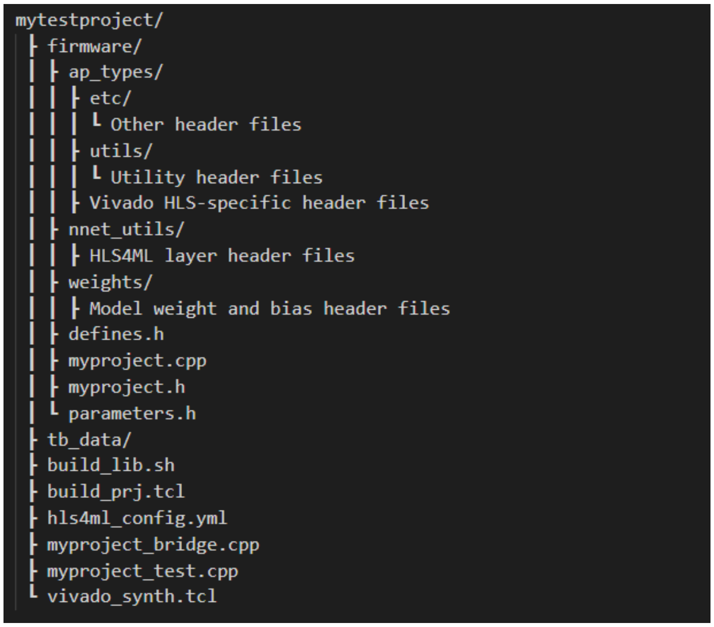
Each step in the HLS4ML workflow is described below:

ML model: Starting from a pre-trained machine learning model from a supported library, HLS4ML reads the model file and generates a project directory that implements the model in C++ using HLS4ML layers. These layers are specially written to leverage user optimization directives. This conversion step can be fine-tuned by configuring parameters in the YAML configuration file.

HLS4ML project: The generated project directory contains all the required files to perform HLS. Figure 8 shows the file hierarchy after converting a model into an HLS4ML project. The firmware directory contains all the C++ source code separated into Vivado HLS data types (ap_types), HLS4ML layers (nnet_utils), model weights (weights) and model top level (myproject). During the ML model to HLS4ML project conversion process, additional optimizations like pruning or quantization can be applied.

C Simulation: During this step, a simulation of the C++ implementation of the machine learning model is run. C simulation can be used to quickly and easily validate the correctness and functionality of the C++ code prior to synthesizing the design. The simulation outputs are also used during the co-simulation step to verify the correctness of the RTL implementation.

C Synthesis: While there are multiple commercially available HLS tools, the HLS4ML workflow primarily supports the Vivado HLS (now Vitis HLS) tool developed by Xilinx [12]. When the C synthesis step is run, the C++ implementation is converted into HDL—both in Verilog and VHDL—and utilization and timing reports are generated. The HLS workflow is the same for both languages, but results may differ due to implementation differences. The utilization estimates are generally an upper limit as the later Vivado synthesis step will further optimize RTL and decrease utilization. The timing reports provide an estimate for the latency and initiation interval in clock cycles.



```
mytestproject/
 ├ firmware/
 │ ├ ap_types/
 │ │ ├ etc/
 │ │ │ └ Other header files
 │ │ ├ utils/
 │ │ │ └ Utility header files
 │ │ ├ Vivado HLS-specific header files
 │ ├ nnet_utils/
 │ │ ├ HLS4ML layer header files
 │ ├ weights/
 │ │ ├ Model weight and bias header files
 │ ├ defines.h
 │ ├ myproject.cpp
 │ ├ myproject.h
 │ └ parameters.h
 ├ tb_data/
 ├ build_lib.sh
 ├ build_prj.tcl
 ├ hls4ml_config.yml
 ├ myproject_bridge.cpp
 ├ myproject_test.cpp
 └ vivado_synth.tcl
```

Figure 8. File hierarchy of generated HLS4ML project directory.

Co-simulation: During co-simulation, the RTL implementation of the model is simulated using user-provided inputs from the tb_data directory or using zeroes as inputs. In pipelined designs, inputs may be requested before previous transactions complete. In this case, the latency is measured as the number of cycles between the first data input and the last data output. The initiation interval (II) is defined as the number of cycles between data ready signals.

Vivado Synthesis: The Vivado HLS tool outputs a set of HDL files that implement the model in RTL. These files can be used to generate a gate-level implementation by using Vivado synthesis, and later, an IP block. During this process, new resource utilization and timing reports are generated. These reports are more accurate estimations of the final values as they represent the actual gates utilized on the FPGA.

4 Convolutional Layer Implementations

In this section, we will explain the design paradigm and structures used in HLS4ML. We present two different approaches to convolutional layer implementations on FPGAs. The two implementations examined in this section—the line buffer and the encoded convolution implementations—are architected by Dr. Philip Harris at MIT and Dr. Vladimir Loncar at CERN, respectively [13] [14].

4.1 Streaming Design Paradigm

The ability for field programmable gate arrays (FPGAs) to perform multiple operations in parallel make them well suited for neural network inference. To exploit the parallel nature of FPGAs, we make use of stream processing wherein computations occur as data is produced or received. Stream processing contrasts batch processing, often used in graphics processing units, in which data is aggregated into batches and then processed all at once.

Under the streaming paradigm, each module, which in our case is a neural network layer, is designed to perform computations on a single pixel. Since layers do not necessarily have the same consumption rate or latency, we are required to buffer values either in an array or in a stream. In this implementation, these buffers are implemented as first in, first out (FIFO) buffers in hardware. FIFO buffers only support sequential access and have additional constraints on the read (pop) and write (push) operations but use fewer resources than array-based buffers [14].

Vivado HLS provides directives called pragmas that are directly inserted in the C++ code and assist the compiler in optimizing the RTL design. In particular, the pragmas *dataflow*, *pipeline,* and *stream* are important for stream processing.

The *dataflow* pragma directs the compiler to pipeline at a function-level and increases concurrency and overall design throughput. Ordinarily, functions are executed sequentially and block the execution of downstream functions. With the *dataflow* pragma, functions can overlap in operation if there are no data dependencies present. Suppose we have a two functions, *A* and *B*. If function *A* computes a value *k*, which is then used in function *B*, then there exists a data dependency between function *A* and *B*. In this case, function B cannot execute until function A has computed and written value *k*.

The *pipeline* pragma applies instruction-level parallelism to reduce latency in loops. Pipelining a loop runs all operations concurrently and enables the loop to accept new inputs every *N* clock cycles, where *N* is called the initiation interval (II).

The *stream* pragma is used to specify a variable as streaming and implements it using a FIFO, instead of random-access memory (RAM). Selection of an appropriate FIFO depth is important. A shallow FIFO buffer may cause issues by deadlocking the pipeline, a state where data cannot flow through the pipeline due to unfulfilled data dependencies. Too deep and hardware resources are wasted.

Under the streaming paradigm, inputs can be processed with low latency since operations can occur concurrently and data is constantly flowing through the system. However, this model also presents significant challenges in scheduling these operations as to not cause dependencies or deadlocking the pipeline. However, HLS tools can help automate the process. Through intelligent usage of directives, in the form of pragmas, users are able further tune the pipelining process.

4.2 Convolutions on FPGAs

A direct implementation of convolution requires six nested loops over an input feature map of size *(H, W, C)* for *N* output feature maps and convolutional kernel of size (*J*, *K*) [1]. We can model the computation for a single pixel of output **Y** as a function of the input **X**, weights **Θ**, and bias **β** [14]. In Equation (3), we compute a single pixel of output **Y**. The input feature map **X** has dimensions $H \times W \times C$, the weight matrix Θ has dimensions $J \times K \times C \times N$ and the bias vector β has dimension N [14].

$$\forall v, u, n \in [1, V] \times [1, U] \times [1, N]$$
$$Y[v, u, n] = \beta[n] + \sum_{c=1}^{C} \sum_{j=1}^{J} \sum_{k=1}^{K} X[v + j, u + k, c]\Theta[j, k, c, n] \tag{3}$$

Implementing the convolution described in Equation (3) would result in high latency due to the additional clock cycle penalty for entering or exiting a loop [14]. This penalty can be address by pipelining the loops via parallelization using the *unroll* directive. However, due to the size of the convolution layer loops, the size of the RTL implementation and resource utilization would become enormous [14].

On other hardware platforms, like software processors (CPUs or GPUs), convolutions are mapped as General Matrix Multiplications (GEMMs). This mapping is often performed using algorithms like *im2col* or *kn2row* [15]. For example, in the *im2col* algorithm, if we consider each possible valid position of the convolutional kernel and each corresponding input window (patches), we can construct a kernel patch matrix by flattening the kernels into row vectors and stacking them and an input patch matrix by flattening each input window into a column vector and appending them as shown in Figure 9. Performing a dense multiplication of the kernel matrix and the input patch matrix yields an output patch matrix which contains all the output channels in row-major order.

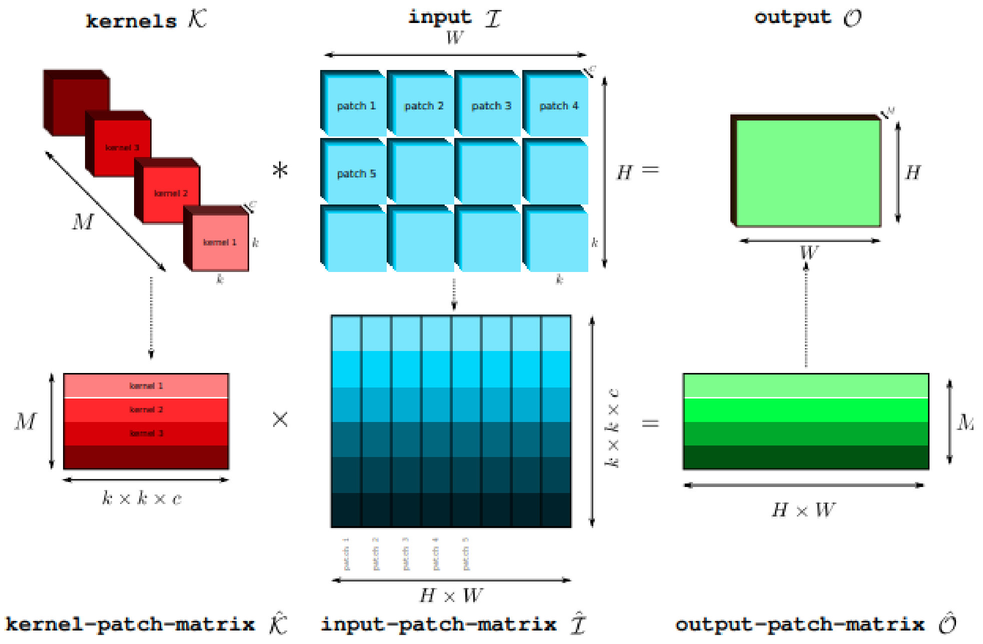


Figure 9. Multiple Channel Multiple Kernel (MCMK) convolution using the im2col algorithm [15].

The approach taken by HLS4ML adapts the GEMMs approach, but instead of batch processing the entire matrix at once, we only compute the output for a single input window or single column vector at a time. This coincides with the streaming processing approach as we can construct the column vectors as each pixel is read and enables usage of existing matrix-vector multiplication routines in HLS4ML.

4.3 Line Buffer

The line buffer implementation is an additional implementation for HLS4ML convolutional layers introduced by this thesis. It uses a chain of shift registers to keep track of previously seen pixels and a sliding window to buffer the kernel elements. Additionally, a pair of counters is used to keep track of the position of the current pixel in the image.

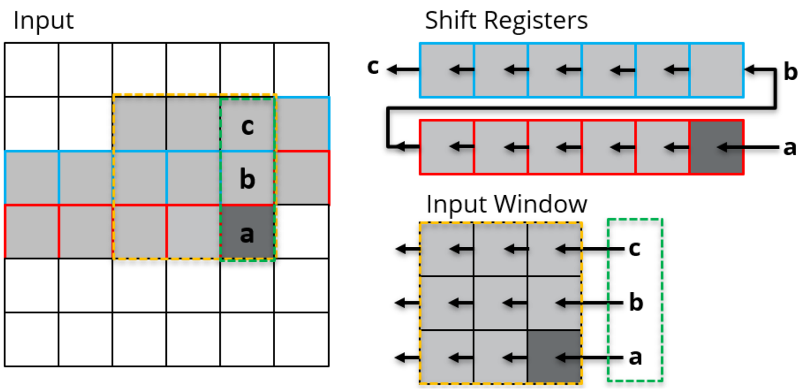


Figure 10. Color coded diagram demonstrating the buffer update process when a new pixel is read. We allocate two shift registers (red and blue) and an input window buffer (yellow). Reading pixel "a" causes pixel "b" and "c" to be popped from the shift register chain. Pixels "a", "b", and "c" are then used to update the input window.

Figure 10 demonstrates the two update operations that the line buffer performs each time a new pixel is read from the input streams. We name the two operations (1) *shift line buffer* and (2) *kernel shift.*

We will assume that the input is a single channel to simplify the discussion, but for a multi-channel input, the operations are simply replicated for each channel independently. We will also assume that the input is two-dimensional. A one-dimensional convolution using the line buffer implementation will perform similarly, but only requires the *kernel shift* operation.

During the *shift line buffer* operation, the new pixel is pushed into the end of the shift register chain. This push operation causes all the contents of the shift register to shift by one element downstream. In the event where a shift register is full, the element at the head of the shift register will be popped. When this occurs, the popped pixel is pushed at the end of the next shift register in the chain. This process propagates through the entire shift register chain.

At the same time, we perform the *kernel shift* operation. This operation updates the contents of the input window. In the previous operation, *shift line* buffer, each time a pixel is read from the input stream a push and pop operation are propagated through the shift register chain. The pixels that are popped from the shift registers are stacked with the input pixel into a column vector and are pushed as the rightmost column of the input window. Any pixels that are popped from the input window are dropped. Both operations are completed in a single clock cycle.

Given a convolution with a kernel size of $J \times K$ and an input image of size $H \times W$, the minimum required size of the shift register is an array of $J - 1$ shift registers of depth W. Each shift register keeps track of a row's worth of pixels. Only $J – 1$ shift register are required as the input window also contains the last K pixels popped from the topmost shift register.

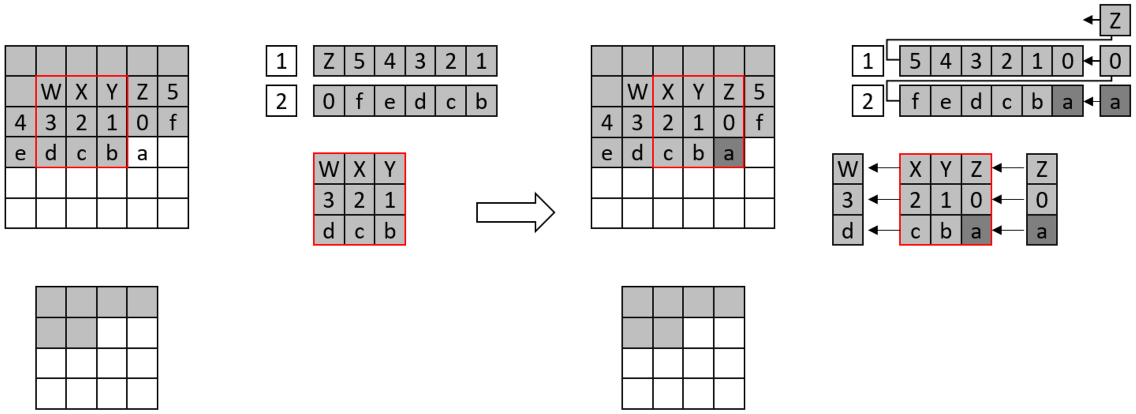


Figure 11. Snapshot of two steps of the line buffer implementation. Previous state (left) and state after reading a pixel from the input stream (right). Input image is 6x6 and the kernel is 3x3. The pixels contained in the input window buffer are outlined in red.

Figure 11 illustrates both operations occurring in tandem. The input image is of size 6x6 and the convolution kernel is of size 3x3. When pixel **a** is read from the input stream, indicated by the transition from the left state to the right state in the figure, we push it into shift register #2. This causes pixel **0** to be popped from shift register #2 and then pushed into shift register #1. And subsequently, pixel **Z** is popped from shift register #1. The input window is updated with a column vector composed of the input pixel **a** and popped pixels **0** and **Z.**

The output pixel can be computed when the pixel position pointers meet certain conditions. These pointers are updated after each iteration of reading the input stream and updating the internal state, regardless of whether an output pixel is computed.

The conditions to compute an output pixel is modeled by the following set of equations:

$$X_{pixel} > J - 1$$

$$Y_{pixel} > K - 1$$

$$mod(X_{pixel} - J - 1, X_{stride}) = 0$$

$$mod(Y_{pixel} - K - 1, Y_{stride}) = 0$$

When these conditions evaluate to true, the output pixel is computed by performing a dense multiplication with the weight vector.

4.4 Encoded Convolution

The encoded convolution is the current implementation for convolution in the HLS4ML framework. It is a direct adaptation of the GEMMs approach discussed in Section 4.2, but only considers a single column of the input matrix, or input window, at a time.

We assume that $J = K$ (square kernel) and that the stride is less than the kernel width and height. In this implementation, we allocate a stream for each element in the kernel, resulting in an array of K^2 streams. The depth of these streams is determined by a function of the output width and the kernel width. Each pixel in the input image is assigned a pre-computed K^2 bit instruction which is used to mask which streams are updated during the update iteration.

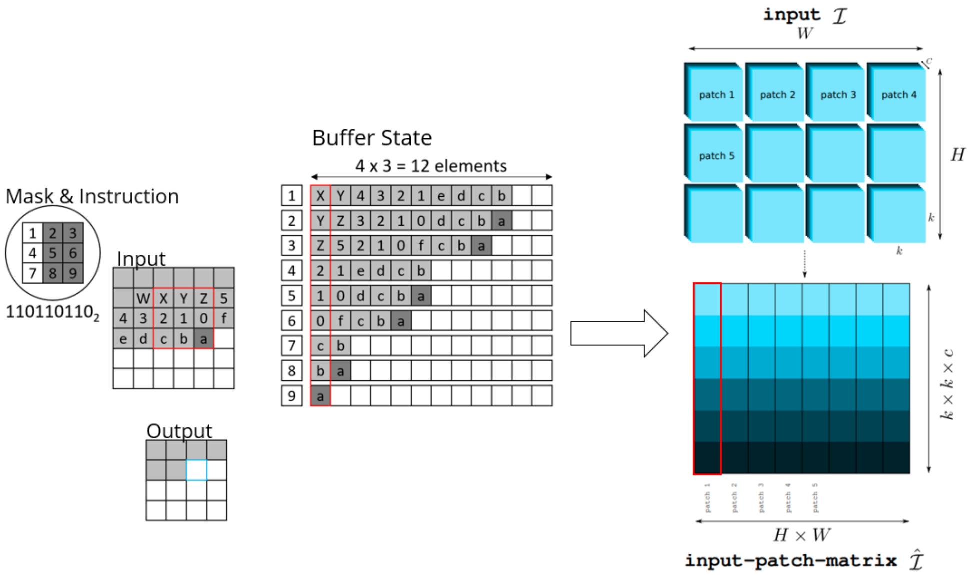


Figure 12. Side-by-side comparison of the encoded buffer and the im2col input patch matrix. Encoded convolution constructs the input patch matrix using mask instructions but processes the matrix one column at a time.

As shown in Figure 12, each column-wise slice of the stream array represents a single input window and a column of the input matrix. The mask instructions construct the input matrix by copying the input pixel to the streams—or matrix rows—to which it contributes.

We compute the mask instruction based on all the possible input windows which contain the pixel. Figure 13 illustrates the mapping from mask indices to the instruction bit order and the computation of an example instruction. In this example, four different input windows contain the highlighted pixel. If we examine just the blue input window, we find that the pixel occupies position 1 in the window. Therefore, in the mask, we set position 1 to have a mask value of 1 to indicate that the highlighted pixel contributes to the stream corresponding to index 1. This process is repeated for each input window that contains the pixel.

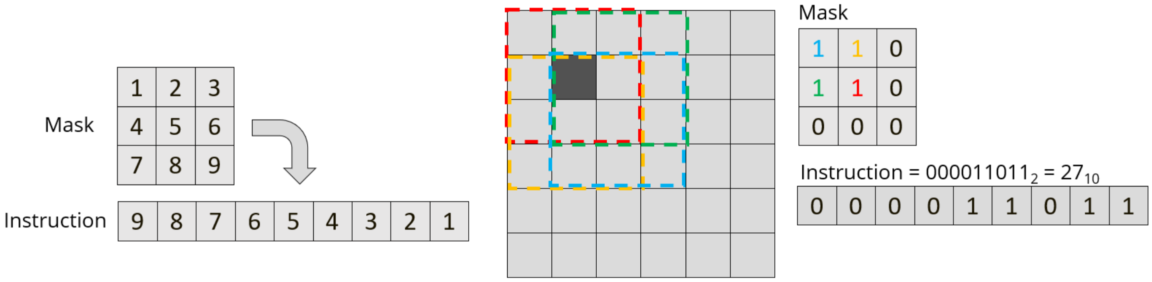
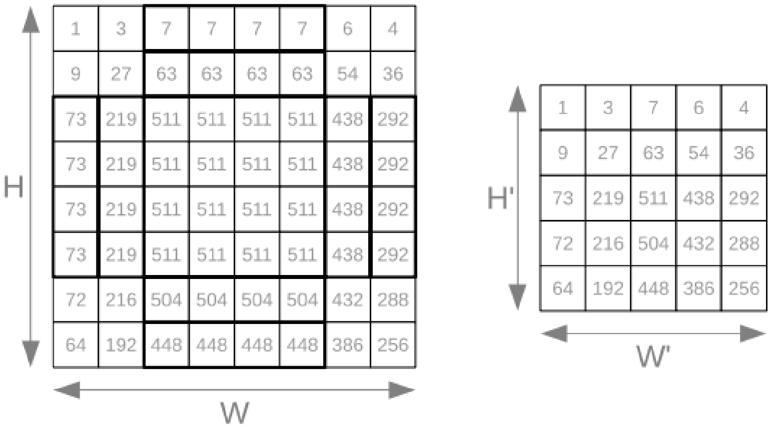


Figure 13. Bit mapping (left) and computation of a mask instruction (right). Colored boundaries indicate input windows that contain the highlighted pixel. The mask for a given pixel position is computed as the superposition of the pixel in each input window.

There are two operations performed every cycle when using the encoded convolution implementation:

1. *compute scaled indices*
2. *compute output*

The *compute scaled indices* operation performs an instruction lookup based on the current pixel position. These instructions are computed during the HLS4ML model conversion process and are saved as part of the layer parameters. Since the image is of size $H \times W$, the instructions array is HW elements long; however, this can be compressed significantly by removing duplicate instructions as shown in Figure 14.



1	3	7	7	7	7	6	4
9	27	63	63	63	63	54	36
73	219	511	511	511	511	438	292
73	219	511	511	511	511	438	292
73	219	511	511	511	511	438	292
73	219	511	511	511	511	438	292
72	216	504	504	504	504	432	288
64	192	448	448	448	448	386	256

1	3	7	6	4
9	27	63	54	36
73	219	511	438	292
72	216	504	432	288
64	192	448	386	256

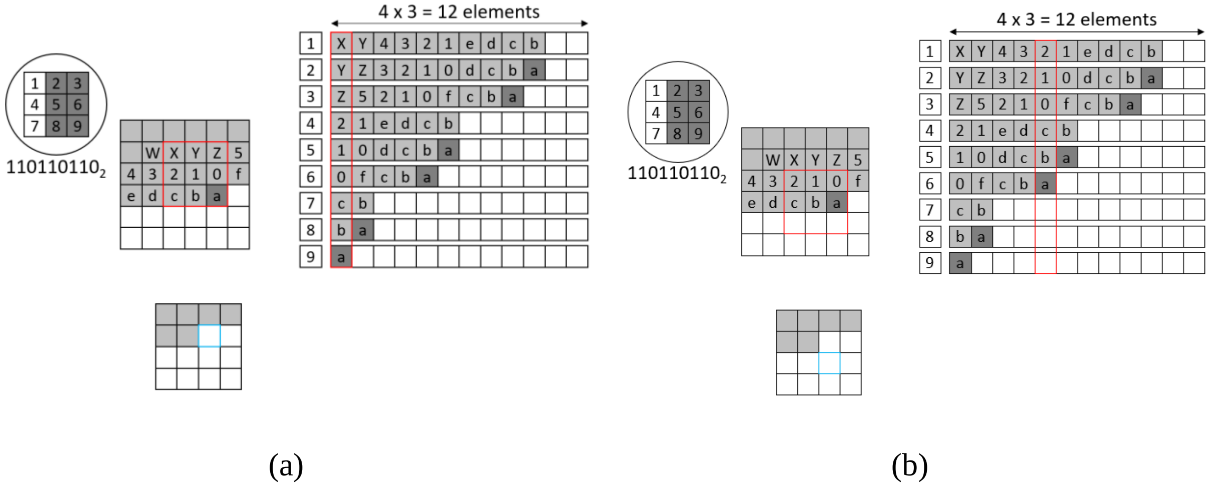
Figure 14. Compression of the instruction array with a 3x3 kernel and unit stride. The left image shows the binary mask instructions by pixel. Bolded regions indicate duplicated instructions. On the right image, we compress the duplicated instructions into a compressed array of size H'×W'. [14]

During the instruction lookup, the pixel width and height indices are remapped to the compressed coordinates and are used to index into the instruction array. The bits of the instruction are organized in row-major order wherein the bit index increases as we move along a given row.

After the instruction is retrieved for the current pixel, it is used to update the layer's internal state during the *compute output* operation. Each bit in the instruction corresponds to a specific stream. In Figure 15, the decimal instruction is 438_{10} which corresponds to a binary value of 110110110_2. It masks streams 1, 4, and 7 and copies the input pixel to streams 2, 3, 5, 6, 8, and 9.

Figure 15 show different slices of the internal buffer and their corresponding input windows overlaid on the input image in red. If we look at a slice in the middle of the buffer as shown in Figure 15b, we can see the incomplete column vector and its corresponding input window scope.

When the last stream index is written, we can compute an output pixel by reading a column of pixels from the stream array and compute the dense multiplication with the weight matrix. This approach does not need to perform complex comparison logic to determine when an output pixel can be computed as it is encoded as part of the precomputed instructions.



(a) (b)

Figure 15. Snapshot of the encoded convolution layer's internal state. Figure 15a shows a fully populated input window slice and Figure 15b shows an incomplete input window slice. Input image is 6x6 and kernel size is 3x3. The current pixel's instruction is shown in the top-left (110110110b = 438d). The right side shows the internal state for each stream. The red outline indicates the input window scope. The blue outlined pixel in the output indicates the current output pixel.

4.5 Differences in Implementation

While both implementations use different structures to buffer data—the line buffer uses shift registers, and the encoded convolution uses streams—they both synthesize to shift registers during gate-level implementation.

The primary difference between the line buffer and encoded implementations is how they buffer all the pixels between the first and last element of the input window and the number of replicated pixels. The line buffer allocates $K - 1$ buffers of depth W for the rows of the image, while the encoded implementation allocates K^2 buffers of depth $K(W - K + 1)$ for the elements in the input window.

However, the line buffer implementation has a clear use case in scenarios where there is a non-square kernel or when the convolutional stride exceeds the size of the kernel. In the more common case, with a square kernel and unit stride, the resource utilization and latency of the implementations are more important.

In the next section, we will further examine the resource utilization and latency of both implementations in this scenario and make a recommendation for each implementation.

5 Performance

In this section, we will discuss the metrics, test setup, and performance results from comparing the line buffer and encoded convolution implementations.

5.1 Metrics

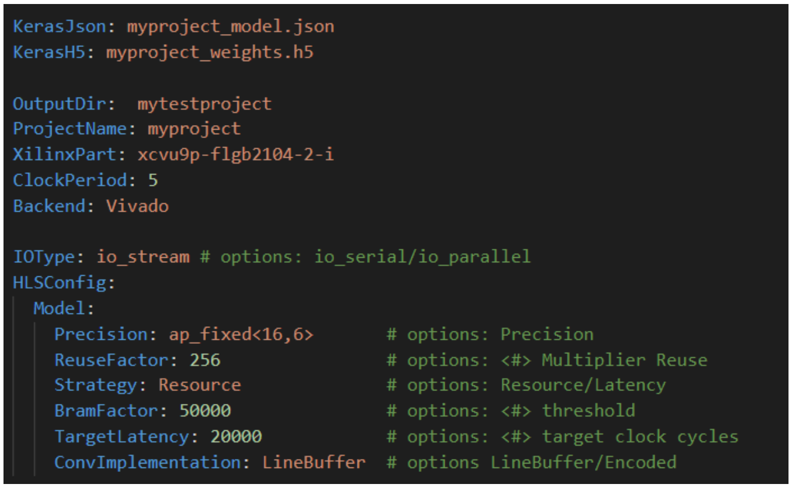
There are two primary metrics by which we compare the performance of the two implementations: (1) resource utilization and (2) latency and initiation interval.

5.1.1 Resource Utilization

Resource utilization refers to how much of the on-chip resources the RTL or gate-level implementation uses. There are four different types of resources that are of particular interest in the utilization reports: flip-flops, lookup tables, digital signal processors, and block RAMs.

- Flip-flops (FFs) are used to save a logical state between clock cycles. On a triggering clock edge, the flip-flop will take a snapshot of the logical value on its input and maintain it on the output until the next triggering clock edge.
- Lookup tables (LUTs) implement combinational logic as a truth table. They come in several variants depending on the number of inputs.
- Digital signal processors (DSPs) are used to perform arithmetic-intensive operations. A single DSP unit can perform a multiply-accumulate (MAC) operation in one clock cycle.
- Block RAMs (BRAMs) are synchronous memory primitives that are used to store and fetch data. They have limited read and write ports that can be used to access the memory every clock cycle.

In addition to the device primitives, there are HLS4ML parameters that affect the resource utilization: strategy, quantization and reuse factor (RF). An example configuration file is shown in Figure 16.



```
KerasJson: myproject_model.json
KerasH5: myproject_weights.h5

OutputDir:  mytestproject
ProjectName: myproject
XilinxPart: xcvu9p-flgb2104-2-i
ClockPeriod: 5
Backend: Vivado

IOType: io_stream # options: io_serial/io_parallel
HLSConfig:
  Model:
    Precision: ap_fixed<16,6>       # options: Precision
    ReuseFactor: 256                # options: <#> Multiplier Reuse
    Strategy: Resource              # options: Resource/Latency
    BramFactor: 50000               # options: <#> threshold
    TargetLatency: 20000            # options: <#> target clock cycles
    ConvImplementation: LineBuffer  # options LineBuffer/Encoded
```

Figure 16. An example YAML configuration file. This configuration will be optimized for resources, use a fixed point <16,6> data format and use a reuse factor of 256.

HLS4ML supports two different strategy configurations: resource and latency. The resource strategy is targeted towards larger models and optimizes for resource utilization by reusing existing hardware to complete operations in multiple stages. In contrast, the latency strategy optimizes for minimum latency by further unrolling (parallelizing) operations. However, this option is limited to small models as the unrolling process greatly increases resource utilization and can quickly reach the Vivado HLS partitioning limit. Changing the strategy also changes the underlying implementation of some layers.

Quantization refers to the precision of the data type used for the inputs, weights and biases. In this study, we use fixed-point precision <16, 6> which is 16 bits wide and has 6 integer bits. Increasing precision (using more bits) can increase model performance at the cost of greater resource utilization.

Reuse factor specifies how many times each DSP primitive is reused for dense multiplication operation. A reuse factor of 1 fully parallelizes the multiplication operation. However, for large models, low reuse factor is not feasible as the number of DSP units on the FPGA are limited. The DSP utilization is inversely proportional to the reuse factor; a reuse factor of 4 would use a quarter of the DSPs as a reuse factor of 1.

5.1.2 Latency and Initiation Interval

Latency is defined as the number of clock cycles that it takes an operation to resolve. This is measured as the time from when the first data element is fed into the FPGA to when the last result element is retrieved from the FPGA.

Initiation interval (II) describes the number of clock cycles before a new set of inputs can be accepted. It is also the inverse of the throughput—one output per II clock cycles. The initiation interval is not necessarily the same as the latency due to function and loop pipelining and is often lower than the latency.

5.2 Comparison Model

We use the CNN model architecture described by Table 1 to benchmark the performance of both convolution implementations. In this study, we opted to use a dummy CNN architecture instead of existing CNNs, as our objective is to study the resource utilization of the convolutional layers. The size of the models, in terms of weights, is scaled by increasing the number of output feature maps **N**. The first Conv2D layer uses a 5×5 kernel and subsequent Conv2D layers use a 3×3 kernel. All pooling layers use a pooling window of size 2×2. Lastly, we have a single fully-connected layer with 8 output neurons.

Table 1. Benchmark model architecture. Models are scaled by increasing the number of output feature maps N. Each Conv2D layer uses unit stride and is grouped with BatchNorm and Leaky ReLU activation layers. Each convolution uses "same" padding which adds additional zero padding to maintain the input size.

Layer	**Output Dimensions**	**Parameters**
Input	(16, 16, 3)	0
Conv2D, 5x5	(16, 16, N)	80N
Pooling2D, 2x2	(8, 8, N)	0
Conv2D, 3x3	(8, 8, 2N)	$18N^2 + 10N$
Conv2D, 3x3	(8, 8, 2N)	$36N^2 + 10N$
Pooling2D, 2x2	(4, 4, 2N)	0
Flatten	32N	0
Dense	8	256N + 8
	Total Weights	$54N^2 + 356N + 8$

We examine four different configurations of the dummy model: N = 4, N = 8, N = 32, and N = 64. These models range from a total of 2,256 learnable parameters to 243,336 learnable parameters. Models are named based on the number of output feature maps **N** of the first Conv2D layer and will be denoted as model<**N**>.

For latency optimized convolution, we examine metrics for model4 and model8. Using latency optimized convolution for more than N = 8 with our CNN architecture exceeds the Vivado HLS partitioning limit of 4,096 unrolled loop iterations.

For resource optimized convolution, we examine metrics for model8, model16, and model64.

5.3 Testing Configuration

For both implementations, the reuse factor is chosen to best match the optimizations strategy.

- For the resource-optimized strategy, we choose an initiation interval target of 10,000 cycles.
- For latency-optimized strategy, we use the lowest reuse factor possible for the bottleneck (slowest) layer and adjust the remaining reuse factors to be less than or equal to the bottleneck layer's initiation interval.

All studies are performed using the *io_stream* IOType parameter and *ap_fixed<16, 6>* data types. We target a Xilinx Virtex Ultrascale+ VU9P FPGA module (part xcvu9p-flgb2104-2-i) with a clock speed of 5 nanoseconds. We use Vivado HLS 2019.2 as the backend for HLS.

For the latency and initiation interval measurements, we examine both the RTL latency estimation after C synthesis and the co-simulation results for an all-zero input. The C synthesis estimates are based on the absolute longest path through the design. The co-simulation timing metrics are actual measured values based on the simulation data set. We also measure the average transaction time by averaging the time between successive transactions.

For resource utilization, we look at only the post-Vivado synthesis utilization reports. During Vivado synthesis, the RTL design is transformed into a gate-level representation, so post-synthesis utilization estimates are more accurate than those at earlier stages.

5.4 Results

5.4.1 Resource-Optimized

In this section, we compare the performance of both implementations on three different models: model8, model32 and model64. These three models have 6,232, 66,376, and 243,336 parameters, respectively.

Table 2. Resource-optimized timing metrics. Latency and initiation interval are measured in clock cycles. Bolded values are the lower of the pair. Wall Clock latency is measured using a clock period of 5 ns.

	Model8		**Model32**		**Model64**	
Timing Metrics	**Line Buffer**	**Encoded**	**Line Buffer**	**Encoded**	**Line Buffer**	**Encoded**
Clock Period (ns)	4.299	4.299	4.366	4.366	4.374	4.374
Clock Uncertainty	0.62	0.62	0.62	0.62	0.62	0.62
Avg Transaction Time	**9,545**	9,609	**9,545**	9,609	**9,545**	9,609
Cosim Latency	**23,030**	23,290	**23,030**	23,290	**23,183**	23,333
Cosim II	**13,487**	13,683	**13,487**	13,683	**13,640**	13,726
Csynth Latency	**15,627**	15,730	**15,564**	15,667	**15,656**	15,759
Csynth II	**14,801**	14,901	**14,801**	14,901	**14,801**	14,901
Wall Clock Latency (µs)	**115.15**	116.45	**115.15**	116.45	**115.91**	116.66

In terms of latency, both implementations performed similarly with resource-optimized strategy. The difference in clock cycles is less than 1% of each other.

Table 3. Resource-optimized utilization. Bolded values indicate the lower of the pair. SRLs are Shift Register LUTs. Bolded values are the lower of the pair.

	Model8		**Model32**		**Model64**	
Resources	**Line Buffer**	**Encoded**	**Line Buffer**	**Encoded**	**Line Buffer**	**Encoded**
LUT	**20,749**	30,082	**85,663**	110,625	**189,609**	261,718
FF	**30,230**	34,495	**95,669**	101,200	**229,161**	244,617
DSP	91	91	649	649	2,065	2,065
BRAM	**59**	134	**288**	363	**754**	829
SRL	**3,904**	7552	**15,040**	30,208	**41,483**	74,059
Clock Period	**3.431**	3.516	**3.692**	4.698	**3.513**	4.627

However, if we look at the resource utilization, the difference between the two implementations becomes more pronounced. In terms of FF and LUTs, we observe an 30% reduction in LUTs and between 5% to 12% reduction in FFs. Since both implementations use the same reuse factors, there is no difference in DSP utilization. Across all three models, there is a difference in utilization of 75 BRAMs. Lastly, in terms of shift register LUTs (SRLs), we observe a 45% to 50% reduction in utilization.

5.4.2 Latency-Optimized

In this section, we compare the performance of both implementations on two different models: model4 and model8. These two models have 2,256 and 6,232 weights, respectively.

Since latency strategy fully parallelizes the convolutional layers, we can only examine the small models as the larger models—like model32 or model64—will not fit within the FPGA's limited resources. Table 4 compares the timing metrics and

Table 5 compares the resource utilization.

Table 4. Latency-optimized timing metrics. Latency and initiation interval are measured in clock cycles. Bolded values are the lower of the pair. Wall Clock latency is measured using a clock period of 5 ns.

	Model4		**Model8**	
Timing Metrics	**Line Buffer**	**Encoded**	**Line Buffer**	**Encoded**
Clock Period (ns)	4.299	4.299	**4.292**	4.366
Clock Uncertainty	0.62	0.62	0.62	0.62
Avg Transaction Time	**1,204**	1,460	**1,204**	1,460
Cosim Latency	**2,694**	3,086	**2,657**	3,102
Cosim II	**1,492**	1,628	**1,455**	1,644
Csynth Latency	**1,746**	1,748	**1,706**	1,708
Csynth II	1,204	1,204	1,204	1,204
Wall Clock Latency (µs)	**13.47**	15.43	**13.28**	15.51

While the C synthesis latency and initiation interval estimates are the same for both models, we observe a significant reduction of approximately 12% in co-simulation latency and initiation interval. The average transaction time, as measured by the co-simulation timestamps, shows a reduction of 256 clock cycles or 18%.

Table 5. Latency-optimized utilization. SRLs are Shift Register LUTs. Bolded values are the lower of the pair.

	Model4		**Model8**	
Resources	**Line Buffer**	**Encoded**	**Line Buffer**	**Encoded**
LUT	**51,912**	58,287	**142,113**	152,616
FF	**31,771**	33,188	**67,801**	73,456
DSP	195	195	522	522
BRAM	**26**	101	**46**	121
SRL	**2,048**	3,776	**3,904**	7,552
Clock Period	**6.605**	6.791	**5.897**	5.940

In terms of resources, we observe a similar reduction in LUT and FF utilization, but to a lesser degree—about 10%—compared to the resource-optimized approach. Like the resource-optimized strategy, the difference of 75 BRAMs is present as well. The longest path, as denoted by the clock period, exceeds the target of 5 nanoseconds, and will require a slower clock speed to meet timing.

5.5 Super Wide Input

For images with a significantly larger width than kernel, the encoded convolution approach uses K^2 more buffer elements to keep track of previously read pixels.

In these results, we examine the performance of both convolutional implementations on model with a super wide input. The model examined here is different from the previous test CNN models and is summarized in Table 6. For this model, we focus primarily on the resource utilization, particularly BRAM and SRLs, between the two implementations.

Table 6. Super wide CNN architecture.

Layer	**Output Dimensions**	**Reuse Factor**	**Parameters**
Input	(1024, 32, 3)	-	-
Conv2D, 3x3	(1024, 32, 32)	15	2,560
Conv2D, 3x3	(1024, 32, 128)	12	37,504
Pooling2D, 4x2	(256, 16, 128)	-	-
Pooling2D, 4x2	(64, 8, 128)	-	-
Pooling2D, 4x2	(16, 4, 128)	-	-
Pooling2D, 4x2	(4, 2, 128)	-	-
Flatten	1024	-	0
Dense	8	8192	8,200
		Total Weights	48,264

We choose an architecture quickly increases the number of parameters of the convolutional layer and then reduces the spatial dimensions to avoid the Vivado HLS partitioning limit of 4096 elements.

Table 7. Resource-optimized resource utilization for super wide CNN model.

	Super wide CNN	
Resources	**Line Buffer**	**Encoded**
LUT	**488,488**	628,229
FF	257,830	**231,438**
DSP	3,394	3,394
BRAM	**10,940**	10,941
SRL	**31,936**	88,064
Clock Period	**3.681**	5.113

The total resource utilization after hardware synthesis is described by Table 7. We observe a 23% reduction in LUTs, but a 11% increase in FF utilization. Overall, combined FF and LUT utilization decreased by 13%. The SRL usage decreases by 64% when using the line buffer implementation compared to the encoded implementation.

We attribute the difference in FF utilization to the implementation of the Pooling2D layers—which use the same computing algorithm as the convolutional layers. The encoded Pooling2D use fewer FFs and more LUTs than line buffer Pooling2D. Since the pooling layers are more numerous in our example architecture, their contribution has a greater effect on the overall utilization.

5.6 Discussion

The line buffer implementation shows a significant reduction in overall LUT and FF utilization compared to the current encoded convolution implementation. It also halves the shift register LUT (SRL) utilization.

This difference in SRLs can be attributed to the difference in approach of both implementations. For each input pixel, the line buffer approach replicates the pixel at most twice: once for pushing into the shift register and another for updating the input window. On the other hand, the encoded convolution approach replicates pixels up to K^2 times—or the number of elements in the kernel—which requires additional primitives for buffering. As we increase the size of the input, we observe that the difference in allocated SRLs grows.

The difference in BRAMs—75 BRAM primitives—occurs in the implementation of the first Conv2D layer. Instead of using SRL primitives to implement the streams used to buffer pixels, Vivado implements each stream using a BRAM instance. Since the first Conv2D layer uses a 5 × 5 kernel for 3 input feature maps, it generates a total of 75 streams to record the internal state.

In terms of wall time, the line buffer implementation has a wall clock latency of 115 microseconds when using resource optimizations and a wall clock latency of 13 microseconds when using latency optimizations. These latency values can be further reduced by streaming multiple images instead of a single image at a time.

Given the significant resource and potential latency savings, we recommend using the line buffer implementation for convolution operations.

6 Integration into HLS4ML

In Section 3, we discussed HLS4ML's role in streamlining firmware implementations of machine learning algorithms. In this section, we will discuss how we generate the HLS4ML projects from trained ML models and how the line buffer convolution implementation is integrated.

6.1 HLS4ML modules

The HLS4ML framework is divided into several modules which each perform a specific step of the model conversion. These modules are described below:

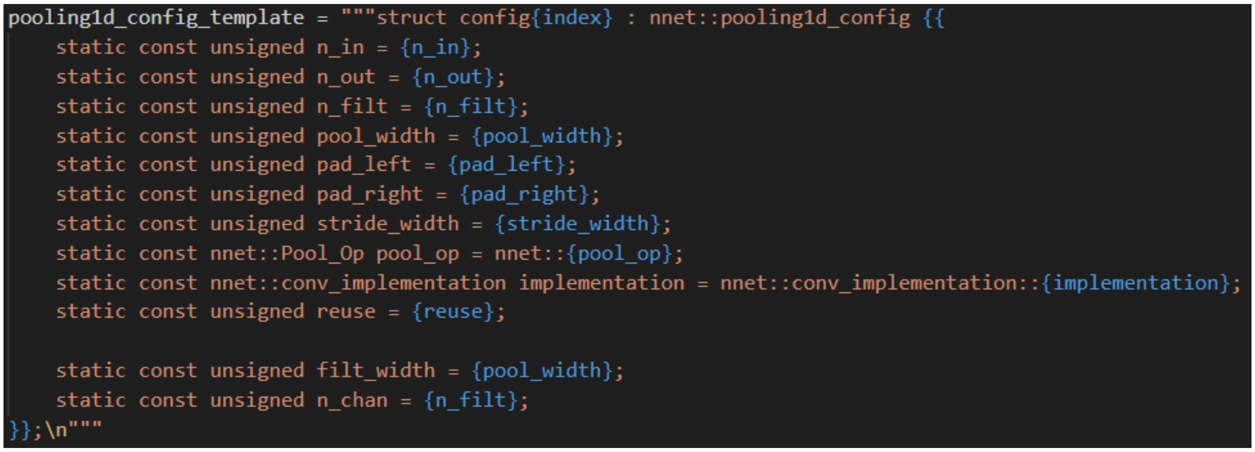
Converters: The *converters* module contains a collection of Python modules that perform the conversion of the trained ML model to a HLS4ML model dictionary. Each supported ML library (Keras, Tensorflow, ONNX, PyTorch) have separate files dedicated to converting the library's model.

Model: The *model* module defines the HLS4ML configuration, model, and layer definitions. The configuration class is used to read the YAML configuration and store the parameters internally. Parameters, like reuse factor or strategy, are stored in an internal attribute dictionary that is attached to the model object. The model class builds and optimizes the layer graph and defines Python API routines.

Each supported layer is implemented as a separate Python class. A layer class is composed of three functions: initialization, C++ configuration and template configuration. The initialization function assigns the layer-specific parameters such as shape, weights, bias, or reuse factor. The C++ and template configurations define how the layer appears in the generated C++ project and which parameters are defined in its C++ parameters, respectively.

Templates: The *templates* module contains the configuration and function templates, as well as all the HLS4ML project files. These templates implement C++ structs with missing fields for each layer as shown in Figure 17. During the writing process, these fields are populated with layer-specific parameters and written to the HLS4ML project. This module also contains the library of header files for the HLS layer C++ implementations.

Writer: The *writer* module generates the C++ project from the HLS4ML model object. It is composed of functions for writing various components of the C++ project such as the top-level project, testbench, weights and biases and parameters. The *writer* module operates in two stages. First, it copies the project skeleton from the *template* module into a new project directory. Second, it builds the C++ model layer-by-layer using the predefined layer templates and filling in missing values with layer parameters.



```
pooling1d_config_template = """struct config{index} : nnet::pooling1d_config {{
    static const unsigned n_in = {n_in};
    static const unsigned n_out = {n_out};
    static const unsigned n_filt = {n_filt};
    static const unsigned pool_width = {pool_width};
    static const unsigned pad_left = {pad_left};
    static const unsigned pad_right = {pad_right};
    static const unsigned stride_width = {stride_width};
    static const nnet::Pool_Op pool_op = nnet::{pool_op};
    static const nnet::conv_implementation implementation = nnet::conv_implementation::{implementation};
    static const unsigned reuse = {reuse};

    static const unsigned filt_width = {pool_width};
    static const unsigned n_chan = {n_filt};
}};\n"""
```

Figure 17. Pooling1D configuration template. The blue fields are populated with layer-specific parameters during the writing step.

Report: The *report* module implements utility functions for fetching, parsing, and printing HLS-generated report files. These report files can be found in the Vivado HLS project directory at a fixed path but can be more easily accessed by calling one of the API functions from this module.

6.2 Convolution Integration

Although the line buffer implementation seems to be the best implementation strategy across the board for the limited cases examined in this thesis, we chose not to completely replace the other approach. Instead, we added the line buffer as an alternative.

To support both implementations of convolution, we add the *ConvImplementation* configuration parameter. This parameter, which is defined in the YAML configuration, supports the *linebuffer* or *encoded* values and is used to select at HLS4ML compile time which implementation is used. Internally, the *ConvImplementation* is written to the project's parameters header file as a C++ enum class. Several modifications to the HLS4ML modules are required to implement this new parameter.

hls_model.py: The YAML configuration is stored as a Python dictionary object during the converter step. However, HLS4ML only exposes parameters that are part of the HLSConfig class to the layer definitions. On initialization, the HLSConfig object initializes its fields and parses the YAML config to retrieve the parameters. We add three new fields to HLSConfig for the *ConvImplementation* parameter that define its value on model, layer type, or individual layer level. Additionally, we implement a "get" access function which returns the parameter value starting from the bottom of the hierarchy. These steps allow the YAML configuration parameters to be accessible within the HLS4ML framework.

hls_layer.py and vivado_templates.py: Next, we define how the parameters are used. The *ConvImplementation* parameter affects the convolution, pooling, and separable convolution layers. For each of these layers, we add an *implementation* parameter to the template like that in

Figure 17. Similarly, in the layer definition, we use the accessor function to retrieve and set the *implementation* value from the HLS configuration.

nnet_conv_stream.h: In the convolution stream header file, we define an enum class for the *implementation* parameter that has one of two values: *linebuffer* or *encoded*. For each layer's header file, stored in the *templates* module, we add a wrapper function which calls one of the implementation top-level function based on the value of the layer's *implementation* parameter.

6.3 Optimization Functions

To achieve the best utilization results for both implementations, we make optimizations to the reuse factor for each model in the benchmark model. This optimization is implemented by another configuration parameter, *TargetCycles*, which is used at model conversion to compute the reuse factor based on the layer's dimensions. The *TargetCycles* parameter is discussed in further detail in Appendix B.

HLS4ML also implements optimizations as functions in the *optimizers* submodule within the *model* module. These optimization functions are called on every node—or layer—in the model graph and must implement two functions: match and transform. The match function returns a Boolean value that determines whether the current node should be optimized using the current optimization function. The transform function implements the optimization routine and is called if the match function returns true.

An example optimization function is the **bn_fuse.py** function. This function performs the batch normalization fuse operation which simplifies the batch normalization formula from Equation (2) when a batch normalization layer is paired with a convolutional layer. The batch normalization operation first normalizes the input and scales and shifts based on learned parameters γ and β. Since the normalization mean and standard deviation are fixed during inference, we can reduce the operations to a single bias vector.

$$y_i \longleftarrow \gamma\hat{x}_i + \beta = \gamma\frac{x_i - \mu_{Bk}}{\sqrt{\sigma_{Bk}^2 + \epsilon}} + \beta = \frac{\gamma}{\sqrt{\sigma_{Bk}^2 + \epsilon}}x_i - \gamma\frac{\mu_{Bk}}{\sqrt{\sigma_{Bk}^2 + \epsilon}} + \beta \qquad (4)$$

The batch normalization can be reformulated into a single scalar and single bias vector as shown in Equation (4). The single scalar vector can be combined with the convolutional layer's weights and so we require only a single bias vector to be stored. This reduces both the number of layers (from two to one) and the amount of storage, typically BRAMs, required to perform the batch normalization operation.

To use an optimization function, it must also be imported and registered in the initialization file for the *optimizer* submodule. This will add it to the list of optimization functions that are called on the model graph during the optimization routine.

7 Conclusion

In this thesis, we examine an alternate line buffer implementation for convolution layers in HLS4ML and compare its performance against the existing encoded implementation. We establish the necessity for efficient convolutions on FPGAs and introduce the HLS tools to achieve it. Using an example, scalable CNN architecture, we study the resource utilization and latency metrics of both implementations and show that, post Vivado synthesis, there is a 30% reduction in LUTs, 10% reduction in FFs and nearly 50% reduction in shift register LUTs when resource optimized. We also show a 12% decrease in latency and initiation interval using the line buffer implementation with latency optimizations. For single image streaming, we achieve a latency of 113 microseconds for resource optimized and 13 microseconds for latency optimized designs. Further, we describe the process of integrating a convolution implementation via a configuration parameter into the HLS4ML framework.

8 Next Steps

Reduction in resource utilization enables larger models to be implemented using the limited on-board FPGA resources. Alternatively, models can be duplicated across FPGA die slices (or Super Logic Regions) if their utilization fits within the die's resources. Each Super Logic Region (SLR) can operate independently and contain all the components necessary to run. Fitting models onto a single SLR best leverages spatial locality and the internal high-speed interconnects. This allows for higher overall processing throughput as multiple models can be run in parallel and independently.

Resource surpluses can be used to increase parallelization by decreasing the reuse factor. This allows us to complete convolution operations in fewer clock cycles. However, the number of DSP units is limited by the number of multiplication operations in the layer. For large models, this is not an issue since the number of multiplications far exceeds the number of available DSP units. For small models, or more latency sensitive models, we might be interested in running faster than a fully parallelized implementation.

One idea to further accelerate small models is to process more than one pixel at once. This would require that multiple input windows be processed simultaneously. We can adapt the line buffer to read and process multiple pixels by increasing the width of the input window buffer to hold multiple input windows.

For example, if we consider two pixels at a time and a 3×3 kernel, we will need an input window buffer of size 3×4 pixels. For each pixel, we update the shift register chain and the input window buffer with new elements. Then, we need to extract each individual input window from the buffer and compute the dense multiplication result.

With this strategy, we will spend a couple more clock cycles to prepare data but will spend the same number of cycles to compute the result since all dense multiplies can occur concurrently.

This will consume additional resources for each instance of a dense multiplier but can reduce the overall inference latency.

9 Beyond the Line Buffer

We present four areas for future optimization beyond the line buffer convolution: reuse factor, single stream processing, per-channel and per-layer quantization, and sparse convolution.

While the *TargetCycles* parameter attempts to balance the initiation interval for all layers in a model, there are cases where the measured initiation interval in co-simulation is significantly different from the target. The exact formulation for the reuse factor when specifying *TargetCycles* is discussed in Appendix B and is based on empirical patterns observed in a few models. Further studies on a wider variety of models are necessary to better balance and more accurately achieve the targeted initiation interval in co-simulation and hardware implementations.

Currently, the buffers between layers are implemented in hardware as an array of streams. However, this approach can use a significant amount of BRAM primitives as each individual stream is implemented as at least one BRAM. For CNN models with many output feature maps, BRAM utilization can drastically increase. One proposed approach to reduce the number of BRAMs is to use a single stream between layers by also packing channel data. The single stream takes clock cycles equal to the number of input feature maps to fully read the input. In cases where the reuse factor is much greater than the number of input feature maps, the increase in latency is not as significant.

Quantization is an area of active research and, typically, refers to the reduction of precision used for model weights or activations. The reduction in precision often leads to poorer model performance, but also can lead to significant resource savings as fewer resources are necessary to store data. Research using quantization at different granularities, specifically on a per-layer or per-channel basis for convolution, has shown to reduce model size without significant performance loss and achieve significant speedups on CPU and DSP platforms [16].

Lastly, sparse convolutions are another area of active research for applications where convolutions operate on sparse inputs or pruned weights. Since most elements are zero, compute cycles are wasted computing trivial convolutions and performance suffers from extraneous memory accesses. Previous research has proposed architectures for improving the spatial locality, and input and output reuse for sparse operations [17] [18]. Implementing these architectures for sparse matrix multiplications has shown to result in lower latency and power savings compared to the traditional *im2col* approach.

10 References

[1] M. P. J. S. F. B. Kamel Abdelouahab, "Accelerating CNN inference on FPGAs: A Survey," *arXiv:1806.01683,* 2018.

[2] "Welcome to hls4ml's documentation," [Online]. Available: https://fastmachinelearning.org/hls4ml/. [Accessed 13 May 2021].

[3] Wikipedia, "Neural Network," en.wikipedia.org, [Online]. Available: https://en.wikipedia.org/wiki/Neural_network. [Accessed 20 May 2021].

[4] H. E. Geoffrey, "Connectionist Learning Procedures," *Artificial Intelligence,* vol. 40.1, pp. 185-234, 1989.

[5] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE,* vol. 86, no. 11, pp. 2278-2324, 1998.

[6] A. Krizhevsky, I. Sutskever and G. E. Hinton, "ImageNet Classification with Deep Convolutional," *NIPS'12: Proceedings of the 25th International Conference on Neural Information Processing Systems,* vol. 1, pp. 1097-1105, 2012.

[7] K. He, X. Zhang, S. Ren and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," *arXiv:1502.01852,* 2015.

[8] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," *ICML'15: Proceedings of the 32nd International Conference on International Conference on Machine Learning,* vol. 37, pp. 448 - 456, 2015.

[9] S. Santurkar, D. Tsipras, A. Ilyas and A. Madry, "How Does Batch Normalization Help Optimization?," *arXiv:1805.11604* , 2018.

[10] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Gee Hock Ong, Y. Tat Liew, K. Srivatsan, D. Moss, S. Subhaschandra and G. Boudoukh, "Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?," *FPGA '17: Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays,* pp. 5 - 14, 2017.

[11] Wikipedia, "High-level Synthesis," en.wikipedia.org, [Online]. Available: https://en.wikipedia.org/wiki/High-level_synthesis. [Accessed 19 May 2021].

[12] Xilinx, "Vitis High-Level Synthesis," [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html. [Accessed 19 May 2021].

[13] P. Harris, D. Rankin and K. Lin, "PCH_CNNAlveo_15_10," 16 October 2020. [Online]. Available: https://indico.cern.ch/event/966337/contributions/4069922/attachments/2124874/3577283/PCH_CNNAlveo_15_10.pdf.

[14] T. Aarrestad, V. Loncar, N. Ghielmetti and et al., *Fast convolutional neural networks on FPGAs with hls4ml,* arXiv:2101.05108, 2021.

[15] A. Vasudevan, A. Anderson and D. Gregg, "Parallel Multi Channel convolution using General Matrix Multiplication," *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP),* pp. 19-24, 2017.

[16] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv:1806.08342,* 2018.

[17] I. Fedorov, R. P. Adams, M. Mattina and P. N. Whatmough, "SpArSe: Sparse Architecture Search for CNNs on Resource-Constrained Microcontrollers," *Thirty-third Conference on Neural Information Processing Systems* , 2019.

[18] Z. Zhang, H. Wang, S. Han and W. J. Dally, "SpArch: Efficient Architecture for Sparse Matrix Multiplication," *arXiv:2002.08947,* 2020.

11 Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 1934360.

First and foremost, I would like to thank both of my advisors, Scott Hauck and Shih-Chieh Hsu for their constant support and guidance. Scott's insights have been an incredible help in completing this project. Shih-Chieh played a pivotal role in helping me progress forward and has been a source of great motivation. Without either of them, this work would not have been possible.

I am grateful to the HLS4ML team. Their collective knowledge and expertise have been a tremendous boon to the development of this work.

Thank you to Philip Harris for fielding all my HLS questions and for assisting me in better developing the CNN implementation that is the contribution of this work. Many thanks to Vladimir Loncar for familiarizing me with the HLS4ML and advising me on the best approach to integrate my work into the greater HLS4ML framework. I feel very fortunate to have had the opportunity to work with them both.

I would like to thank my parents, Shyh-Chung Lin and Wen-Ling Lin, and my sister, Sharon Lin, who have always been cheering me on and encouraging me to be the best I can be. They all have been a great inspiration for me in pursuing this degree.

Thank you to the rest of my friends and family who have supported me both directly and indirectly. Collectively, their support has undoubtedly helped me weather through the unusual circumstances of this past year.

Appendices

Appendix A : Relevant Code

- Master branch of HLS4ML: https://github.com/Keb-L/hls4ml
- Convolutional Neural Network branch: https://github.com/Keb-L/hls4ml/tree/cnn_merge
- Pull Request #332 contribution: https://github.com/fastmachinelearning/hls4ml/pull/332

Appendix B : Optimal Reuse Factor

Reuse factor is one of the most important tunable parameters in the HLS4ML configuration. It serves as a knob by which users can change the level of parallelization in the RTL implementation. However, it requires significant user involvement through manual tuning, after project generation, to achieve the best results. The reuse factor is defined as a global value that is applied equally to all layers, but not all layers execute with the same throughput. We can further optimize resource utilization without incurring any latency penalties by leveraging this fact.

If we have two consecutive convolution layers, one with an 8×8 pixel output and a second with a 4×4 output, then we expect the first layer to take four times as long to process its input pixels as the second layer. The first layer requires 64 dense multiplications, and the second layer requires 16 dense multiplications to compute the result.

The initiation interval of the RTL implementation is based on the layer that has the longest initiation interval. Therefore, given the previous example, the second layer spends three-quarters of its time waiting for the previous layer to generate an output. By increasing the reuse factor, we can slow the execution of the second layer to match the throughput of the first layer and minimize the number of idle cycles. This also reduces parallelization and uses fewer resources without impacting the overall initiation interval or throughput.

Balancing reuse factor across layers within a model is the concept behind the *TargetCycles* parameter introduced by HLS4ML Pull Request #332. The *TargetCycles* parameter sets an initiation interval clock cycle target. For each supported layer (Convolutional and Dense), the reuse factor is estimated according to the Equation (5). The *TargetCycles* value is described by T_{Target}. We define the number of clock cycles required to move data around as $T_{overhead}$. And lastly, we define $N_{multiplies}$ as the number of dense multiplications calls in the layer.

$$RF = \frac{T_{Target} - \mathrm{T}_{\mathrm{overhead}}}{N_{multiplies}} \qquad (5)$$

Further studies are necessary to refine the computation of the optimal reuse factor. While the *TargetCycles* parameter has shown to significantly reduce resource utilization, it does not necessarily achieve the target initiation interval.