BNN Design and Implementation using FINN

Cheng-Yu Chen, Yunqi Yu
{chc032, yuy031}@ucsd.edu

Abstract

Convolutional neural networks have been proven to be successful in visual recognition in a wide range of applications to enable machine vision on all kinds of devices. Traditionally these neural networks use floating point parameters and operations to classify images. For modern CNNs the number of floating pointer parameters is typically within the range of millions and the number of floating point operations is around billions. [1] The number also increases as the neural network scales in size and depth.

Using low precision arithmetic operations in neural networks significantly reduces hardware utilization in memory, allowing for potentially faster and more energy-efficient neural network inference on image classification tasks. This work explores the performance and parameter tradeoff in binarized neural network architecture proposed in [2] using a framework for building scalable BNN called FINN provided in [3].

1. Introduction

For neural networks on application specific hardware to achieve acceptable accuracy rate, it is proven to be unnecessary to use floating point parameters or operations. Instead, one or two bit quantization for weights and activations as shown in Figure 1 is enough for neural networks to perform visual classification tasks given a set of performance requirements which might include frames per second for real time image classification and certain accuracy rate. [2] It is proven to be fast and energy efficient especially for FPGAs to use binary operations, achieving a higher performance whilst maintaining a small memory consumption by keeping the parameters on-chip for large networks.

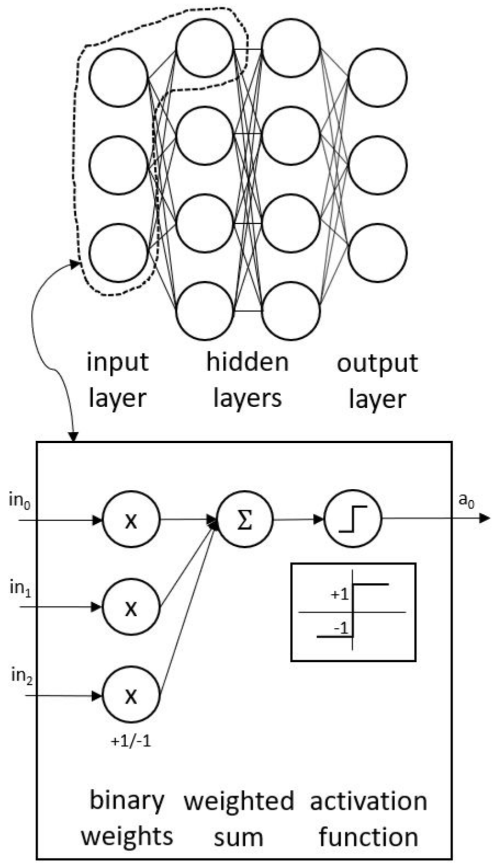

Figure 1. Binarized Neural Network with weighted summations on binarized weights and activation function as a sign function

In this work, we choose FINN, a framework for building scalable and fast BNN inference to implement a binarized neural network on an FPGA PYNQ-Z2 board for its fast classification with low latency. This is achieved by the framework's novel approach on mapping the BNN onto the FPGA and a customizable architecture allowing throughput and resource tradeoffs as per the application's demand.

2. Background

FINN converts a trained BNN into FPGA using a user specific throughput target and a trained BNN to synthesize and optimize the BNN specific operator such as using popcount for accumulation, batchnorm-activation as threshold and boolean XNOR for max pooling. It then produces a synthesizable C++ representation of the BNN streaming architecture using FINN hardware library including a matrix-vector-threshold unit, a sliding window unit, a pooling unit and other helper library modules.

The matrix-vector-threshold unit is the essence of the binarized neural network as it computes matrix-vector product which makes up the majority of operations in the design. The unit acts as a fully connected layer on its own and can also be used as part of the convolution layers. The overview of the unit is shown in Figure 2.

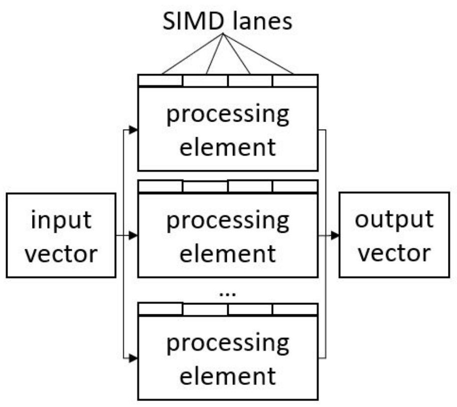

Figure 2. Matrix-vector-threshold unit architecture

The process element in the matrix-vector-threshold unit takes in an input vector and computes the dot product between the input vector and a row of the synaptic weight matrix which is kept in the on-chip memory distributed between the process elements. The dot product between the two bit vectors can be implemented with an XNOR gate which is hardware efficient and fast to compute. The dot product produces a single-bit output. The number of set bits in the output is then counted using popcount and added to the accumulator before being thresholded to produce the final output vector. The process element flow is shown in Figure 3.

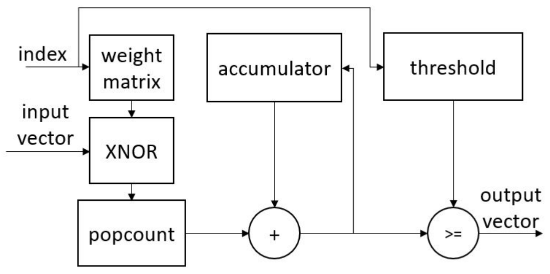

Figure 3. Process element data flow

3. Hardware Implementation

We used FINN's BNN-PYNQ to rebuild the hardware bitstream, trained a network using Fashion-MNIST dataset, loaded the weights onto the BNN-PYNQ overlay and successfully ran the demo on the PYNQ-Z2 board. The number of process elements and number of SIMD lanes are the two parameters explored while rebuilding the hardware of a convolutional network topology. We varied PE and SIMD in the first layer of the architecture in the config.h file and synthesized the topology in Vivado 2018.2 in Ubuntu 18.04.5 to generate resource utilization. During the export of the parameter and the generation of the config.h file, we are able to know the number of operation of each layer which is 2*Matrix Height*Matrix Weight. We also know how much parallelism is used in the hardware accelerator. It is just the number of operation the layer can compute in each clock cycle which is 2*SIMD*PE. The latency of the layer is the rate of the number of ops of the layer divided by the operations we can do in a cycle. The results are shown in Table 1.

Convolutional Layer	Latency	FF	LUT
SIMD = 32 PE = 32	28224	17857	23114
SIMD = 64 PE = 32	14112	18357	37587
SIMD = 32 PE = 64	14112	32765	43887
SIMD = 64 PE = 64	7056	35438	77712

Table 1. Performance in Latency and Resource Utilization

Even though we can achieve lower latency with more PE and SIMD as seen from the synthesized report in Table 1, we need to carefully design layers in the network taking into account other conditions. The latency is used as a feedback to the choice of SIMD and PE values for each layer. We eventually would like to have all layers having similar latency since the overall architecture is using dataflow between layers, meaning that the throughput is limited by the slowest layer.

We trained a four-layer fully connected network topology LFC for classifying the Fashion-MNIST dataset using two implementations which are 1-bit weight matrix 1-bit activation function and 1-bit weight matrix 2-bit activation function. The LFC architecture is shown in Figure 4.

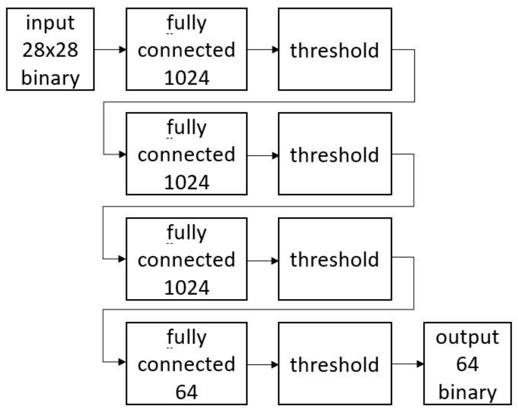

Figure 4. LFC Network Architecture

The network accepts training data as Nx1x28x28 numpy array, meaning that the training data should be comprised of 1 bit color*28 pixels*28 pixels scaling from -1 to 1 using grayscale binary images. The output is encoded using the one-hot encoder that outputs a 10-bit one-hot vector indicating the digit. The network is then trained on Nvidia Quadro RTX 4000, taking around 6 hours with a reduced epoch of 150. We then converted the trained weights from real floating point values into binary values, packed them into .bin files and loaded the weights onto the PYNQ-Z2 board for the demo.

4. Results and Discussion

The two implementations of LFC give less than ideal accuracy results shown in Table 2. The implementation with 2 bit activation function has slightly less accuracy than the implementation with 1 bit activation. This is unexpected and it might be due to the reduced epoch imposed by the limited resource for training.

LFC	Accuracy	Latency
W1A1	84.09	8.41usec
W1A2	83.65	8.41usec

Table 2. LFC Network Accuracy and Latency on Fashion-MNIST

Since we are using the optimized hardware framework, the inference took 8.41 microsecond as compared to a pure software implementation using PYNQ's arm core. Using a pure software implementation, the inference time is around 4 orders of magnitude higher than the hardware inference time as shown in Figure 5.

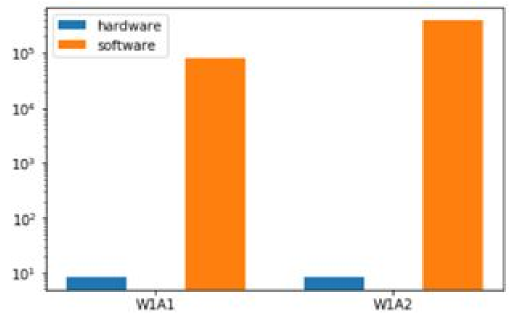

Figure 5. Image Inference Time in Hardware and Software Implementation

5. REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In Proc. NIPS, pages 1097{1105, 2012.
[2] M. Courbariaux and Y. Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. CoRR, abs/1602.02830, 2016.
[3] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. ACM/SIGDA ISFPGA, 2017.
[4] https://github.com/Xilinx/finn-hlslib
[5] https://github.com/Xilinx/BNN-PYNQ