FAST CONVOLUTIONAL NEURAL NETWORKS ON FPGAS WITH HLS4ML

Thea Aarrestad, Vladimir Loncar[*], Nicolò Ghielmetti[†], Maurizio Pierini, Sioni Summers
European Organization for Nuclear Research (CERN)
CH-1211 Geneva 23, Switzerland

Jennifer Ngadiuba
California Institute of Technology
Pasadena, CA 91125, USA

Christoffer Petersson[‡], Hampus Linander
Zenseact
Gothenburg, 41756, Sweden

Yutaro Iiyama
ICEPP, University of Tokyo
Tokyo, Japan

Giuseppe Di Guglielmo
Columbia University, New York,
NY 10027, USA

Javier Duarte
University of California San Diego
La Jolla, CA 92093, USA

Philip Harris, Dylan Rankin
Massachusetts Institute of Technology
Cambridge, MA 02139, USA

Sergo Jindariani, Kevin Pedro, Nhan Tran
Fermi National Accelerator Laboratory
Batavia, IL 60510, USA

Mia Liu
Purdue University
West Lafayette, IN 47907, USA

Edward Kreinar
HawkEye360
Herndon, VA 20170, USA

Zhenbin Wu
University of Illinois at Chicago
Chicago, IL 60607, USA

Duc Hoang
Rhodes College
Memphis, TN 38112, USA

April 30, 2021

ABSTRACT

We introduce an automated tool for deploying ultra low-latency, low-power deep neural networks with convolutional layers on FPGAs. By extending the `hls4ml` library, we demonstrate an inference latency of 5 μs using convolutional architectures, targeting microsecond latency applications like those at the CERN Large Hadron Collider. Considering benchmark models trained on the Street View House Numbers Dataset, we demonstrate various methods for model compression in order to fit the computational constraints of a typical FPGA device used in trigger and data acquisition systems of particle detectors. In particular, we discuss pruning and quantization-aware training, and demonstrate how resource utilization can be significantly reduced with little to no loss in model accuracy. We

[*] Also at Institute of Physics Belgrade, Serbia
[†] Also at Politecnico di Milano, Italy
[‡] Also at Chalmers University of Technology, Sweden

show that the FPGA critical resource consumption can be reduced by 97% with zero loss in model accuracy, and by 99% when tolerating a 6% accuracy degradation.

***K*eywords** deep learning · FPGA · convolutional neural network

1 Introduction

The `hls4ml` library [1, 2] is an open source software designed to facilitate the deployment of machine learning (ML) models on field-programmable gate arrays (FPGAs), targeting low-latency and low-power edge applications. Taking as input a neural network model, `hls4ml` generates `C/C++` code designed to be transpiled into FPGA firmware by processing it with a high-level synthesis (HLS) library. The development of `hls4ml` was historically driven by the need to integrate ML algorithms in the first stage of the real-time data processing of particle physics experiments operating at the CERN Large Hadron Collider (LHC). The LHC produces high-energy proton collisions (or *events*) every 25 ns, each consisting of about 1 MB of raw data. Since this throughput is overwhelming for the currently available processing and storage resources, the LHC experiments run a real-time event selection system, the so-called Level-1 trigger (L1T), to reduce the event rate from 40 MHz to 100 kHz [3–6]. Due to the size of the buffering system, the L1T system operates with a fixed latency of $\mathcal{O}(1\ \mu s)$. While `hls4ml` excels as a tool to automatically generate low-latency ML firmware for L1T applications, it also offers interesting opportunities for edge-computing applications beyond particle physics whenever efficient, e.g. low power or low latency, on-sensor edge processing is required.

The `hls4ml` software is structured with a set of different back-ends, each supporting a different HLS library and targeting different FPGA vendors. So far, new development has been focused on the Vivado HLS [7] back-end targeting Xilinx FPGAs. We have demonstrated this workflow for fully-connected, or dense, neural networks (DNNs) [1], binary and ternary networks [8], boosted decision trees [9], and graph neural networks [10, 11]. The `hls4ml` library accepts models from TensorFlow [12], Keras [13], PyTorch [14], and via the ONNX interface [15]. It has recently been interfaced to QKeras [16], in order to support quantization-aware training (QAT) allowing the user to better balance resource utilization and accuracy.

The `hls4ml` design focuses on fully-on-chip deployment of neural network architectures. This avoids the latency overhead incurred by data transmission between the embedded processing elements and off-chip memory, reducing the overall inference latency. Conversely, this approach constrains the size and complexity of the models that the HLS conversion can easily support. Nevertheless, complex architectures can be supported, as discussed in [10, 11] in the case of graph neural networks.

In this paper, we introduce support for convolutional neural networks (CNNs), through the implementation of streaming-based novel convolutional and pooling layers.

Given the larger number of operations associated to each convolutional layer, a successful deployment on FPGA relies on model compression, through pruning and quantization. The `hls4ml` library supports both these forms of compression through removal of all zero-multiplications during the firmware implementation (a feature of HLS we take advantage of when designing the layer implementation), and through its interface with QKeras [16].

We demonstrate the QKeras+`hls4ml` workflow on a digit classifier trained on the Street View House Numbers (SVHN) dataset [17], with a depth and input size appropriate for the latency- and resource-restricted triggering systems at LHC.

This paper is organized as follows: Section 2 describes related works. Section 3 introduces the stream-based implementation of CNN layers; Section 4 describes the SVHN dataset. The benchmark model is introduced in Section 5, while results obtained by pruning and quantization (after and during training) are presented in Sections 6 and 7, respectively. Section 8 discusses the model porting to FPGAs. Conclusions are given in Section 9.

2 Related work

An early attempt to deploy convolutional neural networks (CNNs) on FPGAs for particle physics was shown in [18], and surveys of other existing toolflows for mapping CNNs on FPGAs are given in [19–22]. The FINN [23, 24] framework from Xilinx Research Labs is designed to explore quantized CNN inference on FPGAs, with emphasis on generating dataflow-style architectures customized for each network. It includes tools for training quantized NNs such as Brevitas [25], the FINN compiler, and the finn-hlslib Vivado HLS library of FPGA components for QNNs. The fpgaConvNet library [26–29] converts CNNs specified in Caffe [30] or Torch formats into generated Xilinx Vivado HLS code with a streaming architecture. FP-DNN [31] is a framework that takes TensorFlow [12]-described CNNs as input, and generates the hardware implementations on FPGA boards with RTL-HLS hybrid templates. DNNWeaver [32]

is an open-source alternative, which also supports CNNs specified in Caffe format and automatically generates the accelerator Verilog code using hand-optimized Verilog templates with a high degree of portability. Caffeine [33] is another CNN accelerator for Caffe-specified models targeting Xilinx devices that support a co-processing environment with a PCIe interface between the FPGA and a host. Snowflake [34] is a scalable and efficient CNN accelerator with models specified in Torch [35] and a single, sequential computation architecture designed to perform at near-peak hardware utilization targeting Xilinx system-on-chips (SoCs). In [36], an FPGA-based accelerator design to execute CNNs is proposed, leveraging TENSORFLOW for model description and exploiting reuse along all dimensions with a 1D systolic array of processing elements. The NullHop [37] accelerator architecture takes advantage of sparse computation in convolutional layers to significantly speed up inference times. A flexible, efficient 3D neuron array architecture for CNNs on FPGAs is presented in [38], describing a technique to optimize its parameters including on-chip buffer sizes for a given set of resource constraint for modern FPGAs. Vitis AI [39] is Xilinx's development platform for AI inference on Xilinx hardware platforms, consisting of optimized IP cores, tools, libraries, models, and example designs for both edge devices and Alveo cards.

Our approach is distinct from many of those above with its emphasis on being a completely open-source and multi-backend tool. In addition, a fully on-chip design is embraced in order to target the microsecond latency imposed in LHC physics experiments.

3 Convolutional layers implementation in hls4ml

A direct implementation of a two-dimensional convolutional layer (Conv2D) requires six nested loops over image height H, width W, number of input channels C, number of output filters N, and filter height J and width K [22]. In particular, calculating one element of the $V \times U \times N$ output tensor $\boldsymbol{Y}$ of a Conv2D layer from the $H \times W \times C$ input tensor $\boldsymbol{X}$, $J \times K \times C \times N$ weight tensor $\boldsymbol{W}$, and length-N bias vector $\boldsymbol{\beta}$ requires three nested loops[4],

$$\boldsymbol{Y}[v,u,n] = \boldsymbol{\beta}[n] + \sum_{c=1}^{C}\sum_{j=1}^{J}\sum_{k=1}^{K} \boldsymbol{X}[v+j, u+k, c]\, \boldsymbol{W}[j,k,c,n]\,, \tag{1}$$

and repeating the calculation for all output elements $\{u,v,n\} \in \times [1,V] \times [1,U] \times [1,N]$ requires three additional nested loops. For simplicity, we assume $J = K$ (square kernel) in the remainder of this paper.

Without additional optimizations, a plain implementation of these nested loops would result in high latency because, in the register transfer level (RTL) implementation, one clock cycle is required to move from an outer loop to an inner loop, and another one to move from an inner loop to an outer loop. This is usually addressed with loop pipelining. However, pipelining an outer loop requires completely parallelizing (or *unrolling*) all nested inner loops, which significantly increases the size of the RTL implementation and the resources used. This approach is then feasible only for very small input sizes or model architectures. Utilizing this direct approach, the total number of unrolled loop iterations (the product K^2VUN) was limited to be less 4,096 to avoid the Vivado HLS partitioning limit.

While the direct implementation has the advantage of not requiring extra memory for temporary storage, on most modern computing architectures convolution is implemented using general matrix multiplication (GEMM), with algorithms like `im2col` and `kn2row` [40]. In im2col, each input window is flattened into a column vector and stacked together to form the input matrix, while the kernels are flattened into row vectors and concatenated to form the weight matrix. Matrix multiplication can then be performed using the accelerated library available on the platform, for example using the routines from basic linear algebra subprograms (BLAS). While this approach can be implemented on FPGAs using HLS, the design choices of `hls4ml`, particularly the decision to store all tensors on the chip itself, mean this approach requires additional $\mathcal{O}(K^2HWC)$ units of memory, either as block random access memory (BRAM) or registers, to store the input matrix. Additionally, to achieve the lowest latency, `hls4ml` completely partitions the input arrays into individual registers as this allows access to each element within the same clock cycle. This strategy works well for fully connected layers but in case of convolutional layers the input tensor is usually much larger. Following this strategy, one would quickly reach the partitioning limit of Vivado HLS. Relaxing this constraint and using block or cyclic partitioning to create multiple array slices presents another challenge as the access pattern between consecutive layers of the model has to be consistent, otherwise scheduling issues arise and the design may fail to meet timing constraints.

To avoid these limitations, we have implemented convolutional layers using streams. Streams are synthesized in hardware as first in, first out (FIFO) buffers and as they do not require additional address management, they consume less resources than designs based on arrays. Since streams only allow sequential access, and have additional limitations on the reads and writes from different tasks (`C++` functions), this requires re-implementing most of the neural network layers in `hls4ml` to support sequential processing.

[4]Note that in practice $\boldsymbol{X}$ in equation (1) is shifted by e.g. $\left(\frac{J+1}{2}, \frac{K+1}{2}\right)$ in order to be symmetric around (v,u).

Our implementation uses an approach similar to the im2col algorithm. However, it does not build the entire input matrix, and rather considers one column vector at a time. This allows us to reuse the existing matrix-vector multiplication functions of `hls4ml`. In order to use streams for this implementation, a special `C++` class `hls::stream<>` provided in Vivado HLS is used. Given an $H \times W \times C$ input image tensor, we create a stream of HW items where each item is an array containing the C elements. This scheme allows us to efficiently read from the stream and construct column vectors. Because it usually takes one cycle to read or write one element of the stream, the latency of the layer will be at least HW cycles.

Processing input sequentially through streams requires that we buffer all values that we wish to reuse at a later stage as an internal state. For a two-dimensional convolution, we need to buffer all values between the first and last element of the convolutional kernel, or *sliding window*, as shown on the left in Fig. 1. The buffer can be defined as an array in `C++` and implemented by the HLS compiler as a shift register, however this approach requires keeping track of the position in the array, further complicating the implementation. We choose a simpler approach using streams. We create K^2 streams, corresponding to the size of the sliding window, and buffer values at the appropriate position in the window as they stream in. The depth of these streams is determined by the width of the output image and the square kernel size. Once we reach an element that is at the last position of a sliding window, we can compute one output by reading from the buffer. This is highly efficient as we can read the entire column vector in one clock cycle. With the column vector prepared, we can invoke the multiplication with the weight matrix and store the result in the output stream.

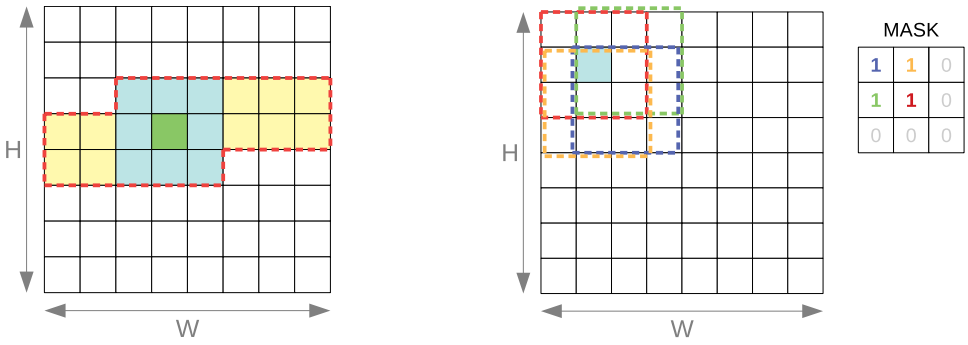

Figure 1: The image on the left shows an illustration of the sliding window buffer. All elements (yellow) of one kernel window (blue) are buffered to compute one output element (green). The right image shows the computation of the binary mask (an instruction) for one input element. The highlighted element (light blue) contributes four times to the sliding window in different positions, with the mask having bits set at the appropriate locations. The bits of the mask are concatenated and stored in a 9-bit unsigned integer, in this example the number 27 (000011011 in binary).

While the algorithm described so far allows us to process larger inputs than a plain implementation would, significant resources are allocated for accounting, e.g. the position of the element in the sliding window or handling of the corners of the input image, and this prevents pipelining of loops at the desired latency. We address this issue by eliminating all branching code that handles these special cases, along with their associated state variables, leaving only the internal sliding window buffer. Instead, we pre-compute the positions in the sliding window where a given input element is used, and store this information as a binary mask, represented as a K^2-bit unsigned integer. In the mask we set bits corresponding to every position in the sliding window where the input element is used, and leave the remaining bits unset (equal to 0), as illustrated on the right in Fig. 1. This mask can be used as an instruction on how to populate the sliding window buffer, eliminating the need for all branching code. The procedure is applied to every element of the input image, and stored in the instruction array. The instruction array can be significantly compressed by eliminating duplicates and translating the position of the element in the input array to the compressed array. As an example of the compression scheme, Fig. 2 illustrates how every convolution with a 3×3 kernel and unit stride can be represented with only $H' \times W' = 5 \times 5$ instructions, regardless of the input image size ($H \times W$).

For the pooling layers, a similar technique is used to collect the data in sliding window buffers. As the most common form of pooling assumes a stride equal to the pooling region size (i.e. no overlaps between pooled regions), we create a simpler and more optimal instruction-encoding scheme for this case. Unlike convolution, in both max and average pooling operation, we do not need the position of elements in the sliding window, only which window they belong to. This allows us to create a simple lookup table of $H + W$ elements without the need for translation of the position of the input element.

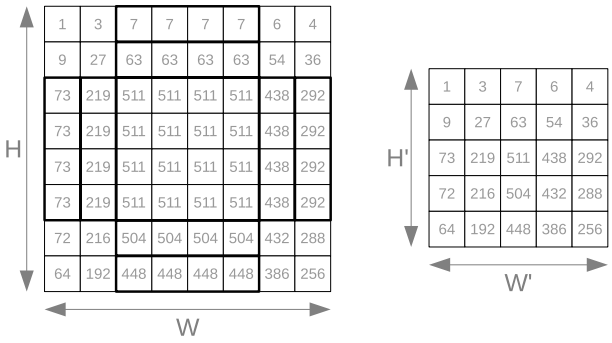

Figure 2: Example of compression of the instruction array. The left image shows the binary mask corresponding to each pixel, here represented as an integer rather than as a bit sequence. The shown values are specific of a 3×3 kernel with unit stride. Instruction duplicates are highlighted by the bold-line rectangles. On the right image, we reduce the instruction array by computing the instruction array of a 5×5 image, which has no duplicates, and translating the position of the element in the input array to the compressed array.

4 Dataset

To demonstrate the functionality of CNNs in `hls4ml`, we consider as a benchmark example a digit classifier trained on the Street View House Numbers (SVHN) Dataset [17]. The SVHN dataset consists of cropped real-world images of house numbers extracted from Google Street View images, in a format similar to that of the MNIST [41] dataset. However, it presents a much more challenging real-world problem, as illustrated by the examples shown in Fig. 3. The numbers are part of natural scene images, possibly with other digits appearing as a background on the two sides of the central one, different colors and focus, orientation, etc.

All the images are in RGB format and have been cropped to 32×32 pixels. Unlike MNIST, more than one digit can be present in the same image. In these cases, the center digit is used to assign a label to the image, which is then used as ground truth when training the classifier. Each image can belong to one of 10 classes, corresponding to digits "0" through "9." As a preprocessing step, we divide each pixel by the max RGB value of 255 in order to have numbers in the range between zero and one. We then standardize the input images to have a mean of zero and unit variance by applying a per-pixel scaling factor computed from the full training dataset. The same scaling is applied to the test set.

The SVHN dataset consists of 604,388 images for training (of which 531,131 are considered extra data that are slightly easier to classify) and 26,032 images for testing.

Training is performed using a k-fold cross-validation procedure. The training dataset is split in 10 training and validation samples such that 10% of the training set is used for validation and the remaining 90% for training. Training and validation is then repeated k times until each fold of the training set has been used to evaluate the accuracy of the model. Model-performance figures of merit (e.g., accuracy, true and false positive rates, etc.) are defined considering the mean across the 10 folds on the test set. The corresponding uncertainty is quantified through the standard deviation across the 10 folds.

Figure 3: Examples of digit images extracted from the SVHN train (three leftmost images) and test (three rightmost images) datasets.

5 Baseline model

Keeping in mind that the model is designed for deployment on the resource limited environment of an FPGA, we limit the depth and complexity of the baseline model while preserving reasonable performance. As a target, we aimed at a test error close to 5%, where state-of-the-art test error lies between 1–5% [42–47]. To reduce the overall model latency

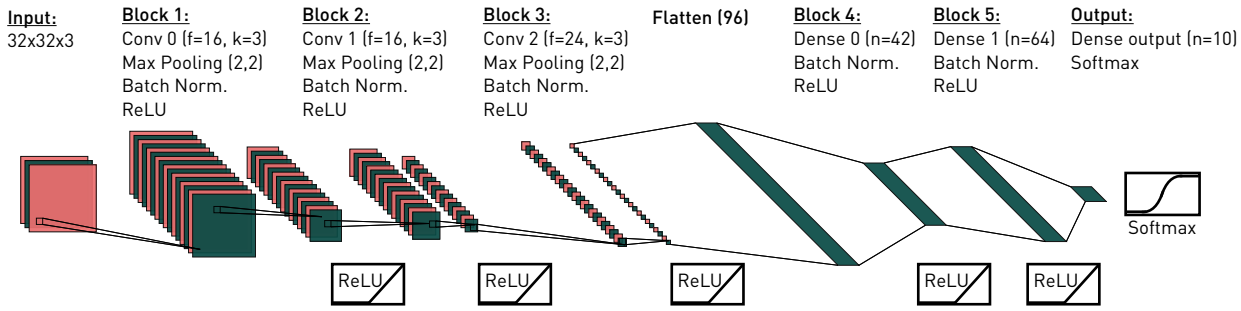

Figure 4: The neural network architecture, chosen through a Bayesian optimization over the hyperparameters, for classifying digits from the SVHN dataset. Each convolutional block consists of a convolutional layer, max pooling, batch normalization, and ReLU activation. The convolutional layers in the three convolutional blocks use 16, 16, and 24 filters, respectively, and each has a kernel size of 3×3. The pooling layers have a size of 2×2. The convolutional blocks are followed by two fully-connected layers consisting of 42 and 64 neurons, with batch normalization and ReLU activation. The bias term is removed from all layers except the final output layer.

Table 1: Number of trainable weights, floating-point operations, energy consumption and layer size in bits for each convolutional or dense layer (not including the activation layers). Batch normalization and pooling layers are not included as they are negligible in size and energy consumption in comparison. The energy is estimated assuming a 45 nm process using QTOOLS. The total energy and bit size includes all model layers.

Layer name	Layer type	Input shape	Weights	MFLOPs	Energy [nJ]	Bit size
`Conv 0`	Conv2D	(32, 32, 3)	432	0.778	1,795	3,456
`Conv 1`	Conv2D	(15, 15, 16)	2,304	0.779	1,802	18,432
`Conv 2`	Conv2D	(6, 6, 16)	3,456	0.110	262	27,648
`Dense 0`	Dense	(96)	4,032	0.008	26	32,256
`Dense 1`	Dense	(42)	2,688	0.005	17	21,504
`Output`	Dense	(64)	65	0.001	4	5,200
Model total			**12,858**	**1.71**	**3,918**	**170,816**

as much as possible, models with fewer large layers (wider) are preferred over models with several smaller layers (deeper). This is due to the parallel nature of the FPGA, making it more resource-efficient to process one large layer in parallel over several small ones sequentially. The dependency of inference latency and resource consumption for increasing depth and width will be further discussed in Section 8.

A Bayesian optimization over the model hyperparameters is performed using KERAS TUNER [48]. The first few layers are chosen to be 2D convolutional blocks. Each block consists of a convolutional layer followed by a max pooling layer, a batch normalization [49] layer, and a rectified linear unit (ReLU) [50, 51] activation function. The optimization range is set so that the maximum number of loop iterations per layer is below the unroll limit described in Section 3 in order to achieve the lowest possible latency. Pooling layers are used to keep the size of the final dense layers small.

The convolutional blocks are followed by a series of fully-connected layers, the amount of layers and their size again determined through the hyperparameter optimization. A final ten-node dense layer, activated by a softmax function, returns the probability for a given image to be assigned to each of the ten classes. The result of the Bayesian optimization, shown in Fig. 4, consists of three convolutional blocks and two dense layers. The convolutional layers in the three blocks have 16, 16, and 24 filters, respectively, and each has a kernel size of 3×3. The pooling layers have a size of 2×2. The two hidden dense layers consist of 42 and 64 neurons, with batch normalization and ReLU activation. The model is implemented in TENSORFLOW [12], using the KERAS API [13]. To reduce the number of required operations, the bias term is removed from all layers, except for the final output layer, while keeping batch normalization on to prevent internal covariate shift [49].

We refer to this model as the Baseline Floating-point (BF) model. The number of floating-point operations (FLOPs) and weights for each convolutional or dense layer is listed in Table 1. In addition, an estimate of the per-layer energy consumption and the layer size in bits is quoted. These estimates are obtained using QTOOLS [16], a library for estimating model size and energy consumption, assuming a 45 nm process [52]. Despite the first dense layer having the most weights, the number of FLOPs and the energy consumption is significantly higher in the convolutional layers due to the much larger number of multiply-accumulate operations performed. The per-layer summaries does not include

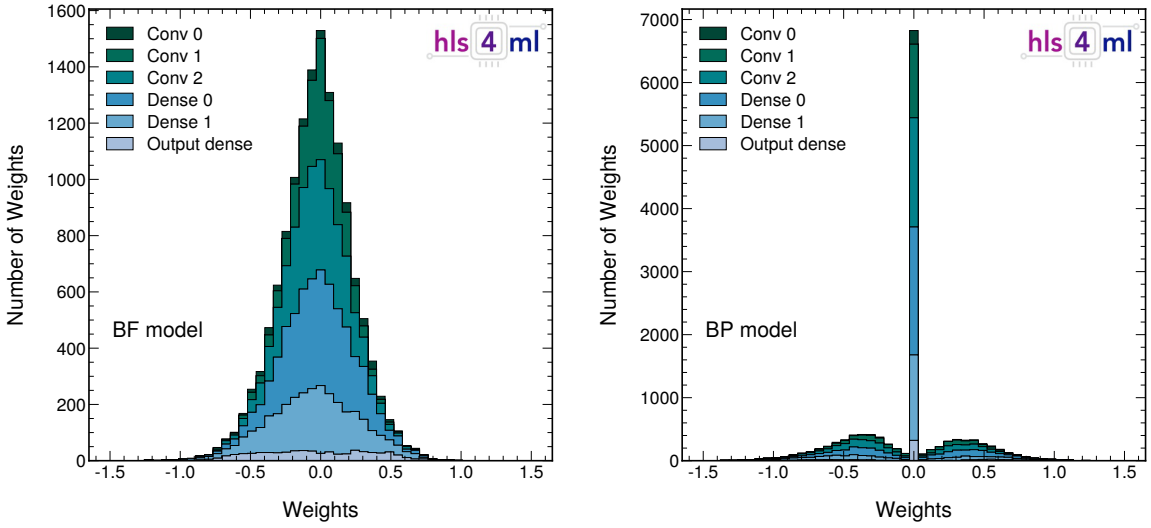

Figure 5: The weights per layer for the the Baseline Floating-point (BF) model (left) and the Baseline Pruned (BP) model (right). The BP model is derived by starting from the BF model and repeating the training while applying a pruning procedure with a target sparsity of 50% for each layer.

results for batch normalization or pooling layers, as the contribution from these are negligible in comparison. The total model energy and bit size includes all layers of the model.

The training is performed minimizing the categorical crossentropy loss [53] using the Adam optimizer [54]. The optimal learning rate is obtained using the hyperparameter optimization described above, found to be 0.003, and is set as the starting learning rate. If there is no improvement in the loss for five epochs, the learning rate is reduced by 90% until a minimum learning rate of 10^{-6} is reached. The batch size is 1,024 and the training takes at most 100 epochs. Early stopping is enabled when no improvement in the validation loss is observed across ten epochs.

6 Compression by pruning

Weight pruning is an established strategy to compress a neural network and consequently reducing its resource utilization. One strategy, magnitude-based pruning, consists of eliminating redundant weights in the weight tensors by setting the value of the smallest weights in a tensor to zero [55–59, 1]. All zero-weight multiplications are omitted by the HLS library when translating the network into firmware, consequently saving significant FPGA resources.

Pruning is enforced using the TENSORFLOW pruning API, a KERAS-based interface consisting of a simple drop-in replacement of KERAS layers. A sparsity of 50% is targeted, meaning only 50% of the weights are retained in the pruned layer and the remaining ones are set to zero. Before pruning, the weights of each layer are initialized to the weights of the corresponding model without pruning (i.e. *fine-tuning pruning*), ensuring the model is in a stable minimum before removing weights deemed unimportant. Each model is pruned starting from the 10th epoch, with the target sparsity gradually increasing to the desired 50% with a polynomial decay of the pruning rate [60].

By pruning the BF model layers as listed in Table 1 to a target sparsity of 50%, the number of FLOPs required when evaluating the model, can be significantly reduced. We refer to the resulting model as the Baseline Pruned (BP) model.

The distribution of the weight values per layer for the BF and BP models are shown in Figure 5. The effect of pruning is seen by comparing the two distributions: the smallest magnitude weights of the BF weight distribution migrate to the spike at zero in the BP weight distribution, while the two tails remain populated, with most of the weights falling in the interval $[-1.5, 1.5]$.

Figure 6 compares the classification performance of the BF and BP models. Specifically, it shows the receiver operating characteristic (ROC) curves and the corresponding area under the curve (AUC) for each digit. In addition, we consider the model accuracy, i.e. how often the predictions (after taking the $\arg\max$ of the output neurons) equals the labels. For each ROC, the solid line corresponds to the mean across the 10 folds and the uncertainty to the standard deviation.

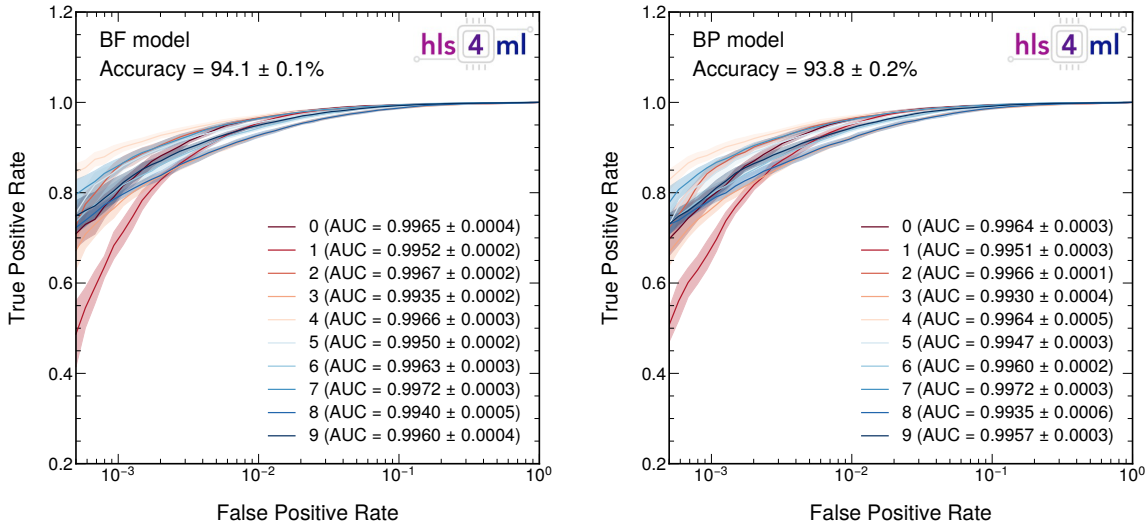

Figure 6: ROC curves of false positive rate (FPR) versus true positive rate (TPR) for the Baseline Floating-point (BF) model (left) and the Baseline Pruned (BP) model (right). Training is performed using k-fold cross validation, with $k = 10$. For each digit, the solid line corresponds to the mean and the band to the standard deviation across the 10 folds. The area under the curve (AUC) is reported in the legend and is defined as the mean across the ten folds with an uncertainty given by the standard deviation.

The mean accuracy and standard deviation across the 10 folds is also reported on the plot. Despite removing 50% of the weights for the BP model, the model accuracy is comparable between the BF and BP models.

These models serve as our reference models. The accuracy, latency and resource consumption of these will be discussed in Section 8. In general, we observe a significant reduction in FPGA resource consumption for pruned models, as zero-weight multiplications are optimized away by the HLS compiler. Because pruning has little impact on the model accuracy (as demonstrated in Figure 6), pruning is always recommended before translation into FPGA firmware with `hls4ml`.

7 Compression by quantization

To further limit the model footprint, we reduce the numerical precision of the model weights before FPGA deployment. During training, one typically relies on single- or double-precision floating-point arithmetic, i.e. 32 or 64 bit precision. However, when deploying a deep neural network on FPGA, reduced precision fixed-point arithmetic (quantization) is often used in order to minimize resource consumption and latency. It has been shown that deep neural networks experience little accuracy loss when QAT is applied, even up to binary quantization of weights [61].

When a quantized model is deployed on an FPGA, all its weights, biases, and activation functions are converted to fixed-point precision before being deployed. This is referred to as post-training quantization (PTQ). The chosen precision is a new tunable hyperparameter. The `hls4ml` library allows users to specify different numerical precisions for different components of the network (known as *heterogeneous quantization*). For instance, it is found that severe PTQ of the activation functions typically results in a greater reduction of accuracy than severe PTQ of the weights [8]. By default, `hls4ml` assumes 16 total bits for every layer, 6 of which are dedicated to the integer part ($\langle 16, 6 \rangle$ precision).

In this paper, we consider two approaches to network quantization: PTQ of a floating-point model, and QAT, resulting in a model already optimized for fixed-point precision. Both methods will be described in the following and the result on hardware discussed in detail in Section 8. To summarize, we observe a significant reduction in accuracy using PTQ, with no prediction power remaining below a bit width of 14. Using QAT, however, high accuracy is maintained down to extremely narrow bit widths of 3–4. The latency and resource consumption are similar for the two methods (with certain caveats that will be discussed in Section 8), and QAT is therefore the preferred solution for model quantization before deployment with `hls4ml`.

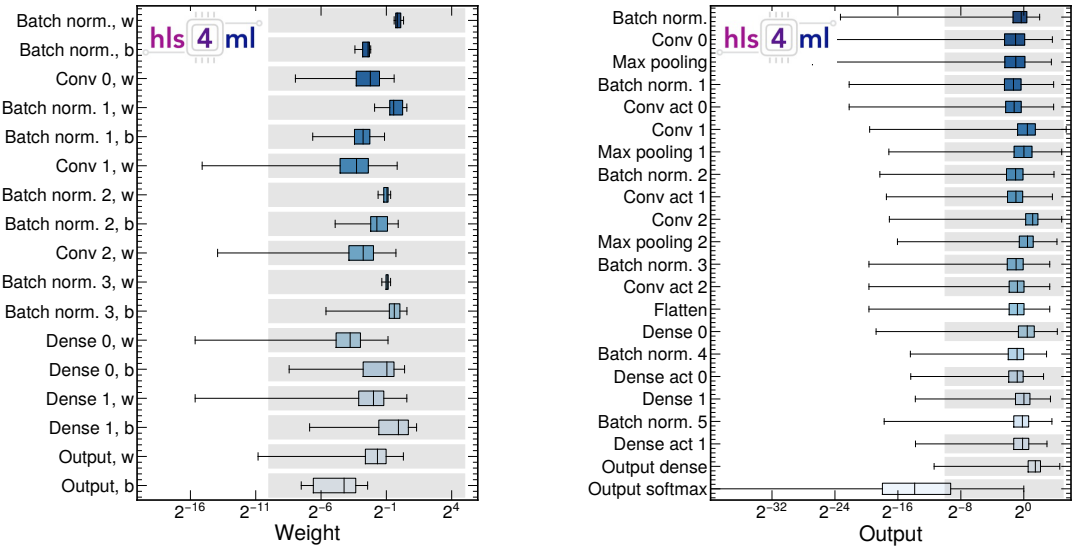

Figure 7: Layer weight (left) and output (right) numerical values per layer for the Baseline Floating-point (BF) model for a subset of the test data. The gray band represents the coverage of the default precision of $\langle 16, 6 \rangle$.

7.1 Post-training quantization

The `hls4ml` library converts model weights and biases from floating-point to fixed-point precision, applying the same quantization to the whole network or setting the precision per layer and per parameter type. Bit width and number of integer bits must be tuned carefully to prevent compromising the model accuracy. For each component, an appropriate precision is selected by studying the floating-point weight profiles, i.e. the range of input or output values spanned by the testing data for the trained model, component by component. In order to minimize the impact of quantization on accuracy, the precision can be tuned so that the numerical representation adequately covers the range of values observed in the floating-point activation profile.

As an example, the by-layer weight profiles of the BF model is shown in Fig. 7, for both layer weights (left) and outputs (right) using the testing data. The last letter in the label indicates which type of weight is being profiled, where w is for weights and b is for bias. Learnable bias parameters are only included in the final dense layer. The other bias terms are introduced by the fusing of a batch normalization and a dense layer. The grey bands illustrate the numerical range covered by the default $\langle 16, 6 \rangle$ precision. No gray band is visible for the Flatten layer as no operations changing the data is performed. Also no gray band is visible for the 4th and 5th batch normalization layer outputs as these are fused with the dense layers.

Typically, extreme PTQ results in a sizeable accuracy loss. The increased spacing between representable numbers enforces a severe weight rounding that leads to a significant reduction in the model accuracy once the resolution becomes too coarse. The amount of compression one can reach by this procedure is balanced by the need to preserve the model accuracy, and how much model accuracy reduction can be tolerated is an application-specific question.

We use PTQ to generate a range of compressed models, further discussed in Section 8, scanning bit widths from 16 to 1, with 6 integer bits.

7.2 Quantization-aware training

QAT [62] is an efficient procedure to limit accuracy loss while reducing the numerical precision of the network components. Here, quantized weights and biases are used in the training during the forward pass, while full precision is used in the backward pass in order to facilitate the drift towards the optimal point in the loss minimization (known as the *straight-through estimator*) [63]. The `hls4ml` library supports QAT through its interface to QKERAS [16].

We train a range of quantized QKERAS models using the same architecture as in Fig. 4, imposing a common bit width across the model. We scan the bit width from 16 to 3, as well as train a ternary and a binary quantized model. We refer

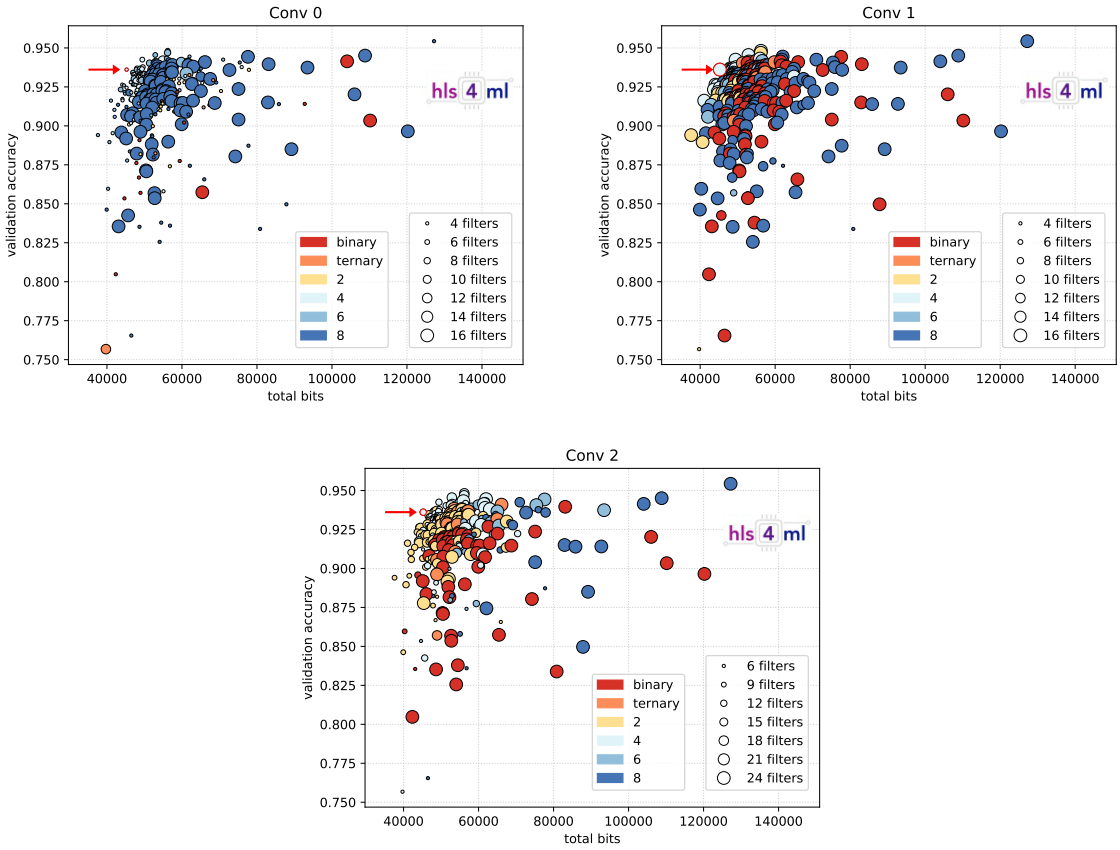

Figure 8: Relation between accuracy and model bit size for an ensemble of trial models, resulting from a Bayesian optimization performed over layer quantizers and number of filters, using AUTOQKERAS. Each figure corresponds to a given quantization and filter configuration tested for the first (top left), second (top right) and third (bottom) convolutional layer. The size of each marker corresponds to the number of filters tested for that layer, and the color to the quantization (binary, ternary or mantissa quantization). The red arrow indicates the model yielding the best accuracy versus size trade-off.

to these models as QKeras (Q) models. In addition, we train pruned versions of these models, targeting a sparsity of 50%. These are referred to as QKeras Pruned (QP) models.

Only convolutional layers, dense layers, and activation functions are quantized. The batch normalization layers are not quantized during training, as support for the QKERAS quantized equivalent of the KERAS batch normalization layer is not supported in `hls4ml` at the time of this writing. Support for this is planned for a future version of `hls4ml`. Batch normalization layers in the QAT models are therefore set to the default precision of $\langle 16, 6 \rangle$ by `hls4ml`. The final softmax layer is also kept at the default precision of $\langle 16, 6 \rangle$ in order to not compromise the classification accuracy.

Finally, we define a heterogeneously quantized model using AUTOQKERAS [16], a library for automatic heterogeneous quantization. The AUTOQKERAS library treats the layer precision as a hyperparameter, and finds the quantization which minimizes the model bit size while maximizing the model accuracy. By allowing AUTOQKERAS to explore different quantization settings for different parts of a given network, we obtain an optimal heterogeneously quantized QKERAS model. A Bayesian optimization is performed over a range of quantizers available in QKERAS, targeting a 50% reduction in model bit size. At the same time, the number of filters per convolutional layer and neurons per dense layer is re-optimized as quantization tends to lead to a preference for either (1) more filters as information is lost during quantization or (2) less filters due to some filters effectively being the same after quantization.

The optimization process is shown in Fig. 8, where the model bit size versus the model validation accuracy is shown for all the models tested in the automatic quantization procedure, showing the different quantization configurations for each of the convolutional layers. The size of the markers correspond to the number of filters used for a given convolutional layer in that trial. The colors correspond to different type of quantizers (binary, ternary of mantissa quantization using

Table 2: Per-layer heterogeneous quantization configuration obtained with AUTOQKERAS, the total estimated energy consumption and model size in bits of the AutoQ (AQ) model. The energy is estimated assuming a 45 nm process using QTOOLS.

Model	Precision per layer												Energy [nJ]	Bit size
	Conv2D	ReLU	Conv2D	ReLU	Conv2D	ReLU	Dense	ReLU	Dense	ReLU	Dense	Softmax		
AQ	$\langle 4,0\rangle$	$\langle 3,1\rangle$	$\langle 4,0\rangle$	$\langle 3,1\rangle$	$\langle 4,0\rangle$	$\langle 8,4\rangle$	$\langle 4,0\rangle$	$\langle 4,2\rangle$	$\langle 4,0\rangle$	$\langle 8,2\rangle$	$\langle 6,0\rangle$	$\langle 16,6\rangle$	465	45,240

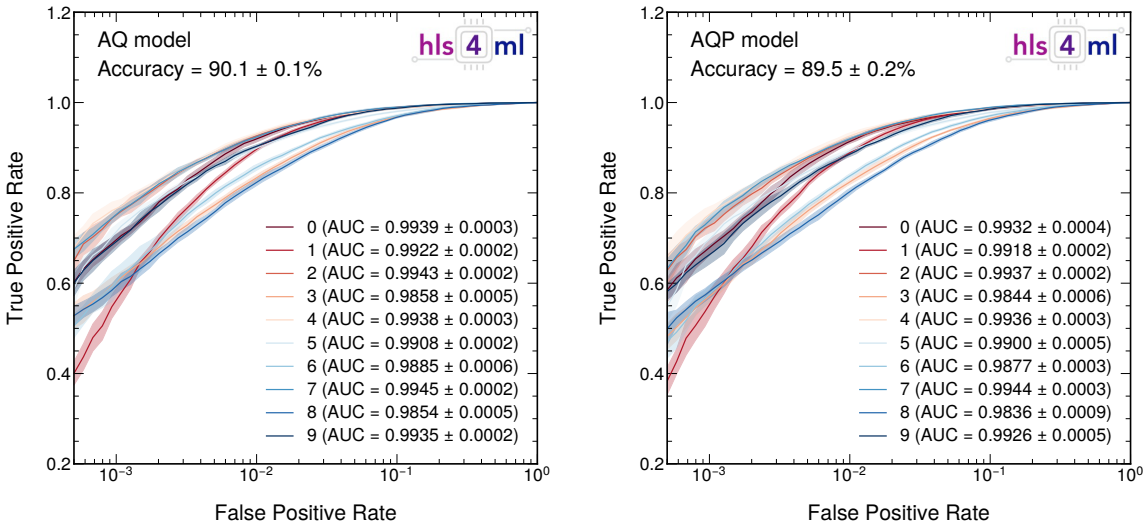

Figure 9: ROC curves of false positive rate (FPR) versus true positive rate (TPR) for the AutoQ (AQ) model (left) and the AutoQ Pruned (AQP) model (right). Training is performed using k-fold cross validation, with $k = 10$. For each digit, the solid line corresponds to the mean across the ten folds and the band to the standard deviation. The mean area under the curve (AUC) across the ten folds is reported in the legend. The mean accuracy and it's standard deviation across the ten folds is reported as well.

different bit widths). The model yielding the best accuracy versus size trade-off is marked by a red arrow. The number of filters per convolutional layer for the selected model is (4, 16, 12), compared to the original (16, 16, 24) for the BF and BP models, and the number of neurons per dense layer is (15, 16) compared to (42, 64) in the original model. Table 2 summarizes the quantization configuration found to be optimal by AUTOQKERAS, and the corresponding model energy consumption estimated using QTOOLS. We note that this model uses almost 90% less energy than the original. We train two versions of this model: an unpruned version (AQ), and a pruned version (AQP). The latter model is the same as AQ, but additionally pruned to a target sparsity of 50%.

Figure 9 shows the ROC curves for the AQ and AQP models. The curves show a slightly lower classification accuracy than those in Fig. 6, with AUCs differing by approximately 1%.

The numerical values spanned by the AQ model is shown in Figure 10 for layer weights (left) and outputs (right). In contrast to those showed in Fig. 7, different bit widths are now used for the different layers, in correspondence with the bit width used in QKERAS.

Figure 11 summarizes the effects of pruning and quantization. Here, we show the median accuracy and upper and lower quartiles across the 10 folds of the unpruned (red) and pruned (green) quantized models, for different choices of bit widths and for the AQ (AQP) models. The unquantized baseline models are shown for reference (BF or BP). For bit widths above four, pruning to 50% sparsity has very little impact on the model accuracy. At very low bit widths, however, pruning negatively impacts the model performance. The accuracy is constant down to four bit precision, with marginal accuracy loss down to three bits. Using ternary quantization, the model accuracy drops to 87–88% and has a higher statistical uncertainty. When quantizing down to binary precision, the model accuracy is reduced to 72% for the unpruned model and 64% for the pruned model. The significant reduction in accuracy due to pruning for binary networks is due to too little information being available in the network to accurately classify unseen data. A large spread in model accuracy for the binary network across the 10 folds is observed, indicating that the model is less robust to fluctuations in the training dataset. As demonstrated in [8], this can be mitigated by increasing the model size (more

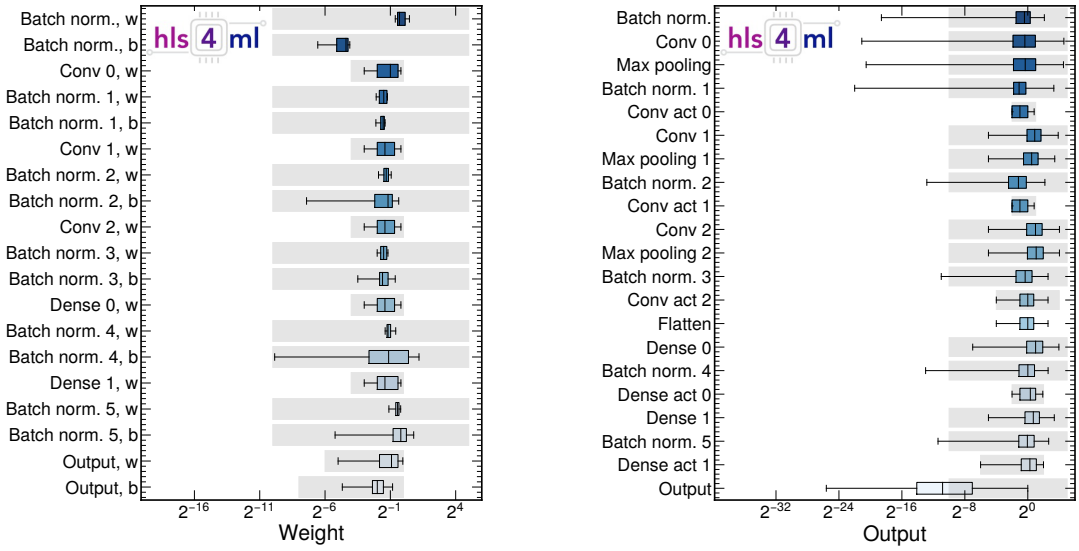

Figure 10: Layer weights (left) and output (right) numerical values per layer for the AutoQ (AQ) model using a subset of the training data. The gray band represents the range covered by the fixed precision per layer in `hls4ml`.

filters and neurons per layer). The AQ models obtain a slightly lower accuracy than the baselines, but uses, as will be demonstrated in Section 8, significantly fewer resources.

Due to the results above, it is recommended that users prune and quantize models using QAT through QKERAS, before proceeding with FPGA deployment with `hls4ml`.

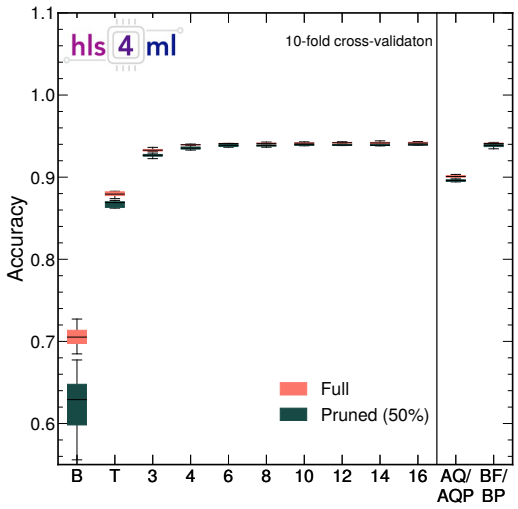

Figure 11: Accuracy for unpruned (red) and pruned (yellow) models for binary (B) and ternary (T) precisions, homogeneously quantized models with bit widths between 3 and 16, the heterogeneously quantized models AutoQ (AQ) and AutoQ Pruned (AQP), compared to the Baseline Floating-point (BF) and Baseline Pruned (BP) models. The black line represents the median, and the box extends from the lower to upper quartile values across the 10 folds.

8 FPGA porting

The models described above are translated into firmware using `hls4ml` version 0.5.0, and then synthesized with Vivado HLS 2020.1, targeting a Xilinx Virtex UltraScale+ VU9P (`xcvu9pflgb2104-2L`) FPGA with a clock frequency of 200 MHz. For the QKERAS quantized models, the sign is not accounted for when setting the bit width per layer during QAT, so layers quantized with total bit width b in QKERAS are therefore implemented as fixed-point numbers with total bit width $b + 1$ in `hls4ml`. We compare the model accuracy, latency, and on-chip resource consumption. The accuracy after translating the model into `C/C++` code with `hls4ml` (solid line) for the different models, is shown in Figure 12 and compared to the accuracy evaluated using KERAS. No pre-synthesis results are shown for the BF and BP models, as these are quantized *during* synthesis. Nearly perfect agreement in evaluated accuracy before and after synthesis is observed for the Q and QP models and the translation into fixed-point precision is lossless.

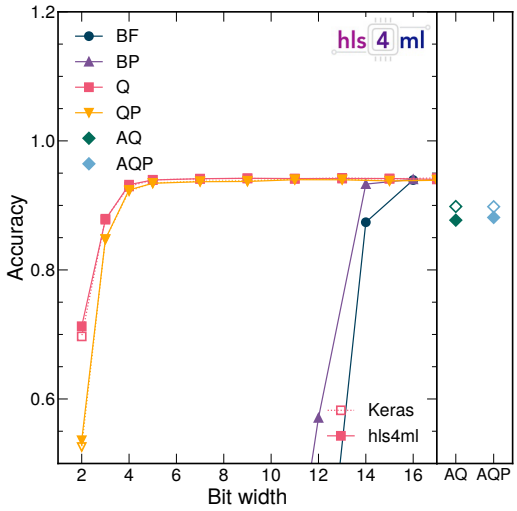

Figure 12: Model accuracy as a function of bit width for the Baseline Floating-point (BF), Baseline Pruned (BP), QKeras (Q) and QKeras Pruned (QP) models. The heterogeneously quantized models AutoQ (AQ) and AutoQ Pruned (AQP) are shown in the sidebar.

While the accuracy of the Q and QP models trained via QAT remains high down to a bit width of three, the accuracy of the PTQ models fall off sharply with decreasing bit width and have almost no discrimination power for bit widths smaller than 14. PTQ has a higher negative impact on the unpruned models, indicating that rounding errors are the biggest cause for accuracy degradation. The heterogeneously quantized models AQ and AQP have slightly lower accuracy than the baseline $\langle 16, 6 \rangle$ model.

We then study the resource consumption and latency of the different models after logic-synthesis. The resources available on the FPGA are digital signal processors (DSPs), lookup tables (LUTs), BRAMs, and flip-flops (FFs). In Fig. 13, the resource consumption relative to the total available resources is shown. Here, a fully parallel implementation is used where each multiplier is used exactly once, which can be achieved by setting the *reuse factor* R [1] to 1 for each layer in `hls4ml`.

The DSP consumption is slightly higher for the Q and QP models than the BF and BP models due to the batch normalization layers in the QAT models being fixed to $\langle 16, 6 \rangle$.

Below a bit width of 10, the DSP consumption is significantly reduced as multiplications are performed using LUTs. DSPs are usually the limiting resource for FPGA inference, and we observe that through QAT, the DSP consumption can be reduced from one hundred percent down to a few percent with no loss in model accuracy (as demonstrated in Fig. 12). Above a bit width of 10, almost all the DSPs on the device are in use for the Q and QP models. This routing is a choice of Vivado HLS during optimization of the circuit layout. This is also the reason why pruning appears to have relatively little impact for these models: the DSPs are maximally used and the remaining multiplications are performed with LUTs. The QP models use significantly fewer LUT resources than the unpruned equivalent. The point where most multiplications are moved from DSPs to LUTs is marked by a steep drop in DSP consumption starting at a bit width of 10.

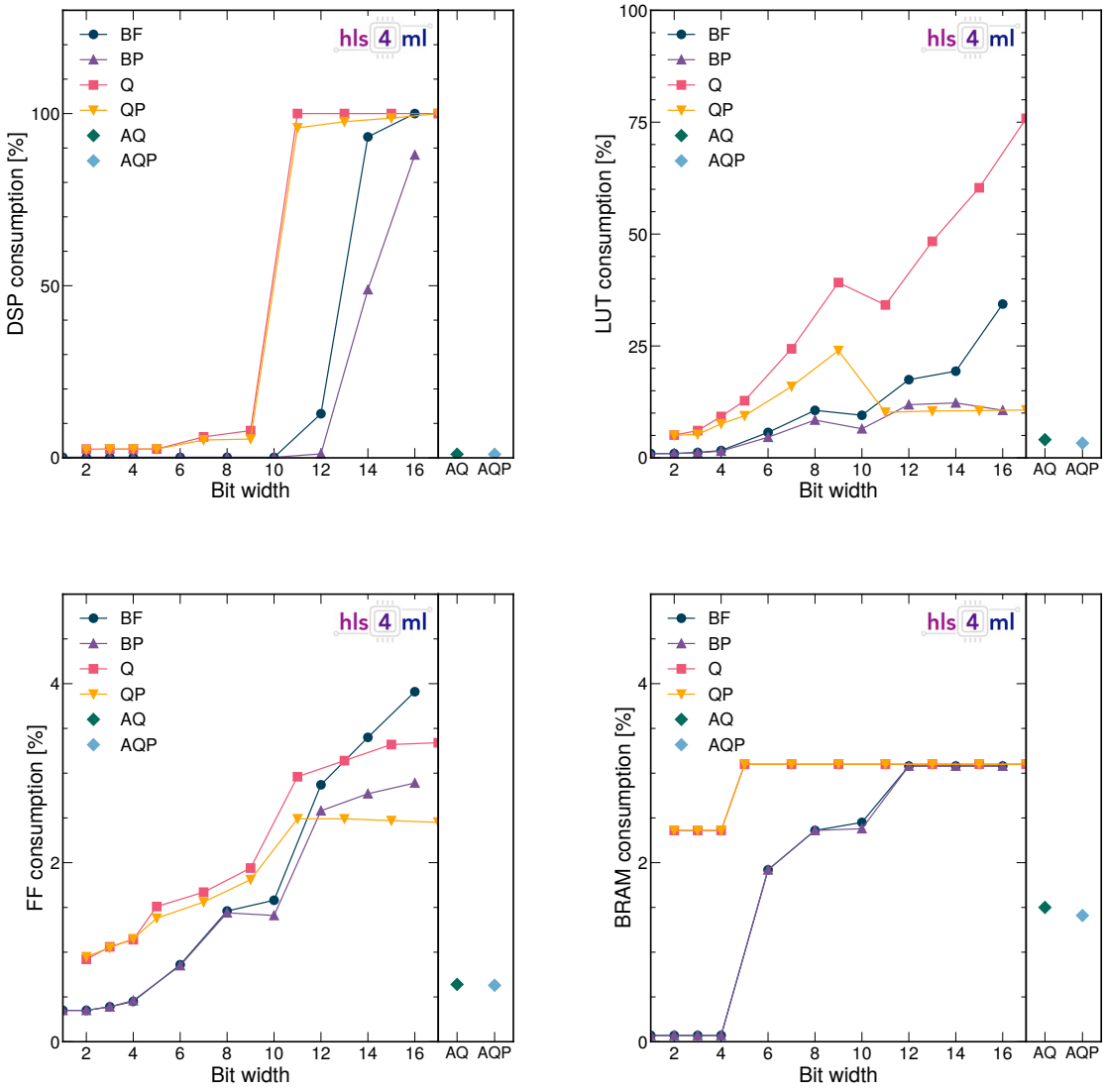

Figure 13: Resource consumption as a function of bit width for the Baseline Floating-point (BF), Baseline Pruned (BP), QKeras (Q), and QKeras Pruned (QP) models. The heterogeneously quantized AutoQ (AQ) and AutoQ Pruned (AQP) models are displayed in the right sub-plot. The model DSP (top left), LUT (top right), FF (bottom left) and BRAM (bottom right) consumption is shown.

The heterogeneously quantized models, AQ and AQP, consume very little FPGA resources, comparable to that of the Q and QP models quantized to a bit width of three. All models use very few FFs, below 4% of the total budget. The BRAM consumption is also small and below 4% for all models. Some dependence on bit width can be traced back to how operations are mapped to the appropriate resources through internal optimizations in HLS. Depending on the length and the bit width of the FIFO buffers used for the convolutional layer sliding window, HLS will decide whether to place the operation on BRAMs or LUTs and migration between the two is expected. Most of the BRAMs, are spent on *channels*, the output of different layers.

The latency and II for all models is shown in Figure 14. A total latency of about 5 μs is observed for all models, similar to the II. The latency is independent of bit width when running at a fixed clock period. We leave it for future studies to explore running the board at higher clock frequencies.

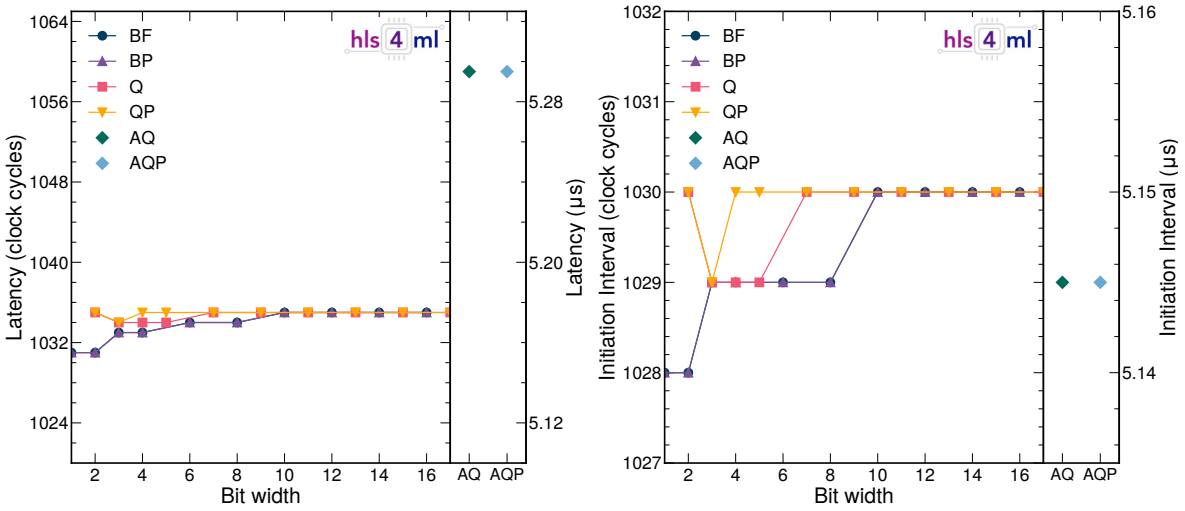

Figure 14: The model latency (left) and initiation interval (right) as a function of bit width for the Baseline Floating-point (BF), Baseline Pruned (BP), QKeras (Q), and QKeras Pruned (QP) models. The heterogeneously quantized AutoQ (AQ) and AutoQ Pruned (AQP) models are displayed in the right sub-plot.

A summary of the accuracy, resource consumption and latency for the Baseline Floating-point (BF) and Baseline Pruned (BP) models quantized to a bit width of 14, the QKeras (Q) and QKeras Pruned (QP) models quantized to a bit width of 7 and the heterogeneously quantized AutoQ (AQ) and AutoQ Pruned (AQP) models, is shown in Table 3. Resource utilization is quoted as a fraction of the total available resources on the FPGA, and the absolute number of resources used is quoted in parenthesis. The accuracy of the post-training quantized BF and BP models drops below 50% for bit widths narrower than 14 and can not be used for inference. The QAT models, Q and QP, quantized to a bit width of 7 maintain a high accuracy despite using only a fraction of the available FPGA resources. The models using the fewest resources are the AQ and AQP heterogeneously quantized models, reducing the DSP consumption by 99% while maintaining a relatively high accuracy. Finding the best trade-off between model size and accuracy in an application-specific way can be done using AutoQKeras, as demonstrated in Sec. 7.

Table 3: Accuracy, resource consumption and latency for the Baseline Floating-point (BF) and Baseline Pruned (BP) models quantized to a bit width of 14, the QKeras (Q) and QKeras Pruned (QP) models quantized to a bit width of 7 and the heterogeneously quantized AutoQ (AQ) and AutoQ Pruned (AQP) models. The numbers in parentheses correspond to the total amount of resources used.

FPGA: Xilinx Virtex UltraScale+ VU9P

Model	Accuracy	DSP	LUT	FF	BRAM	Latency [cc]	II [cc]
BF 14-bit	0.87	6,377 (93.2%)	228,823 (19.4%)	80,278 (3.4%)	66.5 (3%)	1,035 (5.2 μs)	1,030
BP 14-bit	0.93	3,341 (48.9%)	145,089 (12.3%)	65,482 (2.8%)	66.5 (3%)	1,035 (5.2 μs)	1,030
Q 7-bit	0.94	175 (2.5%)	150,981 (12.8%)	35,628 (1.5%)	67.0 (3%)	1,034 (5.2 μs)	1,029
QP 7-bit	0.94	174 (2.5%)	111,152 (9.4%)	32,554 (1.4%)	67.0 (3%)	1,035 (5.2 μs)	1,030
AQ	0.88	72 (1.0%)	48,027 (4.0%)	15,242 (0.6%)	32.5 (2%)	1,059 (5.3 μs)	1,029
AQP	0.88	70 (1.0%)	38,795 (3.3%)	14,802 (0.6%)	30.5 (1%)	1,059 (5.3 μs)	1,029

To further reduce the resource consumption, the reuse factor R can be increased. This comes at the cost of higher latency. The model latency and resource consumption as a function of bit width and for different reuse factors for the QP models are shown in Figure 15. The latency and II increase with R, while the DSP consumption goes down. The LUT consumption is minimally affected by the reuse factor, consistent with the results reported in [1]. The BRAM consumption is the same for all reuse factors, around 3%, and therefore not plotted. The corresponding study for the BF, BP and Q models can be found in Appendix A.

A summary of the latency and resource consumption for different reuse factors for all the models at a fixed bit width of 16 is shown in Figure 16. The latency has a linear dependence on the reuse factor, as expected because each multiplier is used in series one reuse factor at the time. The DSP consumption decreases as $\sim 1/R$ for all models. The first point

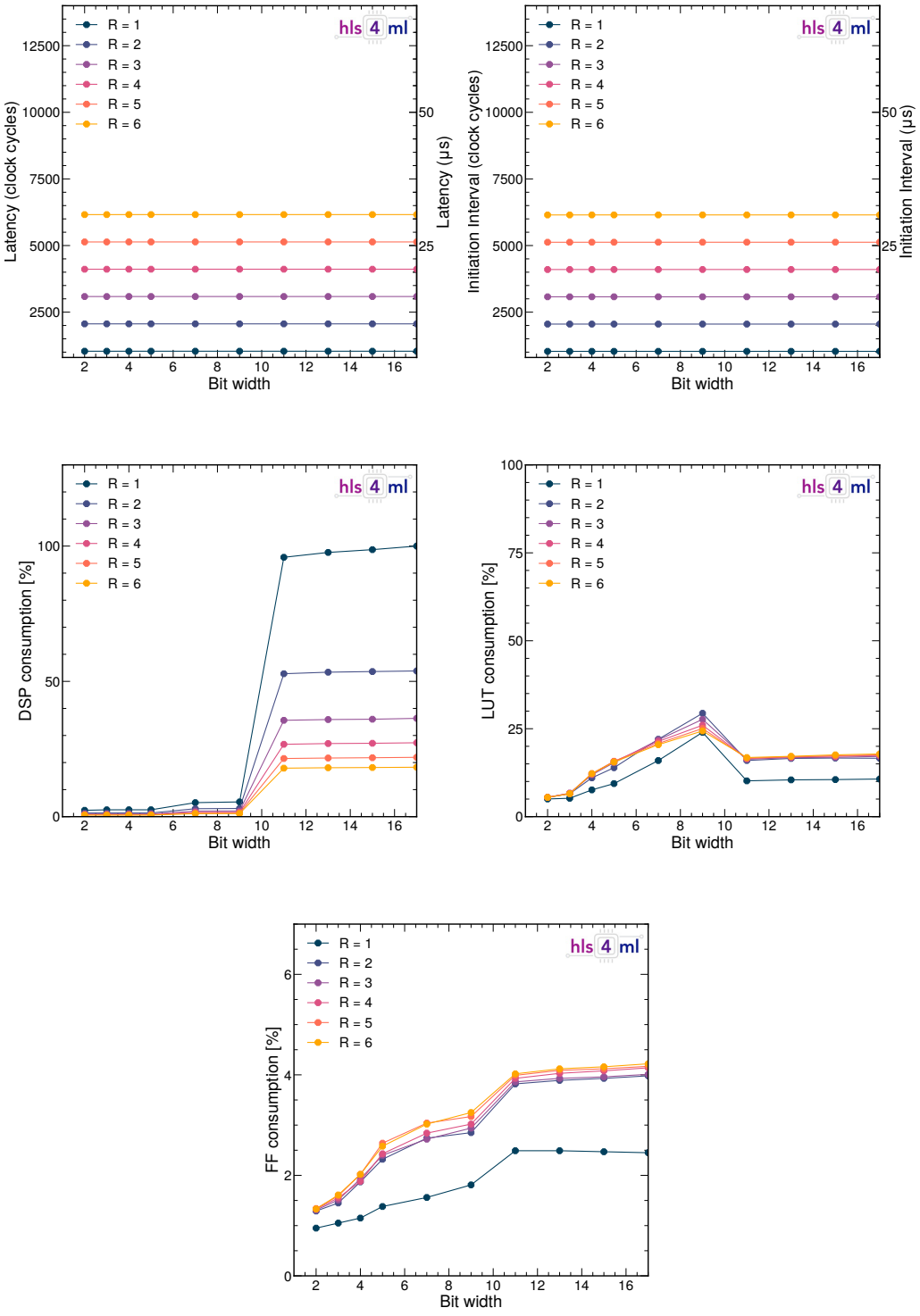

Figure 15: Latency (top left), initiation interval (top right), DSP (middle left), LUT (middle right), FF (bottom) consumption as a function of bit width and for different reuse factors for the QKeras Pruned (QP) models.

deviates from this as the maximum number of DSPs are in use, effectively reaching a plateau. The LUT consumption

is high for a reuse factor of one, complimenting the ceiling reached in DSP consumption at a reuse of one, since the multiplications that do not fit on DSP are moved to LUTs. The FF consumption is flat as a function of reuse factor. The BRAM consumption does not depend on the reuse factor and is the same for all models, around 3%. We leave it up to `hls4ml` users to find the optimal trade-off between inference latency and resource consumption for a given application through tuning of the reuse factor.

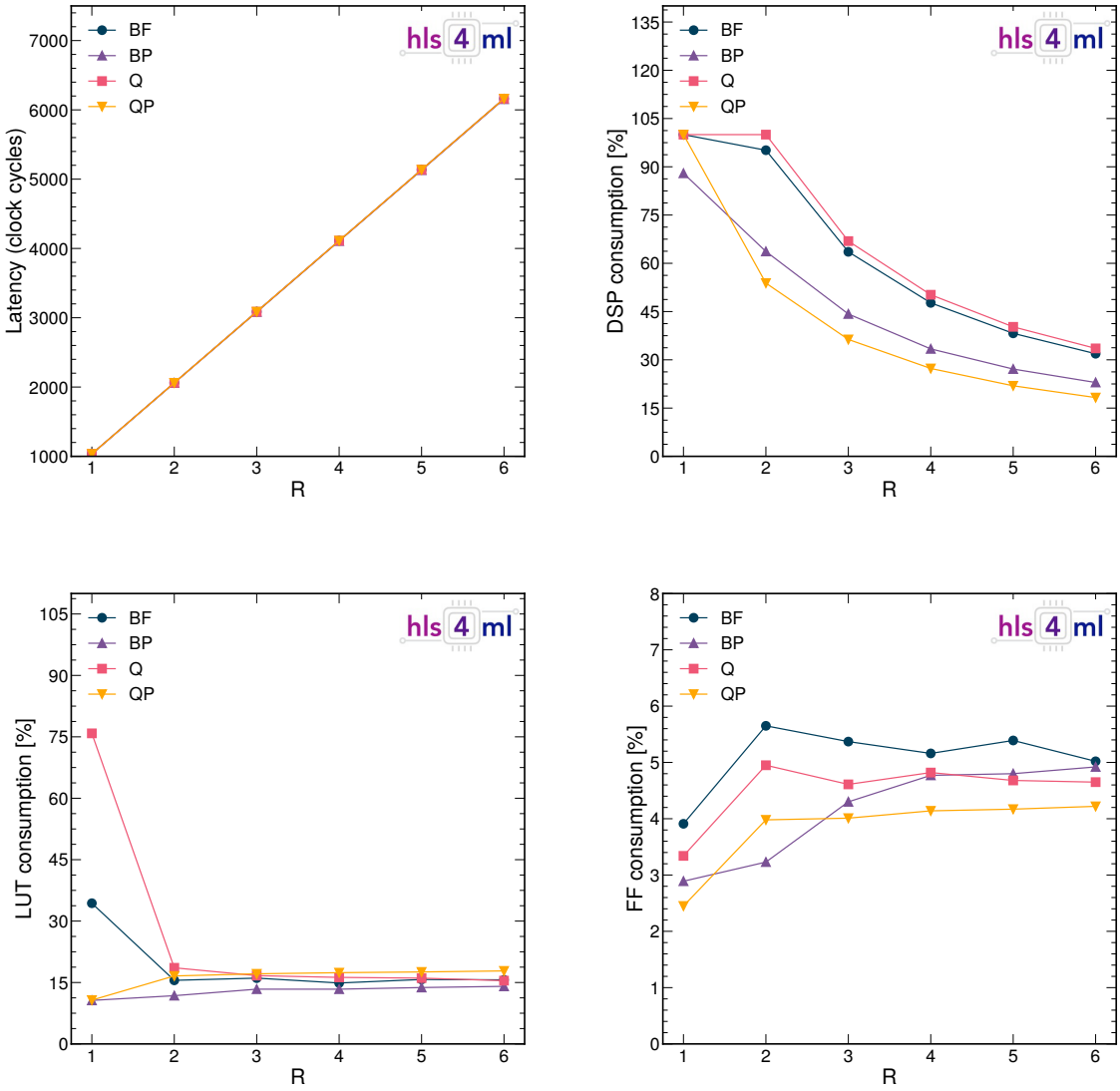

Figure 16: The model latency (top left), DSP (top right), LUT (bottom left) and FF (bottom right) consumption as a function of reuse factor for a fixed bit width of 16 for the Baseline (BF), Baseline Pruned (BP), QKeras (Q), and QKeras Pruned (QP) models.

Although particle physics experiments mostly use large FPGAs, the `hls4ml` library can be readily used for smaller FPGAs, like those found on system-on-chip (SoC) or internet-of-things (IoT) devices, through increasing the reuse factor. To demonstrate this, we synthesize and deploy the smallest model that retains the original model accuracy, QP 7-bit, onto a low-cost TUL PYNQ-Z2 development board, equipped with a Xilinx Zynq XC7Z020 SoC (FPGA part number `xc7z020clg400-1`). This FPGA is significantly smaller than the Xilinx Virtex UltraScale+ VU9P, and consists of 13,300 logic slices, each with four 6-input LUTs and 8 FFs, 630 kB of BRAM, and 220 DSP slices. As expected, a large reuse factor is needed in order to fit the QP 7-bit model onto the Zynq XC7Z020. For a clock frequency of 100 MHz, the resulting inference latency is 171 μs and up to 2,831 image classifications per second. This

implementation uses a total of 91% of the LUTs, 97% of the DSPs, 33% of the FFs, and 44% of the BRAM. A summary is provided in Table 4. This demonstrates the flexibility of `hls4ml` to accommodate SoC/IoT use cases, which can demand smaller FPGAs and tolerate millisecond latencies.

Table 4: Resource consumption and latency for the QP 7-bit model on a Xilinx Zynq XC7Z020 SoC. A clock frequency of 100 MHz is used.

FPGA: Xilinx Zynq XC7Z020 SoC

	DSP	LUT	FF	BRAM (18 kb)	Latency [cc]	II [cc]	frame/s
Available	220	53,200	106,400	280	—	—	—
Used	213 (97%)	48,259 (91%)	35,118 (33%)	122 (44%)	17,085 (171 μs)	16,385	2,831

Finally, in Fig. 17 we study the resource consumption and latency as a function of the input size for a single convolutional layer with varying number of filters and kernel sizes. Three input channels are always assumed and the input height (H) and width (W) is varied between 10 and 256, such that the input size is $H \times W \times 3$. A precision of $\langle 16, 6 \rangle$ is assumed for all models to illustrate the dependency of latency/resources on the given layer configurations, although, as we have demonstrated above, resources can be significantly reduced using QAT. The DSP and LUT consumption is constant as a function of the input size, but increases with the number of filters used and the kernel size, due to the higher number of multiplications that need to be performed simultaneously. The latency increases linearly with the input size, but does not depend on the kernel size or the number of filters. We also show the latency as a function of the depth of the model in Fig. 18. For simplicity, we assume an input size of $30 \times 30 \times 3$, 16 filters and a kernel size of 3×3 for each convolutional layer. The precision is fixed to $\langle 16, 6 \rangle$ or $\langle 7, 1 \rangle$. The DSP consumption scales linearly with the model depth until the maximum number of DSPs are used. When all DSPs are in use, multiplications are moved onto LUTs, seen as a change of slope in the LUT consumption versus model depth. The inference latency increases linearly with the model depth.

Figures 17 and 18 summarize how input size and model architecture affects the inference latency and resource consumption. Through increasing the reuse factor, smaller FPGAs can be targeted through a trade-off between latency and resource consumption. Support for QAT through and pruning further reduce the model footprint. The `hls4ml` library is therefore capable of providing generic, multi-backend support for a wide range of hardware and latency constraints.

9 Conclusions

We have presented the extension of `hls4ml` to support convolutional neural network (CNN) architectures for transpilation to FPGA designs, through a stream-based implementation of convolutional and pooling layers. A fully on-chip design is used in order to provide for microsecond latency applications, like those at the CERN Large Hadron Collider. Taking as a benchmark example a CNN classifier trained on the Street View House Numbers Dataset, we show how compression techniques at training time (pruning and quantization-aware training) reduce the resource utilization of the FPGA-converted models, while retaining to a large extent the floating-point precision baseline accuracy. Once converted to FPGA firmware using `hls4ml`, these models can be executed with 5 μs latency and a comparable initiation interval, while consuming less than 10% of the FPGA resources. We demonstrate the flexibility and scalability of `hls4ml` to accommodate CNN architectures of varying sizes, and offer solutions both for small System-on-Chip (SoC) FPGAs and for the larger FPGAs used in particle physics experiments. This work enables domain machine learning (ML) specialists to design hardware-optimized ML algorithms for low-latency, low-power, or radiation-hard systems, for instance particle physics trigger systems, autonomous vehicles, or inference accelerators for space applications.

Acknowledgement

We acknowledge the Fast Machine Learning collective as an open community of multi-domain experts and collaborators. This community was important for the development of this project.

M. P., S. S. and V. L. are supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (Grant Agreement No. 772369). S. J., M. L., K. P., and N. T. are supported by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy (DOE), Office of Science, Office of High Energy Physics. P. H. is supported by a Massachusetts Institute of Technology University grant. Z. W. is supported by the National Science Foundation under Grants No. 1606321 and 115164. J. D. is supported by the DOE, Office of Science, Office of High Energy Physics Early Career Research program under Award No. DE-SC0021187.

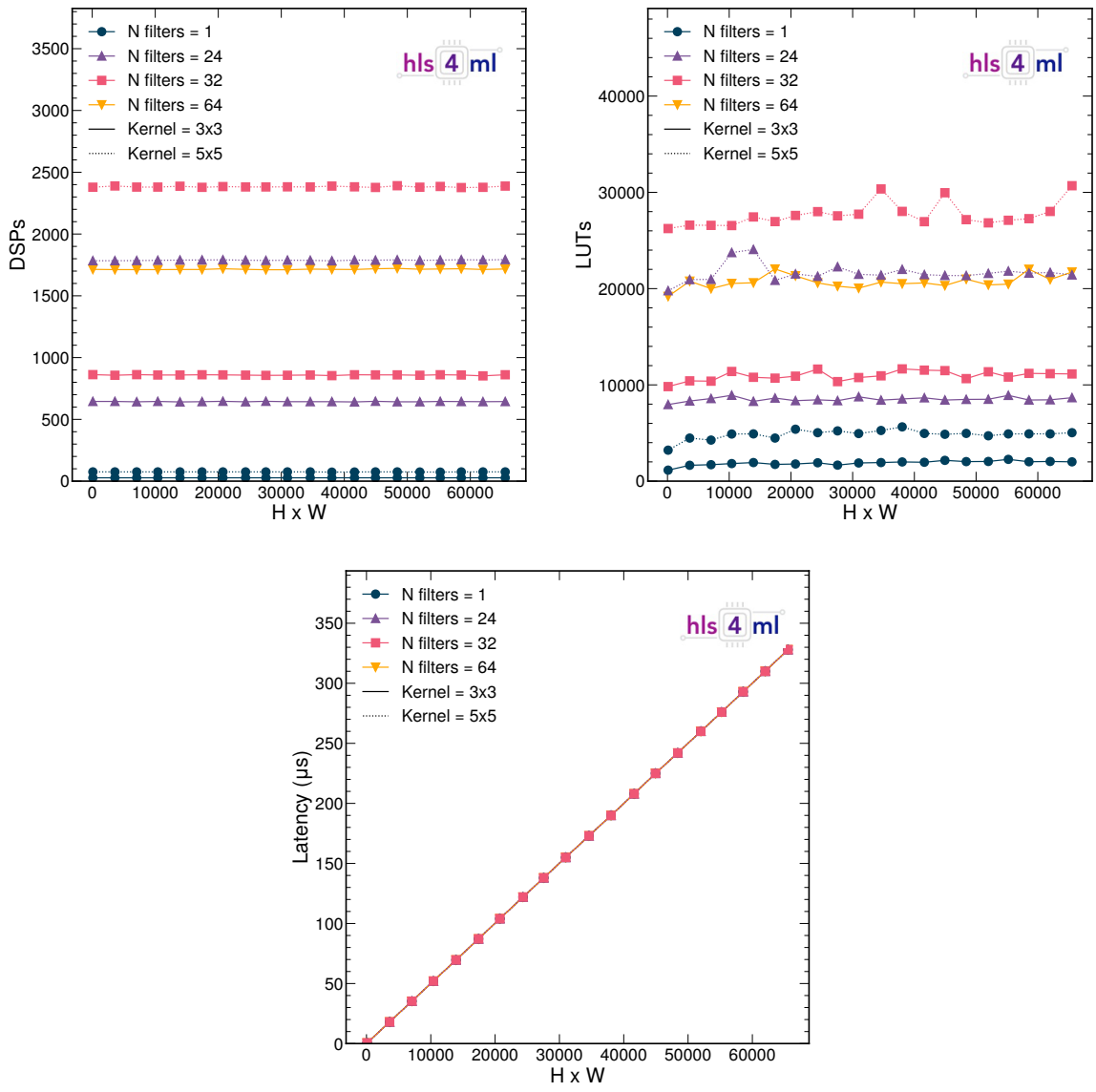

Figure 17: The DSP usage (top left), LUT usage (top right) and latency (bottom) as a function of the input image height (H) and width (W) for a single convolutional layer with varying kernel size and number of filters. Three color channels are assumed such that the input size corresponds to ($H \times W \times 3$). A default precision for all weights and outputs of $\langle 16, 6 \rangle$ is assumed.

Code availability statement

The `hls4ml` library is available at `https://github.com/fastmachinelearning/hls4ml` and archived in the Zenodo platform at doi:10.5281/zenodo.4161550. The work presented here is based on the Bartsia release, version 0.5.0. For examples on how to use `hls4ml`, the notebooks in `https://github.com/fastmachinelearning/hls4ml-tutorial` serve as a general introduction. The QKERAS library, which also includes AUTOQKERAS and QTOOLS, is available at `https://github.com/google/qkeras`.

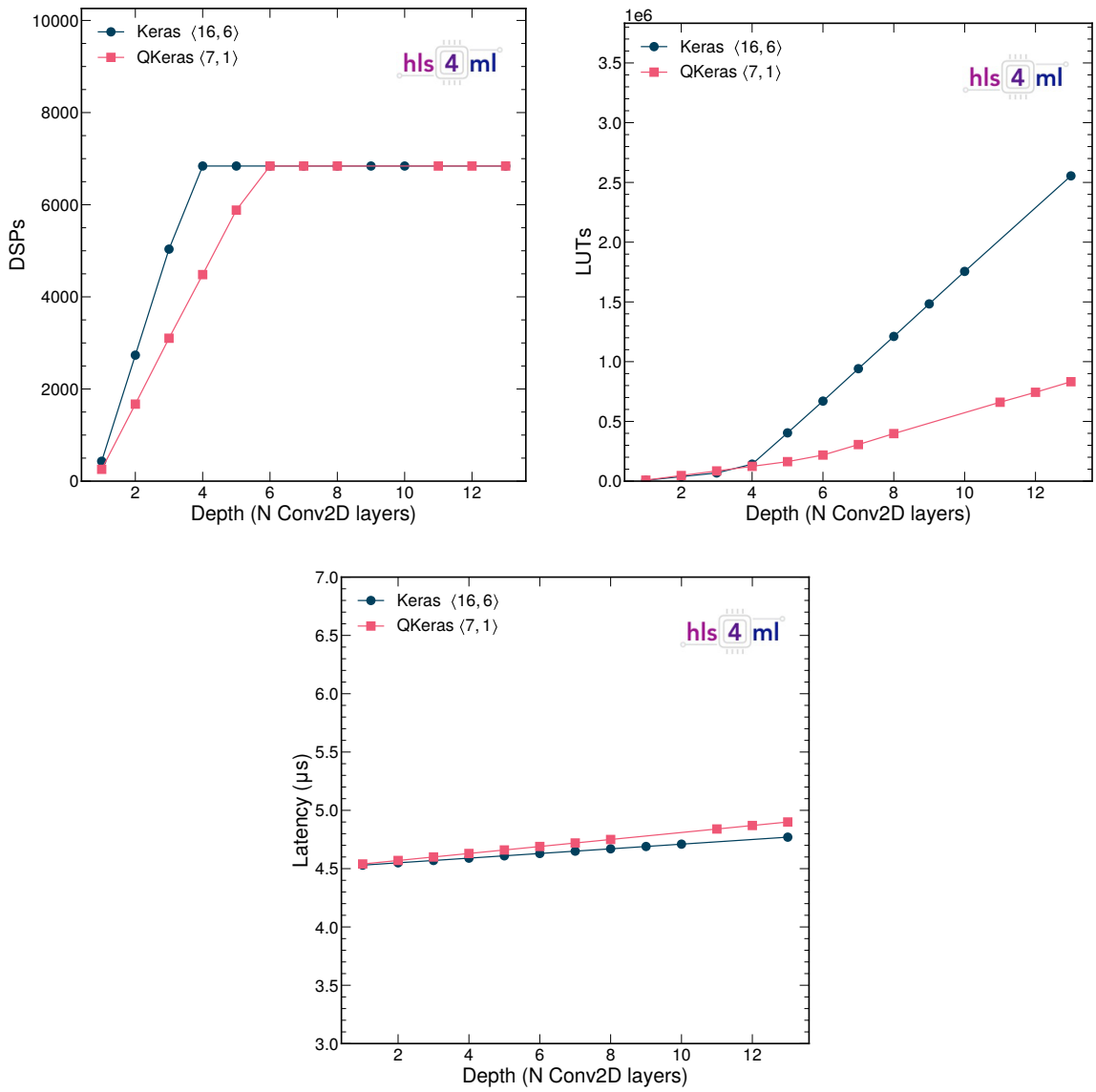

Figure 18: The DSP usage (top left), LUT usage (top right) and latency (bottom) as a function of the model depth for models using a precision of $\langle 16, 6 \rangle$ and $\langle 7, 1 \rangle$. An input size of $(30 \times 30 \times 3)$ is always assumed and each convolutional layer consists of 16 filters and a kernel size of (3×3).

Data availability statement

The data that support the findings of this study are openly available. The SVHN dataset [17] can be downloaded at `http://ufldl.stanford.edu/housenumbers` or through TensorFlow Datasets at `https://www.tensorflow.org/datasets/catalog/svhn_cropped`.

A Performance versus bit width and reuse factor

Figures 19-21 show the model latency, initiation interval, DSP, LUT and FF consumption as a function of bit width and for different reuse factors for the BF, BP and Q models, respectively. A similar behaviour is observed for all models, where the latency roughly scales with one unit of reuse factor and the DSP consumption scales as the inverse of the reuse factor. The BRAM consumption does not depend on the reuse factor.

For the BF models in Fig. 19, there is an unexpected drop in DSP consumption at a bit width of 12 for models using a reuse factor of one. For this model, only 13% of the DSPs (corresponding to 876 units) are used, whereas the same mode with a reuse factor of six uses 19% of the available DSPs (corresponding to a total of 1,311). We would expect models with higher reuse factors to use fewer resources and not vice versa. This unexpected behaviour is only observed for one data point. We have investigated things to the extent possible, but can only map the results back to how resources are allocated within HLS.

For the Q models in Fig. 21, the DSP consumption (middle left) of the models using a reuse factor of one and those using a reuse factor of two overlap above a bit width of ten. The reason for this is that the maximum number of DSPs are reached for both model types, and multiplications are therefore forced to other resources. This effect can be seen in the LUT consumption (middle right), where the model using a reuse factor of 1 uses significantly more LUTs than the other models.

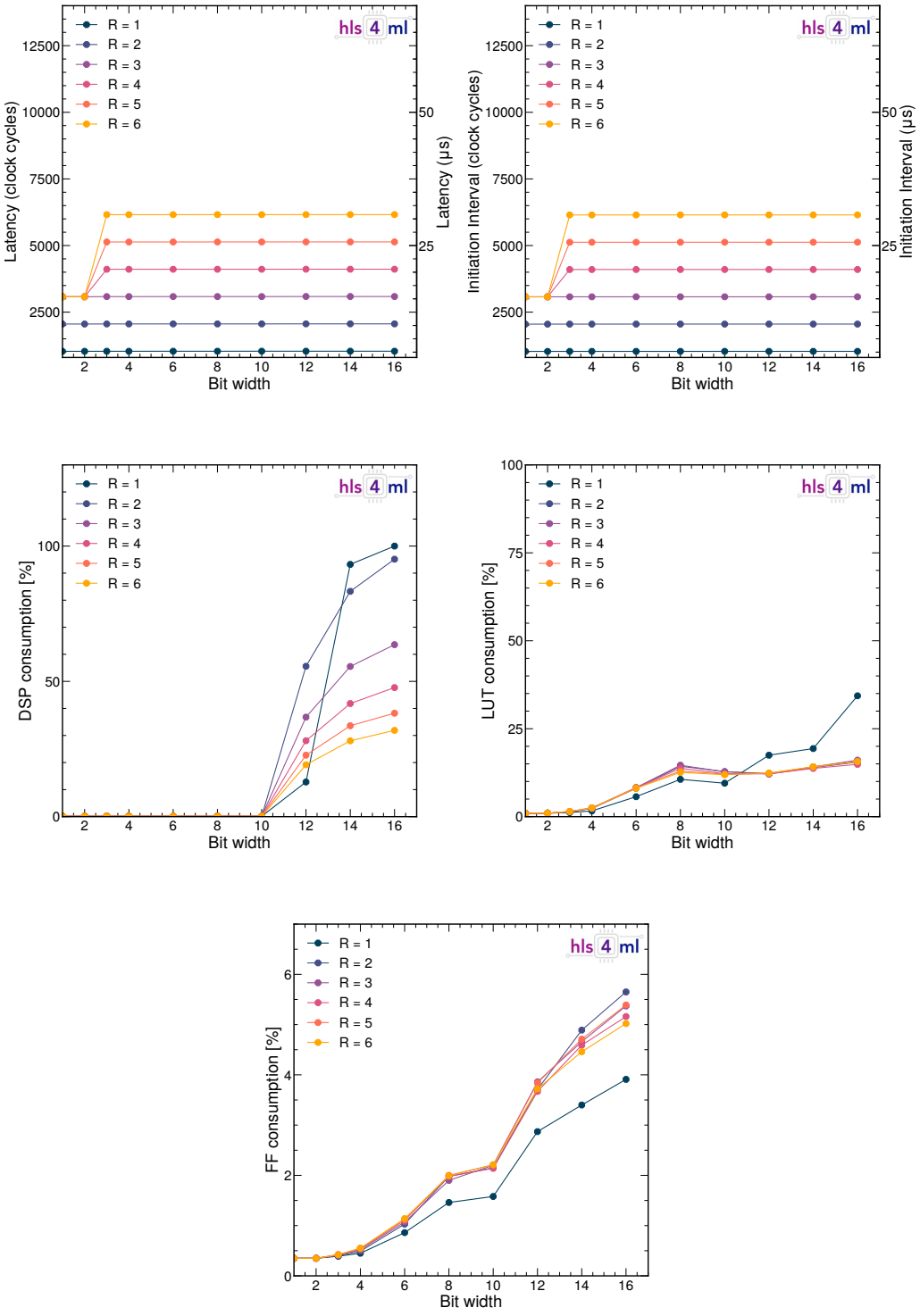

Figure 19: The model latency (top left), initiation interval (top right), DSP (middle left), LUT (middle right) and FF (bottom) consumption as a function of bit width and for different reuse factors for the Baseline Floating-point (BF) model.

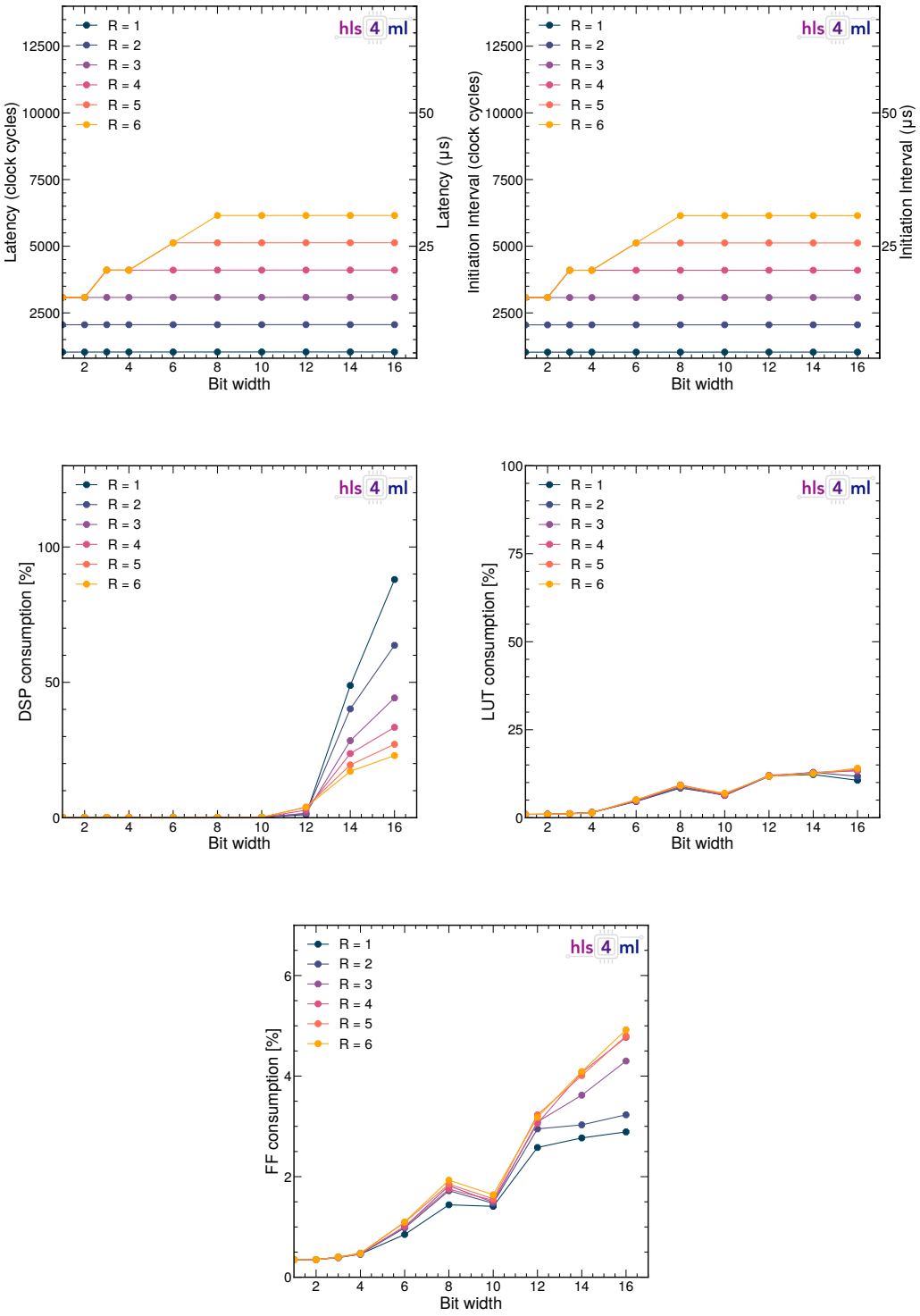

Figure 20: The model latency (top left), initiation interval (top right), DSP (middle left), LUT (middle right) and FF (bottom left)consumption as a function of bit width and for different reuse factors for the Baseline Pruned (BP) model.

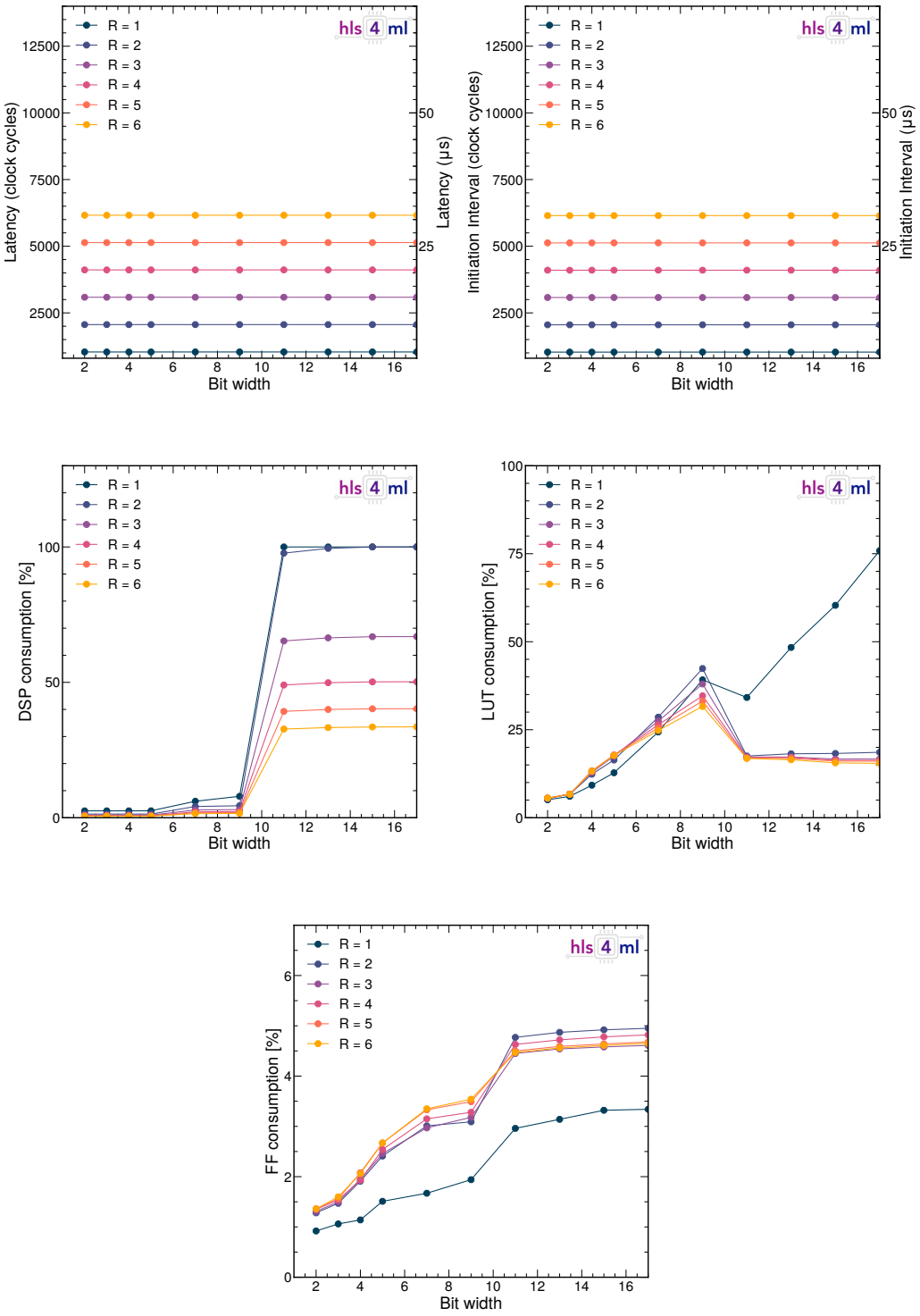

Figure 21: The model latency (top left), initiation interval (top right), DSP (middle left), LUT (middle right) and FF (bottom) consumption as a function of bit width and for different reuse factors for the QKeras (Q) model.

References

[1] J. Duarte et al., "Fast inference of deep neural networks in FPGAs for particle physics", *J. Instrum.* **13** (2018), no. 07, P07027, doi:10.1088/1748-0221/13/07/P07027, arXiv:1804.06913.

[2] V. Loncar et al., "fastmachinelearning/hls4ml: aster", 10, 2020. doi:10.5281/zenodo.4161550, https://github.com/fastmachinelearning/hls4ml.

[3] ATLAS Collaboration, "Operation of the ATLAS trigger system in Run 2", *J. Instrum.* **15** (2020) P10004, doi:10.1088/1748-0221/15/10/P10004, arXiv:2007.12539.

[4] ATLAS Collaboration, "Technical Design Report for the Phase-II Upgrade of the ATLAS TDAQ System", ATLAS Technical Design Report CERN-LHCC-2017-020. ATLAS-TDR-029, 2017.

[5] CMS Collaboration, "Performance of the CMS Level-1 trigger in proton-proton collisions at $\sqrt{s} = 13$ TeV", *J. Instrum.* **15** (2020) P10017, doi:10.1088/1748-0221/15/10/P10017, arXiv:2006.10165.

[6] CMS Collaboration, "The Phase-2 upgrade of the CMS Level-1 trigger", CMS Technical Design Report CERN-LHCC-2020-004. CMS-TDR-021, 2020.

[7] Xilinx, "Vivado design suite user guide: High-level synthesis", 2020. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug902-vivado-high-level-synthesis.pdf.

[8] J. Ngadiuba et al., "Compressing deep neural networks on FPGAs to binary and ternary precision with hls4ml", arXiv:2003.06308.

[9] S. Summers et al., "Fast inference of Boosted Decision Trees in FPGAs for particle physics", *J. Instrum.* **15** (2020), no. 05, P05026, doi:10.1088/1748-0221/15/05/P05026, arXiv:2002.02534.

[10] Y. Iiyama et al., "Distance-weighted graph neural networks on fpgas for real-time particle reconstruction in high energy physics", *Frontiers in Big Data* **3** (2021) 44, doi:10.3389/fdata.2020.598927.

[11] A. Heintz et al., "Accelerated charged particle tracking with graph neural networks on FPGAs", in *3rd Machine Learning and the Physical Sciences Workshop at the 34th Annual Conference on Neural Information Processing Systems*. 12, 2020. arXiv:2012.01563.

[12] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous systems", (2015). arXiv:1603.04467. Software available from tensorflow.org.

[13] F. Chollet et al., "Keras", 2015. https://keras.io.

[14] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems 32*, H. Wallach et al., eds., p. 8024. Curran Associates, Inc., 2019. arXiv:1912.01703.

[15] Open Neural Network Exchange Collaboration, "ONNX". https://onnx.ai/, 2017. https://onnx.ai/.

[16] C. N. C. Jr. et al., "Automatic deep heterogeneous quantization of deep neural networks for ultra low-area, low-latency inference on the edge at particle colliders", (2020). arXiv:2006.10159.

[17] Y. Netzer et al., "Reading digits in natural images with unsupervised feature learning", in *NIPS 2011 Workshop on Deep Learning and Unsupervised Feature Learning*. 2011.

[18] T. Boser, P. Calafiura, and I. Johnson, "Convolutional neural networks for track reconstruction on FPGAs", in *NIPS 2017 Demonstrations*. 2017.

[19] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions", *ACM Comput. Surv.* **51** (2018) doi:10.1145/3186332, arXiv:1803.05900.

[20] K. Guo et al., "A survey of FPGA-based neural network inference accelerators", *ACM Trans. Reconfigurable Technol. Syst.* **12** (2019) doi:10.1145/3289185, arXiv:1712.08934.

[21] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review", *IEEE Access* **7** (2019) 7823–7859, doi:10.1109/access.2018.2890150, arXiv:1901.00121.

[22] K. Abdelouahab, M. Pelcat, J. Serot, and F. Berry, "Accelerating cnn inference on FPGAs: A survey", (2018). arXiv:1806.01683.

[23] Y. 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*. ACM Press, 2017. arXiv:1612.07119. doi:10.1145/3020078.3021744.

[24] M. Blott et al., "FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks", *ACM Trans. Reconfigurable Technol. Syst.* **11** (12, 2018) doi:10.1145/3242897, arXiv:1809.04570.

[25] Alessandro et al., "Xilinx/brevitas: Release version 0.4.0", March, 2021. doi:10.5281/zenodo.4606672, https://doi.org/10.5281/zenodo.4606672.

[26] S. I. Venieris and C. S. Bouganis, "fpgaConvNet: A toolflow for mapping diverse convolutional neural networks on embedded FPGAs", in *NIPS 2017 Workshop on Machine Learning on the Phone and other Consumer Devices*. 2017. arXiv:1711.08740.

[27] S. I. Venieris and C. S. Bouganis, "fpgaConvNet: Automated Mapping of Convolutional Neural Networks on FPGAs", in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, p. 291. ACM, 2017. doi:10.1145/3020078.3021791.

[28] S. I. Venieris and C. S. Bouganis, "Latency-Driven Design for FPGA-based Convolutional Neural Networks", in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, p. 1. Sept, 2017. doi:10.23919/FPL.2017.8056828.

[29] S. I. Venieris and C. S. Bouganis, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs", in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, p. 40. IEEE, 05, 2016. doi:10.1109/FCCM.2016.22.

[30] Y. Jia et al., "Caffe: Convolutional architecture for fast feature embedding", in *Proceedings of the 22nd ACM International Conference on Multimedia*, MM '14, p. 675. ACM, New York, NY, USA, 2014. arXiv:1408.5093. doi:10.1145/2647868.2654889.

[31] Y. Guan et al., "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates", in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, p. 152. 04, 2017. doi:10.1109/FCCM.2017.25.

[32] H. Sharma et al., "From high-level deep neural models to fpgas", in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, p. 1, IEEE. 2016.

[33] R. DiCecco et al., "Caffeinated fpgas: Fpga framework for convolutional neural networks", in *2016 International Conference on Field-Programmable Technology (FPT)*, pp. 265–268. 2016. arXiv:1609.09671. doi:10.1109/FPT.2016.7929549.

[34] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello, "Snowflake: A model agnostic accelerator for deep convolutional neural networks", arXiv:1708.02579.

[35] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A Matlab-like environment for machine learning", in *BigLearn, NIPS Workshop*. 2011.

[36] K. Majumder and U. Bondhugula, "A flexible FPGA accelerator for convolutional neural networks", (2019). arXiv:1912.07284.

[37] A. Aimar et al., "Nullhop: A flexible convolutional neural network accelerator based on sparse representations of feature maps", *IEEE Transactions on Neural Networks and Learning Systems* **30** (2019), no. 3, 644–656, doi:10.1109/TNNLS.2018.2852335.

[38] A. Rahman, J. Lee, and K. Choi, "Efficient FPGA acceleration of convolutional neural networks using logical-3D compute array", in *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, p. 1393. 2016.

[39] Xilinx, "Vitis AI", 2020. https://github.com/Xilinx/Vitis-AI.

[40] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication", in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pp. 19–24. 2017. doi:10.1109/ASAP.2017.7995254.

[41] Y. LeCun and C. Cortes, "MNIST handwritten digit database", 2010. http://yann.lecun.com/exdb/mnist/.

[42] E. D. Cubuk et al., "Autoaugment: Learning augmentation policies from data", 2019.

[43] T. DeVries and G. W. Taylor, "Improved regularization of convolutional neural networks with cutout", 2017.

[44] S. Liang, Y. Khoo, and H. Yang, "Drop-activation: Implicit parameter reduction and harmonic regularization", 2020.

[45] S. Zagoruyko and N. Komodakis, "Wide residual networks", 2017.

[46] K. Zhang et al., "Residual networks of residual networks: Multilevel residual networks", *CoRR* **abs/1608.02908** (2016) arXiv:1608.02908.

[47] P. Sermanet, S. Chintala, and Y. LeCun, "Convolutional neural networks applied to house numbers digit classification", 2012.

[48] T. O'Malley et al., "Keras Tuner". https://github.com/keras-team/keras-tuner, 2019.

[49] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", in *32nd International Conference on Machine Learning*, F. Bach and D. Blei, eds., volume 37, p. 448. PMLR, Lille, France, 07, 2015. arXiv:1502.03167.

[50] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines", in *27th International Conference on Machine Learning (ICML)*, p. 807. Omnipress, Madison, WI, USA, 2010.

[51] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks", in *14th International Conference on Artificial Intelligence and Statistics (AISTATS)*, G. Gordon, D. Dunson, and M. Dudík, eds., volume 15, p. 315. JMLR, Fort Lauderdale, FL, USA, 4, 2011.

[52] M. Horowitz, "1.1 computing's energy problem (and what we can do about it)", volume 57, pp. 10–14. 02, 2014. doi:10.1109/ISSCC.2014.6757323.

[53] I. Goodfellow, Y. Bengio, and A. Courville, "Deep Learning". MIT Press, 2016. http://www.deeplearningbook.org.

[54] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization", in *3rd International Conference on Learning Representations (ICLR) 2015, Conference Track Proceedings*. 2015. arXiv:1412.6980.

[55] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding", in *4th International Conference on Learning Representations (ICLR) 2016, Conference Track Proceedings*. 2016. arXiv:1510.00149.

[56] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal Brain Damage", in *Advances in Neural Information Processing Systems 2*, D. S. Touretzky, ed., p. 598. Morgan-Kaufmann, 1990.

[57] C. Louizos, M. Welling, and D. P. Kingma, "Learning sparse neural networks through l_0 regularization", in *6th International Conference on Learning Representations*. 12, 2018. arXiv:1712.01312.

[58] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both Weights and Connections for Efficient Neural Networks", in *Advances in Neural Information Processing Systems 28 (NIPS 2015)*. 2015. arXiv:1506.02626.

[59] T. Yang, Y. Chen, and V. Sze, "Designing Energy-Efficient Convolutional Neural Networks using Energy-Aware Pruning", in *CVPR*. 2017. arXiv:1611.05128.

[60] M. Zhu and S. Gupta, "To prune, or not to prune: Exploring the efficacy of pruning for model compression", in *6th International Conference on Learning Representations (ICLR) 2018, Workshop Track Proceedings*. 2017. arXiv:1710.01878.

[61] I. Hubara et al., "Quantized neural networks: Training neural networks with low precision weights and activations", *J. Mach. Learn. Res.* **18** (2017) 6869, arXiv:1609.07061.

[62] B. Jacob et al., "Quantization and training of neural networks for efficient integer-arithmetic-only inference", doi:10.1109/CVPR.2018.00286, arXiv:1712.05877.

[63] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training deep neural networks with binary weights during propagations", in *Advances in Neural Information Processing Systems*, C. Cortes et al., eds., volume 28, p. 3123. Curran Associates, Inc., 2015. arXiv:1511.00363.