Generalized Machine Learning Quantization Implementation for High Level Synthesis Targeting FPGAs

Matthew Trahms

A thesis submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Electrical and Computer Engineering

University of Washington 2022

Committee:

Scott Hauck

Shih-Chieh Hsu

Program Authorized to Offer Degree:

Department of Electrical and Computer Engineering

University of Washington

Abstract

Generalized Machine Learning Quantization Implementation for
High Level Synthesis Targeting FPGAs

Matthew K Trahms

Chair of the Supervisory Committee:
Dr. Scott Hauck
Department of Electrical and Computer Engineering

The Large Hadron Collider produces a large amount of data while operating, approximately one petabyte of data per second. The collider is currently undergoing an upgrade to collide more particles and produce even more data. In order to handle this large quantity of data, high throughput and low latency algorithms are required to filter interesting collision results out of the rest of the data collected by the sensors attached to the collider. Machine learning algorithms can be used for this filtering task with comparable accuracy to the traditional filtering algorithms and provide a wide range of accelerator options. FINN and hls4ml are frameworks to deploy machine learning models on Field Programmable Gate Arrays for high throughput, low latency acceleration options. FINN utilizes Brevitas, a quantization aware training library. Using Brevitas, I trained a particle tracking network and demonstrated equivalent accuracy at lower bit precision than post training quantization. As a cross organizational project, the hls4ml and FINN teams collaborated to develop the QONNX standard for quantized machine learning model representation. In order to integrate QONNX into hls4ml, I implemented new transformations to support the unique structures of QONNX.

CONTENTS

Introduction

The Large Hadron Collider produces approximately one petabyte of data per second when in operation [1]. Machine learning algorithms are being explored as a way to process data while the Large Hadron Collider is operating. In order to process the large quantities of data produced by the Large Hadron Collider, high algorithm throughput is needed. Field programmable gate arrays (FPGAs) are being explored as a way to accelerate machine learning algorithms for this task. This thesis will explore two tools, FINN and hls4ml, used to deploy machine learning algorithms on FPGAs, as well as different strategies to reduce the logical complexity of the calculations involved, and propose a new model format for low bit width machine learning models.

This thesis is organized into 6 sections. Section 1 introduces the problem space for the rest of the paper: high energy physics tasks at the Large Hadron Collider. Section 2 provides an overview of machine learning, relevant models for this paper, and quantization. Section 3 introduces FINN, an FPGA machine learning tool. Section 3 also explores an example of quantization aware training as well as deploying a model onto an FPGA. Section 4 introduces hls4ml, explains the process to optimize and synthesize a model, and introduces work on QONNX, a unified quantized machine learning model format.

1 The Large Hadron Collider

The Large Hadron Collider (LHC) is the largest and most powerful particle accelerator in the world. The LHC collides particles to measure the resulting particle showers emitted by the collision [2]. The quantity of particle collisions provide an interesting challenge for data processing, because of the large quantity of data generated in a small time window [1]. This is a

suitable application for a high throughput, low latency algorithm to filter unnecessary data. Deep learning networks are being explored as a method for filtering because of the ready availability of acceleration options [3].

The LHC is composed of multiple detector experiments that measure different aspects of the collisions. This section will introduce the two relevant detectors for the rest of this paper: ATLAS and CMS, both general purpose detectors.

1.1 ATLAS

The ATLAS detector is the largest general purpose detector at the LHC [4] with the focus on interactions involving massive subatomic particles [5]. ATLAS is composed of several layers of detectors: an inner layer consisting of pixel detectors and trackers, electromagnetic and hadron calorimeters, a muon spectrometer, and all surrounded by superconducting magnet systems to direct charged particles [5].

All of the detector layers produce a lot of data which is then fed to the trigger and data acquisition system. The trigger is a process to determine when a special event is occurring and the data should be processed. This decision needs to be made in less than 2.5 microseconds and the system processes up to 100,000 events per second. From there, the events go to the second level trigger system, which conducts detailed analyses of each event in 200 microseconds and sends 1000 events per second to data storage for offline analysis [5].

1.2 CMS

The CMS detector is another large general purpose detector at the LHC [6]. Like ATLAS, CMS is composed of several layers: a tracker to track the position of particles through a magnetic field, an electromagnetic calorimeter, a hadronic calorimeter, a large magnet to direct particles, and muon detectors [6].

Similar to ATLAS, each of the CMS detector layers produce a lot of data which is fed to the trigger and data acquisition system. Similarly to ATLAS, CMS uses a two tiered trigger system. The level 1 trigger selects 100,000 interesting events per second with new data entering the data processing pipeline every 25 nanoseconds. From there, the data is transferred to the high level trigger, where 100 events per second are selected from the 100,000 events selected by the level 1 trigger. The latency of this process is less than a tenth of a second for each event [6].

2 Machine Learning

Machine learning is the process of designing algorithms that can automatically improve through training [7]. This thesis will be focused on deep learning, a subset of machine learning defined by using neural networks with three or more layers [8]. Machine learning algorithms, also known as models, are often composed from different functions, also known as layers [9]. Some of these layers have trainable parameters known as weights [10]. These parameters are trained using a set of sample data with known results and a loss function to determine the magnitude of error of the output produced by the model [10]. This section will explore different layer types, explain algorithms relevant to the Large Hadron Collider, and provide some context for the challenges deploying machine learning models on FPGAs.

2.1 Layer Types

Machine learning models can be composed of many different types of functions, also known as layers. Different layer types implement different functions. These functions operate on multi-dimensional matrices known as tensors [11]. Some layers implement simple nonlinear functions. These functions are known as activation functions [12]. They apply a function to each element in a tensor. An example of an activation function is the rectified linear unit (ReLU). Figure 1 shows the graph of the output of the ReLU function compared to the input. This function will be applied to each element in a ReLU layer.

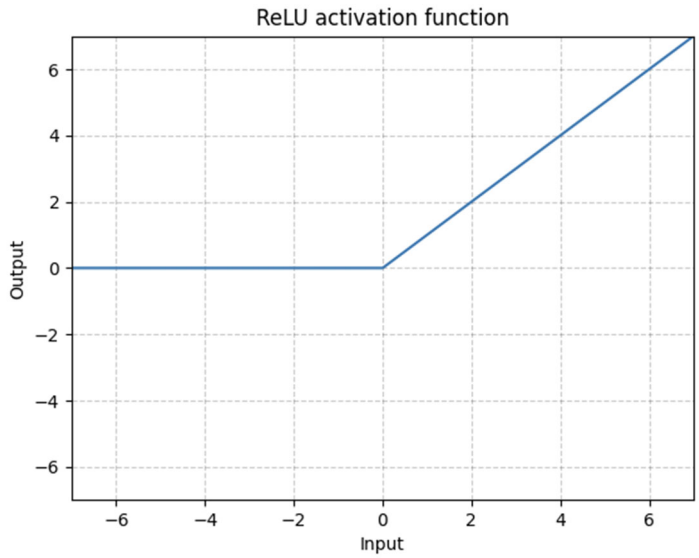

Fig. 1: The output of the ReLU activation function as a function of the input. [13]

These nonlinear functions allow machine learning models to approximate nonlinear functions by applying nonlinear transformations to the outputs of different layers in the model [12].

The main layer types used in models are known as dense layers and convolutional layers. Dense layers, also known as linear layers or fully connected layers, can be visualized as a set of neurons connected to each other with different pathways [14], similar to neuron structures within a brain.

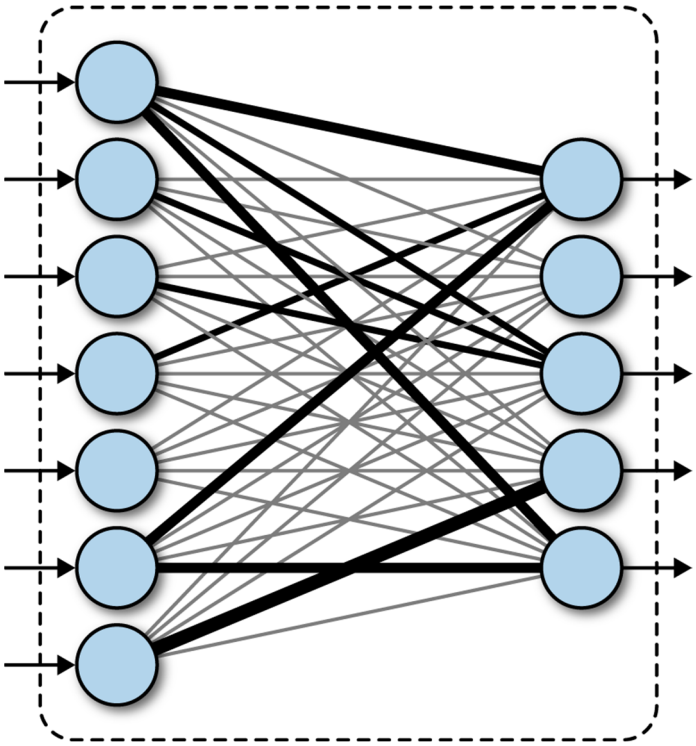

Fig. 2: A visualization of the structure of a fully connected layer [14]

The blue circles on the left in the figure above represent the input neurons to this layer. The blue circles on the right hand side of the figure represent the output neurons. The paths between the input neurons and output neurons are the neuron connections. Each neuron connection is weighted [14]. Some paths may be weighted stronger than others. In the figure above, this is represented with the thickness of the line. These weights are learned during the training process [10]. Additionally, dense layers may have a learned bias term to add an additional offset to each output of the layer [10]. The result of the entire layer can be calculated by doing a cross product multiplication of the inputs with the array of weights, and adding in the bias term [15].

While dense layers do an all-to-all connection between inputs and outputs, convolutional layers apply a small series of filters, or kernels, over an input [16]. These filters contain learned values to detect features within the image. This results in an output resembling a heatmap of the feature detected by the kernel. Figure 3 shows this process of applying a convolutional kernel to an input tensor.

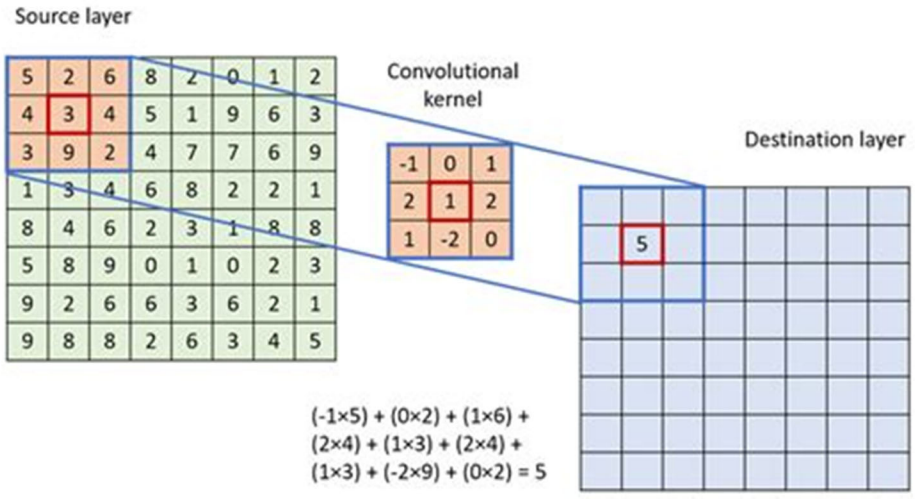

Fig. 3: Applying a convolutional kernel to an input tensor [17]

Convolutional layers have an advantage over dense layers for image processing due to the reduced number of parameters required for training and a lower reliance on feature location within the image, due to the filters of the convolutional layer being applied at all positions of the input tensor [16].

2.2 Network Architectures

There are many possible combinations of the machine learning layers discussed in the previous section. Some models are feedforward networks, meaning that an input feeds directly to an output through a series of layers; FACILE, which will be discussed in more detail in this section, is an example of a feedforward network. Alternatively, complex networks may have branches

and joins within the network. Figure 4 shows a residual network architecture, which is an example of a complex, non-feedforward architecture.

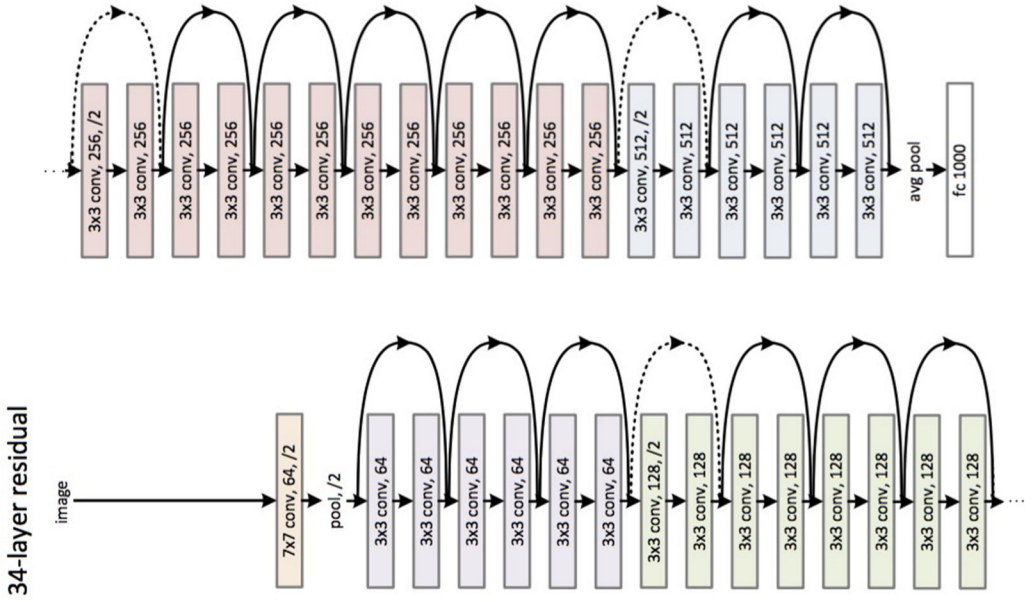

Fig. 4: A residual network architecture. Each box represents a different layer, while each arrow represents the flow of data from one layer to another. Notice that the intermediate outputs feed not only to the layer directly after, but also to additional layers in the architecture. Dotted lines represent transitions between blocks with different dimensions which are resolved with linear projection. [18]

This section will discuss two relevant network architectures for high energy physics: the FACILE network architecture, and the interaction network GNN architecture. We will cover the problem that is being addressed with the network, how the network is structured to solve the problem, the training process, and the results when compared to a traditional algorithm for solving the problem.

2.2.1 FACILE

The FACILE network architecture is a simple regression architecture. The network is designed to estimate energy deposited by particles in each cell of the CMS hadron calorimeter (HCAL) [3]. Hadron calorimeter reconstruction is used as a prototype application because of the high accuracy requirement for reconstruction. The machine learning algorithm must have high accuracy because HCAL measures sensitive events, such as a Higgs boson decaying into bottom quarks [3]. If the machine learning algorithm cannot reconstruct these events with similar accuracy to the traditional algorithm, then it will not make sense to replace the traditional algorithm with a machine learning algorithm. This section will cover the network architecture, how the network is trained, and performance relative to the traditional algorithm.

The network takes in 15 inputs consisting of information about the charge deposited, coordinates, and gain. The network consists of a pattern of batch normalization, densely connected, and rectified linear unit layers. The dense layers have a decreasing number of neurons going from the input densely connected layer to the output dense layer [3]. This structure is illustrated in figure 5.

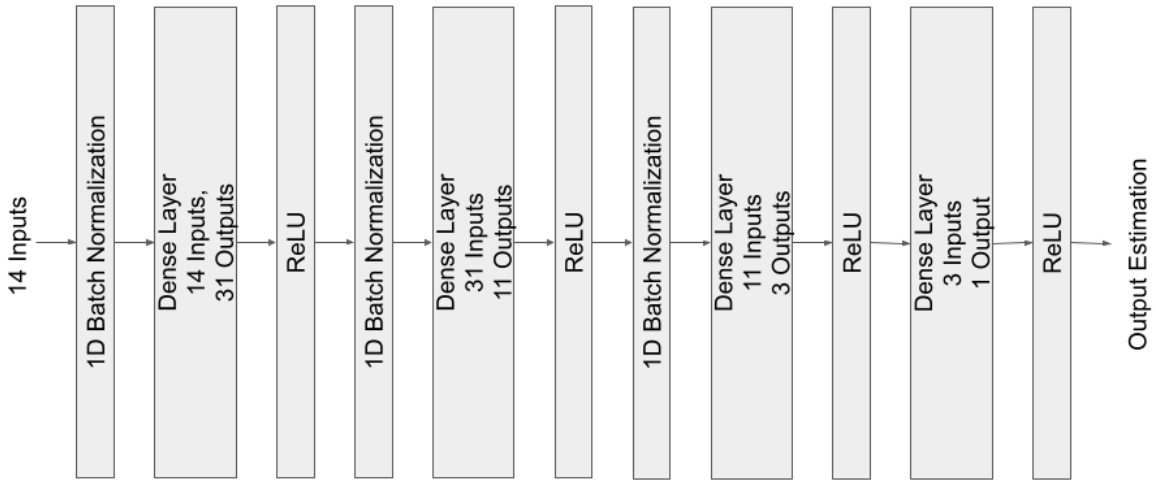

Fig. 5: The HCAL FACILE network architecture is a small, feedforward model used to estimate the energy of particles deposited in each cell of the CMS hadron calorimeter

The model is trained on a sample exceeding 1 million HCAL events separating into training and testing sets [3]. Network loss is evaluated using a mean squared error function and compared to the non-machine learning algorithm.

2.2.2 Interaction Network Graph Neural Network

Tracking the path of charged particles produced by particle collisions is essential to high energy physics tasks such as particle reconstruction and vertex finding [19]. Current state of the art algorithms produce lower accuracy predictions tracking charged particles as the particle beam intensity increases and more particles collide within the same window of time [19]. Machine learning is being explored as a solution to the problem of decreased accuracy due to increased beam intensity [19]. The LHC is undergoing an upgrade which will increase the intensity of the beam and result in more collisions per second [20]. This task will become more important as beam intensity increases.

Particle paths, or tracks, can be represented as mathematical graphs [19]. Detector hits can be viewed as nodes and track segments can be viewed as edges [19]. Nodes represent individual detector hits and have data associated with them such as position and energy. Edges represent possible paths of particles from one node to another. The task of this network is to determine which paths of edges represent accurate particle tracks [19]. For the problem of particle tracking, a series of nodes and edges are given to the network and the network needs to determine the

correct edges within the graph [19]. Graphs are generated by connecting nodes closest to each other geometrically [21].

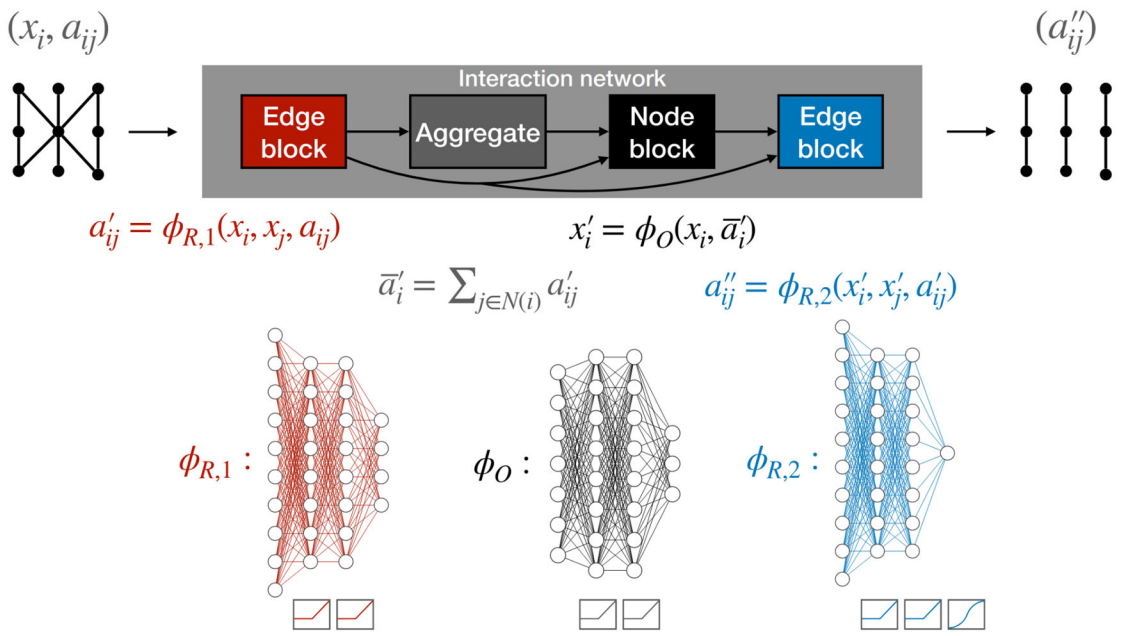

Fig. 6: The structure of an interaction network. Notice how the graph input is transformed from the structure on the left to the structure on the right as it passes through the network. [21]

Graph neural networks (GNNs) work by performing a series of graph transformations on a graph dataset input to transform the graph nodes and edges based on learned parameters [19]. Interaction networks are a subset of GNNs that are capable of reasoning about nodes and their relations to each other. Figure 6 is a representation of an interaction network. The *x* terms represent nodes while the *a* terms represent edges between nodes. *Φ* terms represent the multilayer perceptrons performing these transformations [19]. This is accomplished with a relational submodel that makes predictions based on edges and an object submodel that makes predictions based on nodes [19]. Together, these submodels predict node and edge and transform

the graph using a message passing system to alter the graph [19]. In this case the relational model and object models are both simple fully connected networks with ReLU activation layers [19].

This model is trained on the TrackML dataset. The dataset is a simulated set of proton collision events developed for a particle tracking challenge [21]. The collisions are simulated at high luminosity conditions, similar to those expected at the LHC [21].

2.3 Quantization

Most calculations done for machine learning algorithms are calculated using floating point binary numbers. Floating point binary numbers use a format to describe fractional components as well as integer components. Standard floating point numbers are 32 bits long and broken into three main sections: the sign, the exponent, and the fraction [22]. This structure is shown in figure 7.

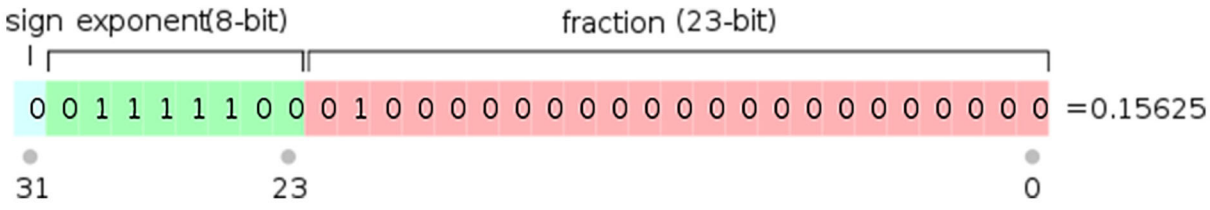

Fig. 7: The 32 bit floating point format, notice the different sections of the number and their bit widths [23].

The sign bit indicates whether the number is positive or negative, with zero being positive and one being negative [22]. The exponent serves effectively as an exponential multiplier of the fractional part of the number, with the value represented by the exponent being two to the power of the number represented in the exponent field. Additionally, the value in the exponent field is

actually represented as the binary value in that field minus 127 [22]. The fractional portion of the number, also known as the mantissa, is the number itself with each bit location representing a position after the binary point. For example, the leftmost bit in the fraction section can be viewed as 0.5, the next one as 0.25, and the next as 0.125. This pattern continues. Additionally, the fraction section has an implicit one in the one's position to make the total format of the number resemble scientific notation [22]. This format allows for a representation of a large range of numbers with a sliding scale of precision based on the magnitude of the number represented [22].

Floating point allows for the representation of a large range of numbers, but the hardware required to implement floating point operations is more complex than the hardware required to implement integer operations [24]. In order to implement machine learning models on FPGAs, it is desirable to reduce the hardware complexity required to implement these operations in order to ensure that the hardware will fit on the targeted FPGA and be able to run with a faster clock and higher throughput [24]. In order to accomplish this, the weights of the deployed model need to be converted to something closely resembling an integer value. This process is known as quantization. There are two main approaches to quantization values to integer values: fixed point quantization and scaled integer quantization [24][25].

Fixed point quantization works on a similar principle to floating point numbers, although much simpler to implement in hardware. Fixed point numbers have a number of bits dedicated to the integer portion of the number and a number of bits dedicated to the fractional portion of the number [24]. These fixed point numbers can represent fractional numbers to some extent, but have a limited range and precision when compared to floating point numbers [24]. However, if

the weights of the model being quantized are all within a set range, a fixed point quantization scheme may be suitable for acceleration because the hardware to implement arithmetic operations with fixed point numbers is very similar to the hardware required to implement the same operations on integers [24].

Scaled integer quantization works by shifting and scaling numbers to fit within an integer range. In order to quantize a number in a scaled integer format, a scale factor needs to be determined in addition to a potentially optional offset value [25]. The offset value shifts all quantized values into the range for. Once the number is shifted, it is then scaled using the scale factor. This ensures that all values should fall within the integer range of the selected bit width. From there, values are rounded or truncated to produce integers [25]. Inputs that fall outside the quantizable range may overflow if not handled. A common way to handle these inputs is to hold them to the boundary of quantization [25]. That way the value is represented as closely as possible to the original value. This quantization scheme is convenient because the hardware required to implement operations on numbers quantized in this way is nearly identical to the hardware required to implement operations on integers [25].

Whether quantization is implemented with fixed point quantization or scaled integer quantization, some loss of precision will occur with the model conversion [21]. There are two main approaches to the quantization process: post training quantization (PTQ) and quantization aware training (QAT) [21]. Post training quantization takes the floating point trained model and truncates/rounds the model to fit within the quantization parameters [21]. In most cases, this causes a drop in accuracy because the model weights are changed from the trained values in

order to fit within the quantization range. On the other hand, quantization aware training provides some information about the quantization parameters to the training process. The training process uses this information to ensure that the weights are already adjusted to the quantization range. This allows the training process to minimize or eliminate the drop in accuracy from quantizing the model [21]. Later sections will compare the accuracy of machine learning models with quantization aware training to models using post training quantization.

3 FINN

FINN is a framework developed primarily by Xilinx Research for transforming, synthesizing, and deploying machine learning models on FPGAs [26]. FINN trains to achieve extremely low bit width models for acceleration. Lower bit width models consume less FPGA area and can generally be accelerated to higher throughputs and lower latencies than higher bit width models using FINN. Additionally, FINN has specific requirements for layer ordering for high throughput and resource utilization [27], meaning that existing model architectures may require minor modifications to achieve reasonable performance in FINN. FINN is based around 5 main components: Brevitas, FINN ONNX, FINN transformations, High Level Synthesis, and deployment tools.

Brevitas is the quantization aware training library developed by Xilinx in conjunction with FINN for quantization aware training (QAT). This library allows users to train at the target precision of the final implementation, which tends to increase the accuracy of the model while performing inference. Users can create their model in Brevitas, train at a low, scaled integer bit width, and export it to a format that FINN can accelerate.

FINN provides a utility to convert Brevitas models directly to an internal representation known as FINN ONNX. FINN ONNX, based on ONNX, adds support for quantization and additional layers which are related to the hardware representations of layer types, such as multi-threshold layers [28]. Multi-threshold layers provide a way to represent the remaining unquantized floating point operations as a set of comparators. FINN also provides transformation utilities to replace, reorder, and combine layers within the FINN ONNX model in equivalent ways [29]. For example, sequential matrix multiplication layers can be combined, which reduces the total resources required to synthesize the model. Once the FINN ONNX model is optimized by combining layers, then FINN provides utilities to synthesize and deploy the model on an FPGA [30].

This section will explain the different components of FINN as background. Once the background on the FINN framework is explained, this section will explain the process of converting an existing machine learning model architecture into a FINN accelerated machine learning implementation on an FPGA.

3.1 Brevitas

Xilinx developed a quantization aware training library known as Brevitas [28]. Brevitas is based on the Pytorch machine learning library. Brevitas has layer specific replacements for major machine learning layers, such as linear and activation layers. Brevitas uses a scaled integer quantization scheme [31], meaning that network inputs are converted to integers by scaling and shifting them using scale and shift values learned on the dataset [31]. By doing this conversion to

integers, Brevitas and FINN can execute machine learning operations on integer values instead of floating point values. Brevitas is useful for training networks for synthesis and deployment in FINN, as well as quantization aware training as part of numerical studies. This section will cover the process of converting model architectures from Pytorch models to Brevitas models, as well as specific implementations for high energy physics.

Brevitas is an extension of the Pytorch machine learning framework [31], which means that converting Pytorch models to Brevitas quantized models can be done by swapping the Pytorch layers for quantized Brevitas layers and retraining. Brevitas has layer replacements available for matrix multiplication layers, activation layers, and convolution layers [31]. When constructing these layers, the constructor takes an additional argument that serves as the quantizer. Brevitas supports different quantization schemes depending on the desired bit width and network function [31]. This paper will focus on the scaled integer quantization scheme, and its usage in quantization aware trained networks for high energy physics.

The scaled integer quantization scheme works by generating an offset and scale factor for each of the input values to the machine learning network [31]. The offset and scale values work by offsetting the input to change the input range and then scaling the input to fit within the integer range of the bit width that is desired for the quantization aware training. If an input that exceeds the minimum or maximum value of the scaled integer representation is given as an input, then the value is clamped to the limit value [31]. This works to ensure that all input values to the network are within the expected range, while also preventing overflow.

Being based on Pytorch, Brevitas provides an interface to do numerical studies on quantization aware training for applications outside of FINN. One such example of a network where this was applied was the Interaction Network GNN discussed in section 2.2.2 of this paper. The interaction network was designed in Pytorch and synthesized at various bit widths in the hls4ml project using post training quantization [21]. There were questions around the accuracy that was achievable at lower bit width using quantization aware training. In order to answer those questions, I converted the unquantized Pytorch layers to Brevitas quantized layers. Once implemented, I trained the network on an equivalent training set as the unquantized model. I trained the model in a quantization sweep with a step of every two bits from two bits up to eighteen bits quantized. Once the models were trained, they were saved and evaluated against the post training quantization models of the same bit widths and the results were graphed.

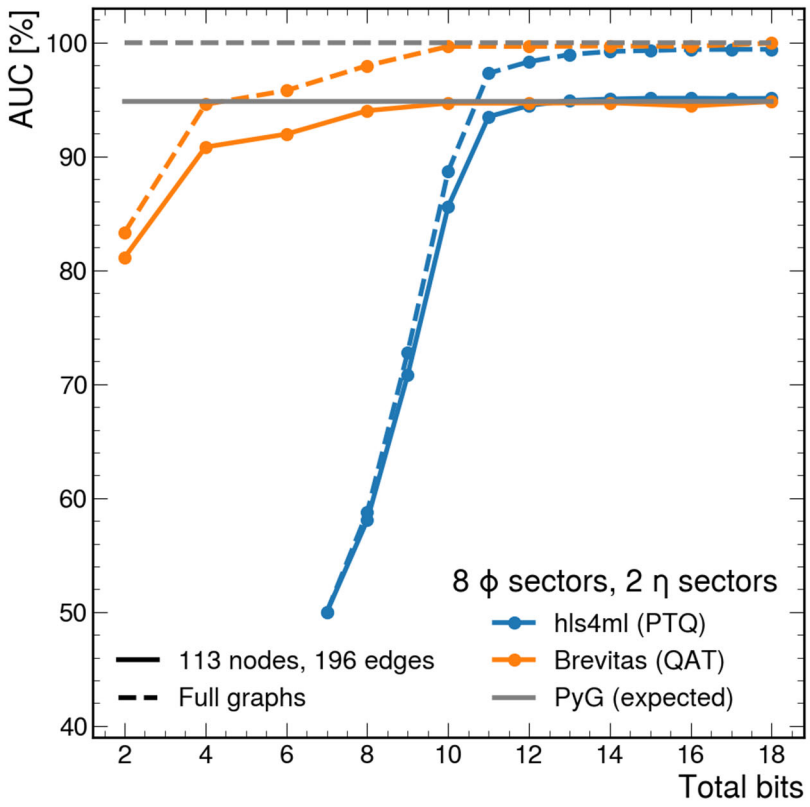

Fig. 8: The AUC of the quantization aware trained interaction network GNN trained in Brevitas compared with the post training quantization version quantized in hls4ml [21]

As figure 8 illustrates, quantization aware training in Brevitas maintains higher Area Under the Curve (AUC) at lower bit widths than post training quantization at comparable or even higher bit widths. AUC is a measurement of classifier accuracy. The "curve" in this context is a plot where the y axis is the true positive rate and the x axis is the false positive rate. Points along the curve can be calculated by varying the threshold for the classifier at which it considers an output as a positive result. Networks with a higher AUC are more accurate than those with lower AUC scores [32]. The two comparisons being done here are the evaluation with full graphs and evaluation with graphs truncated at the 95th percentile of graph size [21]. Larger graphs are reduced in size by not processing any more nodes and edges than are in the 95th percentile. These nodes and edges are assumed to be falsely classified. The reason for truncation in this context is to reduce the total graph size in order to reduce the hardware resource consumption of a fixed latency FPGA model [21]. In both cases, the quantization aware trained model either exceeds or closely matches the post training quantization model, with strong performance at extremely low bit widths.

3.2 FINN Flow

In order to convert a Brevitas model to hardware, FINN includes a series of steps to optimize and synthesize the machine learning model to hardware [33]. Broadly speaking, these stages can be categorized into 3 different stages of transformation: optimization, implementation, and synthesis. Once synthesized, FINN also provides a convenient interface to deploy and evaluate

models on supported FPGA boards [30]. This section will explain the different stages of the FINN flow more in depth, from a Brevitas model to evaluating on an FPGA.

Brevitas provides a direct function to export the model to a format known as FINN ONNX [28]. This is the first stage of transformation. The model is changed from a modified version of a Pytorch internal representation to a modified ONNX internal representation. This modified ONNX internal representation will remain in use until the model is synthesized to an FPGA. During this process, unquantized floating point operations are converted to multi-threshold layers [29]. The next section will explain multi-threshold layers in more depth. For the purposes of this section, multi-threshold layers can be understood as a generalized floating point conversion.

From there, the model proceeds into the optimization stage of the FINN flow. This stage is focused around reordering and combining layers in order to reduce the total number of layers and hardware requirements to synthesize the model [33]. Specific transformation functions exist to combine common combinations of layers into mathematically equivalent layers [33]. For example, multiplication operations can be absorbed into multi-threshold layers by changing the threshold values within the multi-threshold layer, while transpose operations can be absorbed into resize operations by changing the order of the outputs [34]. Reordering functions exist to reorder specific layer combinations in an attempt to maximize the number of combinations that can occur within the network [33]. For example, add operations with a constant term can be moved past multiplication operations by changing the value of the constant term, which may place a multiplication operation in a position where it can be absorbed by a multi-threshold operation. This system is able to combine a large number of layer combinations; however not

everything can be combined successfully [27]. This is the main reason that FINN is sensitive to layer orderings within the network. If specific orderings of layers occur within the network, the network may not be able to be optimized to as low an FPGA resource consumption or as high of a throughput as a functionally equivalent model with different layer orderings [27].

Once layer reordering and combination is finished, the model continues to the implementation stage. In this stage, the model is converted to high level synthesis (HLS) compatible C implementations of the layers [33]. From there, the layers can be synthesized into an FPGA deployable dataflow. Simulations can occur at key points during this synthesis flow. From there, the dataflow representation of the machine learning network can be deployed on an FPGA. Inference can be performed using a Xilinx PYNQ interface to measure potential throughput as well as test inference to ensure that the network performs as expected [30].

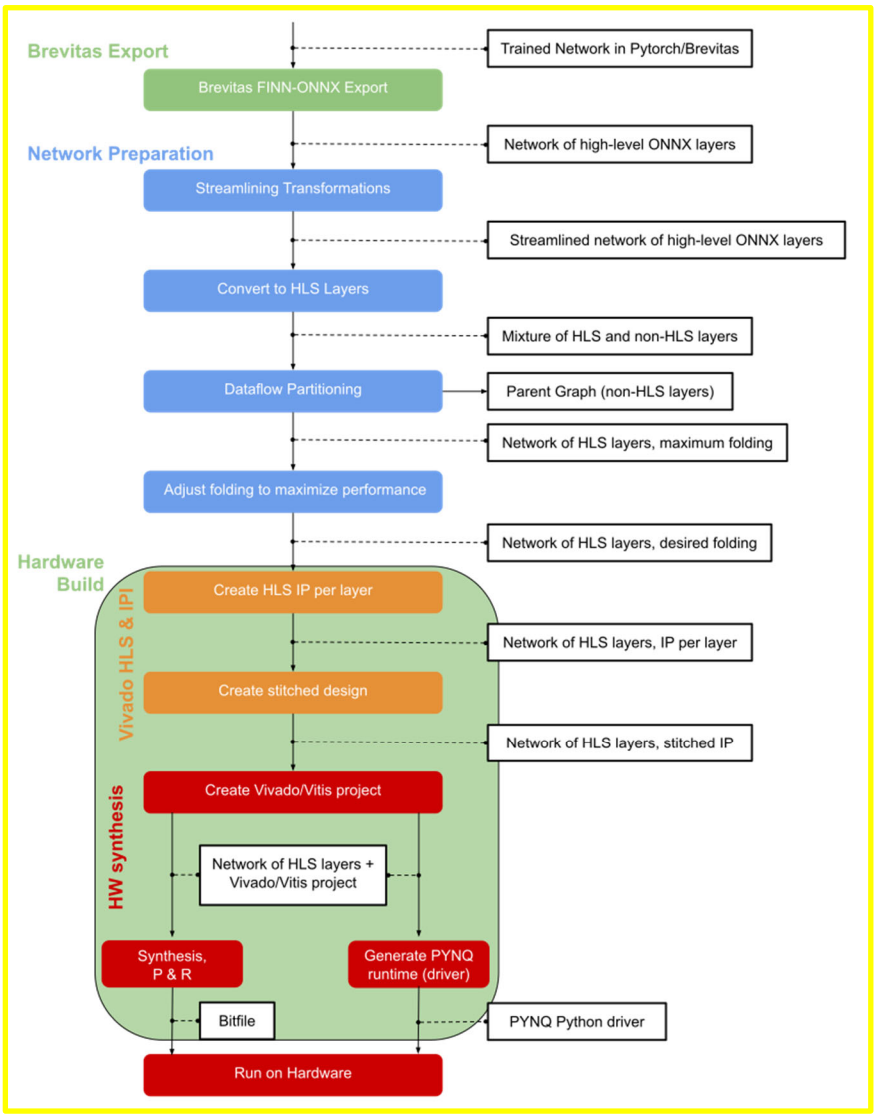

Fig. 9: The complete FINN flow from end to end. [35]

3.3 Multi-Threshold Layers

While quantized replacements for many machine learning layer types exist in Brevitas, not every operation has a quantized replacement. For example, FINN does not have specific implementations of quantized activation functions. Activation functions are implemented as multi-threshold layers as a way to abstract activation functions and Multi-Threshold layers exist

to convert the remaining floating point operations to a representation that is equivalent, while being simpler to implement in hardware [25].

Multi-threshold layers work by converting floating point operations into equivalent sets of comparisons with constants sweeping over a range [25]. Every comparison where the input is greater than the constant registers as true. As these thresholds are met, they are accumulated to form the output result. For example, a ReLU layer may have a minimum threshold of one in order to ensure that the input value is greater than zero in order to output anything greater than one. From there, the thresholds would increase by one. This can be accomplished using a finite number of comparators because there is a limited range of values in a given quantization scheme. For example, a multi-threshold layer with 8-bit quantization does not need to compare from negative infinity to infinity. Comparisons only need to exist in the range from -128 up to 127. As the input increases in value, the accumulated output of the multi-threshold layer also accumulates a higher and higher result [25]. As discussed in previous sections, these thresholds can be modified in order to combine multi-threshold layers with other operations in the network.

3.4 Implementation of the FACILE Architecture in FINN

With this understanding of the FINN flow, from network construction to acceleration, this thesis can now explore the process of translating and accelerating an existing network architecture for comparison to existing network acceleration methods. The FACILE network architecture exists to estimate the energy of particles based on Hadron Calorimeter (HCAL) hit data [3]. This is an example of an application requiring high throughput inference [3]. The FACILE architecture also serves as a point of comparison between FINN and hls4ml, because acceleration of the FACILE

architecture is well documented both using hls4ml on FPGAs and using GPU acceleration [3]. This section will cover the procedure to convert the FACILE network to a Brevitas implementation, the modifications to the architecture to facilitate FINN synthesis of the model, the synthesis process, and throughput results from the synthesized model.

The FACILE network architecture was initially implemented in Tensorflow. The architecture was initially replicated in Pytorch by Vladimir Ovechkin. The initial implementation of the network can be visualized with the figure below.

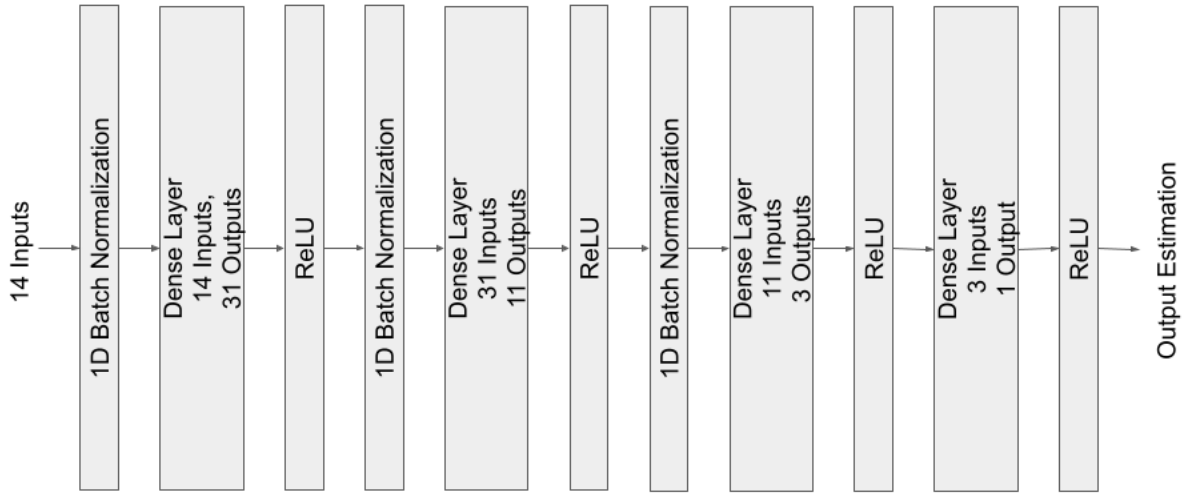

Fig. 10: The unquantized, original FACILE network architecture

I modified the network to substitute in quantized versions of the layers using Brevitas. Additionally, I added a quantized layer at the start of the network to set the quantization for multi-threshold layers at the start of the network. I performed additional layer reorderings after receiving advice from the FINN development team to allow for maximum layer reordering and combination for ease of synthesis. The final network architecture is represented by figure 11.

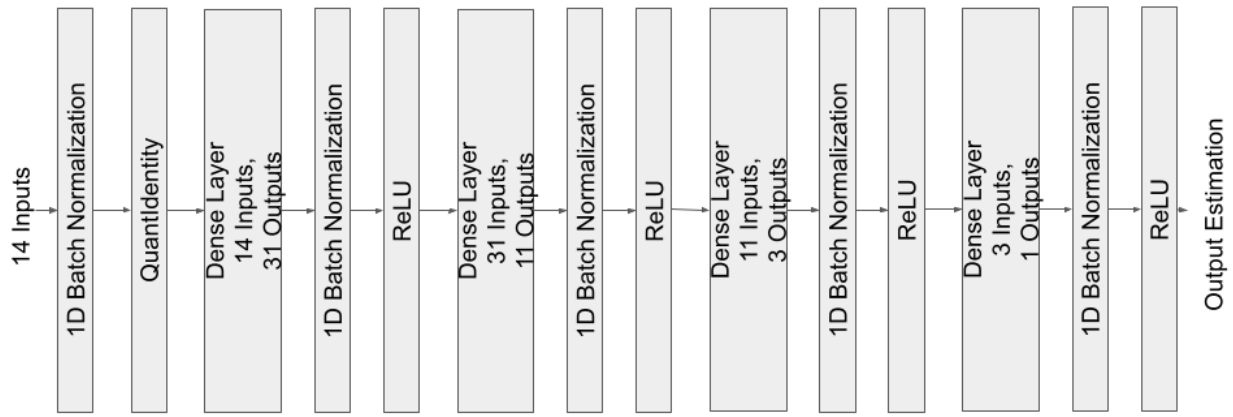

Fig. 11: The modified, FINN compatible FACILE network architecture

I then trained the quantized model and compared it to the unquantized version using the train and validate mean square error values. The quantized model was able to achieve equivalent error values at 8 bits of quantization and near equivalent error values at 4 bits of quantization. Table 1 shows the specific error results at these different precisions. Validation error is significantly higher than train error. This is likely due to overfitting on the train dataset. This could be caused in part by the transition from Keras to Pytorch and an error being made converting the training script. Because the validation error is significantly higher for unquantized and quantized performance, the overfitting should not be a problem for the purposes of this thesis, as it is not the quantization that is causing the increase in validation error.

Table 1: Train and validation errors of FACILE models at different levels of quantization

Quantization Level	**Train Error**	**Validation Error**
Original, unquantized model	.16	6
8 bit quantization	.1	5
4 bit quantization	.21	9.3

I deployed the quantized model on an AVnet Ultra96-V2 FPGA board, which uses a Xilinx Zynq UltraScale+ MPSoC ZU3EG A484 FPGA. The model was able to synthesize at both 8 and 4 bit precision. Testing the model using the PYNQ interface yielded a throughput of 4,000 inferences per second. Synthesis in hls4ml was done using an Alveo U250, which is a datacenter accelerator card, while the Ultra96-V2 is a small consumer grade FPGA. The hls4ml synthesized version was able to achieve 80 million inferences per second of throughput [3]. Compared to the hls4ml synthesized version, the FINN version was not able to achieve nearly as high levels of throughput, however that is likely due in large part to the differences in FPGA. The Ultra96-V2 is designed for edge applications and is easily acquirable as a consumer, while the Alveo U250 is a datacenter focused FPGA designed for high throughput applications.

4 hls4ml

High Level Synthesis for Machine Learning (hls4ml) is another tool to generate FPGA implementations of machine learning models. The driving force behind the development of hls4ml has been the potential for applications for high energy physics [36]. FINN and hls4ml are quite similar but differ in a few key ways. Both hls4ml and FINN target Xilinx FPGAs using the Vivado HLS synthesis tool, both require network quantization in order to synthesize networks, and both use an internal representation and transformations to optimize the machine learning model for FPGA deployment [36]. Unlike FINN, hls4ml uses fixed point quantization where FINN uses scaled integer quantization [21]. Additionally, while FINN uses multi-threshold layers to convert activation functions, hls4ml has individual implementations of activation functions in HLS [37]. FINN and hls4ml are similar enough to inspire a collaboration. This section will start with background on the model ingestion, the internal representation, and the

optimization passes. Then this section will introduce QONNX, a collaborative effort between the FINN and hls4ml teams to develop a standard representation for quantized machine learning models. Finally, this section will discuss the development effort to integrate QONNX support into hls4ml.

4.1 Model Ingestion and HLSModel

hls4ml supports multiple machine learning model formats, such as QKeras, Pytorch, and Tensorflow models [37]. In order to accommodate these different machine learning model formats, hls4ml converts the machine learning model into an internal representation known as HLSModel [37]. This process is known as model ingestion. During this process, the machine learning model is treated as a graph with layers being viewed as nodes. Each node is recreated and weights are translated into the HLSModel format. If nodes have associated quantization information, that is also translated into the HLSModel format, otherwise the quantization information is assumed to be the default quantization specified during the ingestion process for the model. By doing this ingestion, hls4ml is able to accept a wide variety of machine learning model formats and perform optimizations largely independent of the format of the machine learning model [37].

4.2 Optimization Passes

Once the model is converted to HLSModel, hls4ml performs a series of optimization passes to merge layers and improve the synthesizability of the model. This process is very similar to the streamlining process in FINN. Some layers can be combined together. For instance, batch

normalization layers can be combined with dense and convolutional layers by combining the biases and weights. These transformation passes are implemented in Python and are executed before the model is converted to a HLS transformation. Each transformation is implemented as its own class with two functions: match and transform [47]. The match function takes the node being transformed and returns whether this transformation can be run on the node. In most cases, that check consists of checking the type of node on which this transformation is being executed, checking attributes of the node, and potentially attributes of the surrounding nodes. The transformation function takes the node and the whole model and performs the optimization pass [37]. In the case of a fusion operation, this consists of modifying the attributes of either the current node or the parent and removing the other one. Transformations are registered in a map of passes. When it is time to optimize the model, the optimization passes are iterated through and called on each node in the graph. If the transformation matches the node based on the match function, then the transformation is applied to the node. By iterating through the transformations and the nodes, the optimizer is able to stop applying transformations when no more valid transformations are found.

4.3 QONNX

QONNX is a generic quantized machine learning model format developed as a collaboration between the hls4ml and FINN teams. It is based on the ONNX machine learning model representation format with a major addition: quant nodes. Quant nodes serve as an interface to convey quantization information without requiring different quant nodes for different layer types. QONNX is able to achieve this by simply wrapping unquantized nodes into a format that takes 2 inputs where one input is the quant node containing the weight quantization information.

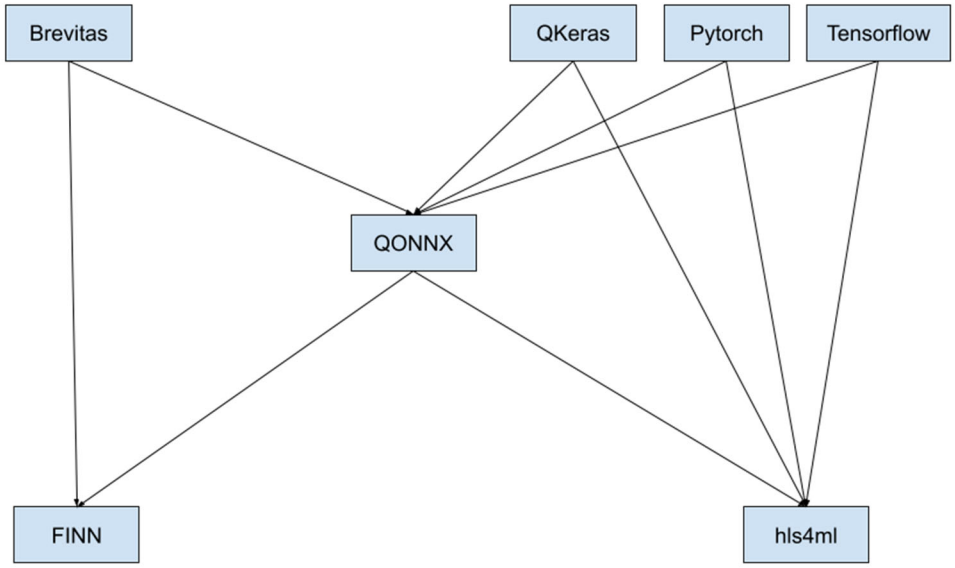

Fig. 12: The planned approach to creating a shared model format between hls4ml and FINN.

Figure 12 shows the different possibilities for interoperability using the QONNX model format. Models otherwise unsupported by FINN and hls4ml can be ingested and synthesized by using QONNX as an intermediate representation. This allows for post training quantized models in Pytorch and Tensorflow to be used in FINN, as well as adding support for QKeras models. Additionally, this approach allows for Brevitas quantization aware trained models to be synthesized in hls4ml.

Quant nodes can feed into nodes to specify how the weights should be quantized. They can also be placed between nodes to specify the quantization of downstream nodes. This means that both input and weight quantization can be handled with a single type of node. Figure 13 shows how quant nodes interact with surrounding nodes. Notice the structure of quant nodes feeding into the convolution nodes independently as a way to quantize weights. Additionally, quant nodes can sit

between conv nodes and their inputs as a way to identify quantization attributes within the model. The application of quant nodes to dense layers is similar to the application to conv nodes.

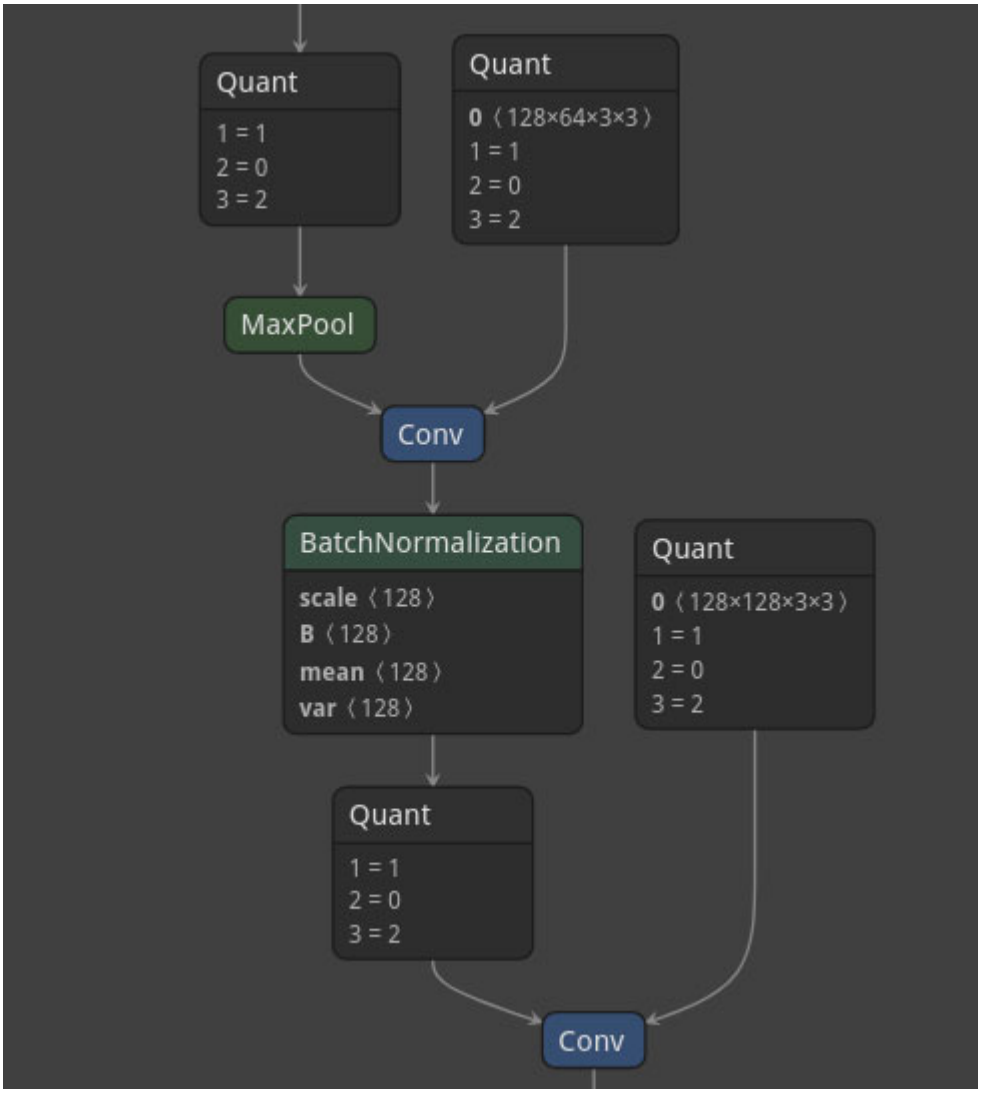

Fig. 13: One portion of a QONNX model graph. Notice how a quant node feeds into the Conv nodes as a way to quantize weights. Other quant nodes feed into the MaxPool and out from the BatchNormalization layers as a way to specify quantization parameters within the model.

FINN and hls4ml will support the QONNX format in the near future, and there are tools in process to convert from Brevitas and QKeras to QONNX [38]. The next section will go into detail around the process of implementing support for QONNX into hls4ml.

4.3.1 QONNX Ingestion and Transformation

As mentioned previously, weights and quantization information are transferred into HLSModel layer by layer. This aspect makes QONNX an outlier in terms of model ingestion. QONNX models don't store quantization information within each layer. Quant nodes store the quantization information for layers and exist as separate nodes within the QONNX graph. The approach to handle that oddity is to create a new quant node class in HLSModel. This quant node class is unsynthesizable and exists only as an intermediate representation. Once the full model is ingested as HLSModel, a new set of optimization passes need to be applied. These optimization passes are designed to wrap the quantization information from the quant nodes into the nodes that they are quantizing. Additionally these nodes need to remove the quant nodes, since they are not synthesizable. Once this is accomplished, the QONNX ingested HLSModel can go through the same set of transformations as other HLSModels. Figure 13 showcases the process for translating a QONNX graph into an HLSModel internal representation.

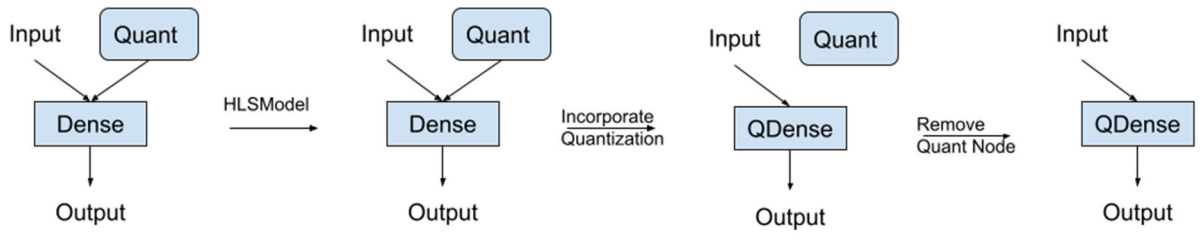

Fig. 13: The QONNX conversion process to HLSModel

5 Conclusion

This thesis examined the increase in accuracy possible using the same machine learning model quantized to the same bit width using quantization aware training instead of post training quantization. This thesis demonstrated that by using quantization aware training, comparable results were achievable with approximately ½ the number of bits used in computation when compared to post training quantization. Additionally, this thesis followed the process to reproduce, train, and deploy a machine learning model using the FINN toolset developed by Xilinx. From this, the thesis covered the efforts to create a cross-organizational standard for quantized machine learning network representation known as QONNX. This thesis covered the development effort to integrate QONNX model ingestion into the hls4ml project, specifically the necessary transformations to convert from a QONNX representation of a model to something synthesizable.

6 Future Work

While a first pass at integrating QONNX ingestion into hls4ml has been integrated, this is still highly in development. There are many features to test and expand with the current version. Model output parity has been verified for smaller models; however latency, throughput, and resource utilization has not been verified to be equivalent. Additionally, there are limitations to the QONNX models that can be ingested. There is a current bug with a convolutional model being tested where array allocation is extremely high for a kernel, which keeps the model from synthesizing. Additionally, this has only been tested with smaller, feedforward architectures. Testing this process with different network architectures will be necessary to ensure the robustness of the ingest process. Special optimization procedures also exist for QKeras and

QONNX models. Eventually, these different optimization passes should be combined into a unified set of optimization passes with just a couple unique passes to integrate quantization layers into the layers being quantized. Finally, this ingestion process should be brought forward into the new hls4ml architecture.

7 Acknowledgements

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

I would like to thank both of my advisors, Dr. Scott Hauck and Dr. Shih-Chieh Hsu for their guidance and support. It has been a pleasure working with both of them, starting as an undergraduate research assistant, up through graduation, and all through this thesis. Scott's technical insights and ability to know which questions to ask to clarify concepts has been incredibly helpful in driving my understanding through this project. Shih-Chieh's insights into the physics and mechanisms behind the Large Hadron Collider were invaluable to my understanding of the projects I was working on. I would not have been able to do this without them.

Thank you to Nhan Tran for onboarding me onto the hls4ml team and always being helpful when I was looking for places to contribute. Thank you to Jovan Mitrevsky for working with me to understand the hls4ml framework and working with me to find specific contributions that I could make to the QONNX ingestion into hls4ml. Thank you to Javier Duarte and Abdelrahman Elabd for working with me to find a place for my work on quantization aware training of the particle tracking GNN in the paper in progress.

I am also thankful to the FINN team at Xilinx and Yaman Umuroglu for answering my questions around FINN and helping me implement FACILE in FINN.

I would like to thank my parents, Rob Trahms and Lisa Trahms who have always encouraged me to be the best version of myself. They have been a great inspiration, support, and motivation while pursuing this degree.

Thank you to my friends and family who supported me not only pursuing this degree, but in my life outside of school, and in my future plans and aspirations.

References

[1] "CERN Data Centre passes the 200-petabyte milestone," CERN. https://home.cern/news/news/computing/cern-data-centre-passes-200-petabyte-milestone.

[2] CERN, "The Large Hadron Collider | CERN," *Cern*, 2019. https://home.cern/science/accelerators/large-hadron-collider.

[3] D. Rankin *et al*., "FPGAs-as-a-Service Toolkit (FaaST)," *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2020, pp. 38-47, doi: 10.1109/H2RC51942.2020.00010.

[4] "About," *ATLAS Experiment at CERN*. https://atlas.cern/about.

[5] "ATLAS Fact Sheets," *ATLAS Experiment at CERN*. https://atlas.cern/resources/fact-sheets.

[6] "Detector | CMS Experiment," *cms.cern*. https://cms.cern/detector.

[7] T. M. Mitchell, *Machine learning*. Singapore: Mcgraw-Hill, 1997.

[8] IBM Cloud Education, "What is Deep Learning?," *www.ibm.com*, May 01, 2020. https://www.ibm.com/cloud/learn/deep-learning.

[9] "List of Deep Learning Layers - MATLAB & Simulink," *www.mathworks.com*. https://www.mathworks.com/help/deeplearning/ug/list-of-deep-learning-layers.html.

[10] "Weight (Artificial Neural Network)," *DeepAI*, May 17, 2019. https://deepai.org/machine-learning-glossary-and-terms/weight-artificial-neural-network.

[11] "Tensor," *DeepAI*, May 17, 2019. https://deepai.org/machine-learning-glossary-and-terms/tensor.

[12] "Activation Function," *DeepAI*, Sep. 27, 2020. https://deepai.org/machine-learning-glossary-and-terms/activation-function.

[13] "ReLU — PyTorch 1.8.0 documentation," *pytorch.org*. https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html.

[14] D. Battini, "Learn Coding Neural Network in C#: Define layers and activations," *Tech-Quantum*, Mar. 14, 2019. https://www.tech-quantum.com/learn-coding-neural-network-in-c-define-layers-and-activations/.

[15] "Linear — PyTorch 1.10 documentation," *pytorch.org*. https://pytorch.org/docs/stable/generated/torch.nn.Linear.html#torch.nn.Linear .

[16] "What are Convolutional Neural Networks?," *www.ibm.com*. https://www.ibm.com/cloud/learn/convolutional-neural-networks.

[17] Podareanu, Damian & Codreanu, Valeriu & Aigner, Sandra & Leeuwen, Caspar & Weinberg, Volker. (2019). Best Practice Guide - Deep Learning. 10.13140/RG.2.2.31564.05769.

[18] Tiwari, Mohit & Tiwari, Tripti & Kassab, Manal & Roy, Anit & Chaudhary, Deepa & Onyema, Edeh. (2020). Detection of Coronavirus Disease in Human Body Using Convolutional Neural Network. 29. 2861-2866.

[19] G. DeZoort *et al.*, "Charged Particle Tracking via Edge-Classifying Interaction Networks," *Computing and Software for Big Science*, vol. 5, no. 1, Nov. 2021, doi: 10.1007/s41781-021-00073-z.

[20] "New technologies for the High-Luminosity LHC," *CERN*. https://home.cern/science/accelerators/new-technologies-high-luminosity-lhc .

[21] A. Elabd *et al.,* "Graph Neural Networks for Charged Particle Tracking on FPGAs," submitted to *Frontiers*

[22] "IEEE Standard for Floating-Point Arithmetic," 2020, doi: 10.1109/ieeestd.2019.8766229.

[23] "Difference between fixed and floating point," *Electrical Engineering News and Products*, Sep. 15, 2017. https://www.eeworldonline.com/difference-between-fixed-and-floating-point/ .

[24] L. S. Rosa, C. F. M. Toledo, and V. Bonato, "Accelerating floating-point to fixed-point data type conversion with evolutionary algorithms," *Electronics Letters*, vol. 51, no. 3, pp. 244–246, Feb. 2015, doi: 10.1049/el.2014.3791.

[25] T. Alonso *et al.*, "Elastic-DF: Scaling Performance of DNN Inference in FPGA Clouds through Automatic Partitioning," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 15, no. 2, pp. 1–34, Jun. 2022, doi: 10.1145/3470567.

[26] "What is FINN?," *finn*. https://xilinx.github.io/finn/about.

[27] T. B. Preuser, G. Gambardella, N. Fraser, and M. Blott, "Inference of quantized neural networks on heterogeneous all-programmable devices," *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Mar. 2018, doi: 10.23919/date.2018.8342121.

[28] "Brevitas Export — FINN documentation," *finn.readthedocs.io*. https://finn.readthedocs.io/en/latest/brevitas_export.html.

[29] "Network Preparation — FINN documentation," *finn.readthedocs.io*. https://finn.readthedocs.io/en/latest/nw_prep.html.

[30] "Hardware Build and Deployment — FINN documentation," *finn.readthedocs.io*. https://finn.readthedocs.io/en/latest/hw_build.html.

[31] "Brevitas," *GitHub*, Feb. 22, 2022. https://github.com/Xilinx/brevitas.

[32] Google, "Classification: ROC Curve and AUC | Machine Learning Crash Course," *Google Developers*, 2019. https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc.

[33] M. Blott *et al.*, "FINN- R," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 3, pp. 1–23, Dec. 2018, doi: 10.1145/3242897.

[34] "Xilinx/finn," *GitHub*, Feb. 20, 2022. https://github.com/Xilinx/FINN

[35] "End-to-End Flow — FINN documentation," *finn.readthedocs.io*. https://finn.readthedocs.io/en/latest/end_to_end_flow.html

[36] T. Aarrestad *et al.*, "Fast convolutional neural networks on FPGAs with hls4ml," *Machine Learning: Science and Technology*, vol. 2, no. 4, p. 045015, Jul. 2021, doi: 10.1088/2632-2153/ac0ea1.

[37] "hls4ml," *GitHub*, Feb. 22, 2022. https://github.com/fastmachinelearning/hls4ml.

[38] H. Borras, Y. Umuroglu, "QONNX and FINN," *finn*, Nov. 03, 2021. https://xilinx.github.io/finn/2021/11/03/qonnx-and-finn.html.

Appendix 1 - Publications

For each of the publications below, I am listed as an author and the material closely relates to the topics of my thesis. For the first three (*FPGAs-as-a-Service Toolkit (FaaST)*, *FPGA-Accelerated Machine Learning Inference as a Service for Particle Physics Computing*, *GPU coprocessors as a service for deep learning inference in high energy physics*) I was an undergraduate at the University of Washington. I was a graduate student for the latest publication (*Graph Neural Networks for Charged Particle Tracking on FPGAs*). As an undergraduate, I worked to implement a REST server for FPGA machine learning inference using AWS instances. This work provided useful insights for *FPGAs-as-a-Service Toolkit (FaaST)* as well as *GPU coprocessors as a service for deep learning inference in high energy physics*. That research was especially relevant to the FACILE comparison using an Alveo U250. Those measurements were accomplished using a datacenter GPU and recorded in the same paper. Also as an undergraduate, I worked on measuring the throughput and accuracy of Microsoft's Brainwave machine learning service. My research helped publish the *FPGA-Accelerated Machine Learning Inference as a Service for Particle Physics Computing*. My research used ResNet-50 and is most relevant in the description of a residual network and in the discussion of metrics for analyzing network performance. As a graduate student, I converted the particle tracking interaction network to Brevitas for quantization aware training purposes, which allowed me to compare and contrast quantization aware training to post training quantization in *Graph Neural Networks for Charged Particle Tracking.* The section on the interaction network GNN is based on that research entirely.

D. Rankin, S. Hauck, S. Hsu, M. Trahms *et al*., "FPGAs-as-a-Service Toolkit (FaaST)," *2020 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC)*, 2020, pp. 38-47, doi: 10.1109/H2RC51942.2020.00010.

J. Duarte, S. Hauck, S. Hsu, M. Trahms *et al.* "FPGA-Accelerated Machine Learning Inference as a Service for Particle Physics Computing," 2019 *Comput Softw Big Sci* **3,** 13, https://doi.org/10.1007/s41781-019-0027-2

J. Krupa, S. Hauck, S. Hsu, M. Trahms *et al.* "GPU coprocessors as a service for deep learning inference in high energy physics," *Mach. Learn.: Sci. Technol.* **2** 035005

A. Elabd, S. Hauck, S. Hsu, M. Trahms *et al*. "Graph Neural Networks for Charged Particle Tracking on FPGAs," 2022 submitted to *Frontiers* *https://arxiv.org/abs/2112.02048*

Appendix 2 - Software Repositories

Brevitas and FINN implementation of FACILE - https://github.com/kf7lsu/pytorchFACILE

Brevitas Tracking Interaction Network - https://github.com/kf7lsu/interaction_network_paper

QONNX ingest into hls4ml - https://github.com/fastmachinelearning/hls4ml/blob/ingest-qonnx/hls4ml/model/optimizer/passes/matmul_const_to_dense.py