Chameleon: A Generalized Reconfigurable Open-Source Architecture for Deep Neural Network Training

Mihailo Isakov, Alan Ehret, Michel Kinsy
Department of Electrical and Computer Engineering
Boston University
Boston, United States of America
{mihailo, ehretaj, mkinsy}@bu.edu

***Abstract*—We present Chameleon, an architecture for neural network training and inference design exploration on FPGAs. While there exists a great number of different network types and optimizations, it is not always clear how these differences impact the hardware implementation of neural networks. Chameleon is created with extensibility and experimentation in mind, supporting a number of activations, neuron types, and signal bitwidths. Furthermore, Chameleon is created to be modular, allowing designers to easily swap existing parts for improved ones, speeding up research iteration. While there exists a large number of inference architectures, we focus on speeding up training, as training time is the bottleneck for neural network architecture exploration. Chameleon therefore aims to help researchers better understand the bottlenecks in training deep neural networks and create models that circumvent these barriers. Finally, Chameleon is designed to be simple, without requiring a compiler or reconfiguration to function. This allows quick localized changes to the architecture and facilitates design exploration. We present synthesis results on an Altera Cyclone V SoC and show the design resource usage. We finish with an evaluation by training a network on the Wisconsin Breast Cancer dataset. The RTL and synthesis files for the architecture will be open-sourced upon publication at *http://ascslab.org/research/abc/chameleon/index.html*.**

***Index Terms*—neural networks, training, hardware, architecture, FPGA, RTL, design exploration, open-source**

I. Introduction

While there exists a number of high-performance inference and training architectures, neural network accelerator design exploration is still cumbersome for a variety of reasons. The majority of published papers do not open-source their designs, preventing researchers from effectively comparing different approaches and optimizations. Additionally, parameter exploration for new architectures is restricted. While there are several open-sourced designs out there, they are often limited to a certain network type [1], or even a single topology [2].

Other designs support a larger network design space, but are too complex and difficult to modify. DnnWeaver [3] generates deep neural network (DNN) architectures from a Caffe specification using a library of hand-written Verilog templates. While supporting a number of layer and neuron types, DnnWeaver forces the user to conform to its framework by modifying the generator and not the generated Verilog code. We aim to provide a simpler approach to testing designs by removing the design generation step from the workflow, and allowing direct modification of the underlying architecture.

Different architectures commonly utilize novel machine learning improvements to speed up their designs. For example, the FINN architecture [4] is a binary neural network accelerator that utilizes the fact that neural networks can have binary (-1 or +1) weights without sacrificing accuracy [5]. While achieving state-of-the-art results in inference latency, FINN does not permit larger design exploration.

We propose a generalized design, one that does not utilize all possible optimizations and keeps its options open, while permitting optimizations to be added later on through extensibility and modularity. Furthermore, we do not take advantage of possible hardware time-multiplexing options such as having only one matrix multiplier module used by both the forward and backward pass. Instead, we directly map the backpropagation algorithm to hardware. The reason for this is twofold: (1) it allows a designer to have a clearer picture of the area utilization of each part of the algorithm, and (2) allows independent optimization of both datapaths (forward and backward pass).

The majority of existing deep neural network and convolutional neural network (CNN) accelerators focus on inference in order to increase the battery life of mobile devices or allow computation on the edge. While inference is important, we choose to focus on training, as training time of large neural networks is the bottleneck in neural network research and exploration. Most neural networks are trained on GPUs, due to the parallel nature of neural networks. For that reason, Chameleon was built to focus on training. While GPUs may offer fast matrix multiplications and convolutions, we believe that certain neural network architectures (i.e. sparse or low-precision networks) can take advantage of the FPGA fabric, reconfigurability, and distributed resources, allowing certain network topologies to train faster.

With many papers out there focusing on optimizing processing elements [6], memory bandwidth and storage [7], locality [8], etc. we feel that a naive benchmark architecture is needed to compare different optimizations more equally.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

By building a transparent and direct implementation of the forward pass and backpropagation algorithms, we hope to gain a greater insight into the bottlenecks of neural network training and inference algorithms.

II. Related work

Open source. DnnWeaver [3] is an open-source DNN accelerator generator which converts high-level Caffe network specifications into synthesizable designs. DnnWeaver uses a library of optimized hand-written templates to generate designs, and supports both Xilinx and Altera families of chips. ZynqNet [2] is an open-source OpenCL network accelerator. It consists of the custom ZynqNet CNN topology, and an accelerator implemented for that specific network. FINN [4] is a binary neural network [5] accelerator with sub-microsecond latency for MNIST image classification. The design is open-sourced on Github.

Parametrizable. A significant number of FPGA CNN and DNN implementations support different layer types and activations [8]–[10], requiring the user to only reconfigure the design in order to change the network topology. In [11], the authors aim to bridge the gap between efficient but slow-to-design hand-written accelerators and inefficient HLS accelerators. By creating a set of efficient and modular CNN primitives, their compiler can adapt to specifics of CNN topologies and quickly produce accelerators. A similar neural network design automation tool is proposed in DeepBurning [12]. The authors show that it is very difficult to create a single generalized architecture that can provide good efficiency on a vast number of different types of neural networks. They present DeepBurning, which consists of an RTL-level accelerator generator and a supporting compiler. DeepBurning supports a large number of networks like conventional DNNs and CNNs, RNNs, NiN [13], Hopfield networks, etc.

Inference. The majority of published FPGA DNN accelerators focus on inference only [9]–[11], as it is assumed that the training is performed offline using GPUs. In NeuFlow [9], the authors implement a systolic array based neural network accelerator that can reconfigure specific networks or even layers. Their systolic cells can perform multiplications, convolutions, neuron activations and routing. Authors also develop a compiler to run Torch models on the accelerator. In [14], the authors implement a layer-multiplexing neural network with one physical layer. The network uses fixed-point values for activations, weights, and biases, and simulates different layers, one at a time. The size of the layers in the network is restricted by the number of physical neurons that are synthesized.

Training. Training neural networks requires significantly more compute power, on-chip storage and bandwidth than inference does. Early FPGA implementations were primarily limited by the amount of logic available, leading to multi-FGPA solutions [15] and stochastic computation solutions [6]. While some inference networks could still fit on a single FPGA, backpropagation would require different logic for inference, training, and updates, with some early solutions opting for time-multiplexing these stages and reconfiguring the FPGA for each stage [16]. In [17], the authors implement a backpropagation architecture which time-multiplexes layers, similar to the inference implementation from [14]. They implement only one physical layer of neurons, and the architecture uses that single layer to process all the network layers. The architecture is restricted by the available on-chip resources and has a hard limit on the maximum number of neurons in any layer. In [18] the authors test out how precision affects training, and create a systolic array based accelerator, taking advantage of the low bitwidth stochastic quantization. This approach is further explored in ZipML, where authors train networks with weights as low as two bits [19].

III. Training Neural Networks

Backpropagation is the most widely used method of training neural networks. Training neural networks typically consists of a forward pass, where an input is run through the network, and backwards pass, where the network is updated to better approximate the target output. We briefly illustrate the backpropagation procedure here.

Forward pass is typically performed as:
for an activation function f, input a^0, weight matrices W^1 to W^L, and biases b^1 to b^L, we calculate layer activations as:

$$z^x = W^x a^{x-1} + b^x \tag{1}$$

$$a_x = f(z_x) \tag{2}$$

Once we have performed a forward pass, we can compare the top layer activation a^L with the target activation t and calculate the error as $e = t - a^L$. From there, using the network cost function C, we can calculate the delta signals δ as:

$$\delta^L = \nabla C \odot f'(z^L) \tag{3}$$

$$\delta^{l-1} = ((W^l)^T \delta^l) \odot f'(z^{l-1}) \tag{4}$$

This procedure is the reason for the name backpropagation, as we are propagating deltas down the layers, starting from the top. The weights and biases can be updated from there as:

$$\frac{\partial C}{\partial w_{ij}^l} = a_j^{l-1}\delta_i^l, \quad \frac{\partial C}{\partial b_i^l} = \delta_i^l \tag{5}$$

IV. Architecture

Chameleon top level architecture consists of the forward pass module, the backwards pass module, the activation stack, and dataset module, as illustrated in Figure 1. Initially, we assume that the whole dataset is stored in DRAM, and we fetch samples and corresponding labels into on-chip BRAM. Since we can shuffle the dataset in advance, the prefetcher incrementally loads portions of the DRAM into BRAM cache. The cache module feeds the samples from the dataset to the forward pass module, and the sample labels to the backwards pass module.

The forward pass calculates the activations of all layers in the network and sends each layer's pre-activation value (i.e. the value before the activation function is applied) to the

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

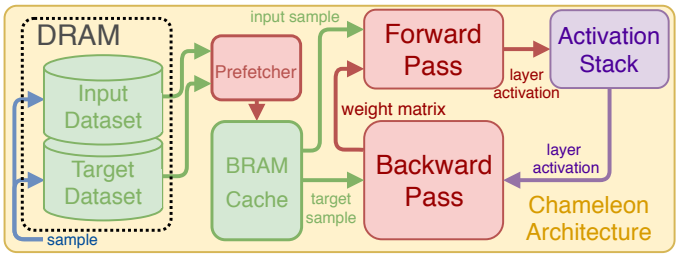

Fig. 1. Top level view of our proposed architecture.

activation stack module which stores them. This is important as the backward pass will require the values of both the activation function and its derivative to work, and we choose to calculate those on the fly rather than store them both. Once all layers are processed for a specific input, the backwards pass module reads activations from the module in the descending order, starting from the output layer down to the input layer. The backwards pass module learns by comparing the targets received from the dataset module, and the activations received from the stack module. From them it calculates the error signal for each output neuron and (1) updates it's weight matrices, which it feeds to the forward pass module, and (2) propagates the error signal to the lower layers, repeating the process. The architecture is fully pipelined, allowing independent weight updates and forward pass inference. While running inference and at the same time changing the weight matrix might lead to race conditions, our arbitration mechanism prevents this.

Additionally, there is the issue of stale reads and training on stale models. Here the design's forward pass may start working on some samples before the backward pass has finished integrating changes in the weight matrix. This may lead the backward pass to optimize a stale model, which has been shown to lead to a decreased accuracy. We disregard this effect for two reasons: (1) the staleness we see in the model can be measured in only a couple of samples, far lower than the reported staleness needed to hurt accuracy [20], and (2) this effect is similar to the momentum used in the optimizer [21], and can be removed with careful momentum tuning. Stale reads can therefore be ignored as long as a good momentum is picked.

A. Conventions used

All values in our design use a fixed-point format. We have opted for fixed-points instead of floating-points as there is enough research [22] to show that training on sufficiently precise fixed-points does not reduce accuracy. In the architecture, all signal bitwidths are defined by their integer width, and the fraction width. We chose to have a unified fraction bitwidth for the sake of simplicity, but are planning to remove this constraint in the future.

B. Dataset Cache

The cache module holds both inputs and targets to the network. It is implemented in BRAM, and loads its values from a prefetcher, which loads the values from DRAM. The inputs are accessed with the sample number which serves as the index. As the forward pass processes the samples, this counter is incremented, and wraps around when the end of the dataset is reached. We assume that the dataset is shuffled in advance.

C. Forward Pass Module

One of the main components of the architecture is the forward pass module, which is used for calculating the activations of all hidden and output layers in the network. This module accepts a vector of inputs and a matrix of weights, and outputs the activations of all the layers in the network, one layer at a time. The forward pass module is illustrated in Figure 2.

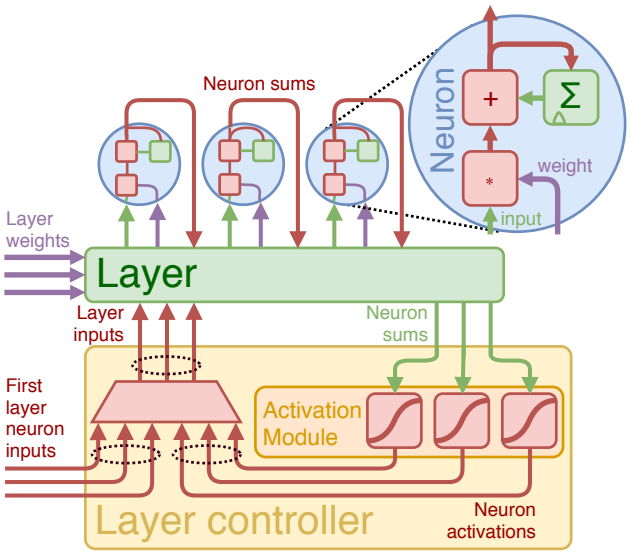

Fig. 2. Forward pass module architecture.

Initially, the forward pass requires the input layer's input values, and the first weight matrix. The module then calculates the activations of the first hidden or output layer. In the case that there are multiple layers in the network, the forward pass module uses it's output as it's activation input and accepts a new weight matrix (but no neuron inputs). The forward pass module continues until it calculates the activation of the final, output layer. It then accepts new input layer activations and starts from the beginning.

Our implementation time-multiplexes layers. We synthesize a fixed number of neurons, and process one layer at a time. The layer module contains a number of physical neurons, while the layer controller module applies activation functions to individual neuron results, and chooses which input to feed to the layer - a new input layer activation, or the previous layer's activation. If a certain layer has fewer neurons than the number of physical neurons we synthesized, the extra neurons are simply turned off. This number of neurons must be equal to or greater than the largest layer in the network, prohibiting training larger networks in this design.

Neuron Module: We opted for creating physical neurons instead of following the conventional approach of performing matrix-vector multiplications in order to further explore this design choice. Our neurons consist of simple multiply-accumulate (MAC) units which iterate through the inputs and

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

appropriate matrix rows. The neurons keep track of results and raise the overflow/underflow flag when appropriate. Neurons also accept an input value describing the number of neurons in the previous layer, in order to allow topologies where layers have different numbers of neurons. Furthermore, neurons can be deactivated for layers with smaller numbers of neurons. We choose to use overflows instead of saturating logic for (1) hardware simplicity, as verilog does not offer built-in saturation logic, and (2) because we want to be aware of saturation/overflows happening and hurting network performance. Figure 2 shows the zoomed-in schematic of a neuron.

Activation Function: We implement multiple activation functions, including the sigmoid activation, rectified linear units (ReLU), and leaky rectified linear units. The sigmoid activations are implemented as lookup tables (LUT) with a parametrized input and output bitwidth. The choice of input bitwidth b_i directly affects the number of elements in the lookup table, while the output bitwidth b_o determines the memory for each LUT element. The total memory needed for the lookup table is $2^{b_i} b_o$ bits. We notice that for the sigmoid activation, a majority of values in the lookup table is repeated for a sequence of inputs, pointing to a possible optimization for storing values. The ReLU activation function is far simpler, and we implement it as a combinational circuit.

We can see three sigmoid activation functions inside the layer controller module in Figure 2.

D. Activation Stack Module

The activation stack is a simple stack module which saves activations received from the forward pass module, and reads them out to the backwards module in a reversed order. The forward pass stores one activation at a time, while the backwards pass reads pairs of activations, one from the current layer being trained, and one from the layer below.

E. Backwards Pass Module

The backpropagation module is the most complex module of the design. It consists of three modules, the weight controller, the error calculator, and the error propagator, as shown in Figure 3. The error calculator determines the errors of the top, output layer, while the error propagator takes these errors and calculates the errors of the lower layers of the network. The weight controller is charged with applying the errors as updates to the weight matrices. Additionally, the weight controller stores and serves the weight matrices to the forward pass module discussed above. The whole backwards pass module tracks overflows which is useful for parameter tuning.

Error Calculator: Calculating the delta values in a neural network depends on the layer being processed. In case of the top layer L, for a target y, neuron sum z^L and activation $a^L = f(z^L)$, we calculate the error as:

$$\delta^L = (y - a^L) \odot f'(z^L) \tag{6}$$

In order to calculate the top layer error, we use the error calculator module, as seen in Figure 3. The error calculator accepts target values from the top level dataset module, along

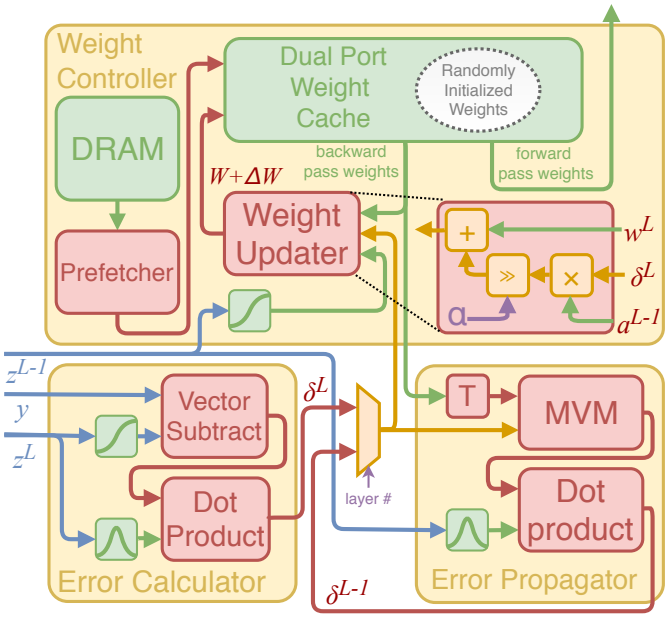

Fig. 3. Proposed backpropagation module architecture. The s-shaped and bell-shaped boxes are the sigmoid activation function and it's derivative. Both are implemented as lookup tables. The T block is the transpose, and the MVM block is the matrix-vector-multiplication module. In the weight updater module, the $\times$ box is the tensor product module, and the $>>$ symbol is the shift right operation. The shift operation applies the learning rate $alpha$ to the updates before they are added to the weights.

with the values calculated by the forward pass, and calculates the top layer delta value according to Eq. 6.

Error Propagator: In case of lower layers, we calculate the error δ^{L-1} using the error from the previous layer δ^L and the weight matrix w as:

$$\delta^{L-1} = w^T \delta^L \odot f'(z^{L-1}) \tag{7}$$

The error propagator module is used for calculating the errors for layers below the top one. It accepts previous (above) delta signals δ^L, the weight matrix w, and this (lower) layer's neuron sum z^{L-1}, and calculates δ^{L-1} according to Eq. 7.

For the first hidden layer below the output layer, the error propagator module will take the error calculator's error signals. For the layers below it however, it will use it's own outputs, which can be seen as the multiplexer between the error calculator and propagator in Figure 3. The error calculator sends out the delta signal to the weight controller, which uses it to update the weight matrices of the network.

Weight Controller: The weight controller serves two purposes: (1) it updates the weight matrices as new errors come in, and (2) serves the weight matrices to both the forward and backward pass modules. Both reading and writing must happen independently, but we do not allow reads during writes. Initially, the weight controller starts with randomized weight matrices with values close to zero. Since we are using fixed point values with n bits for the fraction, we make sure that we pick random numbers in the $[-k, k]$ range, where $k << 2^n$. This is essential to prevent early saturation of neurons, which can prohibit learning.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

While serving weight matrices, the weight controller is also charged with updating them as new errors come in. The weight updater module inside the weight controller serves to calculate the updated weights according to the incoming error from the error calculator or propagator modules. For a cost function C and a weight w_{ij}^L in the L-th layer with coordinates i and j, we are interested in finding $\frac{\partial C}{\partial w}$. We calculate the weight update Δw_{ij}^L as:

$$\Delta w_{ij}^L = \frac{\partial C}{\partial w_{ij}^L} = a_k^{L-1}\delta_j^L \tag{8}$$

This product is calculated in the cross product module. The module is generalized and accepts two vectors of parametrizable lengths m and n, and returns a matrix of size $m \times n$. The tensor product calculates the matrix in tiles, and consists of T_m and T_n multipliers, where T_m and T_n are parametrizable tile sizes. The updates are further multiplied with the learning rate. Researchers rarely require high precision in choosing a learning rate, often increasing it or decreasing it whole orders of magnitude. We implement the learning rate as a shift operator, operating on individual matrix elements. We apply the learning rate after the $a_k^{L-1}\delta_j^L$ in order to decrease numerical errors. Finally, the updates are applied to the weight matrix by adding them together in the vector adder module. We have parametrized the module with a variable number of adders, affecting throughput and latency.

V. Extensibility and Modularity

A goal of Chameleon is painless extensibility. We are exploring using high level synthesis (HLS) languages such as MyHDL [23] for two reasons: (1) as a powerful hardware verification language (HVL) that can tap into the vast amount of Python math and neural network libraries for testing purposes, and (2) as a mocking framework [24], quickly establishing a non-synthesizable golden model that can be gradually "hardened".

Having a suite of high-level parametrizable tests with powerful assertion logic that can be run without the designers interference is common in all branches of software development. Hardware testing has fewer open-source frameworks and many designers rely on manually checking waveforms and testbenches. Furthermore, gradient descent is notoriously difficult to debug, as small errors, such as off-by-one mistakes, may not prevent a system from learning, but only slightly affect its performance. Having a robust and user-friendly set of architecture-independent tests allows the designer to proceed with certainty and focus on improving performance rather than maintaining correctness.

We also propose the use of mocking in our neural net designs. According to Fowler [24], mocks are "objects preprogrammed with expectations which form a specification of the calls they are expected to receive". We extend this definition to mean: mocks are software modules that have correct behaviours but are not synthesizable and have arbitrarily defined latencies between inputs and outputs. We use mocks as a way to quickly define a correct, golden, latency-insensitive design, which we will later "harden" by converting HLS mocks into verilog modules with the same behavior but different latency requirements. With a full test suite and a golden model, designers can quickly make localized changes, test functionality, and observe the throughput and area changes.

A. Implementing Recurrent Layers

To show the proposed extensibility, we implement recurrent neural network (RNN) layers in Chameleon. Unlike the fully-connected networks, recurrent neural networks possess memory, i.e., their outputs are dependent on both the inputs and their state. The inputs to the network at each sample are not independent, but are a temporal sequence such as sound, video, or stock market data. Instead of serving the whole sequence at the same time, which would require a very large input size, the sequence is fed to the network one sample at a time.

In RNNs, each neuron stores a scalar value and we mark all these values as s. Given the activation function f, weight matrices W_i, W_s, W_o, inputs x_t and states s_t at timestep t, we can write the new state and output equations as:

$$s_t = f(W_x x_t + W_s s_{t-1}) \tag{9}$$

$$o_t = W_o s_t \tag{10}$$

In Figure 4 we see a recurrent neural network unfolded across time. This way, we can treat previous states as inputs to the network, and implement the RNN similarly as we did with fully-connected networks. Here the inputs i_t and the states s_{t-1} need to be concatenated, as well as the weight matrices W_x and W_s.

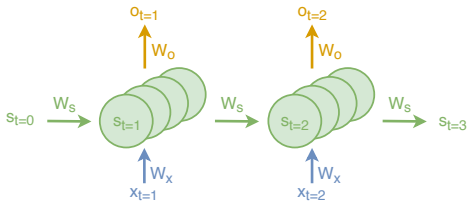

Fig. 4. A recurrent neural network unfolded across timesteps.

When a fully-connected layer is stacked on top of the recurrent layer, we see that the output weights become unnecessary, as the fully-connected layer will again multiply the RNN outputs with it's own weights. We rewrite equation 10 as $o_t = s_t$. This further simplifies our hardware implementation, as the user can choose to add a fully-connected layer if needed.

In Figure 5 we show the upgraded forward pass module architecture that enables processing both fully-connected and recurrent layers. The architecture has two changes: (1) we add a *Neuron Memory* register that stores the activations of RNN layers, and (2) we replace the multiplexer choosing between the outside inputs and layer outputs with the *Concat / Mux* module. The *neuron states* register stores the states of recurrent layers between two full forward passes. The *Concat / Mux* module is tasked with picking between forwarding (1) outside inputs when processing the first layer of the network,

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

(2) previous layer's outputs when processing a hidden layer, (3) concatenating the outside inputs with neuron states when the first layer of the network is recurrent, or (4) concatenating the previous layer's outputs with neuron states when a hidden layer is recurrent.

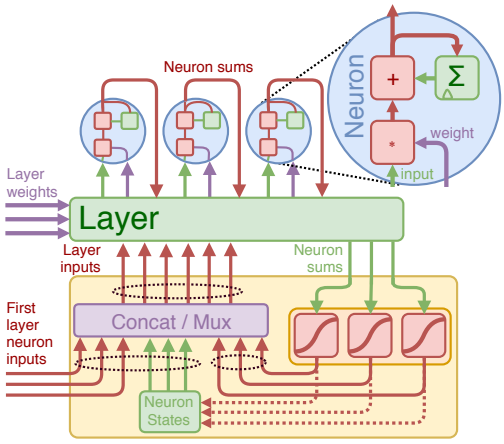

Fig. 5. New forward pass architecture enabling processing both fully-connected and recurrent layers.

Training recurrent neural networks is complicated by the fact that gradients must be propagated not only down the layers, but also back through time. We leave the implementation of BPTT for future work.

VI. Evaluation

We present synthesis results on an Altera Cyclone V SoC and show the design resource usage. We compare two designs with different numbers of neurons and tiling parameters in Table I.

Parameters: The majority of the building blocks of this design are parametrizable. The parameters were determined through iterative latency measurements described in the previous subsection V.A. The majority of these parameters are shared to the smallest building blocks but we omit them to increase legibility.

Module name	Parameter	Design 1	Design 2
Top module	Neuron #	4	8
Top module	Activation bitwidth	9	9
Top module	Weight bitwidth	16	16
Top module	Delta bitwidth	9	9
Top module	Fraction bitwidth	8	8
Top module	Learning rate shift	4	4
Top module	Activation function	sigmoid	sigmoid
Error propagator	Tiling row	2	4
Error propagator	Tiling column	2	4
Vector subtract	Tiling	1	2
Vector dot	Tiling	1	2

TABLE I
Module parameters

Resources: We analyze the adaptive logic module (ALM), register, BRAM, and DSP block usage of different modules of the design. First, in Table II we present the memory and logic consumed by each of the top modules, and go more in depth for each of them. We notice that the amount of registers used by the design grows quadratically with the number of neurons. We later identified this to be the result of our inefficient weight controller implementation, which will be updated in the next revision. The updated weight controller will grow linearly with the number of neurons in the network.

	Design 1	Design 2
ALM Usage	3,030	10,453
Registers	5,759	29,123
BRAM Bits	18,624	37,248
DSP Blocks	13	32

TABLE II
Resource usage

Accuracy: We test out the architecture on the Wisconsin Breast Cancer dataset. This dataset contains 569 samples with 30 dimensions each, 357 labeled as benign and 212 as malignant. To confirm that the architecture is working properly, we write a Python script for training fixed-point networks, and confirm that both the design and the script are behaving identically. This is not far-fetched, as the network only performs simple operations such as matrix-vector multiplications, vector additions, subtractions, multiplications, shifts, and lookup table based activation functions. Since the values for our hardware LUT-based activations are derived from the same Python script, we see the same values in both designs. We show the training accuracy of a 30 neuron network with three different bitwidth configurations: (1) 4 integer bits and 16 fractional bits, (2) 4 integer bits with 12 fractional bits, and (3) 4 integer bits with 8 fractional bits. The smoothed out accuracy over 10000 training samples is shown in Figure 6. Notice that there is no added benefit in training with 16 fractional bits over 12, but the accuracy significantly degrades when using only 8 fractional bits. This is in line with the findings in [22].

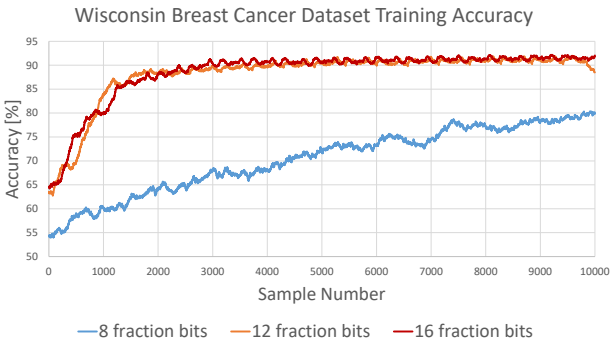

Fig. 6. Chameleon accuracy during training on the Wisconsin Breast Cancer dataset.

VII. Conclusion

In this work we present Chameleon - a hardware architecture for neural network training focusing on extensibility, modularity, and simplicity. We describe the implementation in detail, and offer our vision of how open-sourcing such a design will allow further experimentation, exploration, and modular refinement. We present the synthesis results for two designs and show the achieved accuracy on the Wisconsin Breast Cancer dataset.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

REFERENCES

[1] D. Wang, K. Xu and D. Jiang, "PipeCNN: An OpenCL-based open-source FPGA accelerator for convolution neural networks," 2017 International Conference on Field Programmable Technology (ICFPT), Melbourne, VIC, 2017, pp. 279-282.

[2] David Gschwend. Zynqnet: An fpga-accelerated embedded convolutional neural network. Masters thesis, Swiss Federal Institute of Technology Zurich (ETH-Zurich), 2016.

[3] H. Sharma et al., "From high-level deep neural models to FPGAs," 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, 2016, pp. 1-12.

[4] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. 2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17). ACM, New York, NY, USA, 65-74.

[5] Courbariaux, Matthieu and Yoshua Bengio. BinaryNet: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1. CoRR abs/1602.02830 (2016): n. pag.

[6] K. Kollmann, K. -. Riemschneider and H. C. Zeidler, "On-chip backpropagation training using parallel stochastic bit streams," Proceedings of Fifth International Conference on Microelectronics for Neural Networks, Lausanne, Switzerland, 1996, pp. 149-156.

[7] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, Huazhong Yang, and William (Bill) J. Dally. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17). ACM, New York, NY, USA, 75-84.

[8] DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. SIGPLAN Not. 49, 4 (February 2014), 269-284.

[9] P. Pham, D. Jelaca, C. Farabet, B. Martini, Y. LeCun and E. Culurciello, "NeuFlow: Dataflow vision processing system-on-a-chip," 2012 IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS), Boise, ID, 2012, pp. 1044-1047.

[10] S. Han et al., "EIE: Efficient Inference Engine on Compressed Deep Neural Network," 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA), Seoul, 2016, pp. 243-254.

[11] Yufei Ma, N. Suda, Yu Cao, J. Seo and S. Vrudhula, "Scalable and modularized RTL compilation of Convolutional Neural Networks onto FPGA," 2016 26th International Conference on Field Programmable Logic and Applications (FPL), Lausanne, 2016, pp. 1-8.

[12] Y. Wang, J. Xu, Y. Han, H. Li and X. Li, "DeepBurning: Automatic generation of FPGA-based learning accelerators for the Neural Network family," 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, 2016, pp. 1-6.

[13] M. Lin, Q. Chen, and S. Yan, Network in network, CoRR, vol. abs/1312.4400, 2013. [Online]. Available: http://arxiv.org/abs/1312.4400

[14] S. Himavathi, D. Anitha and A. Muthuramalingam, "Feedforward Neural Network Implementation in FPGA Using Layer Multiplexing for Effective Resource Utilization," in IEEE Transactions on Neural Networks, vol. 18, no. 3, pp. 880-888, May 2007.

[15] C. E. Cox and W. E. Blanz, "GANGLION-a fast field-programmable gate array implementation of a connectionist classifier," in IEEE Journal of Solid-State Circuits, vol. 27, no. 3, pp. 288-299, March 1992.

[16] C. E. Cox and W. E. Blanz, "GANGLION-a fast field-programmable gate array implementation of a connectionist classifier," in IEEE Journal of Solid-State Circuits, vol. 27, no. 3, pp. 288-299, March 1992.

[17] Ortega-Zamorano F., Jerez J.M., Gmez I., Franco L. (2016) Deep Neural Network Architecture Implementation on FPGAs Using a Layer Multiplexing Scheme. In: Omatu S. et al. (eds) Distributed Computing and Artificial Intelligence, 13th International Conference. Advances in Intelligent Systems and Computing, vol 474. Springer, Cham

[18] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37 (ICML'15), Francis Bach and David Blei (Eds.), Vol. 37. JMLR.org 1737-1746.

[19] Zhang, Hantian & Kara, Kaan & Li, Jerry & Alistarh, Dan & Liu, Ji & Zhang, Ce. (2016). ZipML: An End-to-end Bitwise Framework for Dense Generalized Linear Models.

[20] Jeffrey Dean, Greg S. Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Quoc V. Le, Mark Z. Mao, Marc'Aurelio Ranzato, Andrew Senior, Paul Tucker, Ke Yang, and Andrew Y. Ng. 2012. Large scale distributed deep networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12), F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger (Eds.), Vol. 1. Curran Associates Inc., USA, 1223-1231.

[21] Mitliagkas, I., Zhang, C., Hadjis, S., & R, C. (2016). Asynchrony begets momentum, with an application to deep learning. 2016 54th Annual Allerton Conference on Communication, Control, and Computing (Allerton), 997-1004.

[22] Courbariaux, Matthieu & Bengio, Y & David, Jean-Pierre. (2015). Training deep neural networks with low precision multiplications.

[23] Decaluwe, Jan. (2004). MyHDL: a python-based hardware description language. Linux Journal. 2004. 5.

[24] M. Fowler, Mocks Arent Stubs, https://martinfowler.com/articles/mocksArentStubs.html, 2007, [Online; accessed 24-May-2018].

978-1-5386-5989-2/18/$31.00 ©2018 IEEE