Quantization Effects on FINN Convolutional Neural Networks

Nojan Sheybani
Electrical and Computer Engineering
UC San Diego
La Jolla, CA
nsheyban@ucsd.edu

Vansh Singh
Computer Science and Engineering
UC San Diego
La Jolla, CA
vcsingh@ucsd.edu

***Abstract*—We use the FINN and Brevitas frameworks to create, train, and deploy a custom Convolutional Neural Network (CNN) that classifies clothes onto the Pynq-Z2 FPGA by Xilinx. We observe the effects of modulating quantization widths on training convergence, model accuracy, and inference throughput. We deployed a LeNet-inspired quantized CNN that achieves 77% accuracy on the Fashion-MNIST dataset onto Pynq-Z2. During our exploration into Quantized CNNs we find and document the problems we encountered with the FINN and Brevitas frameworks.**

I. Introduction

Since image classification problems have been made solvable with advances in compute and machine learning, they have surfaced in our society at an exponential rate these last few decades. Today, with use-cases from face-unlock on the iPhone to autonomous driving, we see more and more of these classification problems moving from resource-abundant datacenters to resource-bound edge computing devices. Furthermore, we see that the modern image classifications workloads are transitioning from batch-oriented jobs to real-time online classification. Thus edge-computing devices are faced with fewer hardware resources and higher throughput expectations. For this reason, we see specialized hardware on the edge such as ASICs and FPGAs. In this paper we study realtime image classification on an FPGA.

Traditionally, convolutional neural networks (CNNs) have been the most successful image classification algorithms, however their large floating-point convolution operations are very hardware and time expensive. Neural network quantization has been a proven workaround that maintains high performance but is less hardware intensive. In this technique, we substitute the floating-point arithmetic of the CNN with fixed-width operations, thereby decreasing the hardware load and increasing throughput. Quantization has been a successful strategy - Hubara et. al. show that even with Binarized Neural Networks, they achieved an impressive performance of 51% top-1 accuracy on ImageNet [1]. We use the new Xilinx Brevitas and FINN frameworks to create, train, and explore quantization effects on a CNN classification task.

We evaluate our model performance on the Fashion-MNIST dataset [2]. This is a more challenging dataset than MNIST - which is largely considered a solved problem in Machine Learning - and is more indicative of realistic classification workloads [2].

II. Preliminaries

A. PYNQ

PYNQ is an open source development environment developed by Xilinx [3]. The PYNQ environment can be installed on Xilinx hardware, such as the Pynq-Z2 FPGA. This provides a friendly user interface in the form of Jupyter Notebooks that allows developers to deploy and test hardware designs on the Pynq boards. We used the Pynq-Z2 board for all deployment and testing.

DSP slices	220
Memory	512 MB DDR3
Slices	13,300 Logic Slices
LUTs	53,200 6-input LUTs
FFs	106,400
BRAM	630 KB

TABLE I
PYNQ Z2 SPECS

B. Brevitas

Brevitas is a beta-released open-source PyTorch library built by Xilinx that allows developers to create quantization-aware neural networks [4]. It consists of building blocks of quantization-aware modules - such as ReLUs, Convolutions, MaxPools, Batch-Norm etc - that allow for fixed-width training of model parameters. With its simple API, developers can specify the bit widths for Brevitas component inputs, weights, and activations. It also supports exporting models to the ONNX format, which is what Xilinx's FINN framework uses to create HLS for QNNs.

C. FINN

FINN is a project by Xilinx Research in [5], that provides an HLS framework for building quantized neural networks (QNNs). Originally, purely written in C, the FINN project now has a beta-released open-source python framework that enables QNN development and deployment onto PYNQ FPGA boards [6]. For brevity we refer to this python framework as FINN throughout the rest of the paper.

FINN exposes a set of ONNX transformations that support HLS preparation and Vivado HLS generation for quantized models. It also provides tools to deploys generated bitstreams onto a remote Pynq FPGA with an easy-to-use Jupyter development environment. FINN boasts a high performance and efficient implementation of neural networks on FPGA, and provides a streamlined workflow with a high-level API. Although FINN was originally built for binarized neural networks, as discussed in [5], recent developments have offered support for quantized neural networks, which makes FINN much more robust. By building on top of the Pynq ecosystem, Pytorch, the ONNX framework, and Jupyter-Notebook dev tools, the FINN/Brevitas reduce a lot of barriers to Neural Network HLS for the developer.

III. Design

We followed the standard FINN workflow in our development and Pynq deployment process [3], as shown in Figure 1.

Fig. 1. Development Workflow for FINN

A. Developing our Custom CNN

We aimed to create and deploy a Convolutional Neural Network trained on Fashion-MNIST to the Pynq-Z2 board using the Brevitas and FINN frameworks. Both are still quite nascent and are considered to be in "Beta". For that reason, during our model design, we learned a lot about the limitations of the frameworks by confronting bugs head-on.

We designed a LeNet5-inspired model for Fashion-MNIST images using the higher-order abstractions from the Brevitas library [7]. Whereas the original LeNet5 was built over the (1 x 32 x 32) images, Fashion-MNIST has size (1 x 28 x 28). Thus we created QeNet, shown in Table II.

Each convolution layer and forward connected layer is followed by a fixed-width quantized batch normalization layer. Given that we have purely fixed-point operations, we have followed the Brevitas CNV model design and used the Identity activation function after each layer instead of ReLU or Sigmoid.

B. Training

We trained our model on epochs of our 50,000 image training set. We stopped training when we began overfitting to our 10,000 image validation set. For each quantized model, we select the model with best performance on the Validations set for deployment. We save the selected model's weights and export the model to ONNX for FINN transformations.

Layer	Feature Map	Size	Kernel	Stride
Image Input	1	28x28		
Convolution	16	26x26	3x3	1
Convolution	16	24x24	3x3	1
MaxPool	16	12x12	2x2	2
Convolution	32	10x10	3x3	1
MaxPool	32	5x5	2x2	2
Convolution	64	3x3	3x3	1
Convolution	64	1x1	3x3	1
FC		120		
FC		84		
FC		10		

TABLE II
CUSTOM QENET LAYERS

C. Performing FINN Transformations

To create a model bitstream for deployment, our ONNX model must go through FINN transformations that prepare the model for a Vivado HLS build. A FINN transformation is an iterative algorithm that traverses through an ONNX model and updates, inserts, collapses, or removes the individual ONNX nodes according to some defined logic. Table III provides an overview of some of the significant transformations we conducted.

The first class of transformations involve simple commonsense checks that include verifying the input size of every node matches with the output of its upstream node, and the removal of nodes whose output is a constant. They also tidy up our imported model to make it easier to view in the FINN-supported model-visualization tool: Netron. Such transformations include $GiveUniqueNodeNames()$ and $GiveReadableTensorNames()$, and are very handy when trying to debug further down the line.

After the initial pre-processing transformations, we transform floating-point arithmetic into integer operations using thresholds in a process called Streamlining [8]. This process attaches metadata to Brevitas batch normalization units, linear layers, max-pools, and convolutions that ensure they become fully quantized into integer operations during HLS generation. Transformations here include $LowerConvsToMatMul()$, which converts convolutions to matrix multiplication operations and $ConvertBipolarMatMulToXnorPopcount()$, which reduces Binary-width matrix multiplication to a more efficient XnorPopcount operation in hardware [5].

Next, we begin the Dataflow transformations. Internally, they first map our ONNX nodes to HLS components from the $finn - hlslib$ library. Then, they create dataflow partitions across the HLS components to increase inference throughput. An example of an HLS transformation is $InferStreamingMaxPool()$, which provides a streaming interface for MaxPool operations. Once all HLS transformations are done, the $CreateDataflowPartition()$ identifies the dataflow partitions in the onnx-model mapped to finn-hls components. Figure 2 shows the Netron shape of

Transformation	Description	Pre-Processing Stage	Streamlining Stage	Dataflow Stage
InferShapes	Ensure every tensor has a shape	x		
FoldConstants	Replace output of node with const-only inputs with a precomputed result	x		
GiveUniqueNodeNames		x		
GiveReadableTensorNames		x		
RemoveStaticGraphInputs	Remove any top-level graph inputs that have initializers	x		
MoveScalarLinearPastInvariants	Move scalar linear ops past functions invariant to them. eg: f(x*C) –>f(x) * C		x	
LowerConvsToMatMul	Replace Convolution with MatMul and Transpose		x	
ConvertBipolarMatMulToXnorPopcount	As described in FINN paper for BNNs		x	
RemoveUnusedTensors			x	
Streamline			x	
InferBinaryStreamingFCLayer	Convert any XnorPopcount to StreamingFCLayer_Batch finn-hlslib function			x
InferQuantizedStreamingFCLayer	Convert MatMul layers with quantized weights to StreamingFCLayer_Batch finn-hlslib function			x
InferThresholdingLayer	Convert MultiThreshold into a standalone thresholding HLS layer			x
InferStreamingMaxPool				x
RemoveCNVToFCFlatten	Removes an onnx node that implements a simple reshape if between two dataflow nodes			x
CreateDataFlowPartition				x

TABLE III
A SUBSET OF THE TRANSFORMATIONS WE PERFORMED BEFORE MODEL FOLDING AND HLS GENERATION

QeNet after the DataFlow transformations collapsed into the $StreamingDataflowPartition$ node.

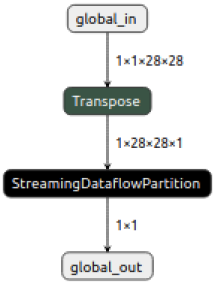

Fig. 2. Transformed Dataflow Partitioned Model

The final transformation before bitstream generation is folding. In folding, we enable hardware parallelism for each child node of the StreamingDataflowOperation component above with SIMD, Processing Elements (PEs), and FIFO depth specification, as done in the original FINN paper [5]. At the time this paper was written, FINN's auto-folding transformation was too naive to work for QeNet's complex model. Thus, we manually tuned the folding factors. These attributes are unique across models, and failure to set these correctly can cause unexpected behavior. Once this transformation is complete, we run the ZynqBuild transformation to generate a bitstream for the model.

IV. IMPLEMENTATION

All files for below can be found on our Github[1].

A. Choosing our Bitwidths

In quantization, we define the bit-widths of fixed-point Neural Network weight widths, activation output widths and input-data widths. We fix the input width in our quantization experiment to 8-bit fixed-point for all models. All models are trained assuming that the input images have been scaled to -1 to 1. We were able to successfully train Quantized CNNs across a variety of activation and weight widths. With our preliminary tests, we were achieving 83% test accuracy with 8-bit width models. Table IV shows our results on these quantized models.

Results	w4a4	w6a6	w8a8
Epochs until convergence	13	17	11
Test Set Performance	82%	83%	81%

TABLE IV
CONVERGENCE AND TEST-SET RESULTS OF LARGE-WIDTH QENETS

However, for reasons further expounded in section VI, HLS generation would fail for for our model with activation or weight widths greater than two. Thus we limit this study to models with the following quantization weight and activation widths: (1,1), (1,2), (2,2). FINN does not allow a larger weight width than activation width thus we could not test (2,1).

[1]https://github.com/nickshey/CSE237C_Final_nsheyban_vcsingh

Results	w2a2	w1a2	w1a1
Epochs until convergence	16	27	12
Test Set Performacne	76%	77%	66%
Pynq-Z2 Throughput (img/second)	10288	10288	10288

TABLE V
RESULTS OF MODELS DEPLOYED TO PYNQ-Z2

B. Training and Testing our Custom CNN

We split Fashion-MNIST into a training set of 50000 images, validations set of 10000 images, and test set of 10000 images. We trained with an Adam optimization algorithm in PyTorch with a learning rate of 1e-4 and momentum of 0 for all three models. We trained on mini-batches of size 50 on epochs of 50000 training images. Models were trained over epochs until validation performance no longer decreased: early stopping with a patience of 1. At this point, we considered the model to have converged. Epochs til convergence are shown in Results.

Our testing and training notebooks can be found in our Github.

C. Deploying Custom CNN on Pynq-Z2

As mentioned before, once the folding transformation is complete, a bitstream can be generated for the model. Upon specifying the platform that is being used (e.g. Pynq-Z1, Pynq-Z2, etc.), the $ZynqBuild()$ transform can be performed. This transform interfaces with Vivado and generates the bitstream for the model. The process takes about one and a half hours per model, so getting data for all of our models proved to be quite time-consuming. Once this transform is complete, we have a bitstream that can be deployed on a Pynq board. Once the network details of the Pynq board are provided, such as the IP address and port, the $DeployToPynq()$ transform can be done and deploy the drivers, .bit, and .hwh files that are necessary for remote execution of an inference on the Pynq board. Finally, we can perform Fashion-MNIST inferences on our Pynq board!

We have built Jupyter notebooks that go through the transformation and deployment process for each of our quantized models. These notebooks can be found in our Github.

V. RESULTS

Table V summarizes the results of our QeNet tests on Fashion-MNIST

A. Convergence and Accuracy

We find no clear trend in our training convergence data. We observe that the fully Binarized Neural Network, w1a1, converged the fastest. Also it had the lowest test-set accuracy. This may be because w1a1 has the smallest size of learnable parameters. The w2a2 model converged in 33% more epochs than the w1a1 model and scored 10% higher in test-set accuracy. Again, this seems reasonable because the w2a2 model has twice the size of learnable parameters. However, results also show that w1a2 took significantly longer to converge.

Given that w1a2 also had the best test-set performance, this may suggest that our patience of 1 in our training was too strict for the other models, and that the others may have also had increased test-set performance if trained longer on the training set. However, we believe this is not likely the case, given that we trained with a mini-batch of 50, over 50,000 images between testing the validation set. From preliminary model testing, we are confident that if validation accuracy on a 10-class labeling dataset does not decrease over 1000 samples of our mini-batch then that is sufficient to conclude the model has converged. Another possible reason is that same optimizer used for w1a1 and w1a2 simply learned much slower on the w1a2 model. We conjecture this may be because the non-linear functions the w1a2 model learns, with different weight and activation widths is more difficult than one where weights and activations have the same width.

B. Inferences on Fashion-MNIST

Here we show sample inferences of our w2a2 model from the Pynq-Z2 board. To make an inference, we generate and run a driver file on the Pynq board that will allow the model to process inputs and generate outputs. We then call a library method to remotely execute the inference on our Pynq IP. Figure 3 shows our QNN correctly inferring a sandal and a bag from the Pynq board.

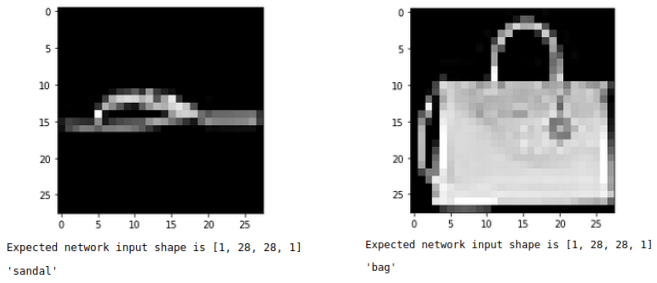

Fig. 3. Correct Inferences of a Sandal and Bag

Likewise, Figure 4 shows an incorrect inference from our Pynq-deployed model.

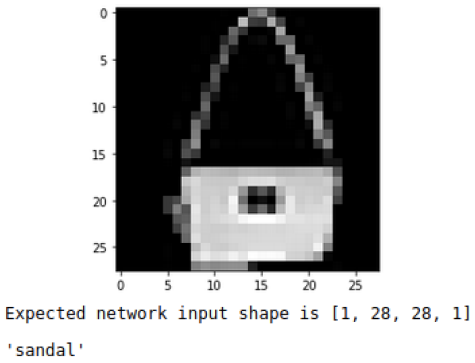

Fig. 4. Incorrect Inference of a Bag as a Sandal

C. Throughput of Quantized Models

The throughput test is found in the aforementioned driver file that is deployed to the Pynq board. This test essentially takes in a specified number of images and runs inferences on all of them through the model deployed on Pynq. Using Python's built in timing package, the test tracks the amount of time per inference and reports the throughput as a unit of images per second. We observe throughput that is quite high, at 10,288 images per second, but we do not find a difference in measured throughput between the different models. We hypothesize that we do not see a difference because, when implemented in hardware, our model sizes are not different enough to constitute a sacrifice in throughput. Had we had more time, we would have liked to find tools that allow us to see the actual hardware usage of our generated bitstreams.

VI. Problems

Both the Brevitas and FINN projects are currently in beta. Thus, we expected to encounter some bugs during our project. A more full list of the issues we faced are linked here[2]. We were fortunate to have direct contact with Yaman Umuroglu and Nick Fraser, two of the lead developers of FINN, towards the tail end of our project. We were able to bring up small bugs to them and talk about how to get around them. With their guidance, we pulled in updates from the unreleased FINN v0.5, coming out this week, which simplified the process of adding a custom neural network. Below, we cover some of the most time-consuming errors we faced.

A. Brevitas

First, even though Brevitas has been made for FINN, we learned from the FINN Gitter that the example on the Getting Started section of the Brevitas Github repository[3] has not been tested and is not officially supported. The Brevitas developers provide a LeNet implementation in Brevitas and PyTorch that users can begin working with. The example had a few minor errors, such as not including all of the necessary packages to successfully compile. After fixing those, we could train this Brevitas model, however, these models would fail during HLS generation. As further described in the FINN section below, we ditched the example LeNet model design and decided to tune an existing brevitas model design to fit our needs.

B. FINN

Upon building our first model, our Jupyter kernel would shut down anytime we tried running a transformation. Sometimes we would re-run some cells and things would work, but we never figured out how to consistently get things to work. This error resulted in tens of hours of debugging to no avail. Completely stuck, we dove into the Github FINN Issues list, previous PRs, codebase, and Gitter community. In a last-ditch scroll through the FINN Gitter archived messages and we found a 10-month old message from the developer that Jupyter kernel panics can be caused by importing PyTorch packages before importing ONNX packages. We changed the order of our import statements and resolved the Jupyter kernel shutdowns. This was by far one of the weirdest errors we have ever encountered and we have no clue how we would have solved it if it weren't for us scrolling through the past ten months of messages. Nevertheless, this fixed a bulk of our initial problems.

Then, we had an issue where the Dataflow partitioning transformations would split our ONNX model into discontinuous pieces as shown in Figure 5. While we were still able to make inferences with the split ONNX model, this was causing errors in HLS generation later down the road. We reached out to the lead developers, and Yaman informed us that this is likely a FINN bug and may be caused by failures in the streamlining transformations. Yaman recommended that we follow existing Brevitas model shapes as closely as possible in our FINN compilation steps, because FINN may not yet generalize to all Brevitas models. Following his advice, instead of creating our models using higher-level Brevitas libraries as shown in the Brevitas repository landing page, we copied the source code of a pre-trained CNV[4] model from the Brevitas repository homepage, and iteratively tuned that model into what we desired for classifying Fashion-MNIST.

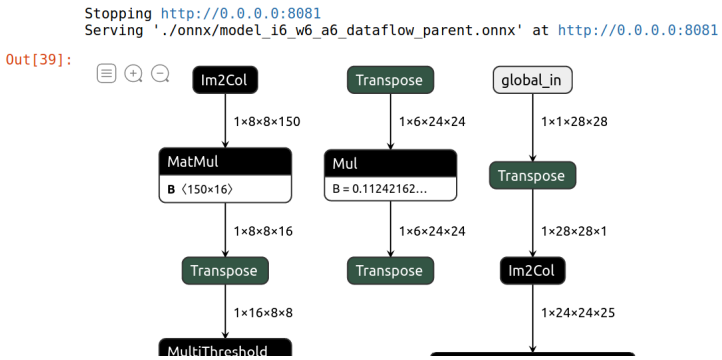

Fig. 5. Dataflow transformations split model

During this iterative process, we learned more about why FINN streamlining, dataflow, and zynqbuild transforms were failing. First, we found that using high bit-widths for quantization would result in a split model as shown above. Specifically if the activation or weight widths of our model were greater than two, the dataflow partitioning transformations would create discontinuous shapes such as in Figure 5. This limited the design-space of our quantization exploration.

Also from the code, we found that some FINN transformations currently expect the last layer before a fully-connected layer to be a Convolution layer, whereas before we had a MaxPool layer before our fully-connected layers. Unfortunately, the example model on the homepage of the Brevitas repository has a max-pool operation as the last layer before a fully connected layer. Their model design caused us

[2]https://bit.ly/3mv2O6H
[3]https://github.com/Xilinx/brevitas
[4]https://bit.ly/3p0g5Wm

to go down this rabbit hole. We wish that this was condition was either asserted or documented earlier. Only when the last layer was changed to be a convolution, did the code raise an assert statement that provided us more detail on what FINN streamlining transformations expect.

Lastly, when we were deploying the model onto the Pynq board and trying to remotely execute an inference, we ran into a strange error where no output file was being generated. After looking through Gitter and manually looking through the files on our board, we were able to find that the driver was not functioning properly. As the driver is a Python file, we were able to manually run it and see what was causing an error. For some reason, our platform type was not correctly set when we ran the $DeployToPynq()$ transformation. This resulted in us having to change the driver file manually to hardcode our platform type as $zynq - iodma$. Upon changed this, we were able to successfully run the driver and generate the output file. This allowed us to remotely execute inferences on the Pynq board and perform a successful throughput test.

VII. Conclusion

In this work we present a successful implementation and exploration of a custom quantized Convolutional Neural Network on the Xilinx Pynq-Z2 FPGA. Through this project, we have been able to explore the capabilities of FINN and Brevitas and suggest improvements that we feel would benefit the respective projects. We conclude that the model with a weight width of 1 and activation width of 2 had the best accuracy, but also took twice as long to converge as the model with weight width and activation width of 2. We also observed that the throughput did not differ between the different quantizations of the model. This allows us to prioritize accuracy without sacrificing throughput when implement our neural network on Pynq. This was a very informative experience and it was interesting to work with open-source projects and get creative with debugging all of the problems that we ran into.

References

[1] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: training neural networks with low precision weights and activations. J. Mach. Learn. Res. 18, 1 (January 2017), 6869–6898.

[2] https://github.com/zalandoresearch/fashion-mnist

[3] "PYNQ - Python productivity for Zynq," PYNQ - Python productivity for Zynq. http://www.pynq.io/.

[4] https://github.com/Xilinx/brevitas

[5] Y. Umuroglu et al., "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17, pp. 65–74, 2017, doi: 10.1145/3020078.3021744.

[6] "FINN," finn. https://xilinx.github.io/finn/.

[7] LeCun Y. LeNet-5, convolutional neural networks. URL: http://yann. lecun. com/exdb/lenet. 2015;20(5):14.

[8] Umuroglu, Yaman & Jahre, Magnus. (2017). Streamlined Deployment for Quantized Neural Networks.