This is the Author-Accepted Version of the paper:

J. Vreča and A. Biasizzo, "Towards Deploying Highly Quantized Neural Networks on FPGA Using Chisel," *2023 26th Euromicro Conference on Digital System Design (DSD)*, Golem, Albania, 2023, pp. 161-167, doi: 10.1109/DSD60849.2023.00032.

Towards Deploying Highly Quantized Neural Networks on FPGA Using Chisel

1st Jure Vreča
Computer Systems Department
Jožef Stefan Institute
Ljubljana, Slovenia
jure.vreca@ijs.si

2nd Anton Biasizzo
Computer Systems Department
Jožef Stefan Institute
Ljubljana, Slovenia
anton.biasizzo@ijs.si

***Abstract*—We present chisel4ml, a Chisel-based tool that generates hardware for highly quantized neural networks described in QKeras. Such networks typically use parameters with bitwidths less than 8 bits and may have pruned connections. Chisel4ml can generate the highly quantized neural network as a single combinational circuit with pipeline registers in between the different layers. It supports heterogeneous quantization where each layer can have a different precision. The full parallelization enables very low-latency and high throughput inference, that are required for certain tasks. We illustrate this on the triggering system for the CERN Large Hadron Collider, which filters out events of interest and sends them on for further processing. We compare our tool against hls4ml, a high-level synthesis based approach for deploying similar neural networks. Chisel4ml is still under development. However, it already achieves comparable results to hls4ml for some neural network architectures. Chisel4ml is available on https://github.com/cs-jsi/chisel4ml.**

***Index Terms*—Chisel, neural networks, quantization, FPGA**

I. Introduction

Artificial neural networks have become a very important branch of machine learning as they can easily be trained to handle a variety of problems. The increasing size of neural networks has however proven a challenge for deployment of such neural networks at the edge. Additionally, in some systems very low latency is required. An example is a data processing and storage triggering system for a particle accelerator, where the triggering system must filter out events of interest to avoid over-loading the computing system. Another example is a network intrusion detection system at the edge [1]. Network intrusion detection is the task of scanning network traffic to detect a potential intrusion. Machine learning approaches have been applied to this task, however such approaches are intractable to compute on battery powered edge devices.

The CERN Large Hadron Collider generates a vast amount of data at rate of several terabytes per second. This presents a significant computing challenge for both real time and offline processing of this data [2]. To make the data rate manageable a processing and storage triggering system was developed based on neural networks that are deployed on FPGAs. As the required latency is below few hundreds of nanoseconds they propose a data flow style accelerator of quantized neural networks where all the weights of the neural network are stored on the FPGA [3]. This eliminates the need for high latency access to DRAM memory, but presents a challenge in itself because the memory capacity of an FPGA is quite limited with respect to the sizes of neural networks. This is where quantization and pruning techniques have proven useful in significantly reducing the size of neural network models [4, 5].

Quantization of neural networks is the process of reducing the required precision of parameters and input features. Instead of the floating-point representation we can use integers of various bitwidths, even as low as a single bit [6]. Pruning on the other hand is the process of removing unneeded parameters (connections) from the neural network, and thus decreasing the model size. It has been shown that even 90% of neural network parameters can be removed this way, with minimal loss of neural network accuracy [7]. Pruning and quantization can also be combined to further reduce the size of the neural network.

Typically, digital hardware is designed with hardware description languages (HDL). Such hardware development, however, can take a long time, and requires domain knowledge. This motivates the development of a generator that could take a high-level description of a neural network, and generate the hardware automatically. One such generator is hls4ml [3], and was developed to solve the motivating example given above. Another generator, FINN [8], was developed by Xilinx. Both of these tools are based on High-Level Synthesis, which can lead to sub-optimal results, as shown by an experimental study conducted by the FINN team [9].

We have decided to tackle the problem with the Chisel Hardware Construction Language (HCL). It is implemented as a framework within the Scala programming language, which gives it many advanced features; such as object orientation and functional programming. In Chisel you write a software program, that when executed constructs the hardware, and exports it as Verilog. Neural networks are good candidates for such an approach since their high-level description can be easily translated by the Scala code into Chisel hardware representation. Chisel4ml tool is a Chisel hardware generator that implements quantized feed-forward neural networks. It is still

The authors acknowledge the financial support from the Slovenian Research Agency (research core funding No. P2-0098). This work is also part of a project that has received funding from the ECSEL Joint Undertaking under grant agreement No 101007273 (DAIS).

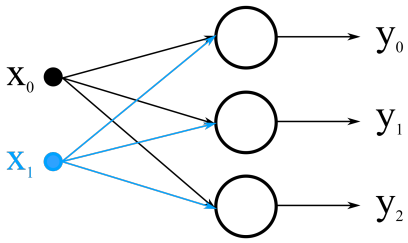

Fig. 1: A schematic representation of a fully-connected layer.

under development. However, it already achieves comparable results to hls4ml for some neural network architectures.

The remainder of this article is organized as follows. In Section II we provide background information on quantized neural networks, in Section III we detail how chisel4ml is implemented, in Section IV we compare the results of the tools, and finally we conclude the paper with Section V.

II. Quantized Neural Networks

Artificial neural networks are computational models that are loosely inspired by biological neurons. They are constructed from a set of layers, which can contain many neurons.

A. Artificial Neural Networks

A single neuron of an artificial neural network can be computed as shown in Eq. (1):

$$y = f(b + \sum_{i}^{N} w_i \cdot x_i), \tag{1}$$

where the vector of weights w and the bias b are constant parameters, x_i are the inputs to the neuron, f is the non-linear function, and y is the (scalar) output. The Rectified Linear Unit (ReLU) is a common non-linear function used in deep learning:

$$f(x) = ReLU(x) = max(0, x). \tag{2}$$

A single neuron however, has a limited learning ability. So we typically combine them in groups called layers, which contain many neurons. A basic type of neural network layer is a fully-connected or dense layer, where all inputs are connected to all neurons. Fig. 1 shows an example fully-connected layer with three neurons and two inputs. Fully-connected layers can be concisely expressed as matrix-vector multiplication, as shown in Eq. (3):

$$\vec{y} = f(W \cdot \vec{x} + \vec{b}) = d(\vec{x}), \tag{3}$$

where W is the matrix of weights, $\vec{x}$ is the input vector, $\vec{b}$ is the vector of constant bias values, and $\vec{y}$ is the output vector. Several such layers can be sequentially connected to form a feed forward neural network:

$$\vec{y} = d_1(d_2(...(d_n(\vec{x})))) = d_1 \circ d_2 \circ ... \circ d_n(\vec{x}) = D(\vec{x}; \theta). \tag{4}$$

Where $\vec{x}$ is again the input vector, $\vec{y}$ is the output vector, functions d_i represent the various layers, θ represents the parameters of the entire neural network, and function D represents the whole neural network. For classification tasks, the final layer typically uses the softmax activation function, as defined in Eq. (5):

$$\sigma(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{K} e^{x_j}} \quad for\ i = 1, 2, \ldots, K, \tag{5}$$

however, at inference the winning class can simply be selected by choosing the largest output value. Therefore computing the softmax is unnecessary for applications which only care about the winning class.

B. Quantization

Function D (the neural network) in Eq. (4) would typically be computed using either standard 32-bit floating-point numbers, or some variant of 16-bit floating-point numbers [10]. It has been shown that neural networks can be accurate also with fixed-point arithmetic, but special consideration is needed.

In general there are two methods of quantization of neural networks:

- Post-Training Quantization (PTQ), and
- Quantization-Aware Training (QAT).

With PTQ methods we first train the neural network using the floating-point representation, and then convert trained parameters into the fixed-point representation after training. This is the most straightforward method, however it can cause severe degradation in neural network accuracy when quantizing bellow 8 bits.

The main idea of QAT is to use the full-precision representation of parameters and input features in the back-propagation step, however quantize them at the forward-propagation step. This is achieved by inserting fake quantization functions[1] into the neural networks computational graph. This is shown in Fig. 2. The left part of Fig. 2 shows the computational graph of Eq. (3), and the right part shows a graph for quantization aware training with fake quantization functions q inserted. Fake quantization functions are called fake, because they don't actually change the datatype of the computation. Instead they take floating-point inputs, and produce floating-point outputs; however, they limit the output values to a set that is representable with fixed-point numbers. After training is completed, the floating-point parameters can be replaced with fixed-point parameters, and the fake-quantization functions removed.

As the parameters values range can differ significantly between layers, a scaling parameter can be introduced to improve results:

$$W \approx \frac{W_q}{S} \tag{6}$$

Where W is the floating-point weight tensor, W_q is a quantized integer weight tensor, and S is a constant scaling factor. The granularity of scaling is either the entire weight tensor, or per neuron scaling can also be performed. Inserting Eq. (6) into Eq. (3) we get:

$$\vec{y} = f(W \cdot \vec{x} + \vec{b}) \approx f(\frac{W_q \cdot \vec{x}}{S} + \vec{b}). \tag{7}$$

[1] Fake quantization is also sometimes called simulated quantization.

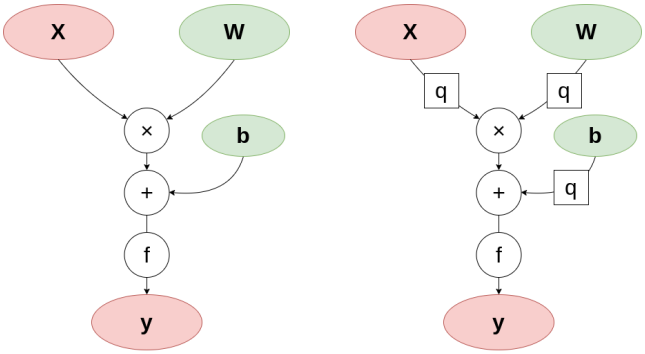

Fig. 2: A Regular neural network computational graph on the left, and a graph for quantization aware training on the right.

If we assume $\vec{x}$ and $\vec{b}$ are quantized, and that the scaling factor S is a power of two constant, this leaves us with only integer operations, that can efficiently be performed in hardware.

An extreme case of quantization are binarized neural networks (BNN) [6]. In BNN the input features, weights, and output features are restricted to two values -1, +1 and are represented by a binary variable. By mapping them to binary values 0, 1 the multiplication operation translates to an XNOR operation as depicted in Fig. 3. Similarly the dot product, which is an important part of an artificial neuron model Eq. (1), translates to a population count of products (XNOR operations). Because the output is binarized as well, the activation function is a sign function. All aforementioned operations can be implemented very efficiently in hardware: multiplication are implemented by a single gate or a lookup table (LUT), population count operation is simpler than addition operation, and sign function is implemented by a comparator. The uses of binarized neural networks are, however, limited to simplier problems.

w	x	w·x
-1	-1	1
-1	1	-1
1	-1	-1
1	1	1

→

w	x	w⊙x
0	0	1
0	1	0
1	0	0
1	1	1

Fig. 3: The mapping of values (-1,+1) to (0,1), where the ⊙ symbol means the XNOR operation.

C. QKeras

QKeras is a python library for quantization aware training [11]. It offers a user friendly way to define and train such neural networks. An example of a neural network definition is given in Listing 1. It defines a 4-layer neural network with 16 inputs, 5 outputs, and 64, 32, 32, and 5 neurons in each layer. The weights of all layers are quantized to 6-bit signed integer numbers and scaling factors are constrained to a power of two. The function *quantized_bits* in Listing 1 represents the quantization operator q in Fig. 2. Additionally a quantizer was introduced at the input of the neural network, to define how the input is quantized. This is required to define the neural network input data type for chisel4ml.

```
nn = Sequential() # nn = neural network model
nn.add(Input(shape=(16,)))
nn.add(QActivation(quantized_bits(bits=11, integer=10,
                                  alpha=1,
                                  keep_negative=True)))
nn.add(QDense(64,
              kernel_quantizer=quantized_bits(bits=6,
                                              integer=5,
                                              alpha='auto_po2',
                                              keep_negative=True),
              use_bias=False))
nn.add(QActivation(activation=quantized_relu(bits=5,
                                             integer=5)))
nn.add(QDense(32,
              kernel_quantizer=quantized_bits(bits=6,
                                              integer=5,
                                              alpha='auto_po2',
                                              keep_negative=True),
              use_bias=False))
nn.add(QActivation(activation=quantized_relu(bits=5,
                                             integer=5)))
nn.add(QDense(32,
              kernel_quantizer=quantized_bits(bits=6,
                                              integer=5,
                                              alpha='auto_po2',
                                              keep_negative=True),
              use_bias=False))
nn.add(QActivation(activation=quantized_relu(bits=5,
                                             integer=5)))
nn.add(QDense(5,
              kernel_quantizer=quantized_bits(bits=6,
                                              integer=5,
                                              alpha='auto_po2',
                                              keep_negative=True),
              use_bias=False))
nn.add(Activation(activation=tf.keras.activations.softmax))
```

Listing 1: An example neural network in QKeras.

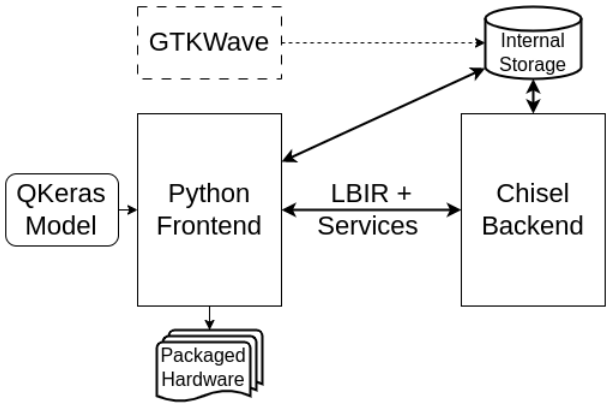

Fig. 4: High-level software architecture of chisel4ml.

III. Chisel4ml

Chisel4ml converts QKeras neural network models to a Verilog hardware description. It supports heterogeneous quantization where each layer can have different precision. To make chisel4ml more user friendly we added a Python frontend to it. The software architecture of chisel4ml is shown in Fig. 4. It is divided into the Python frontend, and the Chisel backend. The Chisel backend is implemented as a server. Communication in between the frontend and the backend is handled by gRPC: a Remote Procedure Call library from Google, and protocol buffers, a binary serialization format.

A custom quantized neural network format, called Low-Bitwidth Intermediate Representation, was developed to provide a unique representation of such networks for chisel4ml. We use protocol buffers to encode quantized neural network models in LBIR format and service requests and responses to and from the server. The backend serves two types of

requests: generate circuit from the model and simulate the circuit. Listing 2 shows the protocol buffers definition file that defines the communication between the frontend and the backend. Both of these remote procedure calls are hidden to the user by the python frontend.

```
service Chisel4mlService {
  rpc GenerateCircuit (GenerateCircuitParams) returns (
     GenerateCircuitReturn) {}
  rpc RunSimulation (RunSimulationParams) returns (
     RunSimulationReturn) {}
}
message GenerateCircuitParams {
    message Options {
        bool isSimple = 1;
        bool pipelineCircuit = 2;
    }
    lbir.Model model = 1;
    Options options = 2;
    bool useVerilator = 3;
    bool genVcd = 4;
    uint32 generationTimeoutSec = 5;
}
message GenerateCircuitReturn {
    message ErrorMsg {
        enum ErrorId {
            SUCCESS = 0;
            FAIL    = 1;
        }
        ErrorId errId = 1;
        string msg = 2;
    }
    uint32 circuitId = 1;
    ErrorMsg err = 2;
}
message RunSimulationParams {
    uint32 circuitId = 1;
    repeated lbir.QTensor inputs = 2;
}
message RunSimulationReturn {
    repeated lbir.QTensor values = 1;
}
```

Listing 2: Protocol buffers definition file of the Chisel backend services.

A. Chisel backend

The Chisel backend is implemented as a server that accepts LBIR neural network description, generates the neural network hardware description (structure), stores it into internal storage, and returns the hardware description ID to the frontend for further tests and simulations. Optionally, the user can specify that the backend stores the hardware simulation waveforms (vcd files) into local storage, which can be manually inspected with a waveform viewing tool, such as GTKWave.

The core of the chisel4ml backend are $Neuron$ object, $ProcessingElement$ class, and $ProcessingPipeline$ class. These represent the neurons, layers, and the entire neural network, respectively.

```
object Neuron {
    def apply[I <: Bits,
              W <: Bits : WeightsProvider,
              M <: Bits,
              A <: Bits : ThreshProvider,
              O <: Bits](in: Seq[I],
                         weights: Seq[W],
                         thresh: A,
                         mul: (I, W) => M,
                         add: Vec[M] => A,
                         actFn: (A, A) => O,
                         shift: Int): O = {
        val muls = VecInit((in zip weights).map{
            case (a,b) => mul(a,b)
        })
        val pAct = add(muls)
        val sAct = (pAct << shift.abs).asTypeOf(pAct)
        actFn(sAct, thresh)
    }
}
```

Listing 3: An abridged version of a neuron in Chisel.

Listing 3 shows an abridged version of neuron object in Chisel. We use Scalas parameterizability extensively, to produce a completely generic neuron. The neuron object takes a sequence of inputs in, sequence of weights $weights$, a threshold value (inverse of a bias value) $thresh$, a multiplication function mul, an addition function add, a activation function $actFn$, and a shift value $shift$. All the inputs are also parameterized by type. This allows us to implement several different types of quantization easily. For instance a multiplication function for signed integers looks like this:

```
def mul(i: SInt, w: SInt): SInt = {
    if(w.litValue == 0.S.litValue) {
        0.S
    } else {
        i * w
    }
}
```

While a multiplication for binarized neural networks looks like this:

```
    def mul(i: Bool, w: Bool): Bool = ~ (i ^ w)
```

Combining the neurons in fully-connected layers is simple in Chisel. Listing 4 shows an abridged version of an processing element in Chisel. It defines inputs and outputs that are a single unified wire, but internally they get broken apart into seperate inputs and outputs by constructing a vector from them. The Neuron object, defined in Listing 3 is then used to construct the individual neurons, and connect them to an appropriate output. Finally, we concatenate the output vector back into a single large wire, that forms the output of the processing element.

```
class ProcessingElement[...](layer: lbir.Layer, ...)
extends Module{
    val io = IO(new Bundle {
        val in  = Input(UInt(inputLayerBitwidth.W))
        val out = Output(UInt(outputLayerBitwidth.W))
    })

    val in_int  = Wire(Vec(inputSize, typeI))
    val out_int = Wire(Vec(outputSize, typeO))
    in_int := io.in.asTypeOf(in_int)
    for (i <- 0 until outputSize) {
        out_int(i) := Neuron[I, W, M, A, O](in_int,
                                            weights(i),
                                            thresh(i),
                                            mul,
                                            add,
                                            actFn,
                                            shift(i))
    }
    io.out := Cat(out_int.reverse)
}
```

Listing 4: An abridged version of a processing element (or layer) in Chisel.

The processing pipeline is the highest level hardware module in chisel4ml. It represents the entire neural network. Listing 5 shows an abridged version of the processing pipeline implementation.

```
class ProcessingPipeline(model: lbir.Model, ...)
extends Module {
    val io = IO(new Bundle {
        val in  = Input(UInt(inputBitwidth.W))
        val out = Output(UInt(outputBitwidth.W))
    })

    // Construct the layers
    val peList = new ListBuffer[ProcessingElementSimple]()
    for (layer <- model.layers) {
        peList += Module(ProcessingElement(layer))
    }

    // Connect the inputs and outputs of the layers
    peList(0).io.in := io.in
    for (i <- 1 until model.layers.length) {
        peList(i).io.in := RegNext(peList(i - 1).io.out)
    }
    io.out := peList.last.io.out
}
```

Listing 5: An abridged version of a processing pipeline in Chisel.

B. Python frontend

The input to the python frontend is a model defined in QKeras. The frontend transforms it into LBIR and hands it to the Chisel backend which then generates the hardware.

The main operations of the python frontend are:

1) **Optimization of the QKeras neural network model** This optional operation performs folding of batch-normalization layers.
2) **Generation of hardware description** In this operation (step) the QKeras neural network model is translated into LBIR which is sent to the Chisel backend. The returned hardware description ID is recorded for future simulations, tests, and packaging of the generated hardware description.
3) **Simulation of generated hardware description** It performs the RTL simulation of the neural network hardware description on the Chisel backend and returns simulation results in numpy arrray format. Simulated results can be easily compared to expected results in order to verify the correct operation of the generated neural network hardware.
4) **Packaging the hardware description.** The generated hardware description is exported to a verilog file in a local storage.

Listing 6 demonstrates the main functionality of the chisel4ml frontend as a python code snippet.

```
from chisel4ml import optimize, generate

opt_model = optimize.qkeras_model(model)
circuit = generate.circuit(opt_model)
res = circuit.predict(X_data)
circuit.package(directory="/my/pkg/dir")
```

Listing 6: The chisel4ml python frontend.

IV. Results

In the paper a 4-layer neural network with 16 input and 5 outputs was studied. The network has 64, 32, 32 and 5 neurons in each of the layers, and every layer uses the ReLU activation, with the exception of the last layer, that uses the softmax activation. The layers do not use biases. In all cases the inputs are quantized to 11 bits, but all other layer parameters are quantized to n-bits; where n varies from 2 to 8 bits. Listing 1 shows such a model definition for $n = 6$. Additionally, all models use prunning, where around 75% of the parameters are removed. Note that the definition of the hls4ml model differs slightly, because of the way hls4ml interprets the models, but the number and size of all parameters and input features are equal. The hls4ml tool allows the user to select a "reuse" factor, which controls the unrolling of the computation. We set this factor to 1, which means it will generate a fully parallel circuit in a same way chisel4ml does. For more details please refer to our code on git repository: https://github.com/jurevreca12/qat_lhc_jets_hlf.

Table I shows the synthesis results of the models generated by chisel4ml 0.1.4 and hls4ml 0.6.0. The synthesis was performed using Xilinx Vivado 2019.2. In each case we targeted a high-end Xiliinx FPGA: xcvu9p-flga2104-2L-e. The columns in the table are:

- the $Bitwidth$ column signifies the number of bits the parameters and the input features were quantized to,
- LUT column gives the number of lookup tables used by the design,
- FF the number of flip-flops,
- DSP the number of digital-signal processing blocks used,
- $BRAM18$ the number of 18-kbit block rams used,
- $CLOCK[ns]$ specifies the maximum path delay in nanoseconds,
- $CLOCK[cycles]$ column specifies number of pipeline stages and thus the number of cycles it takes to compute any single input,
- $DELAY[ns]$ is the number of nanoseconds to compute one inference ($DELAY[ns] = CLOCK[ns] \cdot CLOCK[cycles]$), and finally
- the $FREQ[MHz]$ column specifies the maximum achievable frequency, but also, since the designs can compute a result each cycle, the maximum throughput.

We also provide the information in graph form. Fig. 5 and Fig. 6 plot the number of lookup tables and fliip-flops used by chisel4ml and hls4ml for the various bitwidths. Chisel4ml designs use similar amount of lookup tables as hls4ml for lower bitwidths. For larger bitwidths, however, chisel4ml uses more lookup tables. However, chisel4ml uses less flip-flops then hls4ml for equal bitwidth designs.

Fig. 7 and Fig. 8 show the total delay and the maximum frequency of the circuits, respectively. In general, chisel4ml circuits have a lower delay then comparable hls4ml designs. Conversely, the maximum achievable frequency is higher in most hls4ml designs, since they focus on adding additional pipeline stages. It would be possible to improve the timing performance of chisel4ml generated circuits by adding additional pipeline registers.

Table II shows the accuracy achieved by the various models we trained on the hls4ml_lhc_jets_hlf dataset [2]. We list both the accuracy achieved in the QKeras framework, as well as the accuracy achieved by RTL simulation. The discrepancies in QKeras and RTL accuracy stem from the slight differences in approximating fixed-point arithmetic with floating-point

TABLE I: Synthesis results.

Bitwidth	Tool	LUT	FF	DSP	BRAM18	CLOCK [ns]	CLOCK [cycles]	DELAY [ns]	FREQ [MHz]
2	chisel4ml	3062	127	0	0	2.478	4	9.912	403.55
	hls4ml	2132	379	5	3	3.591	5	17.955	278.47
3	chisel4ml	6005	267	0	0	2.822	4	11.288	354.36
	hls4ml	4934	535	5	3	3.694	5	18.47	270.71
4	chisel4ml	10164	362	0	0	5.324	4	21.296	187.83
	hls4ml	7392	656	5	3	3.694	5	18.47	270.71
5	chisel4ml	14522	485	0	0	5.46	4	21.84	183.15
	hls4ml	8848	1034	5	3	3.694	6	22.164	270.71
6	chisel4ml	13674	581	210	0	5.901	4	23.604	169.46
	hls4ml	9840	1633	44	3	3.694	7	25.858	270.71
7	chisel4ml	18648	916	260	0	5.75	4	23.0	173.91
	hls4ml	11783	1945	81	3	4.112	8	32.896	243.19
8	chisel4ml	23881	801	281	0	6.368	4	25.472	157.04
	hls4ml	13915	2058	125	3	4.074	8	32.592	245.46

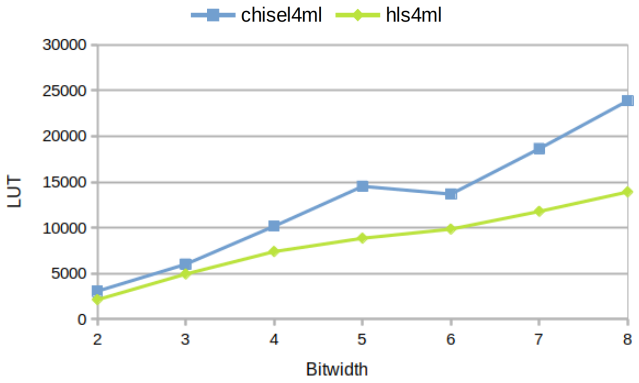

Fig. 5: Number of lookup tables used by bitwidth.

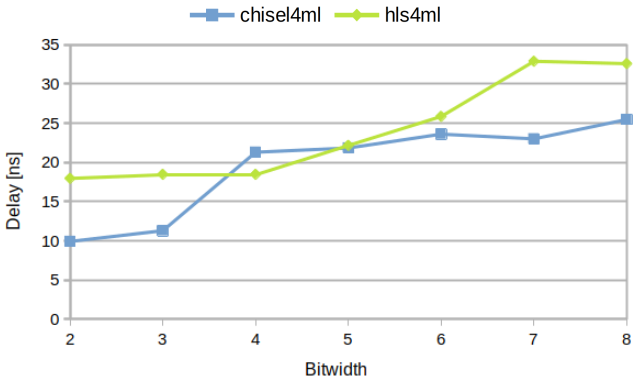

Fig. 7: Total delay of the circuit by bitwidth.

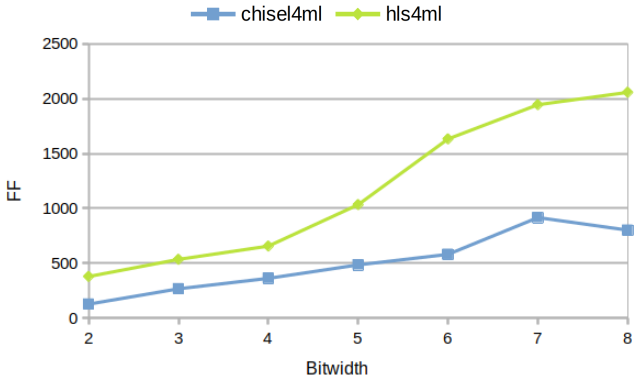

Fig. 6: Number of flip-flops used by bitwidth.

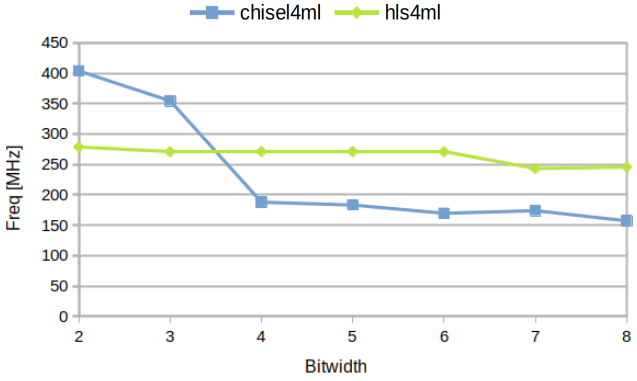

Fig. 8: Maximum achivable frequency by bitwidth.

arithmetic. The *-PY columns show the accuracy achieved in the QKeras framework, and *-RTL the accuracy achieved by simulating the generated RTL (where C4ML is short for chisel4ml). The accuracy discrepancy is lower in chisel4ml, except for the 2 bit case. We also trained a full-precision model without quantization and pruning. It achieved 0.76 accuracy, which is only slightly higher then the quantized versions.

Finally, the generation time between the two tools differs greatly. The average RTL generation time for chisel4ml was around 19 seconds, while the average hls4ml generation time was 7.5 minutes.

TABLE II: Achieved accuracy of the models.

Bitwidth	C4ML-PY	C4ML-RTL	HLS4ML-Q	HLS4ML-RTL
2	0.62	0.42	0.71	0.70
3	0.67	0.67	0.73	0.73
4	0.73	0.72	0.75	0.74
5	0.74	0.73	0.76	0.70
6	0.74	0.73	0.76	0.70
7	0.74	0.73	0.76	0.68
8	0.73	0.72	0.76	0.67

V. Conclusion

We presented chisel4ml a framework for generating highly parallel FPGA implementations of quantized neural networks defined in QKeras. Chisel and Scala allowed us to write the generator very concisely, and still achieve comparable performance with hls4ml, and in some cases, like the total delay, surpassing the performance of hls4ml. Additionally, the speed of RTL generation is substantialy greater. This allows for faster development and debugging. One major drawback of chisel4ml, however, is that we only support fully parallelized circuits. We plan to address this by adding support for sequential processing elements, that will be able to compute the given computation in a sequential manner. We also plan to add support for other layer types, like convolution, as well as the softmax activation function.

Acknowledgment

References

[1] Nour Moustafa and Jill Slay. "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)". In: *2015 Military Communications and Information Systems Conference (MilCIS)*. 2015, pp. 1–6. DOI: 10.1109/MilCIS.2015.7348942.

[2] J. Duarte et al. "Fast inference of deep neural networks in FPGAs for particle physics". In: *Journal of Instrumentation* 13.07 (July 2018), P07027. DOI: 10.1088/1748-0221/13/07/P07027. URL: https://dx.doi.org/10.1088/1748-0221/13/07/P07027.

[3] Thea Aarrestad et al. "Fast convolutional neural networks on FPGAs with hls4ml". In: *Machine Learning: Science and Technology* 2 (Dec. 2021).

[4] Itay Hubara et al. "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations". In: *J. Mach. Learn. Res.* 18.1 (Jan. 2017), pp. 6869–6898. ISSN: 1532-4435.

[5] Yann Lecun, John Denker, and Sara Solla. "Optimal Brain Damage". In: vol. 2. Jan. 1989, pp. 598–605.

[6] Itay Hubara et al. "Binarized neural networks". In: *Advances in Neural Information Processing Systems* (NIPS 2016), pp. 4114–4122. ISSN: 10495258.

[7] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. *Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning*. 2017. arXiv: 1611.05128 [cs.CV].

[8] Yaman Umuroglu et al. "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '17. Monterey, California, USA: Association for Computing Machinery, 2017, pp. 65–74. ISBN: 9781450343541. DOI: 10.1145/3020078.3021744. URL: https://doi.org/10.1145/3020078.3021744.

[9] Syed Asad Alam et al. "On the RTL Implementation of FINN Matrix Vector Unit". In: *ACM Trans. Embed. Comput. Syst.* (July 2022). Just Accepted. ISSN: 1539-9087. DOI: 10.1145/3547141. URL: https://doi.org/10.1145/3547141.

[10] Neil Burgess et al. "Bfloat16 Processing for Neural Networks". In: *2019 IEEE 26th Symposium on Computer Arithmetic (ARITH)*. 2019, pp. 88–91. DOI: 10.1109/ARITH.2019.00022.

[11] Claudionor N. Coelho et al. "Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors". In: *Nature Machine Intelligence* 3.8 (Aug. 2021), pp. 675–686. ISSN: 2522-5839. DOI: 10.1038/s42256-021-00356-5. URL: https://doi.org/10.1038/s42256-021-00356-5.