FINN: A Framework for Fast, Scalable Binarized Neural Network Inference

Yaman Umuroglu[*†], Nicholas J. Fraser[*‡], Giulio Gambardella[*], Michaela Blott[*], Philip Leong[‡], Magnus Jahre[†] and Kees Vissers[*]
[*]Xilinx Research Labs; [†]Norwegian University of Science and Technology; [‡]University of Sydney
yamanu@idi.ntnu.no

ABSTRACT

Research has shown that convolutional neural networks contain significant redundancy, and high classification accuracy can be obtained even when weights and activations are reduced from floating point to binary values. In this paper, we present Finn, a framework for building fast and flexible FPGA accelerators using a flexible heterogeneous streaming architecture. By utilizing a novel set of optimizations that enable efficient mapping of binarized neural networks to hardware, we implement fully connected, convolutional and pooling layers, with per-layer compute resources being tailored to user-provided throughput requirements. On a ZC706 embedded FPGA platform drawing less than 25 W total system power, we demonstrate up to 12.3 million image classifications per second with 0.31 µs latency on the MNIST dataset with 95.8% accuracy, and 21906 image classifications per second with 283 µs latency on the CIFAR-10 and SVHN datasets with respectively 80.1% and 94.9% accuracy. To the best of our knowledge, ours are the fastest classification rates reported to date on these benchmarks.

1. INTRODUCTION

Convolutional Neural Networks (CNNs) have dramatically improved in recent years, their performance now exceeding that of other visual recognition algorithms [14], and even surpassing human accuracy on certain problems [24, 29]. They are likely to play an important role in enabling ubiquitous machine vision and intelligence on all kinds of devices, but a significant computational challenge remains. Modern CNNs may contain millions of floating-point parameters and require billions of floating-point operations to recognize a single image. Furthermore, these requirements tend to increase as researchers explore deeper networks. For instance, AlexNet [14] (the winning entry for ImageNet Large Scale Visual Recognition Competition (ILSVRC) [23] in 2012) required 244 MB of parameters and 1.4 billion floating point operations (GFLOP) per image, while VGG-16 [25] from ILSVRC 2014 required 552 MB of parameters and 30.8 GFLOP per image.

FPGA '17, February 22 - 24, 2017, Monterey, CA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. ISBN 978-1-4503-4354-1/17/02...$15.00
DOI: http://dx.doi.org/10.1145/3020078.3021744

While the vast majority of CNNs implementations use floating point parameters, a growing body of research demonstrates this approach incorporates significant redundancy. Recently, it has been shown [5, 27, 22, 12, 32] that neural networks can classify accurately using one- or two-bit quantization for weights and activations. Such a combination of low-precision arithmetic and small memory footprint presents a unique opportunity for fast and energy-efficient image classification using Field Programmable Gate Arrays (FPGAs). FPGAs have *much* higher theoretical peak performance for binary operations compared to floating point, while the small memory footprint *removes* the off-chip memory bottleneck by keeping parameters on-chip, even for large networks. Binarized Neural Networks (BNNs), proposed by Courbariaux et al. [5], are particularly appealing since they can be implemented almost entirely with binary operations, with the potential to attain performance in the teraoperations per second (TOPS) range on FPGAs.

In this work, we propose Finn, a framework for building scalable and fast BNN inference accelerators on FPGAs. Finn-generated accelerators can perform millions of classifications per second with sub-microsecond latency, thereby making them ideal for supporting real-time embedded applications such as augmented reality, autonomous driving and robotics. Compute resources can be scaled to meet a given classification rate requirement. We demonstrate Finn's capabilities with a series of prototypes for classifying the MNIST, SVHN and CIFAR-10 benchmark datasets. Our classification rate results surpass the best previously published results by over 48× for MNIST, 2.2× for CIFAR-10 and 8× for SVHN. To the best of our knowledge, this is the fastest reported neural network inference implementation on these datasets. The novel contributions are:

- Quantification of peak performance for BNNs on FPGAs using a roofline model.
- A set of novel optimizations for mapping BNNs onto FPGA more efficiently.
- A BNN architecture and accelerator construction tool, permitting customization of throughput.
- A range of prototypes that demonstrate the potential of BNNs on off-the-shelf FPGA platforms.

The rest of this paper is organized as follows: Section 2 provides background on CNNs, BNNs, and their hardware implementations. Section 3 discusses BNNs accuracy and peak performance on FPGAs. Section 4 describes Finn's architecture and optimizations. Section 5 presents the experimental evaluation, and Section 6 concludes the paper.

2. BACKGROUND

This work is focused on *supervised* learning, in which the goal is to find a function, $g(\mathbf{x}_i)$, which approximates a mapping $\mathbf{x}_i \rightarrow y_i \; \forall \; i$, where $\{\mathbf{x}_i, y_i\}$ is an input/output pair known as a training example. Furthermore, only the *inference* problem is studied, the parameters, w, being assumed to have been learned offline.

2.1 Convolutional Neural Networks

A *multilayer perceptron* is a type of Artificial Neural Network (ANN) which has its neurons arranged in multiple layers, with neurons taking the output of all neurons of the previous layer as inputs. Mathematically, the output, $a_{l,n}$, for the n^{th} neuron in the l^{th} layer of a fully connected network is calculated as follows:

$$a_{l,n} = f_{act}(\sum_{s=0}^{S_l} w_{l,n,s} a_{l-1,s} + b_{l,n}) \quad , \tag{1}$$

where $w_{l,n,s}$ is weight of the s^{th} synapse connected to the input of the n^{th} neuron in the l^{th} layer, $b_{l,n}$ is a bias term, f_{act} is the activation function, and S_l is the number of synapses connected to each neuron in the l^{th} layer. Popular activation functions include: the hyperbolic tangent function, $f_{act}(a) = tanh(a)$; and the rectified linear unit (ReLU), $f_{act}(a) = max(0, a)$.

CNNs [15] are a variant of multilayer perceptrons, in which a layer only receives inputs from a small *receptive field* of the previous layer. This approach greatly reduces the number of parameters involved and allows local features (e.g., edges, corners) to be found [15]. A basic 2D convolutional layer in a neural network is similar to a fully connected layer except that: a) each neuron receives an image as inputs and produces an image as its output (instead of a scalar); b) each synapse learns a small array of weights which is the size of the convolutional window; and c) each pixel in the output image is created by the sum of the convolutions between all synapse weights and the corresponding images. The output of the l^{th} convolutional layer, which takes as input S_l images of dimension $R_l \times C_l$, the pixel, $p_{l,n,r,c}$, at location (r, c) of the n^{th} output image is calculated as follows:

$$p_{l,n,r,c} = f_{act}(\sum_{s=0}^{S_l} \sum_{j=0}^{J_l} \sum_{k=0}^{K_l} w_{l,n,s,j,k} p_{l-1,n,r+j,c+k}) \quad , \tag{2}$$

where $J_l \times K_l$ are the dimensions of the convolution window. As discussed in Section 4, a 2D convolutional layer can be reduced to a matrix multiply followed by an elementwise activation function. CNN topologies are composed from a few common primitives: convolutional layers, *pooling* layers and fully connected layers.

Pooling layers can be considered as simple downsamplers of 2D images. A basic max pooling layer divides an image into small sub-tiles of a given window size and then replaces each sub-tile with its largest element. An average pooling layer is similar but uses the average function instead of max.

2.2 Binary Neural Networks

Although floating point numbers are a natural choice for handling the small updates that occur during neural network training, the resulting parameters can contain a lot of redundant information [8]. One of several possible dimensions possessing redundancy is precision [27]. An extreme case are BNNs in which some or all the arithmetic involved in computing the outputs are constrained to single-bit values. We consider three aspects of binarization for neural network layers: binary input activations, binary synapse weights and binary output activations. If all three components are binary, we refer to this as *full binarization*, and the cases with one or two components as *partial binarization*.

Kim and Smaragdis [12] consider full binarization with a predetermined portion of the synapses having zero weight, and all other synapses with a weight of one. They report 98.7% accuracy with fully-connected networks on the MNIST dataset, and observe that only XNOR and bitcount operations are necessary for computing with such neural networks. XNOR-Net by Rastegari et al. [22] applies convolutional BNNs on the ImageNet dataset with topologies inspired by AlexNet, ResNet and GoogLeNet, reporting top-1 accuracies of up to 51.2% for full binarization and 65.5% for partial binarization. DoReFa-Net by Zhou et al. [32] explores reduced precision during the forward pass as well as the backward pass, and note that this opens interesting possibilities for training neural networks on FPGAs. Their results includes configurations with partial and full binarization on the SVHN and ImageNet datasets, including best-case ImageNet top-1 accuracies of 43% for full and 53% for partial binarization.

Finally, the work by Courbariaux et al. [5] describes how to train fully-connected and convolutional networks with full binarization and batch normalization layers, reporting competitive accuracy on the MNIST, SVHN and CIFAR-10 datasets. Training for this work was performed using their open source implementation. We use the acronym CNN to refer to conventional or non-binarized neural networks for brevity throughout the rest of this paper.

2.3 Neural Networks in Hardware

A great deal of prior work on mapping neural networks to hardware exist both for FPGAs and as ASICs. We refer the reader to the work by Misra and Saha [16] for a comprehensive survey. We cover a recent and representative set of works here, roughly dividing them into four categories based on their basic architecture: 1) a single processing engine [20, 31, 4, 2], usually in the form of a systolic array, which processes each layer sequentially; 2) a streaming architecture [28, 1], consisting of one processing engine per network layer; 3) a vector processor [7] with instructions specific to accelerating the primitives operations of convolutions; and 4) a neurosynaptic processor [6], which implements many digital neurons and their interconnecting weights.

Systolic arrays: Zhang et al. [31] describes a single processing engine style architecture, using theoretical roofline models tool to design accelerators optimized for the execution of each layer. Ovtcharov et al. [20] implement a similar style architecture, but achieved a 3× speedup over Zhang et al. [31]. Eyeriss by Chen et al. [4] use 16-bit fixed point rather than floating point, and combine several different data reuse strategies. Each 2D convolution is mapped to 1D convolutions across multiple processing engines, allowing for completely regular access patterns for each processing element. The authors report that their data reuse provides 2.5× better energy efficiency over other methods. YodaNN by Andri et al. [2] have a similar design as Zhang et al. [31] but explore binary weights for fixed sized windows.

Streaming architectures: Venieris and Bouganis [28] proposed a synchronous dataflow (SDF) model for mapping

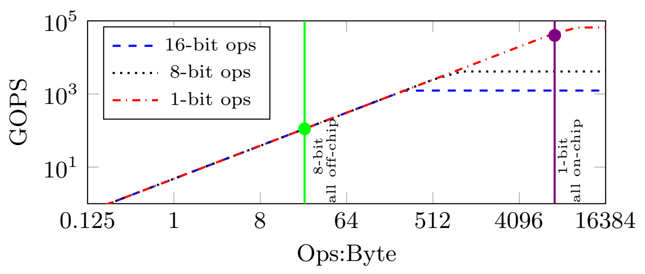


Figure 1: Roofline model for a ZU19EG.

CNNs to FPGAs, which is a similar approach to ours. The main difference is that our design is optimized for BNNs while their design targets conventional CNNs. Their designs achieve up to 1.62× the performance density of hand tuned designs. Alemdar et al. [1] implement fully-connected ternary-weight neural networks with streaming and report up to 255K frames per second on the MNIST dataset, but concentrate on the training aspect for those networks.

Vector processors: Farabet et al. [7] describe a programmable ConvNet Processor (CNP), which is a RISC vector processor with specific macro-instructions for CNNs including 2D convolutions, 2D spatial pooling, dot product and an elementwise non-linear mapping function. The authors also created a tool to compile a high level network description into host code which is used to call the CNP.

Neurosynaptic processors: TrueNorth [6] is a low power, parallel ASIC with 4096 neurosynaptic cores, each implementing 256 binary inputs, 256 neurons and a 256 × 256 array of synapses. An internal spiking router can connect any input on any core to any neuron on any core, allowing many network topologies to be implemented on fixed hardware.

The authors are not aware of any publication that demonstrates end-to-end mapping of BNNs onto FPGAs. In comparison to prior art, the binary network inference engine can significantly increase classification rates, while reducing power consumption and minimizing latency. This currently comes at the cost of a small drop in accuracy for larger networks, however we believe a) there are use cases that do not require the highest level of accuracy, or can be solved with smaller networks (such as classification of playing cards or handwritten digits [15]) and b) that accuracy can be improved by increasing network sizes [27], an ongoing topic in machine learning research.

3. BNN PERFORMANCE AND ACCURACY

3.1 Estimating Performance Using Rooflines

To estimate and compare BNN performance with fixed-point CNN, we use a *roofline model* [30] which considers memory bandwidth, peak computational performance and arithmetic intensity (the number of mathematical operations performed for each byte of off-chip memory read or written). The intersection of the roofline curve with a vertical line for a particular arithmetic intensity, gives the theoretical peak performance point, which is either *compute-bound* or *memory-bound*. In particular, we consider the binarized [32, 22] and 8-bit fixed-point [26] implementations of the popular AlexNet [14], both of which require 1.4 billion operations (GOPS) to classify one image.

Using the methodology described in [17], we develop a roofline model for a Xilinx Zynq UltraScale+ ZU19EG FPGA[1]. The resulting roofline model is depicted in Figure 1. We first observe that the FPGA's compute-bound performance is 66 TOPS for binary operations, which is about 16× higher compared to 8-bit and 53× higher compared to 16-bit fixed point operations. However, reaching the compute-bound peak is only possible if the application is not memory-bound. The compact model size of BNNs provides another key benefit. Since the binarized AlexNet requires only 7.4 MB of parameters (compared with 50 MB for 8-bits), the entire neural network model can be kept in on-chip memory. The arithmetic intensities for the binarized and 8-bit fixed point AlexNet variants are shown with vertical lines. Thus, the BNN is almost able to reach the computational peak, while the peak performance of the fixed-point CNN is bound by the memory bandwidth. Based on these observations, with a design that reaches 75% of the peak, we estimate a throughput of $0.75 \cdot \frac{66 \text{ TOPS}}{1.4 \text{ GOPS}} \approx 35000$ images per second.

Using the same model, it should be possible to extend the comparison to CPUs and GPUs, but little data is available on peak binary synaptic operation performance since BNNs are relatively new. For instance, [5] mentions 6 cycles per 32 synapses (64 binary operations) on recent NVIDIA GPUs, which would yield a computational peak of about 26 TOPS on a Tesla K40 with 2880 cores running at 875 MHz, and 16666 images per second for binarized AlexNet.

3.2 Accuracy–Computation Tradeoffs

A tradeoff between network size, precision and accuracy exists [27] so if one would like to achieve a certain classification accuracy for a particular problem, which approach leads to the most efficient solution? 1) A regular ANN with floating point precision? 2) A larger network, but a BNN? To gain more insight into this issue, we conducted a set of experiments on the MNIST dataset that compare accuracy of floating point and binary precision for the same topology. The binary networks are obtained via replacing regular layers by their binary equivalents, as described by Courbariaux et al. [5]. We also binarize the input images for the BNN as our experiments show that input binarization works well for MNIST. Since the space of possible network topologies that can be trained is infinite, we adopted the approach in [27] to simplify the problem. We fix the network topology to a 3 hidden layer, fully connected network while scaling the number of neurons in each layer, and plot the resulting accuracy in Table 1 along with the number of parameters and operations per frame. A few trends are apparent for this problem and network configuration space: 1) similar to what was found in by Sung et al. [27], as the network size increases, the difference in accuracy between low precision networks and floating point networks decreases; and 2) in order to achieve the same level of accuracy as floating point networks, BNNs require 2–11× more parameters and operations. Note that we show the accuracy for networks trained using 32-bit floating point numbers, but it is likely that this could be reduced to 8-bit fixed point without a significant change in accuracy [10]. Our BNN performance estimates from Section 3.1 suggest a 16× speedup for BNN over 8-bit fixed point, which is greater than the 2–11× increase in parameter and operation size. Thus, we expect that BNNs with compara-

[1]We assume 4.8 GB/s off-chip memory bandwidth, 350 MHz clock and the following operation cost function: 2.5 LUTs for 1-bit, 40 LUTs for 8-bit, 8 LUTs and 0.5 DSPs for 16-bit.

Table 1: Accuracy results - BNN vs floating point NN.

Neurons/layer	Binary Err. (%)	Float Err. (%)	# Params	Ops/frame
128	6.58	2.70	134,794	268,800
256	4.17	1.78	335,114	668,672
512	2.31	1.25	932,362	1,861,632
1024	1.60	1.13	2,913,290	5,820,416
2048	1.32	0.97	10,020,874	20,029,440
4096	1.17	0.91	36,818,954	73,613,312

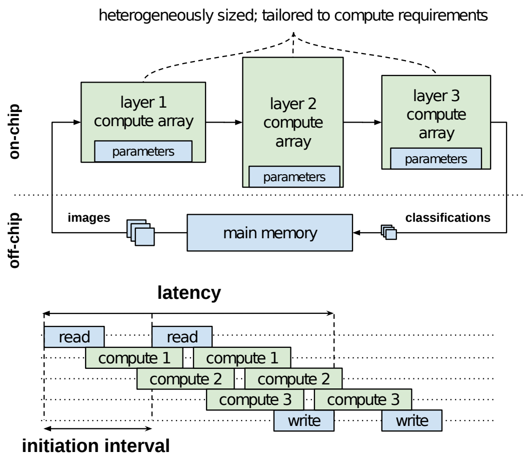


Figure 2: Heterogeneous streaming architecture and schedule.

ble accuracy will be faster than fixed-point networks, even though they may require more parameters and operations.

4. BNNs ON RECONFIGURABLE LOGIC

4.1 Architecture

We adopted a *heterogeneous streaming* architecture as shown in Figure 2 for this work. We build a custom architecture for a given topology rather than scheduling a operations on top of a fixed architecture. Separate compute engines are dedicated to each layer, which communicate via on-chip data streams. Each engine starts to compute as soon as the previous engine starts to produce output. Additionally, owing to the compact model size of BNNs, all neural network parameters are kept in on-chip memory. This avoids most accesses to off-chip memory, minimizes the *latency* (the time to finish classifying one image) by overlapping computation and communication, and minimizes the *initiation interval*: a new image can enter the accelerator as soon as the first compute array is finished with the previous image. The separate mapping of layers to compute arrays also enables heterogeneity. By tailoring compute arrays separately for each layer's requirements, we can avoid the "one-size-fits-all" inefficiencies and reap more of the benefits of reconfigurable computing. This requires a different bitfile when the neural network topology is changed but we consider this an acceptable cost for the performance gains obtained.

A BNN accelerator may have various constraints imposed upon it depending on the use case. User-imposed constraints include the choice of FPGA and platform, desired classification throughput in frames per second (FPS) and clock frequency. Simultaneously, the BNN topology constrains how the compute resources must be allocated to obtain an efficient heterogeneous streaming architecture. FINN offers parameterizable building blocks and a way of controlling the classification throughput, as described in Sections 4.3 and 4.4. To achieve portability, we chose a commercial high level synthesis tool, Vivado High-Level Synthesis (HLS), for the implementation. The tool enables faster development cycles via high-level abstractions, and provides automated pipelining to meet the clock frequency target.

4.2 BNN-specific Operator Optimizations

BNNs have several properties that enable a more efficient mapping to FPGAs without affecting the network accuracy, which we describe in the following subsections. We assume that the methodology described in [5] is used for training all BNNs in this paper, where all BNN layers have the following properties (unless otherwise stated):

- Using 1-bit values for all input activations, weights and output activations (full binarization), where an unset bit represents -1 and a set bit represents +1.
- Batch normalization prior to the activation function.
- Using the following activation function: $\text{Sign}(x) = \{+1 \text{ if } x \geq 0, -1 \text{ if } x < 0\}$

4.2.1 *Popcount for Accumulation*

The regular and value-constrained nature of BNN computations enable computing binary dot products with fewer hardware resources. Let Y be the number of input synapses (or *fan-in*) for a given neuron, with the number of +1-valued synapse inputs denoted as Y_1 and -1-valued synapses as Y_0. As there are only two possible values (-1 and +1) for any synapse input, $Y = Y_0 + Y_1$. Therefore, by counting the number of synapses for only one value, it is possible to infer the summed response for the entire neuron.

The practical consequence for hardware is that the summation of a binary dot product can be implemented by a *popcount* operation that counts the number of set bits instead of accumulation with signed arithmetic. Our experiments with Vivado HLS indicate that popcount-accumulate requires approximately half the number of LUT and FF resources to implement compared to signed-accumulate. For instance, with a target $F_{\text{clk}} = 200$ MHz, a 128-bit popcount-accumulate requires 376 LUTs and 29 FFs, while a 128-bit bipolar-accumulate requires 759 LUTs and 84 FFs.

4.2.2 *Batchnorm-activation as Threshold*

All BNN layers use batch normalization [11] on convolutional or fully connected layer outputs, then apply the sign function to determine the output activation. We show how the same output can be computed via thresholding.

Let a_k be the dot product (pre-activation) output of neuron k, and $\Theta_k = (\gamma_k, \mu_k, i_k, B_k)$ be the batch normalization parameters learned during training for this neuron. The output a_k^b is computed as $a_k^b = \text{Sign}(\text{BatchNorm}(a_k, \Theta_k))$, with $\text{BatchNorm}(a_k, \Theta_k) = \gamma_k \cdot (a_k - \mu_k) \cdot i_k + B_k$. Figure 3 shows the dot product input vs output activation for three example neurons. Depending on parameter values, the plot may be shifted towards the left or right, or be flipped horizontally, but a threshold τ_k for a change in the output activation is always present. Solving $\text{BatchNorm}(\tau_k, \Theta_k) = 0$ we can deduce that $\tau_k = \mu_k - (B_k/(\gamma_k \cdot i_k))$.

To make the thresholds compatible with the positive-only operations in Section 4.2.1), the computed threshold is averaged with the neuron fan-in S to obtain $\tau_k^+ = (\tau_k + S)/2$. Observing how neuron C activates with an opposite sign threshold to neurons A and B in Figure 3, all neurons can be

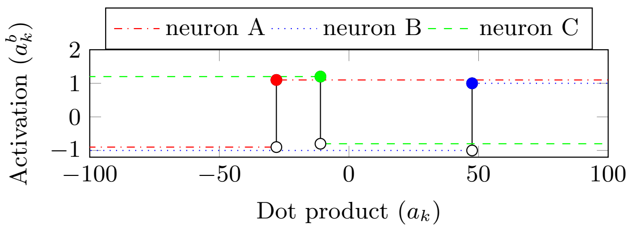


Figure 3: Three examples of binary neuron activations with batch normalization. A slight vertical offset is added for clarity.

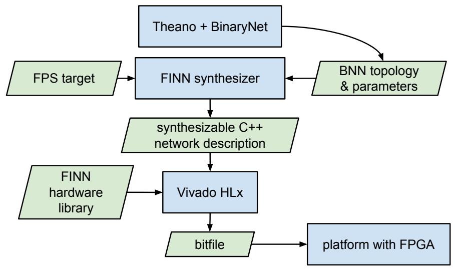


Figure 4: Generating an FPGA accelerator from a trained BNN.

made to activate using a greater-than threshold by flipping the signs of a neuron's weights if $\gamma_k \cdot i_k < 0$.

Using these techniques, we can compute the output activation using an unsigned comparison and avoid computing the batch normalized value altogether during inference. τ_k^+ itself is fixed for a trained network and can be computed from the batchnorm parameters at compile time. Synthesis reports from Vivado HLS for 16-bit dot product output values indicate that regular batchnorm-and-sign activation requires 2 DSPs, 55 FFs and 40 LUTs, whereas the threshold activation we describe here only requires 6 LUTs.

4.2.3 *Boolean OR for Max-pooling*

The networks described in [5] perform pooling prior to activations, i.e. pooling is performed on non-binarized numbers, which are then batch normalized and fed into the activation function. We show that the same layer outputs can be derived by max pooling *after* the activations without having to re-train the network. Let $a_1, a_2, \ldots a_Y$ be the positive dot product outputs that will be processed by max-pooling. In accordance with Section 4.2.2, the output would be computed as $a^b = (\text{Max}(a_1, a_2, \ldots a_Y) > \tau^+)$. Due to the distributivity of Max, the output will be *true* if *any* of $a_1, a_2, \ldots a_S$ are greater than τ^+. Therefore, the same result can be computed as $a^b = (a_1 > \tau^+) \vee (a_2 > \tau^+) \ldots \vee (a_Y > \tau^+)$. As the threshold comparisons are already computed for the activations, max-pooling can be effectively implemented with the Boolean OR-operator. We note that similar principles apply for min-pooling (as Boolean AND) and average-pooling (as Boolean majority function) as well.

4.3 FINN Design Flow and Hardware Library

Figure 4 illustrates the design flow for converting a trained BNN into an FPGA accelerator. The user supplies a FPS target alongside a Theano-trained BNN to the FINN synthesizer. The synthesizer first determines the folding parameters (Section 4.4) to meet the FPS target and applies the optimizations from Section 4.2, then produces a synthesizable C++ description of a heterogeneous streaming architecture. The architecture is composed of building blocks from the FINN hardware library described in the following subsections.

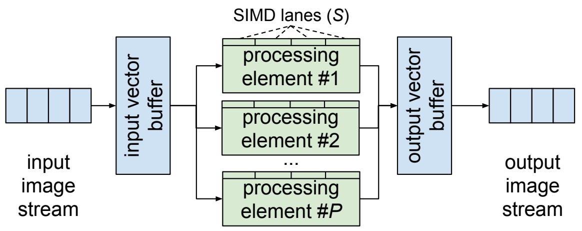


Figure 5: Overview of the MVTU.

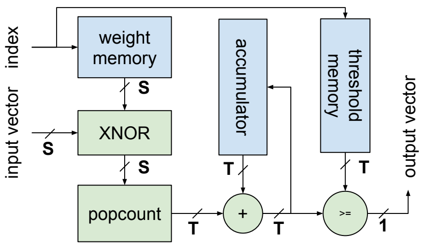


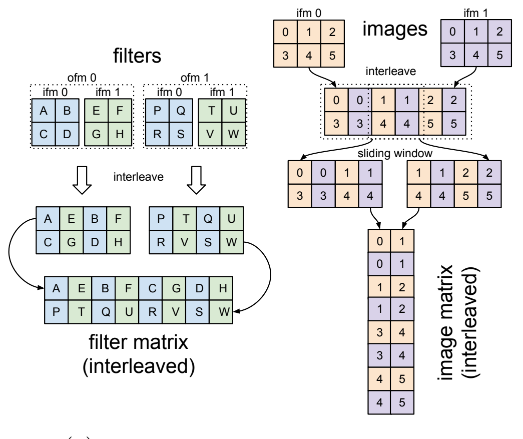
Figure 6: MVTU PE datapath. **Bold** indicates bitwidth.

4.3.1 *The Matrix–Vector–Threshold Unit*

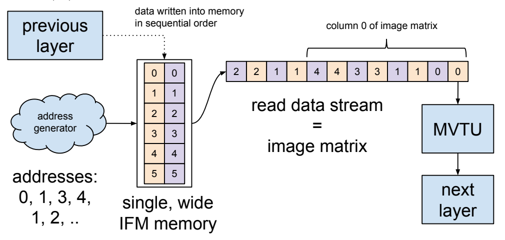
The Matrix–Vector–Threshold Unit (MVTU) forms the computational core for our accelerator designs. The vast majority of compute operations in a BNN can be expressed as matrix–vector operations followed by thresholding. For instance, the pre-activation output $\mathbf{a_N}$ of the fully connected neural network layer at index N is given by matrix-vector product $\mathbf{a_N} = \mathbf{A} \cdot \mathbf{a^b_{N-1}}$ where $\mathbf{A}$ is the synaptic weight matrix and $\mathbf{a^b_{N-1}}$ are the activations from the previous layer. The post-activation output can then be computed by $\mathbf{a^b_N} = \mathbf{a^N} > \tau^+_{\mathbf{N}}$, where the thresholds $\tau^+_{\mathbf{N}}$ are determined as described in Section 4.2.2. Convolutions can also be implemented as matrix–vector products, as will be described in Section 4.3.2. As such, the MVTU implements fully-connected layers as a standalone component, and is also used as part of the convolutional layers.

The overall organization of the MVTU is shown in Figure 5. Internally, the MVTU consists of an input and output buffer, and an array of Processing Elements (PEs) each with a number of SIMD lanes. The number of PEs (P) and SIMD lanes (S) are configurable to control the throughput as discussed in Section 4.4.1. The synapse weight matrix to be used is kept in On-Chip Memory (OCM) distributed between PEs, and the input images stream through the MVTU as each one is multiplied with the matrix. Each PE receives exactly the same control signals and input vector data, but multiply-accumulates the input with a different part of the matrix. In terms of the taxonomy described in [4], this architecture is both *weight stationary* (since each weight remains local to the PE) and *output stationary* (since each popcount computation remains local to the PE).

Figure 6 shows the datapath of an MVTU PE. It computes the dot product between the input vector and a row of the synaptic weight matrix and compares the result to a threshold, producing a single-bit output. The dot product computation itself consists of an XNOR of the vectors, after which the number of set bits in the result is counted (see Section 4.2.1) and added to the accumulator register. Once the entire dot product is accumulated, it is thresholded. The accumulator, adder and threshold memory bitwidth can be scaled down to $T = 1 + \log_2(Y)$ for additional resource savings.



(a) Lowering with interleaved channels.


previous layer
data written into memory in sequential order
address generator
addresses: 0, 1, 3, 4, 1, 2, ..
single, wide IFM memory
column 0 of image matrix
read data stream = image matrix
MVTU
next layer

(b) SWU operation.

Figure 7: Convolution using interleaved channels.

Finally, it is worth pointing out that the MVTU architectural template can also support partial binarization for non-binarized outputs and inputs. Removing the thresolding stage provides non-binarized outputs, while using regular multiply-add instead of XNOR-popcount can handle non-binarized inputs These features are used in the first and last layers of networks that process non-binary input images or do not output a one-hot classification vector.

4.3.2 Convolution: The Sliding Window Unit

Convolutions can be *lowered* to matrix-matrix multiplications [3], which is the approach followed in this work. The weights from the convolution filters are packed into a *filter matrix*, while a sliding window is moved across input images to form an *image matrix*. These matrices are then multiplied to generate the output images.

The convolutional layer consists of a Sliding Window Unit (SWU), which generates the image matrix from incoming feature maps, and a MVTU that actually computes the matrix–matrix product using a different column vector from the image matrix each time. In order to better cater for the SIMD parallelism of the MVTU and minimize buffering requirements, we *interleave* the feature maps such that each pixel contains all the Input Feature Map (IFM) channel data for that position, as illustrated in Figure 7a. Since the dot product to compute a Output Feature Map (OFM) pixel includes all IFMs pixels at a certain sliding window location, those IFM pixels can be processed in any order owing to the commutative property of addition. Note that interleaving the filter matrix has no additional cost since it is done offline, and interleaving the input image can be done on-the-fly in the FPGA. Storing the pixels in this fashion allows us to implement the SWU with a single wide OCM instead of multiple narrow OCMs, and also enables the output of the MVTU to be directly fed to the next layer without any transposition. As illustrated in Figure 7b, the incoming IFM data is simply stored at sequential addresses in a buffer, then the memory locations corresponding to each sliding window are read out to produce the image matrix.

Although not required by any of the networks described in this work, the SWU also pads the images if necessary. One interesting observation is that with the bipolar number representation used in this work, there is no number corresponding to zero. Therefore, in order to maintain a true binary datapath for activations, images must be padded with our representation or either a 1 or a -1. Future work will look into what impact this has on the accuracy of trained networks, but early experiments suggest that there is very little difference in accuracy, with respect to [5].

4.3.3 The Pooling Unit

The Pooling Unit (PU) implements max-pooling as described in Section 4.2.3. To implement $k \times k$ max-pooling on a $D_H \times D_W$ binary image of C channels, the PU contains $C \cdot k$ line buffers of D_W bits each. As with the rest of our component library, the PU operates in a streaming fashion. The input image is gradually streamed into the line buffers. When at least k rows of the image have arrived, each k consecutive bits of the line buffer are OR'ed together to produce horizontal subsampling for each channel. These are then OR'ed together with the other line buffers to produce vertical subsampling, the results are streamed out, and the oldest line buffers are refilled with the next row of pixels.

4.4 Folding

In terms of the MVTU description given in Section 4.3.1, each PE corresponds to a *hardware neuron*, while each SIMD lane acts as a *hardware synapse*. If we were to dimension each MVTU in a network with a number of hardware neurons and synapses equal to the number of neurons and synapses in a BNN layer, this would result in a fully parallel neural network that could classify images at the clock rate. However, the amount of hardware resources on an FPGA is limited, and it is necessary to time-multiplex (or *fold*) the BNN onto fewer hardware synapses and neurons. We now describe how the folding is performed subject to user constraints.

The work by Venieris et al. [28] describes a method for folding neural networks expressed as streaming dataflow graphs, with focus on formalizing the folding and design space exploration. In this work, we consider a simpler variant that only controls the folding of matrix–vector products to achieve a given FPS requirement set by the user, and focus on *how* the folding is implemented in terms of the workload mapping. As almost all computations in BNNs are expressed as matrix–vector multiplications, implementing folding for matrix–vector multiplication already enables a great degree of control over the system throughput. Folding directly affects the resource and power consumption of the final system as well, which we explore in Section 5.

4.4.1 Folding Matrix–Vector Products

Folding matrix–vector products is achieved by controlling two parameters of the MVTU: P the number of PEs, and S the number of SIMD lanes per PE. These determine how the matrix is partitioned between the PEs. A P-high, S-wide tile of the matrix is processed at a time, with each row in the tile mapped to a different PE, and each column to a different SIMD lane. For a $X \times Y$ matrix, we refer to $F^n = X/P$

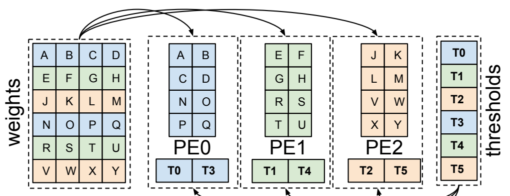


Figure 8: Neuron and synapse folding for MVTU.

as the *neuron fold* and $F^s = Y/S$ as the *synapse fold*. The *total fold* F is then obtained as $F = F^n \cdot F^s$, which is also the number of cycles required to complete one matrix–vector multiply. Note that F^n and F^s should be integers to avoid padding the weight matrix. As an example, Figure 8 shows how a 6×4 weight matrix is partitioned between three PEs with two SIMD lanes each. Here, each matrix-vector multiply will take $F^n \cdot F^s = (6/3) \cdot (4/2) = 4$ cycles.

The same principle applies for convolutional layers, but these always have an inherent amount of folding due to our current matrix–matrix product as multiple matrix–vector products implementation. For convolutional layers, the total fold is $F = F^m \cdot F^n \cdot F^s$, where F^m is a network-dependent constant due to multiple matrix-vector products, and is equal to the number of output pixels from the convolution.

4.4.2 Determining F^n and F^s

Avoiding the "one-size-fits-all" inefficiencies requires tailoring each MVTU's compute resources to layer requirements. The guiding principle here is *rate-balancing* the heterogeneous streaming architecture: the slowest layer (with $II_{\max}$) will determine the overall throughput, so each layer should use a roughly equal number of cycles to process one image. As this is a streaming system, the classification throughput FPS will be approximately $\frac{F_{\text{clk}}}{II_{\max}}$, where F_{clk} is the clock frequency. For a fully-connected layer, the total fold F is equal to the initiation interval (II). Therefore, balancing a fully-connected BNN can be achieved by using F^n and F^s such that $F^n \cdot F^s = \frac{F_{\text{clk}}}{\text{FPS}}$ for each layer. Depending on the BNN and the FPS requirements, the number of memory channels or sliding window generation may constitute bottlenecks. For such cases, we match the throughput of all other layers to the bottleneck in order not to waste resources.

5. EVALUATION

5.1 Experimental Setup

To evaluate FINN, we created a number of prototypes that accelerate BNNs inference on the MNIST [15] (28×28 handwritten digits), CIFAR-10 [13] (32×32 color images in 10 categories) and cropped SVHN [18] (32×32 images of Street View House Numbers) datasets. Each prototype combines a BNN topology with a different use case scenario. We consider three different BNN topologies for classifying the datasets as follows:

- **SFC** and **LFC** are three-layer fully connected network topologies for classifying the MNIST dataset, with different numbers of neurons to demonstrate accuracy-computation tradeoffs (Section 3.2). SFC contains 256 neurons per layer and achieves 95.83% accuracy, while LFC has 1024 neurons per layer and achieves 98.4% accuracy. These networks accept 28x28 binary images and output a 10-bit one-hot vector indicating the digit.
- **CNV** is a convolutional network topology inspired by BinaryNet [5] and VGG-16 [25]. It contains a succession of (3x3 convolution, 3x3 convolution, 2x2 maxpool) layers repeated three times with 64-128-256 channels, followed by two fully connected layers of 512 neurons each. We use this topology for classifying both the CIFAR-10 (with 80.1% accuracy) and SVHN (with 94.9% accuracy) datasets, with different weights and thresholds. Note that the inputs to the first layer and the outputs from the last layer are not binarized; CNV accepts 32x32 images with 24 bits/pixel, and returns a 10-element vector of 16-bit values as the result.

To further demonstrate the flexibility of the framework, we consider two usage scenarios for each BNN topology to guide the choice of parametrization:

- **max** is the maximum performance scenario where it is desirable to reach the peak FPS permitted by the platform, topology and FINN's architecture.
- **fix** represents a scenario with a fixed FPS requirement, which is often determined by an I/O device for real life applications. For instance, consider a 640×480 video stream at 30 FPS, which is to be chopped up into 32×32 tiles for neural network inference. Handling this task with real-time performance would require a BNN inference rate of 9000 FPS, which we set as the requirement for this usage scenario.

We use shortened names to refer to the prototypes, e.g. CNV-fix refers to the prototype that implements the **CNV** topology for the **fix** usage scenario. For each prototype, the folding factors (Section 4.4) were determined to meet the requirements of its usage scenario, and the FINN design flow (Section 4.3) was followed to generate the hardware accelerator. Vivado HLS and Vivado version 2016.3 were used for the bitfile synthesis. A target clock frequency of 200 MHz was used for both Vivado HLS and Vivado, and to run the resulting accelerator unless otherwise stated. The salient properties of the topologies and folding factors for the prototypes are summarized in Table 2.

All prototypes have been implemented on the Xilinx Zynq-7000 All Programmable SoC ZC706 Evaluation Kit running Ubuntu 15.04. The board contains a Zynq Z7045 SoC with dual ARM Cortex-A9 cores and FPGA fabric with 218600 LUTs and 545 BRAMs. The host code runs on the Cortex-A9 cores of the Zynq. It initializes 10000 images with test data in the Zynq's shared DRAM, launches and times the accelerator execution to measure classification throughput, then measures accuracy by comparing against the correct classifications. Two power measurements P_{chip} and P_{wall} are provided for each experiment; P_{chip} using the PMBus interface to monitor the FPGA power supply rails, and P_{wall} using a wall power meter for the total board power consumption. The measurements are averaged over a period of 10 seconds while the accelerator is running.

5.2 Results

Table 3 provides an overview of the experimental results, in terms of classification throughput, latency to classify one image, FPGA resource usage and power. The **max** scenario

Table 2: Summary of workloads.

Topology	Params (Mbits)	Ops (M)	Off-chip I/O (B)	Op.Int. (Ops/B)
SFC	0.3	0.6	112	5970
LFC	2.9	5.8	112	51968
CNV	1.5	112.5	3092	36400

Prototype	Per-Layer Total Fold (F)
SFC-max	13, 16, 16, 16
SFC-fix	12544, 16384, 16384, 2560
LFC-max	104, 128, 128, 128
LFC-fix	13312, 16384, 16384, 10240
CNV-max	8100, 7056, 5184, 7200, 5184, 4608, 8192, 8192, 1280
CNV-fix	16200, 14112, 10368, 14400, 10368, 9216, 16384, 16384, 1280

Table 3: Summary of results from FINN 200 MHz prototypes.

Name	Thr.put (FPS)	Latency (µs)	LUT	BRAM	P_{chip} (W)	P_{wall} (W)
SFC-max	12361 k	0.31	91131	4.5	7.3	21.2
LFC-max	1561 k	2.44	82988	396	8.8	22.6
CNV-max	21.9 k	283	46253	186	3.6	11.7
SFC-fix	12.2 k	240	5155	16	0.4	8.1
LFC-fix	12.2 k	282	5636	114.5	0.8	7.9
CNV-fix	11.6 k	550	29274	152.5	2.3	10

results are perhaps the best summary of the potential of BNNs on FPGAs, with SFC-max achieving 12.3 million classifications per second at 0.31 µs latency while drawing less than 22 W total power. All **fix** results meet and exceed the 9000 FPS requirement by 30% due to folding factors being integers, though lower throughput and power could have been achieved by using a slower clock. We focus on particular aspects of the results in the following subsections.

5.2.1 Maximum Throughput and Bottlenecks

To assess the quality of results for the **max** scenarios, we compare the achieved performance (XNOR–popcount operations per second) with the peak throughput in TOPS indicated by the roofline model. Figure 9 presents a roofline model (Section 3.1) for the ZC706, assuming 90% LUT utilization, 200 MHz clock frequency and 1.6 GB/s of DRAM bandwidth. The vertical lines show the arithmetic intensities for the topologies, and the actual operations per second values from corresponding prototypes with **max** usage scenarios are indicated as points on those lines. All **max** prototypes achieve performance in the TOPS range, but are bottlenecked due to different factors. CNV-max achieves 2.5 TOPS and is *architecture-bound*. The current SWU design does not scale as well as the MVTU and constitutes a bottleneck, which will be addressed in future work. Despite its higher complexity, observe that CNV-max actually requires ~2× fewer LUTs

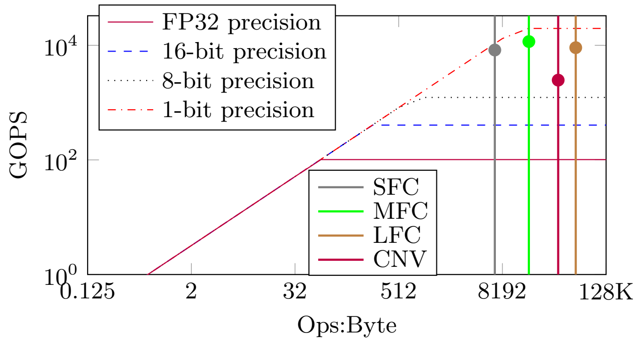


Figure 9: ZC706 roofline with topologies and **max**-datapoints.

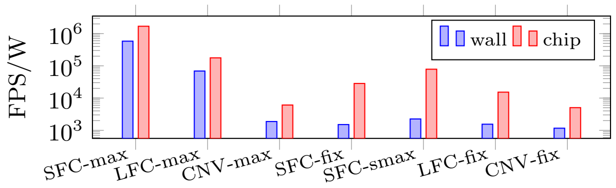


Figure 10: Prototype energy efficiency.

than SFC-max since the folding parameters for CNV-max are chosen in accordance with the maximum performance dictated by the bottleneck. SFC-max achieves 8.2 TOPS and is *memory-bound*. Observe that the SFC arithmetic intensity line intersects the memory-bound (sloped) part of the roofline, thus the performance cannot be scaled up without adding more DRAM memory bandwidth. LFC-max achieves 9.1 TOPS, which is 46% of the roofline, and is *resource-bound*. As folding factors are integers, the smallest increment is 2× which roughly doubles the resource cost. The FPGA has enough LUTs but not enough BRAMs to accommodate doubled resource cost, thus leaving ~30% of BRAMs unused. A 3x512-neuron fully connected topology, labeled MFC in Figure 9, was able to achieve 11.6 TOPS and 6238 kFPS with 95% of the device BRAMs.

5.2.2 Energy Efficiency

It is desirable to minimize the energy spent per image classification, which corresponds to maximizing FPS per Watt when many images are to be classified. To help evaluate the energy efficiency, Figure 10 plots the achieved FPS per Watt for the prototypes for both the wall power and FPGA power readings. In general, we see that the higher FPS prototypes have better energy efficiency, with SFC-max offering 583066 FPS per W of total power and outperforming all other prototypes by at least an order of magnitude. It is also worth noting that the board's idle power consumption is about 7 W, which forms a lower bound on all wall power measurements, and could be improved by e.g. using LPDDR memory.

To maximize energy efficiency with a fixed target FPS, is it better to use a highly parallel design at low clock frequency, or a less parallel design at high clock frequency? We ran an additional experiment to investigate this question by slowing down the SFC-max prototype to meet the **fix** FPS requirement of 9000 FPS. By clocking it at 250 kHz, we obtained a classification throughput of 15731 FPS with 0.2 W of FPGA power. The result is labeled SFC-smax in Figure 10, and is over 2× more energy efficient than SFC-fix. This suggests that a high degree of parallelism benefits energy efficiency as long as the FPGA resources are available.

5.2.3 Resource Efficiency

We consider two aspects of resource efficiency for FINN: how efficiently the compute units are used during runtime (*runtime efficiency*), and how efficiently FPGA resources are turned into compute units (*mapping efficiency*).

To assess runtime efficiency, we divide the FPS-based (actual) operations per cycle ($\frac{\text{FPS}\cdot\text{Ops}}{F_{\text{clk}}}$) by the (peak) number of synaptic operations per cycle from the design ($\sum 2 \cdot P \cdot S$). The prototypes exhibit good runtime efficiency, with ~70% for **CNV**, ~80% for **SFC** and ~90% for **LFC**. The efficiency can be increased further by fine-tuning the folding factors between different layers.

Evaluating the mapping efficiency directly on the proto-

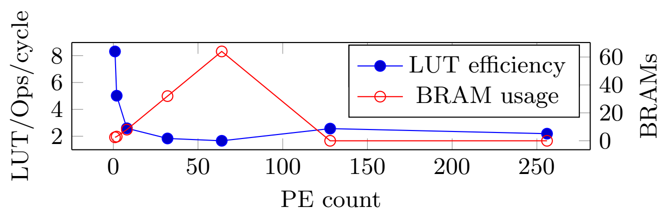


Figure 11: Mapping resource efficiency.

types loses some insight, since **CNV** uses LUTs on SWU and PU, while fully-connected topologies do not. Instead, for a single 256×256 fully-connected layer, we fix $S = 64$ and vary P, and plot the LUTs per synaptic operation in Figure 11, which should be minimized to maximize efficiency. The LUTs per operation decreases with higher P since the fixed-size control logic is amortized between more PEs and reaches a minimum of 1.83 for $P = 64$, but increases again for $P > 64$. To understand why, we also plot the number of BRAMs used in the same figure. Although all designs have the same number of BNN parameters, the number of BRAMs increases with P since each PE needs its own weight and threshold memories. This also means a significant part of the BRAM storage capacity is unused for $1 < P \leq 64$, since the same amount of network parameters is divided between a greater number of memories. This is also visible for SFC-fix and SFC-max, which use the same network parameters, but have almost 10× difference in the number of BRAMs used (15.5 vs 130.5) since SFC-max has more compute elements working in parallel. Here, with $P > 64$, so little of each BRAM is used that Vivado HLS implements the weight and threshold memories using LUTs, which causes the LUTs per operation to increase. Thus, the depth and number of BRAMs, and the LUT-to-BRAM ratio of the FPGA plays a key role in determining how well the resources will be utilized by a BNN. For instance, on another FPGA with the same amount of LUTs but twice the number of half-depth BRAMs, LFC-max could achieve 2× throughput.

5.3 Comparison to prior work

From an application perspective, we suggest that the current best way to compare different platforms is to simply compare their accuracy, FPS and power consumption when working on the same benchmark datasets (MNIST, CIFAR-10 and SVHN). This comparison is provided in Table 4, and is divided into three sections: our results, prior work on low-precision (< 4 bits) networks, and prior work with higher-precision (> 4 bits) networks.

When it comes to pure image throughput, our designs outperform all others. For the MNIST dataset, we achieve an FPS which is over 48/6× over the nearest highest throughput design [1] for our SFC-max/LFC-max designs respectively. While our SFC-max design has lower accuracy than the networks implemented by Alemdar et al. [1] our LFC-max design outperforms their nearest accuracy design by over 6/1.9× for throughput and FPS/W respectively. For other datasets, our CNV-*max* design outperforms TrueNorth [6] for FPS by over 17/8× for CIFAR-10 / SVHN datasets respectively, while achieving 9.44× higher throughput than the design by Ovtcharov et al. [20], and 2.2× over the fastest results reported by Hegde et al. [9]. Our prototypes have classification accuracy within 3% of the other low-precision works, and could have been improved by using larger BNNs.

A recent work by Nurvitadhi et al. [19] compares binary matrix-vector operation performance and efficiency on FPGA, ASIC, GPU and CPU. Their results indicate that CPU and GPUs are severely underutilized for binary synaptic operations, and that FPGAs are only ~8× less energy efficient than ASICs in this case. As they do not provide results on end-to-end network implementations, we do not include them in Table 4. Our 11.6 TOPS MFC prototype (Section 5.2.1) is 20% faster than the 9.6 TOPS reported in their work.

6. CONCLUSION

This work demonstrates the performance and energy efficiency potential of recently proposed BNNs for image classification. They are particularly well-suited for FPGA implementations as parameters can be fit entirely in OCM and arithmetic is simplified, enabling high computational performance. The novel parameterizable dataflow architecture and optimizations presented enable unprecedented classification rates, minimal power consumption and latency, while offering the flexibility of C++ design entry and the scalability required for accelerating larger and more complex networks. We hence believe that this technology is eminently suitable for embedded applications requiring real-time response, including surveillance, robotics and augmented reality. Future work will focus on providing support for non-binary low precision, implementing larger networks like AlexNet, higher performance convolutions, and a more thorough design space exploration. Finally, FINN assumes that all BNN parameters can fit into the available OCM of a single FPGA. Supporting external memory, multi-FPGAs implementations and reconfiguration [28] could improve the utility of our approach.

Acknowledgments

The authors would like to thank the NTNU HPC lab and colleagues at Xilinx Research Labs for their support. This work was supported under the Australian Research Councils Linkage Projects funding scheme (project number LP130101034).

7. REFERENCES

[1] H. Alemdar, N. Caldwell, V. Leroy, A. Prost-Boucle, and F. Pétrot. Ternary Neural Networks for Resource-Efficient AI Applications. *CoRR*, abs/1609.00222, 2016.
[2] R. Andri, L. Cavigelli, D. Rossi, and L. Benini. YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights. *CoRR*, abs/1606.05487, 2016.
[3] K. Chellapilla, S. Puri, and P. Simard. High performance convolutional neural networks for document processing. In *Proc. ICFHR*. Suvisoft, 2006.
[4] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proc. ACM/IEEE ISCA*. IEEE, 2016.
[5] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio. Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.
[6] S. K. Esser, P. A. Merolla, J. V. Arthur, A. S. Cassidy, R. Appuswamy, A. Andreopoulos, D. J. Berg, J. L.

Table 4: Comparison to prior work. Metrics not reported by prior work are indicated by dashes (-), and our estimates by ∼ .

Name	Dataset	Platform	Precision	Err. (%)	kFPS	P_{chip} (W)	P_{wall} (W)	kFPS/P_{chip}	kFPS/P_{wall}	GOPS
SFC-max	MNIST	ZC706	1	4.17	12,361	7.3	21.2	1693.29	583.07	8,265.45
LFC-max	MNIST	ZC706	1	1.60	1,561	8.8	22.6	177.39	69.07	9,085.67
MFC-max	MNIST	ZC706	1	2.31	6,238	11.3	28.5	552	218.8	11,612.86
CNV-max	CIFAR-10	ZC706	1	19.90	21.9	3.6	11.7	6.08	1.87	2,465.5
CNV-max	SVHN	ZC706	1	5.10	21.9	3.6	11.7	6.08	1.87	2,465.5
Alemdar et al. [1]	MNIST	Kintex-7 160T	2	2.24	255.10	0.32	-	806.45	-	∼96.68
Alemdar et al. [1]	MNIST	Kintex-7 160T	2	1.71	255.10	1.84	-	138.50	-	∼448.47
Alemdar et al. [1]	MNIST	Kintex-7 160T	2	1.67	255.10	2.76	-	92.59	-	∼864.03
Park and Sung [21]	MNIST	ZC706	3	-	70	4.98	-	14.06	-	∼210
TrueNorth [6]	CIFAR-10	TrueNorth	1	16.59	1.249	0.2044	-	6.11	-	-
TrueNorth [6]	SVHN	TrueNorth	1	3.34	2.526	0.2565	-	9.85	-	-
CaffePresso [9]	MNIST	Keystone-II	16	-	5	-	14	-	0.357	44.82
CaffePresso [9]	CIFAR-10	Keystone-II	16	-	10	-	14	-	0.714	146.14
CaffePresso [9]	MNIST	Parallella	32	-	0.64	-	5	-	0.129	5.78
CaffePresso [9]	CIFAR-10	Parallella	32	-	0.1	-	5	-	0.019	1.40
Ovtcharov et al. [20]	CIFAR-10	Stratix V D5	32	∼11-26	2.32	-	25	-	0.093	-

McKinstry, T. Melano, D. R. Barch, et al. Convolutional Networks for Fast, Energy-Efficient Neuromorphic Computing. *CoRR*, abs/1603.08270, 2016.

[7] C. Farabet, C. Poulet, J. Y. Han, and Y. LeCun. CNP: An FPGA-based processor for convolutional networks. In *Proc. IEEE FPL*, pages 32–37. IEEE, 2009.

[8] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman coding. *CoRR*, abs/1510.00149, 2015.

[9] G. Hegde, Siddhartha, N. Ramasamy, and N. Kapre. CaffePresso: An Optimized Library for Deep Learning on Embedded Accelerator-based platforms. In *Proc. CASES*, 2016.

[10] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and< 1MB model size. *CoRR*, abs/1602.07630, 2016.

[11] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *Proc. ICML*, pages 448–456, 2015.

[12] M. Kim and P. Smaragdis. Bitwise neural networks. *CoRR*, abs/1601.06071, 2016.

[13] A. Krizhevsky and G. Hinton. Learning multiple layers of features from tiny images. *Technical Report*, 2009.

[14] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proc. NIPS*, pages 1097–1105, 2012.

[15] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proc. of the IEEE*, 86(11):2278–2324, 1998.

[16] J. Misra and I. Saha. Artificial neural networks in hardware: A survey of two decades of progress. *Neurocomputing*, 74(1–3):239–255, 2010.

[17] S. Muralidharan, K. O'Brien, and C. Lalanne. A Semi-Automated Tool Flow for Roofline Anaylsis of OpenCL Kernels on Accelerators. *Proc. Workshop on H2RC*, 2015.

[18] Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. Reading digits in natural images with unsupervised feature learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2011.

[19] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr. Accelerating Binarized Neural Networks: Comparison of FPGA, CPU, GPU, and ASIC. In *Proc. ICFPT*, 2016.

[20] K. Ovtcharov, O. Ruwase, J.-Y. Kim, J. Fowers, K. Strauss, and E. Chung. Accelerating deep convolutional neural networks using specialized hardware, February 2015.

[21] J. Park and W. Sung. FPGA based implementation of deep neural networks using on-chip memory only. In *Proc. IEEE ICASSP*, pages 1011–1015. IEEE, 2016.

[22] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi. XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In *ECCV*, 2016.

[23] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *IJCV*, 115(3):211–252, 2015.

[24] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

[25] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[26] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. B. K. Vrudhula, J. Seo, and Y. Cao. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *Proc. ACM/SIGDA ISFPGA*, pages 16–25, 2016.

[27] W. Sung, S. Shin, and K. Hwang. Resiliency of deep neural networks under quantization. *CoRR*, abs/1511.06488, 2015.

[28] S. I. Venieris and C.-S. Bouganis. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In *Proc. IEEE FCCM*, pages 40–47. IEEE, 2016.

[29] T. Weyand, I. Kostrikov, and J. Philbin. Planet - photo geolocation with convolutional neural networks. *CoRR*, abs/1602.05314, 2016.

[30] S. Williams, A. Waterman, and D. A. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, 2009.

[31] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proc. ACM/SIGDA ISFPGA*, pages 161–170. ACM, 2015.

[32] S. Zhou, Z. Ni, X. Zhou, H. Wen, Y. Wu, and Y. Zou. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *CoRR*, abs/1606.06160, 2016.