OPEN-SOURCE FPGA-ML CODESIGN FOR THE MLPERF™ TINY BENCHMARK

Michaela Blott [1] **Hendrik Borras** [2] **Giuseppe Di Guglielmo** [3] **Javier Duarte** [4] **Nicolò Ghielmetti** [5] **Ben Hawks** [6]
Scott Hauck [7] **Shih-Chieh Hsu** [7] **Ryan Kastner** [4] **Jason Liang** [4] **Andres Meza** [4] **Jules Muhizi** [6 8] **Tai Nguyen** [4]
Rushil Roy [4] **Nhan Tran** [6] **Yaman Umuroglu** [1] **Olivia Weng** [4] **Aidan Yokuda** [7]

ABSTRACT

We present our development experience and recent results for the MLPerf™ Tiny Inference Benchmark on field-programmable gate array (FPGA) platforms. We use the open-source hls4ml and FINN workflows, which aim to democratize AI-hardware codesign of optimized neural networks on FPGAs. We present the design and implementation process for the keyword spotting, anomaly detection, and image classification benchmark tasks. The resulting hardware implementations are quantized, configurable, spatial dataflow architectures tailored for speed and efficiency and introduce new generic optimizations and common workflows developed as a part of this work. The full workflow is presented from quantization-aware training to FPGA implementation. The solutions are deployed on system-on-chip (Pynq-Z2) and pure FPGA (Arty A7-100T) platforms. The resulting submissions achieve latencies as low as $20\,\mu$s and energy consumption as low as $30\,\mu$J per inference. We demonstrate how emerging ML benchmarks on heterogeneous hardware platforms can catalyze collaboration and the development of new techniques and more accessible tools.

1 INTRODUCTION

Efficient implementations of machine learning (ML) algorithms in dedicated hardware devices at the *edge*, or near sensor, offer multiple advantages. Edge processing and data compression can greatly reduce downstream data rates and the energy required for data movement. Furthermore, real-time data processing and interpretation can accelerate decision making, hypothesis testing, and enable just-in-time interventions. These edge ML tasks can have a significant impact on a broad range of applications from *internet of things* (IoT) to Industry 4.0 (Kagermann et al., 2013) and new experimental methods for scientific discovery (Deiana et al., 2022).

To enable broader adoption of these technologies, we present our solutions for the open division of the MLPerf™ Tiny Inference Benchmark v0.7. MLPerf Tiny has two divisions for submitting results: a stricter closed division and a more flexible open division, which allows submitters to alter ML model implementations and training workflows. We participated in the open division to demonstrate the advantages of hardware-AI codesign.

The hls4ml (Duarte et al., 2018; Fahim et al., 2021) and FINN (Umuroglu et al., 2017; Blott et al., 2018b;a) teams aim to democratize low-power, *tiny* (tinyML Foundation, 2019; Banbury et al., 2020), accelerated ML by releasing accessible tools for the codesign of optimized neural networks on field-programmable gate arrays (FPGAs). The hls4ml workflow originates from the Fast Machine Learning for Science community, which focuses on developing tools for scientific applications. FINN is an open-source project from AMD that enables the exploration of efficient ML acceleration on FPGAs. These jointly developed solutions are the product of an ongoing collaboration between the FINN and hls4ml developers with the goal of making FPGA-accelerated tiny ML broadly available.

There are a number of unique features of the hls4ml and FINN workflows. Solutions support extreme flexibility in data type precision. In fact, each solution from the team uses a different precision, from 1- to 12-bit operations. The resulting hardware implementations are configurable, spatial, dataflow architectures that are tailored for speed and efficiency. The code, from end-to-end, is open-source and freely available including tools for design space explo-

---

[1]AMD Adaptive and Embedded Computing Group (AECG) Labs, Dublin, Ireland [2]Heidelberg University, Heidelberg, Germany [3]Columbia University, New York, NY, USA [4]University of California San Diego, La Jolla, CA, USA [5]European Organization for Nuclear Research (CERN), Geneva, Switzerland [6]Fermi National Accelerator Laboratory, Batavia, IL, USA [7]University of Washington, Seattle, WA, USA [8]Harvard University, Cambridge, MA, USA. Correspondence to: Javier Duarte <jduarte@ucsd.edu>.

ration and the final implementations. The workflow includes quantization-aware training (QAT) in QKeras (Coelho et al., 2021; Google, 2020) and Brevitas (Pappalardo, 2021), hyperparameter optimization using Determined AI (Determined AI, 2018) and KerasTuner (O'Malley et al., 2019), and FPGA implementation with hls4ml and FINN including Python APIs for inspection, validation, and deployment. Furthermore, one goal in these solutions is to develop a more unified workflow for quantized neural networks that is built on a common interchange format, called QONNX (Pappalardo et al., 2022; Xilinx, 2021). The envisioned workflow is depicted in Fig. 1.
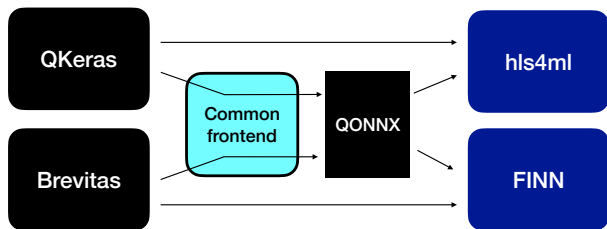

Figure 1. Common hls4ml-FINN FPGA codesign workflow based on QONNX.

The team consists of researchers collaborating across industry and academia. The submissions are available on the TUL Pynq-Z2 platform with a Zynq-7020 system-on-chip (SoC) and Digilent Arty A7-100T platform with an Artix-7 100T FPGA. The latter is the first submission on an FPGA-only platform. Open division submissions are provided for the keyword spotting, image classification, and anomaly detection MLPerf Tiny benchmarks. The resulting submissions on FPGA hardware were optimized for performance and speed with latencies as low as 20 μs. The paper is structured as follows. In Section 2, we briefly describe the benchmark tasks. Section 3 presents the model optimization for performance, latency, and resources. The integration of those models into hardware is detailed in Section 4. We summarize in Section 5.

In this work, in the spirit of the MLPerf process, we do not provide detailed comparisons of our solutions with solutions on other hardware platforms. However, the other submitted solutions for this benchmark can be found on the official MLPerf Tiny results page[1].

1.1 Previous Work

This work is built from a number of previous studies by the hls4ml (Duarte et al., 2018; Summers et al., 2020; Ngadiuba et al., 2020; Coelho et al., 2021; Aarrestad et al., 2021; Iiyama et al., 2021; Elabd et al., 2022) and FINN

---

[1] https://mlcommons.org/en/inference-tiny-07

teams (Umuroglu et al., 2017; Blott et al., 2018b;a). Other open-source efforts have explored ML-FPGA codesign. Surveys of existing toolflows can be found in Venieris et al. (2018); Guo et al. (2019); Shawahna et al. (2019); Abdelouahab et al. (2018). These workflows include fpgaConvNet (Venieris & Bouganis, 2017b;a;c; 2016), FP-DNN (Guan et al., 2017), DNNWeaver (Sharma et al., 2016), Caffeine (DiCecco et al., 2016), Snowflake (Gokhale et al., 2017), Vitis AI (Xilinx, 2021), FixyNN (Whatmough et al., 2019a;b), and others (Rahman et al., 2016; Majumder & Bondhugula, 2019; Hacene et al., 2020; Chang et al., 2021). The hls4ml and FINN workflows are unique with respect to other ML-to-FPGA workflows in two primary ways: the extreme configurability for low and arbitrary bit-precision and optimizing throughput using spatial dataflow architectures that resembles the flow of data through the NN on chip.

2 BENCHMARK TASKS AND MODELS

The MLPerf Tiny benchmark consists of four tasks and reference implementations related to image classification (IC), anomaly detection (AD), keyword spotting (KWS), and visual wake words (VWWs). We submitted solutions for the first three, which we describe in this section. The full set of tasks is described in greater detail in Banbury et al. (2021).

2.1 Image classification

Image classification is an important task for many autonomous and low-power embedded systems. CIFAR-10 (Krizhevsky et al., 2009) is a labeled subset of the 80 Million Tiny Images dataset (Torralba et al., 2008). The low resolution of the images make CIFAR-10 suitable for training tiny image classification models. It consists of 60,000 $32 \times 32 \times 3$ RGB images, with 6,000 images per class. The 10 different classes represent airplanes, cars, birds, cats, deers, dogs, frogs, horses, ships and trucks. The dataset is divided into 50,000 training images and 10,000 testing images.

The reference IC model for the closed division is a customized version of ResNetV1 (He et al., 2016; Banbury et al., 2021) which takes as input $32 \times 32 \times 3$ images and outputs a probability vector of size 10. This customized model is composed of three residual residual stacks rather than four. Each stack consists of three convolutional layers. Moreover, the first convolutional layer is not followed by the pooling layer due to the low resolution of the input data. The number of convolutional filters and strides are also lower compared to the official ResNet.

A subset of 200 images from the CIFAR-10 test set are selected to evaluate the performances of the IC reference implementation. For the v0.5 benchmark a class-unbalanced

subset was chosen, whereas for the v0.7 benchmark the subset was updated to maintain class balance. The reference model achieves 87.0% accuracy across the 200 testing images of the v0.7 benchmark.

2.2 Anomaly detection

Anomaly detection is a task that requires separating normal and anomalous signals in various data formats. For this benchmark as defined by MLPerf Tiny, an unsupervised approach is developed to train the neural network to closely match industrial use-cases where normal behavior may be well defined because it is less feasible to collect every possible anomalous signal and train a binary classifier in a supervised learning approach.

The unsupervised AD model is trained on the DCASE 2020 Challenge Task 2 dataset which employs the ToyAD-MOS (Koizumi et al., 2019) ToyCar dataset. The dataset is comprised of 10 s WAV files. The full set is split into 7,000 normal audio files for training and 2,459 for testing.

Before training on the audio files, we preprocess them into mel spectrograms of 128 bands describing each 32 ms interval. The model is then trained on a sliding window of five frames of the spectrogram yielding an input size of 640. We use an autoencoder NN structure that attempts to recreate the input. We calculate the mean-squared error (MSE) between the input and output and average it over each of the windows ($196\times$). We use a smaller version of the MLPerf Tiny AD autoencoder network that has 128 inputs with an encoder and decoder comprised of two quantized 72-unit fully-connected (FC) layers with batch normalization (BN) and ReLU activation. An FC layer is also used as the output layer. To evaluate the model performance, we average the MSE over each the windows in the audio sample to compute an anomaly score. To set the threshold between normal and anomalous sounds, we use the receiver operating characteristic (ROC) curve and the corresponding area under the curve (AUC) as the quality metric.

2.3 Keyword spotting

Over the last decade, keyword spotting has become increasingly prevalent, especially in modern voice assistants. Running a full speech recognition system only to detect an activation word is often impractical because of power implications and privacy concerns. Instead, modern devices only listen for an activation word, a specific keyword. Since recognizing a limited set of words is significantly simpler than full speech recognition, the keyword spotting system can be run locally on a given device and with low power impact. Keyword spotting is also interesting for general robot control, by setting a vocabulary with words such as "start," "stop," "louder," and "quieter."

For the MLPerf Tiny benchmark the KWS task is based on the Google speech commands dataset V2 (Warden, 2018). The dataset consist mainly of 1 s audio files, each containing one spoken word. In total 105 829 data samples are available, recorded by different speakers. The data samples are partitioned into training, validation, and test sets, such that a given speaker only appears in one of the sets. Additionally, longer files with background noises are included. Overall the dataset contains 35 classes, each representing the utterance of one word. The dataset is however more often used in its twelve-class variant, where ten fixed classes are used and the additional 25 classes are grouped into the unknown class, while also adding a new class called silence, comprised of samples from the included background noises. For MLPerf Tiny, this twelve-class variant is used. Along with sample code for setting up the pre-processing for the dataset MLPerf Tiny also provides a reference model for the KWS task, which is a depthwise separable convolutional neural network (DS-CNN) from Zhang et al. (2017). The DS-CNN model is relatively compact and optimized for low-power microcontrollers. For on-device testing, 1,000 samples from the Google speech commands test test are selected for the benchmark. Over this subset, the reference model achieves an accuracy of 92.2%.

3 MODEL DEVELOPMENT AND CODESIGN

An overview of the models developed for the submission are presented in Table 1. Two models for the IC task one each for the AD and KWS tasks were submitted. Below, we will describe in detail the model architecture and optimizations, both for training and implementation, that were performed for each of the models for the two FPGA hardware platforms considered. To optimize performance, all of the FPGA neural network do not use off-chip memory. However, to measure the performance of the models using the MLPerf Tiny benchmarking suite, the ARM processing system uses off-chip memory which would not necessarily be required in a standalone design. Each model is developed through hardware-software codesign to search for Pareto-optimal solutions in model accuracy and resource usage by tuning a number of design machine learning and hardware architecture parameters. The solutions are not configurable at runtime for optimized performance.

Benchmark	Flow	Prec. [bits]	Params.	Accuracy
IC	hls4ml	8–12	58 115	83.5%
IC	FINN	1	1 542 848	84.5%
AD	hls4ml	6–12	22 285	0.83 AUC
KWS	FINN	3	259 584	82.5%

Table 1. Summary of models submitted for the v0.7 benchmark including benchmark task, tool flow used, precision of model, number of parameters, and performance—by default this is accuracy unless denoted as AUC

3.1 Optimization for IC with hls4ml

To find models that simultaneously accomplished the goals of high accuracy, low latency, low resource usage (such that they can be accommodated on the chosen FPGA platforms), and low power utilization, a sequence of neural architecture search (NAS), QAT with QKeras, configuration and implementation with hls4ml, and model- and hardware-centric optimizations were performed.

3.1.1 Bayesian Optimization

For the NAS, the MLPerf Tiny benchmark reference ResNet-8 model was chosen as a starting point. The model was generalized in several ways to allow a restricted NAS. In particular the tunable hyperparameters include the total number of stacks, the number of filters, filter size, and strides in each convolutional layer, whether average pooling is applied before the final dense layer, and whether whether skip connections are enabled.

We perform several Bayesian optimization (BO) scans using KerasTuner. We consider 1-, 2-, and 3-stack models in separate scans of 100 models each. For each scan, we consider 2, 4, 8, or 16 filters, filter sizes of 1, 2, or 3, and particular strides to allow for valid skip connections. We use a batch size of 32 and train each model for 10 epochs. During training, the input data is normalized by dividing by 256. For each model, we compute the best test accuracy and the number of floating point operations (FLOPs) (Yoshioka, 2020). The results of these BO scans are shown in Fig. 2. We generally find that 1-stack models generally provide a good balance of a smaller number of FLOPs, while maintaining high accuracy. The number of filters has the biggest impact on the accuracy and FLOPs of the models. Larger stride lengths and smaller filter sizes can also reduce FLOPs at the cost of some accuracy. Neither average nor max pooling gave significant improvement.

BO allowed us to narrow down our choices to a very few models by revealing the most important hyperparamter values. From the results of our scans, we found a 1-stack model (3 convolutional layers with 32, 32, and 32 filters, kernel sizes of 3, 3, and 3, and strides of 4, 4, and 1, respectively, and an FC layer with 2048 units) used for the v0.5 submission that achieves a test accuracy of 75.0% for 2.5 MFLOPs and 12.8 MFLOPs, respectively. We also found a 2-stack model with no skip connections (5 convolutional layers, with 32, 4, 32, 32, and 4 filters, kernel sizes of 1, 4, 4, 4, and 4, and strides of 1, 1, 1, 4, 1, respectively, and an FC layer with 2048 units) used for the v0.7 submission that achieves a test accuracy 83.5% for 12.8 MFLOPs. Compared to the reference model, which corresponds to an accuracy of 87.0% for 25.0 MFLOPs, these optimized models represent a substantial reduction of FLOPs for a minor reduction in accuracy. After settling the model architecture, we performed
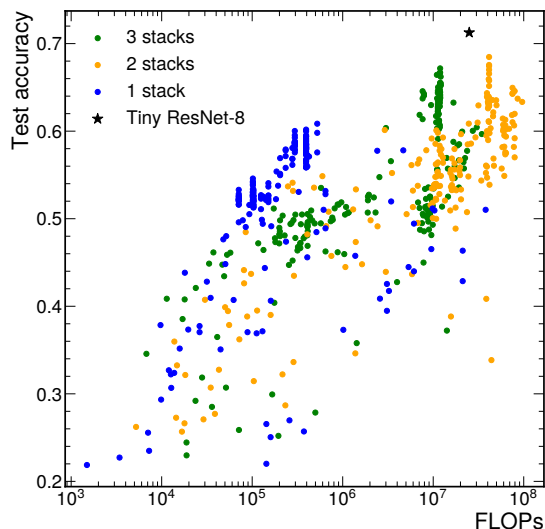

Figure 2. Results of the BO scans for 1-, 2-, and 3-stack models. The star represents the test accuracy achieved after 10 epochs with tiny ResNet-8 model

QAT with QKeras (Google, 2020; Coelho et al., 2021) using a fixed-point precision of 8 total and 2 integer bits. Because the benchmark only measures the top-1 accuracy of the model, it is only necessary to return the class predicted to be highest probability. Since softmax layer is monotonic in the input logits, it is not necessary to find the top-predicted class (i.e. a simple max applied to the logits is sufficient), and thus it is removed for inference.

3.1.2 FIFO Buffer Depth Optimization

By exploiting the dataflow preprocessor directive (or *pragma*) of Vivado HLS, each layer composing a neural network is connected with the rest of the model through first-in first-out (FIFO) buffers. The implementation of the FIFO buffers contribute to the overall resource utilization of the design, impacting in particular the block random access memories (BRAMs) or look-up table (LUT) utilization. Because the neural networks can have complex architectures generally, is hard to know a priori the correct depth of each FIFO buffer. In order to reduce the impact on the resources used for FIFO buffer implementation, an optimization has been developed which aims to correctly size the depth of the FIFO buffers by analyzing the data produced by the register-transfer level (RTL) simulation. We implemented this FIFO buffer resizing within the hls4ml framework as an optimization pass. Through RTL simulation with large FIFO buffers, we estimate the maximum occupation of each FIFO. Once the maximum depth is determined, the optimization pass sets the FIFO buffer depth to that value plus 1. In Table 3,

the FPGA resource usage for the hls4ml IC model is shown with and without the optimization. This optimization significantly reduces the FPGA resources enabling the deployment of larger models.

For FINN an equivalent optimization exists, which was applied to all FINN-based models in this submission. Fundamentally the optimization executes very similar steps to the optimization in hls4ml, running PyVerilator (Wright et al., 2020) on the full design, to perform an estimation for the optimal FIFO buffer depths between the layers of a given neural network design. The found FIFO buffer depths are then saved in the internal ONNX representation and are applied at a later step. Even though the simulation of the whole model in an RTL simulation is time consuming, this approach has proven useful for many FINN models, such that it is now part of the default compiler flow in FINN.

Table 2 shows a summary of the FIFO buffer sizes set with this optimization for both hls4ml and FINN.

Benchmark	Flow	FIFO optimization	FIFO size
IC	hls4ml	enabled	1–1066
IC	FINN	enabled	2–512
AD	hls4ml	disabled	1
KWS	FINN	enabled	32–64

Table 2. Summary of FIFO buffer sizes for models submitted for the v0.7 benchmark. For the hls4ml FIFO optimization the FIFO buffer sizes can take an arbitrary integer values, while for FINN they can only be powers of two. No FIFO optimization was performed for the AD model.

3.1.3 ReLU Layer Merging

As mentioned in the previous section, each dataflow stage consists of a neural network layer, which are linked together by FIFOs that cost BRAMs, LUTs, and flip flops (FFs). By default in hls4ml, each rectified linear unit (ReLU) layer is implemented as its own dataflow stage. Because each additional dataflow stage costs extra logic and FIFOs, we reduce the resource utilization by merging the ReLU activation function into the layer preceding it. Although the layers with the newly merged ReLU functionality use more logic than before, there is still a net decrease in resources. Table 3 shows the resulting resource utilization reductions.

	BRAM [18 kb]		FF		LUT	
Available	280		106 400		53 200	
Without opt.	477	170.4%	79 177	74.4%	66 838	125.6%
With FIFO opt.	278	99.3%	72 686	68.3%	58 515	110.0%
With ReLU opt.	345	123.2%	72 921	68.5%	55 292	103.9%
With all opt.	146	52.1%	66 430	62.4%	46 969	88.3%

Table 3. Resource estimates from Vivado HLS for the IC model with hls4ml for the v0.7 MLPerf Tiny submission

3.2 Optimization for IC with FINN

The model submitted for the IC task with FINN is called CNV-W1A1 from Umuroglu et al. (2017). The model architecture takes inspirations from BinaryNet (Hubara et al., 2016) and VGG-16 (Simonyan & Zisserman, 2015), consisting of first multiple convolutional blocks and then fully connected layers at the end. The whole architecture can be described as follows:

- Three convolutional blocks, consisting of two 3×3 convolutions and one 2×2 max pooling layer at the end. The convolutions in each of these blocks have the following number of channels respectively: 64, 128, 256.
- The network then continues with two fully connected layers with 512 neurons and one output layer with 10 neurons.
- Finally a top-k layer is inserted to calculate the classification result in hardware.

Since the original release of the FINN paper the framework has been extended to support arbitrary bit widths, meaning that weight and activations with more than one bit can also be synthesized. However, bit widths below eight bit are generally recommended for FINN, due to how the underlying activation implementation scales with bit width. As such the CNV model also exists in variants with two bit weights and activations. For the MLPerf Tiny submission, the binary version of the model is used. Here, the weights and activations are quantized to a bipolar representation, with the notable exception being the input layer, which processes the input images as 8-bit data. Consequently the activation function associated with the input layer performs an eight bit calculation, while all other layers of the network work with a binary representation of the weights and activations.

3.2.1 ASHA for IC with FINN

We used the adaptive ASHA algorithm (Li et al., 2020) from Determined AI to search for a more efficient or accurate model. The starting point for the scan was the CNV-W1A1 model. The hyperparameters that were varied were the number of convolutional filters (from 32 to 512), whether or not to pool after convoluational layers, strides (from 1 to 4), kernel sizes (from 1 to 4), pooling size (2 or 4), number of neurons in fully connected layers (from 16 to 512), and activation and weight bit widths (1 or 2), The adaptive scan allocates a set of resources to scan varied hyperparameter configurations, throwing out the worse half based on the specified validation metric and repeating until only one optimal configuration remains. A batch size of 50 was used and each model was trained for up to 100 epochs, although the adaptive ASHA algorithm may terminate training earlier.

For each model, several inference cost metrics are computed

including total bit operations (BOPs) (Baskin et al., 2018; Hawks et al., 2021) and the total number of bits needed to store the weights in memory (WM). BOPs count the number of multiply-accumulate operations in a neural network multiplied with the bit width at which this operation is performed. For a single convolutional layer with b_{w}-bit weights and b_{a}-bit activations containing n input channels, m output channels, and $k \times k$ filters,

$$\mathrm{BOPs} \approx mnk^2(b_{\mathrm{a}}b_{\mathrm{w}} + b_{\mathrm{a}} + b_{\mathrm{w}} + \log_2 nk^2). \quad (1)$$

For computing BOPs for fully connected layers, we set $k = 1$ in Eq. 1. This metric functions as a preliminary estimate for the FPGA resource usage of the network implemented with FINN and thus allows for a first comparison between networks before any synthesis takes place. These two metrics, BOPs and WM, were inspired by the ITU AI for Good Challenge: Lightning-Fast Modulation Classification with Hardware-Efficient Neural Networks (International Telecommunication Union, 2021). A summary inference cost metric also inspired from that competition is defined as

$$C = \frac{1}{2}\left(\frac{\mathrm{BOPs}}{\mathrm{BOPs}_{\mathrm{CNV\text{-}W1A1}}} + \frac{\mathrm{WM}}{\mathrm{WM}_{\mathrm{CNV\text{-}W1A1}}}\right), \quad (2)$$

where the CNV-W1A1 model is taken as reference for comparison. Figure 3 shows the result of the scan in terms of accuracy as a function of the inference cost. Based on our results, the CNV-W1A1 model performs near optimally.

3.3 Optimization for AD with hls4ml

When implementing the reference AD multilayer perceptron (MLP) model on the Pynq-Z2 and Arty A7-100T platforms, the limiting resource was LUTs. The reference floating point implementation was too large to synthesize, therefore, we optimized the architecture using quantization and model compression techniques with minimal AUC performance reduction. As a part of this process, we implemented a generic optimization to fold the batch normalization (BN) into the FC layer and reduced the the depth and width of the autoencoder network.

3.3.1 QDenseBatchnorm Layer

To remove the LUT utilization from the BN computation, we developed a new quantized FC layer in QKeras (Muhizi, 2021) where during the forward pass, we compute the matrix multiplication of the FC layer as well as BN in the same pass and then save a kernel that "folds" the FC kernel with the BN parameters into a folded kernel by transforming the dense kernel with the BN parameters as defined in equation 3. Furthermore, within the same pass, both the BN parameters as well as the folded kernel are updated. During the forward pass, we compute the FC layer outputs then pass the outputs
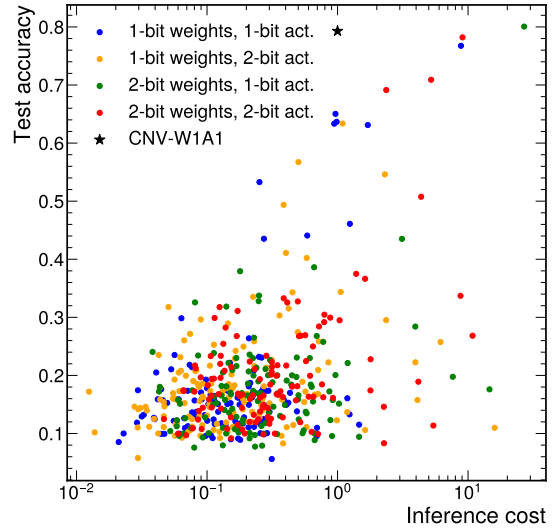

Figure 3. Results of ASHA scan in terms of accuracy as a function of the inference cost hardware metric. The CNV-W1A1 used in the submission is shown for reference (inference cost of 1), with its test accuracy after 100 epochs of training

into a BN pass in order to update the BN parameters. Once we have the new BN parameters, we then fold them into the FC layer parameters as,

$$k_{\mathrm{folded}} = vk_{\mathrm{FC}} \quad (3)$$
$$b_{\mathrm{folded}} = v(b_{\mathrm{FC}} - \mu) + \beta \quad (4)$$

where $v = \gamma\sqrt{\sigma^2 + \epsilon}$ with μ and σ the moving mean and standard deviation and ϵ and β the BN scale and shift parameters. γ represents a learned scale factor in the original kernel. k_{FC} (k_{folded}) represents the original (folded) kernel, and b_{FC} (b_{folded}) is the original (folded) bias.

3.3.2 Reuse Factor, Input Size, and Layer Depth

To further tune the LUT utilization for the model, we varied the parallelization of the algorithm via the *reuse factor* (RF), or how many times each multiplier unit is used, which controls the parallelization of the algorithm. With the goal of a low latency solution, we optimized for the lowest RF factor. We then performed a scan across the available RFs and synthesized onto the Pynq-Z2, while also tracking the overall resource utilization on the FPGA. After the scan, we found that the smallest RF deployable on the FPGA is 144. Table 4 summarizes the optimizations from the reference model to the final submitted model. By reducing the number of hidden layers and width of each layer from 9 to 5 and 128 to 72, respectively, we reduce the key bottleneck, the LUT count, to 161 228. Combined with downsampling of

the input from 640 to 128, our optimizations achieve a final 58.5% utilization of the FPGA LUTs on the Pynq-Z2.

	AUC	FF		LUT	
Available	–	106 400		53 200	
Reference	87.1%	–	–	–	–
With folding	68.1%	161 228	151.5%	221 063	451.5%
With downsampling	81.4%	55 341	52.0%	35 366	66.5%
With all opt.	83.3%	44 300	41.6%	31 094	58.5%

Table 4. Resource utilization from Vivado HLS logic synthesis on the Pynq-Z2 for the hls4ml AD model with various optimizations at 144 reuse factor.

3.4 Optimization for KWS with FINN
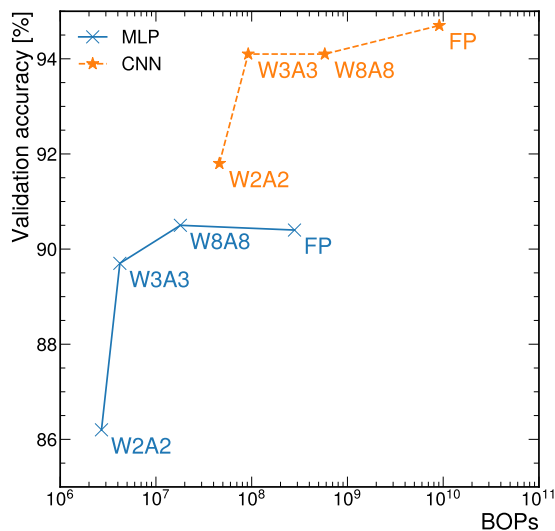
Figure 4. Quantization exploration for keyword spotting. Each data point is annotated with its weight and activation bit width, in the following schema: WnAm, with n the weight bit width and m the activation bit width. Additionally, floating point data points are annotated with FP. The model accuracy on the validation dataset is shown on as a function of the network complexity quantified by BOPs.

The model submitted for keyword spotting and synthesized by FINN is directly inspired by Zhang et al. (2017). Initially, two architectures from this paper were explored, the MLP and the convolutional neural network (CNN). However, given the more complex rectangular convolutions employed in the CNN, the MLP was chosen for its simplicity. The architecture of the MLP consists of three fully connected (FC) layers, each with BN and ReLU activations. One final output layer with 10 neurons is also used. Similar to the image classification network synthesized with FINN an in-hardware top-k node was inserted at the end.

A weighted cross-entropy loss was employed during train-

ing. The re-weighting largely suppresses the unknown label in the dataset to combat the imbalance between classes, where for the 12 class version of the google speech commands V2 dataset, the unknown label is present about 17 times more often than any other label. The exact suppression setting for the unknown label was then found by running an adaptive ASHA hyperparameter search (Li et al., 2020). Additionally, the training was managed using Determined AI (Determined AI, 2018), which allowed for easy integration of the adaptive ASHA algorithm into the general training flow. For the feature extraction, mel-frequency cepstral coefficients (MFCC) were adapted and implemented as done for the closed division reference implementation. The primary optimizations for the KWS submission included investigating the different quantization settings for activations and weights. QAT was performed with Brevitas (Pappalardo, 2021), which is an extension to the popular machine learning framework PyTorch (Paszke et al., 2019). The exploration process is shown in Fig. 4. Here, the model accuracy on the validation dataset is plotted against the network complexity quantified by BOPs. To find the appropriate quantization for the network, a reference model was first trained at floating-point precision. After this, the bit widths of the weights and activations were successively lowered, until the network validation accuracy dropped significantly. This sudden decrease was found for both the CNN and MLP to be below three bits for weights and activations. Thus, 3-bit quantization was chosen for the submission. However, notably the network input is 8 bits.

3.5 Automatic Optimizations for FINN Models

In addition to the optimizations applied to the IC and KWS models described above, FINN automatically applies multiple optimizations before a design is synthesized.

As a first step, FINN applies constant folding. Here, constant initialized tensors are propagated through the ONNX (Bai et al., 2019) computation graph, and nodes that return the constant values are precomputed at compile time. This basic optimization can save significant compute overhead and makes most networks easier for the compiler to handle.

Afterwards, a graph transformation called streamlining is applied. This transformation is a direct application of Umuroglu & Jahre (2017). The operation folds layers, which are usually computed at floating-point precision into integer operations for uniformly quantized neural networks. For FINN, this primarily affects BN layers, which are folded into multi-threshold nodes, which can represent arbitrarily quantized activation functions. In addition to removing time-consuming floating-point operations, this optimization eliminates some runtime computations entirely.

Before layers are converted into generated HLS code,

FINN will minimize the final accumulator datatypes for all threshold-based operations. In particular, FINN minimizes the memory footprint for all activation layers. After the individual layers of a network have been synthesized into Vivado intellectual property (IP) blocks, FINN performs a FIFO buffer optimization to balance its dataflow pipeline. This method is conceptually similar to the recently implemented optimization pass in hls4ml (see Sec. 3.1.2) and achieves similar performance improvements.

4 SYSTEM INTEGRATION

4.1 QONNX Interchange Format

Currently, all networks synthesized with hls4ml were trained with QKeras and all submissions synthesized with FINN were trained with Brevitas. In the future, we plan to be able to synthesize models trained with either quantization aware training (QAT) library in both compiler frameworks. The central component to this process is an interchange format called QONNX (Pappalardo et al., 2022; Xilinx, 2021), which is an extension to the ONNX standard (Bai et al., 2019). It introduces new quantization nodes for arbitrary uniform quantization, as is required by both frameworks. Already now FINN and Brevitas both fully support the QONNX format, since version 0.7 of both frameworks, and the KWS submission already uses this format. QONNX should enable simpler and faster exchange of QAT models between different FPGA-ML flows like hls4ml and FINN.

4.2 ML Accelerators

4.2.1 hls4ml and FINN for Dataflow Architectures

The hls4ml flow generates the C++ code listed on the top of Fig. 5 as the top-level module for the HLS-synthesizable accelerator. The module has memory-mapped registers and interfaces (lines 1–4) that allow the accelerator to be programmed, and to read and write data from the off-chip memory, which is shared between the accelerator implemented on the programmable logic and the application running on the processor core. The `bundle` keyword on the `INTERFACE` pragma specifies the name of the ports as shown in the generated Vivado IP core. The port `CTRL_BUS` groups control registers to program, start, and check the status of the accelerator (line 2). The bandwidth and throughput of the accelerator can be increased by creating multiple ports (lines 3–4) to load and store data in the local buffers of the accelerators. This accelerator style, defined as loose out-of-core with direct memory access to main memory, is typical for high-throughput applications that have clear memory access patterns and have input sizes large enough to make vector-processing impractical (Cota et al., 2015).

To further increase performance, the hls4ml accelerator uses a dataflow implementation to produce and consume data in

```
1  void top_module_axi(input_axi_t in[N_IN], output_axi_t out[N_OUT])
2  {
3      #pragma HLS INTERFACE s_axilite port=return bundle=CTRL_BUS
4      #pragma HLS INTERFACE m_axi port=in offset=slave bundle=IN_BUS
5      #pragma HLS INTERFACE m_axi port=out offset=slave bundle=OUT_BUS
6
7      hls::stream<input_t> in_local("input_1");
8      #pragma HLS STREAM variable=in_local depth=N_IN_LOCAL
9      hls::stream<output_t> out_local("output_1");
10     #pragma HLS STREAM variable=out_local depth=N_OUT_LOCAL
11
12     for(unsigned i = 0; i < N_IN / input_t::size; ++i) {
13         input_t ctype;
14         #pragma HLS DATA_PACK variable=ctype
15         for(unsigned j = 0; j < input_t::size; j++) {
16             ap_ufixed<16,8> tmp = in[i * input_t::size + j];
17             ctype[j] = typename input_t::value_type(tmp >> 8);
18         }
19         in_local.write(ctype);
20     }
21
22     module(in_local, out_local);
23
24     for(unsigned i = 0; i < N_OUT / output_t::size; ++i) {
25         output_t ctype = out_local.read();
26         for(unsigned j = 0; j < output_t::size; j++) {
27             out[i * output_t::size + j] = output_axi_t(ctype[j]);
28         }
29     }
30 }
```
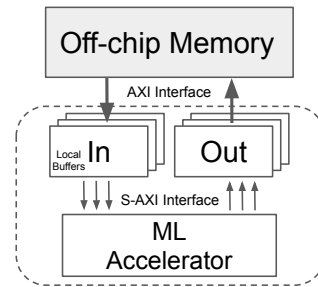

Figure 5. Code for the hls4ml top-level module and accelerator architecture.

a streaming fashion, thus the local buffers are implemented as FIFOs (line 6–9). Finally, the interface uses specialized large-width data types, such as the arbitrary precision integer data type `ap_int<W>` in Vivado HLS, where `W` is a data width (64, 128, 256, etc.) to increase the bandwidth of the data communication between the off-chip memory and local buffers. The data movers in the HLS code are in charge of unpacking and packing the data to and from the ML processing core of the accelerator (lines 11–19 and 23–28).

While FINN produces an accelerator, which looks similar to the one from hls4ml at a conceptual level, the build process is significantly different. Both FINN and hls4ml produce a dataflow style accelerator that can be easily integrated into existing designs using the Vivado IP integrator. Both accelerators exploit a streaming architecture, which keeps the activation data of the neural network on-chip, thus reducing overall data movement. However, while hls4ml builds the whole accelerator from one top level module, FINN builds a similar design by interconnecting multiple IP blocks in Vivado. Here, each IP block represents one layer of the neural network. This means in particular that only the final IP block stitching must be run in series, while the synthesis

of each IP block can be done in parallel.

4.2.2 IP Integration

We used the Xilinx Vivado Design Suite 2019.1 to instantiate and interconnect IP cores from the Vivado IP catalog and the hls4ml and FINN codesign workflows. We first interactively used the IP integrator design canvas to develop an automated flow using the Tcl programming interface. In particular, we integrated most of the IP cores at the advanced extensible interface (AXI) level, but we also worked at the port-and-constraint level to interface the device under test (DUT) with the Embedded Microprocessor Benchmark Consortium (EEMBC) performance and power analysis setup.

Fig. 6 shows the main components for the integration on both Zynq SoCs and pure FPGA chips. A Zynq SoC combines hard cores, e.g., ARM Cortex-A, of the processing system (PS) and with the flexibility of the programmable logic (PL). AXI ports connect the PL with the off-chip memory through the PS. In Fig. 6a, we integrate both FINN and hls4ml accelerators with AXI buses to support both the accelerator control (`s_axi`) and data movement (`m_axi`). Fig. 6b shows the design on a pure FPGA, where we similarly instantiate the accelerator, but we integrate a soft processor (MicroBlaze) on the PL instead. The memory controller (MIG) and the on-chip memory (OCM) are instantiated as soft IPs as well. For our experiments, we sized the MicroBlaze instruction and data cache in the range 1–16 kB and the on-chip memory in the range 32–128 kB to balance BRAM usage and software performance.
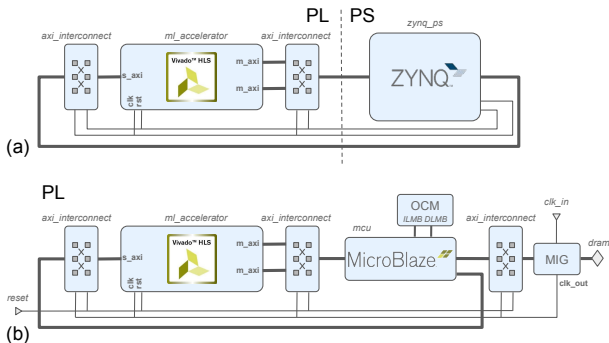

Figure 6. Acceleration integration for (a) SoC and (b) FPGA-only designs

4.2.3 Experimental Results on Development Boards

In our experimental setup we used two off-the-shelf development boards: the TUL Pynq-Z2 and Digilent Arty A7-100T boards. The TUL Pynq-Z2 board is based on a Xilinx Zynq SoC and designed for the Xilinx University Program to support the Pynq software stack. The Zynq SoC on the board (`xc7z020-1clg400c`) combines an ARM dual-core Cortex-A9 processor at 650 MHz with 13,300 logic slices (four 6-input LUTs and eight FFs), 630 kB of BRAM, and 220 DSP slices.

The Digilent Arty A7-100T board is based on Xilinx Artix-7 technology and designed for low-power and low-cost applications. The FPGA chip (`xc7a100t-1csg324`) comes with 15,850 logic slices (four 6-input LUTs and eight FFs), 607.5 kB of BRAM, and 240 DSP slices. Table 5 reports the final resource usage after placement and routing for all designs implemented on both platforms.

We can directly compare the two different solutions—one based on hls4ml and one based on FINN—that were submitted on the same hardware platforms for the IC benchmark task. First, we note some differences in the model design. The hls4ml IC model is a relatively small CNN (58 115 parameters) implemented using fixed-point precision weights and activations with bit widths in the range 8–12, while the FINN IC model is significantly larger (1 542 848 parameters), but implemented with binary weights and activations. Thus while the FINN IC model implements more operations, they are each less computationally expensive.

Another distinction between the models is the chosen resource-latency tradeoff. The hls4ml IC model utilizes 58% fewer BRAMs compared to the FINN IC model for the Pynq-Z2 platform. However the latency is 18.2 times larger, the bulk of which is required by the penultimate convolutional layer (6.6 times longer latency than the next slowest layer). The hls4ml streaming architecture chosen is such that the 32×32 input image size is iterated over sequentially. For each iteration, the inputs are assembled into the corresponding $4 \times 4 \times 32$ input tensor for a single kernel multiplication and up to 16 384 multiplications are performed sequentially, resulting in 32 outputs per kernel multiplication. Thus while the resource usage is kept to a minimum, the worst-case latency scales approximately as $32 \times 32 \times 16 384$ clock cycles. In future submissions, we plan to more efficiently pipeline these operations to substantially reduce the latency of the hls4ml IC model.

4.3 Software Integration

4.3.1 Bare-Metal Setup

In our setup, the processor, or microcontroller, is in charge of initiating the memory with the benchmark data, programming the accelerator, starting it, and waiting for its completion with polling on a register. Finally, we compare the correctness of the accelerator outputs against precomputed reference outputs. The accelerator responds to the initial configuration from the processor, and then autonomously transfers data between off-chip memory and local buffers. The communication between processor and accelerator always uses memory-mapped I/O. The processor can directly

Model	LUT		LUTRAM		FF		BRAM [36 kb]		DSP		Latency [ms]	Energy/inf. [μJ]
Pynq-Z2												
Available	53 200		17 400		106 400		140		220		–	–
IC (hls4ml)	28 544	53.7%	3 756	21.6%	49 215	46.3%	42	30.0%	4	1.8%	27.3	44 330
IC (FINN)	24 502	46.1%	2 086	12.0%	34 354	32.3%	100	71.4%	0	0.0%	1.5	2 535
AD	40 658	76.4%	3 659	21.0%	51 879	48.8%	14.5	10.4%	205	93.2%	0.019	30.1
KWS	33 732	63.4%	1 033	5.9%	34 405	32.3%	37	26.4%	1	0.5%	0.017	30.9
Arty A7-100T												
Available	63400		19 000		126 800		135		240		-	-
IC (hls4ml)	39 126	61.7%	5 877	30.9%	59 184	46.7%	50	37.0%	6	2.5%	33.1	73 166
IC (FINN)	32 096	50.6%	3 154	16.6%	39 962	31.5%	113.5	84.1%	2	0.8%	1.5	3 419
AD	51 429	81.1%	5 780	30.4%	61 639	48.6%	22.5	16.7%	207	86.3%	0.045	98.4
KWS	42 518	67.1%	1 634	8.6%	43 157	34.0%	59.5	44.1%	2	0.8%	0.033	53.7

Table 5. Resource usage, latency, and energy per inference for the submitted models implemented on the Pynq-Z2 and Arty A7-100T platforms.

read and write registers of the accelerator interface that are accessible using pointers in C/C++.

The EEMBC benchmarking framework requires a serial console connected to the board to view and process the standard output from `printf` statements. We also use a programmable baud rate for both the Zynq and Microblaze designs with an AXI universal asynchronous receiver transmitter (UART) Lite IP for the latter.

4.4 EEMBC EnergyRunner™ and Test Harness

In order to submit official results to the benchmark, we must use the benchmarking framework that consists of two pieces of software: the EEMBC EnergyRunner™ (runner) and test harness. The former runs on a host computer and the latter on the DUT.

The runner application runs on a host computer and communicates over a serial connection with the DUT and other hardware required to perform various benchmark measurements. The runner software is responsible for configuring benchmark hardware, sending input samples to the DUT, and calculating benchmark metrics such as latency, network accuracy/AUC, and energy used per inference based on data the DUT and other hardware report.

The test harness is provided as partially implemented C++ code that must be integrated onto the DUT. The harness communicates with the runner, with functionality including basic command parsing and benchmark-related operations already implemented (and unable to be modified). More DUT-specific operations need to be implemented by the submitter, such as loading input data into the accelerator, running a batch size of 1 inference, timer functionality, and general hardware setup. We implemented the required functionality for the Pynq-Z2 and Arty A7-100T in separate instances of the test harness, and ran the test harness as a bare-metal application, programming it and the applicable bootloader into the DUT's memory to launch the application
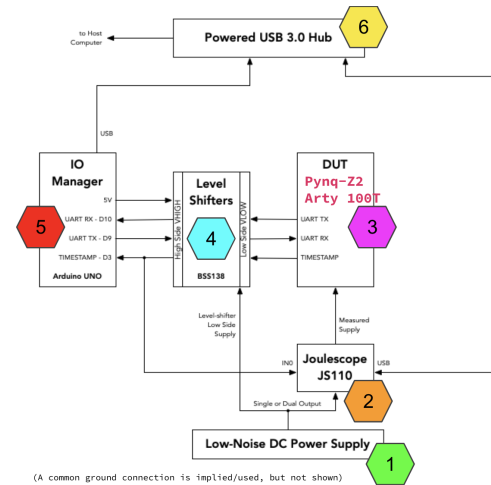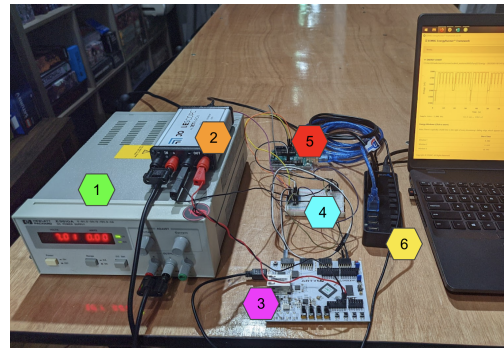

Figure 7. Top: Image of device testing setu. Bottom: Block diagram of testing setup with numerical labels corresponding to the top image.

upon a device restart.

4.4.1 Performance and Accuracy Measurements

When running the benchmark in performance mode, we test the latency of the accelerator, and, as our submission is in the open division, we measure the accuracy over the whole test dataset. The physical setup of performance mode

consists of the DUT, programmed with the test harness, connected over a serial port to a host computer running the runner application. In our case, both the Arty A7-100T and Pynq-Z2 were connected over a USB serial connection to a host PC running either Microsoft Windows 10 or Linux. For the latency test, a total of 5 samples are sent to the DUT one at a time. For each sample, the DUT performs sufficient batch-1 inferences to accumulate at least 10 s of continuous accelerator run time. Once complete, the runner calculates the median latency to perform a single inference over the 5 different samples. To perform the accuracy test, each sample in the entire test dataset for a given benchmark is sent to the DUT one at a time. The DUT runs all single-sample inferences, after which the results are returned to the runner to compute the overall accuracy/AUC. Logs of both benchmarks are recorded, then submitted along with the code into the official MLPerf Tiny v0.7 GitHub repository.

Table 5 shows the latencies and Table 1 shows the measured accuracies. The submitted designs are comparable in accuracy or AUC to the reference models and have latencies ranging from 30 ms to 20 μs, demonstrating the high throughput achievable with this approach.

4.4.2 *Energy Consumption Measurements*

To run the benchmark in energy mode, the hardware setup is more complex than in performance mode, as can be seen in Fig. 7. It is comprised of a host computer, the DUT, an energy monitor that measures and records the power used over a set interval, an IO Manager (Arduino UNO) running firmware to act as a serial bridge between the host computer and DUT for power isolation and serial port stability when power cycling the DUT, and level shifters between the IO Manager and the DUT.

Minor modifications are also made to the test harness when running in energy mode. First, the baud rate of the DUT is changed from 115 200 to 9 600, which is required to communicate with the IO Manager. Additionally, the time measurement protocol is changed from the DUT's internal timer to holding a GPIO pin that is connected to the energy monitor low for at least 10 μs, as the energy monitor manages the timer in this mode.

When running the energy benchmark, the methodology is nearly the same as the latency test, except that the power utilization of the DUT is also recorded by the energy monitor, and the energy per inference is also taken as the median over all of the samples. We used the Joulescope JS110 as our energy monitor, and an HP/Agilent E3610A power supply to power the DUT and energy monitor.

The measured energies per inference for each design are shown in Table 5. They vary from 70 mJ to 30 μJ per inference depending on the task and hardware platform, demonstrating the relatively low energy consumption possible with our workflows.

5 SUMMARY

This paper details the hls4ml-FINN solutions implementing field-programmable gate array (FPGA)-based acceleration for the MLPerf Tiny Inference Benchmark. The goal of the submissions was to demonstrate efficient and low-latency solutions on FPGAs using open-source workflows developed by the hls4ml and FINN teams; the solutions also catalyzed collaboration and the development of more accessible tools towards the democratization of powerful tiny ML.

Solutions were provided for the anomaly detection, keyword spotting, and image classification benchmarks. Model and design space exploration is presented including performance and hardware implementation optimizations. This enabled novel capabilities merged into the tool flows, including layer fusion, FIFO buffer optimization, and the development of the QONNX interchange format for representing flexibly quantized neural networks.

From this work, we demonstrated a common methodology for building FPGA-optimized ML model implementations targeting different benchmarks. First, a 32-bit floating-point precision model was trained to determine the baseline expected performance. Then we generalized the model by introducing additional hyperparameters related to layer sizes, number of layers, pooling choices, and more, and performed a hyperparameter optimization to determine the Pareto-optimal front balancing the performance and inference cost of a given ML model. Next, we quantized the model from 32-bit floating-point to integer/fixed-point precision reducing the bit width until the model performance began to degrade—the smallest bit width that retained the original baseline performance was then chosen. Finally, we synthesized the model for the FPGA platforms with further hardware-specific optimizations to fit it within the FPGA resources with minimal latency and maximal throughput. This approach can be further refined and formalized by integrating with all-in-one, end-to-end workflows like Sherlock (Gautier et al., 2022).

The system-level integration of the AI algorithms is also presented including interfaces to the control platform and setup for measuring performance, latency, and power in the standardized benchmark workflow. The optimized designs are comparable in performance to the reference models and have latencies as low as 20 μs and measure as low as 30 μJ per inference. The end-to-end workflows are publicly available online[2].

---

[2] https://github.com/mlcommons/tiny_results_v0.7/tree/main/open/hls4ml-finn

ACKNOWLEDGMENTS

REFERENCES

Aarrestad, T., Loncar, V., Pierini, M., Summers, S., Ngadiuba, J., Petersson, C., Linander, H., Iiyama, Y., Guglielmo, G. D., Duarte, J., Harris, P., Rankin, D., Jindariani, S., Pedro, K., Tran, N., Liu, M., Kreinar, E., Wu, Z., and Hoang, D. Fast convolutional neural networks on FPGAs with `hls4ml`. 2021, `arXiv:2101.05108`. Submitted to *Mach. Learn.: Sci. Technol.*

Abdelouahab, K., Pelcat, M., Serot, J., and Berry, F. Accelerating cnn inference on FPGAs: A survey. 2018, `arXiv:1806.01683`.

Bai, J., Lu, F., Zhang, K., et al. Onnx: Open neural network exchange. `https://github.com/onnx/onnx`, 2019.

Banbury, C., Reddi, V. J., Torelli, P., Holleman, J., Jeffries, N., Kiraly, C., Montino, P., Kanter, D., Ahmed, S., Pau, D., Thakker, U., Torrini, A., Warden, P., Cordaro, J., Guglielmo, G. D., Duarte, J., Gibellini, S., Parekh, V., Tran, H., Tran, N., Wenxu, N., and Xuesong, X. MLPerf Tiny benchmark. In Vanschoren, J. and Yeung, S. (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021, `arXiv:2106.07597`. URL `https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/file/da4fb5c6e93e74d3df8527599fa62642-Paper-round1.pdf`.

Banbury, C. R., Reddi, V. J., Lam, M., Fu, W., Fazel, A., Holleman, J., Huang, X., Hurtado, R., Kanter, D., Lokhmotov, A., Patterson, D., Pau, D., Seo, J.-s., Sieracki, J., Thakker, U., Verhelst, M., and Yadav, P. Benchmarking tinyml systems: Challenges and direction. 2020, `arXiv:2003.04821`.

Baskin, C., Schwartz, E., Zheltonozhskii, E., Liss, N., Giryes, R., Bronstein, A. M., and Mendelson, A. UNIQ: Uniform noise injection for the quantization of neural networks. 2018, `arXiv:1804.10969`.

Blott, M., Preußer, T., Fraser, N., Gambardella, G., O'Brien, K., and Umuroglu, Y. FINN-R: An end-to-end deep-learning framework for fast exploration

of quantized neural networks. *ACM Trans. Reconfigurable Technol. Syst.*, 11(3), 2018a. ISSN 1936-7406, `doi:10.1145/3242897`, `arXiv:1809.04570`.

Blott, M., Preußer, T. B., Fraser, N. J., Gambardella, G., O'brien, K., Umuroglu, Y., Leeser, M., and Vissers, K. Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11(3):1, 2018b.

Chang, S.-E., Li, Y., Sun, M., Shi, R., So, H. K. H., Qian, X., Wang, Y., and Lin, X. Mix and match: A novel FPGA-centric deep neural network quantization framework. In *27th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2021, `arXiv:2012.04240`.

Coelho, C. N., Kuusela, A., Li, S., Zhuang, H., Aarrestad, T., Loncar, V., Ngadiuba, J., Pierini, M., Pol, A. A., and Summers, S. Automatic deep heterogeneous quantization of deep neural networks for ultra low-area, low-latency inference on the edge at particle colliders. *Nat. Mach. Intell.*, 3:675, 2021, `arXiv:2006.10159`.

Cota, E. G., Mantovani, P., Di Guglielmo, G., and Carloni, L. P. An analysis of accelerator coupling in heterogeneous architectures. In *ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1. IEEE, 2015.

Deiana, A. M., Tran, N., et al. Applications and Techniques for Fast Machine Learning in Science. *Front. Big Data*, 5:787421, 2022, `doi:10.3389/fdata.2022.787421`, `arXiv:2110.13041`.

Determined AI. Determined ai, 2018. URL `https://www.determined.ai`.

DiCecco, R., Lacey, G., Vasiljevic, J., Chow, P., Taylor, G., and Areibi, S. Caffeinated fpgas: Fpga framework for convolutional neural networks. In *2016 International Conference on Field-Programmable Technology (FPT)*, pp. 265. IEEE, 2016, `doi:10.1109/FPT.2016.7929549`, `arXiv:1609.09671`.

Duarte, J., Han, S., et al. Fast inference of deep neural networks in FPGAs for particle physics. *JINST*, 13:P07027, 2018, `doi:10.1088/1748-0221/13/07/P07027`, `arXiv:1804.06913`.

Elabd, A. et al. Graph Neural Networks for Charged Particle Tracking on FPGAs. *Front. Big Data*, 5:828666, 2022, `doi:10.3389/fdata.2022.828666`, `arXiv:2112.02048`.

Fahim, F., Hawks, B., et al. hls4ml: An Open-Source Codesign Workflow to Empower Scientific Low-Power Machine Learning Devices. In *tinyML Research Symposium 2021*, 3 2021, arXiv:2103.05579.

Gautier, Q., Althoff, A., Crutchfield, C. L., and Kastner, R. Sherlock: A multi-objective design space exploration framework. *ACM Trans. Des. Autom. Electron. Syst.*, 27 (4), 2022, doi:10.1145/3511472.

Gokhale, V., Zaidy, A., Chang, A. X. M., and Culurciello, E. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1, 2017, doi:10.1109/ISCAS.2017.8050809, arXiv:1708.02579.

Google. Qkeras, 2020. URL https://github.com/google/qkeras.

Guan, Y., Liang, H., Xu, N., Wang, W., Shi, S., Chen, X., Sun, G., Zhang, W., and Cong, J. 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)*, pp. 152, 2017, doi:10.1109/FCCM.2017.25.

Guo, K., Zeng, S., Yu, J., Wang, Y., and Yang, H. A survey of FPGA-based neural network inference accelerators. *ACM Trans. Reconfigurable Technol. Syst.*, 12, 2019. ISSN 1936-7406, doi:10.1145/3289185, arXiv:1712.08934.

Hacene, G. B., Gripon, V., Arzel, M., Farrugia, N., and Bengio, Y. Quantized guided pruning for efficient hardware implementations of convolutional neural networks. In *2020 18th IEEE International New Circuits and Systems Conference (NEWCAS)*, pp. 206, 2020, doi:10.1109/NEWCAS49341.2020.9159769, arXiv:1812.11337.

Hawks, B., Duarte, J., Fraser, N. J., Pappalardo, A., Tran, N., and Umuroglu, Y. Ps and Qs: Quantization-aware pruning for efficient low latency neural network inference. *Front. AI*, 4:676564, 2021, doi:10.3389/frai.2021.676564, arXiv:2102.11289.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770, 2016, doi:10.1109/CVPR.2016.90, arXiv:1512.03385.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. Binarized neural networks. In

Lee, D., Sugiyama, M., Luxburg, U., Guyon, I., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 29. Curran Associates, Inc., 2016, arXiv:1602.02830. URL https://proceedings.neurips.cc/paper/2016/file/d8330f857a17c53d217014ee776bfd50-Paper.pdf.

Iiyama, Y. et al. Distance-weighted graph neural networks on FPGAs for real-time particle reconstruction in high energy physics. *Front. Big Data*, 3:598927, 2021, doi:10.3389/fdata.2020.598927, arXiv:2008.03601.

International Telecommunication Union. ITU-ML5G-PS-007: Lightning-Fast Modulation Classification with Hardware-Efficient Neural Networks. https://challenge.aiforgood.itu.int/match/matchitem/34, 2021. Accessed: 2022-03-31.

Kagermann, H., Wahlster, W., and Helbig, J. Recommendations for implementing the strategic initiative industrie 4.0 – securing the future of german manufacturing industry. Final report of the industrie 4.0 working group, acatech – National Academy of Science and Engineering, München, 2013. URL http://forschungsunion.de/pdf/industrie_4_0_final_report.pdf.

Koizumi, Y., Saito, S., Uematsu, H., Harada, N., and Imoto, K. Toyadmos: A dataset of miniature-machine operating sounds for anomalous sound detection. In *2019 IEEE Workshop on Applications of Signal Processing to Audio and Acoustics (WASPAA)*, pp. 313. IEEE, 2019.

Krizhevsky, A., Nair, V., and Hinton, G. Cifar-10 (canadian institute for advanced research), 2009. URL http://www.cs.toronto.edu/~kriz/cifar.html.

Li, L., Jamieson, K., Rostamizadeh, A., Gonina, E., Ben-tzur, J., Hardt, M., Recht, B., and Talwalkar, A. A system for massively parallel hyperparameter tuning. In Dhillon, I., Papailiopoulos, D., and Sze, V. (eds.), *Proceedings of Machine Learning and Systems*, volume 2, pp. 230, 2020, arXiv:1810.05934. URL https://proceedings.mlsys.org/paper/2020/file/f4b9ec30ad9f68f89b29639786cb62ef-Paper.pdf.

Majumder, K. and Bondhugula, U. A flexible FPGA accelerator for convolutional neural networks. 2019, arXiv:1912.07284.

Muhizi, J. Add support for qdense_batchnorm in qkeras, 2021. URL https://github.com/google/qkeras/pull/74.

Ngadiuba, J., Loncar, V., Pierini, M., Summers, S., Di Guglielmo, G., Duarte, J., Harris, P., Rankin, D., Jindariani, S., Liu, M., and et al. Compressing deep neural networks on FPGAs to binary and ternary precision with `hls4ml`. *Mach. Learn.: Sci. Technol.*, 2:015001, 2020, `doi:10.1088/2632-2153/aba042`, `arXiv:2003.06308`.

O'Malley, T., Bursztein, E., Long, J., Chollet, F., Jin, H., Invernizzi, L., et al. Kerastuner. `https://github.com/keras-team/keras-tuner`, 2019.

Pappalardo, A. Xilinx/brevitas, 2021, `doi:10.5281/zenodo.3333552`. URL `https://github.com/xilinx/brevitas`.

Pappalardo, A., Umuroglu, Y., et al. QONNX: Representing Arbitrary-Precision Quantized Neural Networks. In *4th Workshop on Accelerated Machine Learning (AccML) at HiPEAC 2022 Conference*, 2022, `arXiv:2206.07527`. URL `https://accml.dcs.gla.ac.uk/papers/2022/4thAccML_paper_1(12).pdf`.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32*, pp. 8024–8035. Curran Associates, Inc., 2019. URL `http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf`.

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

Sharma, H., Park, J., Mahajan, D., Amaro, E., Kim, J. K., Shao, C., Mishra, A., and Esmaeilzadeh, H. From high-level deep neural models to fpgas. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1. IEEE, 2016.

Shawahna, A., Sait, S. M., and El-Maleh, A. FPGA-based accelerators of deep learning networks for learning and classification: A review. *IEEE Access*, 7:7823, 2019. ISSN 2169-3536, `doi:10.1109/access.2018.2890150`, `arXiv:1901.00121`.

Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR 2015, Conference Track Proceedings*, 2015, `arXiv:1409.1556`.

Summers, S. et al. Fast inference of boosted decision trees in FPGAs for particle physics. *JINST*, 15:P05026, 2020, `doi:10.1088/1748-0221/15/05/P05026`, `arXiv:2002.02534`.

tinyML Foundation. About, 2019. URL `https://www.tinyml.org`.

Torralba, A., Fergus, R., and Freeman, W. T. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE transactions on pattern analysis and machine intelligence*, 30(11):1958, 2008.

Umuroglu, Y. and Jahre, M. Streamlined deployment for quantized neural networks. In *International Workshop on Highly Efficient Neural Networks Design (HENND)*, 2017, `arXiv:1709.04060`.

Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P., Jahre, M., and Vissers, K. 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, pp. 65. ACM, 2017.

Venieris, S. I. and Bouganis, C.-S. fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 40. IEEE, 2016, `doi:10.1109/FCCM.2016.22`.

Venieris, S. I. and Bouganis, C.-S. fpgaConvNet: Automated mapping of convolutional neural networks on FPGAs. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 291. ACM, 2017a. ISBN 978-1-4503-4354-1, `doi:10.1145/3020078.3021791`.

Venieris, S. I. and Bouganis, C.-S. 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*, 2017b, `arXiv:1711.08740`.

Venieris, S. I. and Bouganis, C. S. Latency-driven design for FPGA-based convolutional neural networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1, 2017c, `doi:10.23919/FPL.2017.8056828`.

Venieris, S. I., Kouris, A., and Bouganis, C.-S. Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions. *ACM Comput. Surv.*, 51, 2018. ISSN 0360-0300, doi:10.1145/3186332, arXiv:1803.05900.

Warden, P. Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *ArXiv e-prints*, April 2018, arXiv:1804.03209. URL https://arxiv.org/abs/1804.03209.

Whatmough, P. N., Zhou, C., Hansen, P., Venkataramana-iah, S. K., sun Seo, J., and Mattina, M. FixyNN: Efficient hardware for mobile computer vision via transfer learning. In *2nd SysML Conference*, 2019a, arXiv:1902.11128. URL https://mlsys.org/Conferences/2019/doc/2019/69.pdf.

Whatmough, P. N. et al. Arm-software/deepfreeze. https://github.com/ARM-software/DeepFreeze, 2019b.

Wright, A., Pit-Claude, C., Miller, D., and @threonorm. Pyverilator, 2020. URL https://github.com/csail-csg/pyverilator.

Xilinx. QONNX and FINN. https://xilinx.github.io/finn//2021/11/03/qonnx-and-finn.html, 2021.

Xilinx. Xilinx/vitis-ai. https://github.com/Xilinx/Vitis-AI, 2021.

Yoshioka, K. kentaroy47/keras-opcounter, 2020. URL https://github.com/kentaroy47/keras-Opcounter.

Zhang, Y., Suda, N., Lai, L., and Chandra, V. Hello Edge: Keyword Spotting on Microcontroller. 2017, arXiv:1711.07128.