Training Deep Neural Networks in Low-Precision with High Accuracy using FPGAs

Sean Fox*, Julian Faraone*, David Boland*, Kees Vissers†, Philip H.W. Leong*
*School of Electrical and Information Engineering, The University of Sydney, Australia 2006
†Xilinx Inc, San Jose, USA
Email:{sean.fox, julian.faraone, david.boland, philip.leong}@sydney.edu.au, {kees.vissers}@xilinx.com

***Abstract*—Low-precision training for Deep Neural Networks (DNN) has recently become a viable alternative to standard full-precision algorithms. Crucially, low-precision computation reduces both memory usage and computational cost, providing more scalability for Field Programmable Gate Arrays (FPGAs) with limited on-chip memory. In this paper, we describe and test a prototype training accelerator for Zynq All Programmable System on Chip (APSoC) devices using predominantly 8-bit integer numbers. Block floating-point quantisation and stochastic weight averaging techniques are applied during training to avoid any degradation in accuracy. Results of an implementation reveal memory savings and $17\times$ speed-ups over processor only systems on several training tasks including the MNIST and CIFAR10 benchmarks, and online radio-frequency anomaly detection. Moreover, we propose modifications to the stochastic weight averaging low-precision (SWALP) algorithm to achieve a 0.5% accuracy improvement for the abovementioned benchmarks with results within 0.1% of floating-point. We suggest that both inference and training can be deployed in the same package for stand-alone embedded applications.**

I. Introduction

Training deep neural networks (DNNs) requires large amounts of memory and computation, and until now, has almost exclusively been performed with full-precision floating-point (FP32) numbers and many Graphics Processing Units (GPUs). As a result, it is typically difficult to move training into edge devices where there are stricter memory, power and computational constraints (eg. mobile phones, radios, video cameras).

The primary workload in DNN training is general matrix multiplication (GEMM), which depending on the underlying hardware, could be either memory or compute bound. To overcome compute bound problems we want hardware with more logic and higher compute density, and for memory bound problems it's desirable to fit at least one of the matrices in fast on-chip memory. Given that DNN model sizes are steadily increasing, and larger models produce larger matrices, more efficient accelerators are required to satisfy these demands. Low-precision computation offers arguably the best solution, since fewer bits increases the compute density and reduces memory consumption.

The majority of work on low-precision DNNs has focused on the inference part only, to speed-up the run-time. These low-precision network representations are obtained by quantising the weights and activations during training, and can normally achieve FP32 accuracy [1] [2]. In contrast, training with low-precision generally produces worse accuracy than training with FP32. The problem arises in computing gradients and accumulating the weight updates specifically [3] [4]. This is because the representable range of low-precision fixed-point numbers is unsuitably small in most cases. A number of techniques have been developed to address this issue, such as wider accumulators [4], alternate number representations [5], overflow prediction [6], variance reduction and bit centering [7], and weight averaging [8]. In fact in SWALP [8], the authors combine block floating-point quantisation with weight averaging, establishing the first training algorithm capable of achieving floating-point accuracy with only 8-bit arithmetic. Our work is most similar to SWALP, except we compute the weight updates in full-precision.

Although low-precision training techniques like block floating-point quantisation and weight averaging have been described previously, to the best of our knowledge, this is the first paper that demonstrates training with these techniques on Field Programmable Gate Arrays (FPGAs). Specifically, the contributions of this work are:

- The first FPGA implementation of a CNN training technique, which achieves accuracy of floating-point using mostly 8-bit arithmetic.
- A modified version of SWALP which achieves improved accuracy, and a flexible hardware/software partitioning scheme.
- Quantitative analysis of the performance of the resulting system using the CIFAR10 and MNIST benchmarks.
- The first demonstration of a real-time, radio frequency anomaly detector with FPGA-accelerated training.

Our implementation targets the full range of Xilinx Zynq All Programmable System on Chip (APSoC) devices. Zynq comprises many different processing elements in the same package, offering real-time processing and massive integration capabilities for embedded applications. For instance, the ZCU111 board features an ARM processor and FPGA together with special multi-gigabit components for RF signal analysis. Our training accelerator is designed especially for such systems, where we want to combine real-time integration with on-chip training. We refer to these applications as *stand-alone* because they can necessarily be deployed in the field without host communication. In Section V, we demonstrate this use case by training a DNN-based auto-encoder for anomaly detection

in RF networks.

II. Background

In this section, we present the required background and theory on DNN training, Low-Precision Stochastic Gradient Descent (LP-SGD), and Stochastic Weight Averaging.

A. DNN Training

In simple terms, DNNs are powerful multi-layered feature extractors. They transform raw inputs via a sequence of layers into simpler yet more abstract features with linear mappings at the output. In general, computation layers take the form:

$$f(\mathbf{x}) = g(\mathbf{W}^T\mathbf{x} + \mathbf{b}) \tag{1}$$

where $\mathbf{x} \in \mathbb{R}^d$ is the input to the layer, $f(\mathbf{x}) \in \mathbb{R}^k$ is the output of the layer, $\mathbf{W}^{d\times k}$ and $\mathbf{b}^k$ are the trainable weights and bias parameters, and g is a non-linear activation function applied elementwise (eg. the *relu* function, $g(x) = max(0, x)$). Importantly, this simple formulation describes the fundamental computation at the heart of all layers, convolution layers included. For example, a $k_w \times k_h$ convolution operator can be reformulated as standard matrix multiplication by rearranging $k_w \times k_h$ blocks in the input into columns of a matrix. This transform is known as *im2col*, while the reverse transform is called *col2im*. Both transforms are used to implement the fastest known convolution layers on GPUs and CPUs [9].

Training is performed by first establishing and then minimising an objective of the form:

$$H(\mathbf{W}) = \frac{1}{m}\sum_{i=1}^{m} h_i(\mathbf{W}) \tag{2}$$

Here, $h(\mathbf{W})$ is a loss function (eg. mean squared error, cross-entropy etc.) and m is the number of training batches. For ease of notation, we have defined the loss as a function of the weights only.

Stochastic gradient descent (SGD) is commonly used to minimise the objective. On each iteration, SGD will select a random batch from the training set, corresponding to a function $\tilde{h}$ from $\{h_1, h_2, ..., h(m\}$, and update the weights in the direction of steepest descent, according to the following equation and learning rate α.

$$\mathbf{W}^{t+1} = \mathbf{W}^t - \alpha\nabla\tilde{h}(\mathbf{W}^t) \tag{3}$$

Here, $\nabla\tilde{h}(\mathbf{W}^t)$ refers to the weight gradient or alternatively the derivative of the loss with respect to the weight $\frac{\partial\tilde{h}}{\partial\mathbf{W}}$. This needs to be computed for every weight at every layer in the network, and requires application of the chain rule to propagate the loss back through the network. This is called backpropagation and the back propagated loss at each layer is the activation gradient (or deltas) denoted $\boldsymbol{\delta}_l$. If $z(\mathbf{x}) = \mathbf{W}^T\mathbf{x}$, then at each computation layer we must compute the forward path by Equation (1) and the backward path by Equations (4) and (5), for the deltas and weight updates respectively.

$$\boldsymbol{\delta}_{l-1} = \frac{\partial z}{\partial \mathbf{x}}\frac{\partial g}{\partial z}\frac{\partial \boldsymbol{\delta}_l}{\partial g} \tag{4}$$

$$\nabla\tilde{h}(\mathbf{W}) = \frac{\partial z}{\partial \mathbf{W}}\frac{\partial g}{\partial z}\frac{\partial \boldsymbol{\delta}_l}{\partial g} \tag{5}$$

Since $\frac{\partial z}{\partial \mathbf{x}} = \mathbf{W}$ and $\frac{\partial z}{\partial \mathbf{W}} = \mathbf{x}$, then both equations reduce to matrix multiplication by $\tilde{\boldsymbol{\delta}}_l = \frac{\partial g}{\partial z}\frac{\partial \boldsymbol{\delta}_l}{\partial g}$.

In summary: DNN training introduces two matrix multiplications at each computation layer, and furthermore, the input batch at each layer of the forward path must be saved to calculate $\nabla\tilde{h}(\mathbf{W})$ in the backward path. Low-precision numbers can be used to reduce these computation and memory costs.

B. Quantisation

To train networks in low-precision a quantisation function $Q(.)$ is required to convert real-valued weights, activations, and gradients into their rounded versions. A number of quantisation methods have been suggested in the literature [10], [11], [3], [12]. This paper will focus on block floating-point quantisation with stochastic rounding, as implemented in [8].

In block floating-point (BFP), all numbers within a block share the same exponent, which is allowed to vary during training. The exponents are calculated at regular intervals during training and should be set to the largest exponent in a block to avoid overflow [13][5]. For training convolution layers where the inputs are multi-dimensional arrays, a block could represent a batch, an image, or even just one channel. The finer the granularity, the more exponents that must be saved [8].

In stochastic rounding, numbers are rounded up or down at random such that the expected rounding error is zero, i.e. $\mathbb{E}[Q(x)] = x$. If I bits are allocated to represent a quantised number and the exponent for that block is E, then the number of fractional bits is $F = -E + (I - 2)$ and the smallest representable number is $\epsilon = 2^{-F}$ [10]. Let $\lfloor x \rfloor$ be the largest integer multiple of ϵ which is less than or equal to the real number x, then the quantisation function for x is:

$$Q(x, \langle I, E\rangle) = \begin{cases} \lfloor x \rfloor & \text{with prob.} \quad 1 - \dfrac{(x - \lfloor x \rfloor)}{\epsilon} \\ \\ \lfloor x \rfloor + \epsilon & \text{with prob.} \quad \dfrac{x - \lfloor x \rfloor}{\epsilon} \end{cases}, \tag{6}$$

C. Low-precision SGD

It was shown in [4] that low-precision SGD converges to a solution without losing any full-precision accuracy. The authors determined that weights, activations and deltas can be quantised to 8 bits only, while 16 bits are needed for the gradient accumulators in the weight update, i.e. $\nabla\tilde{h}(\mathbf{W}^t)$. Basically, if the bit width of the accumulated gradients is too small, and the learning rate is also small, then the weight update will be eliminated by quantisation causing training to stagnate well before the full-precision version. This is the root cause of accuracy degradation in low-precision models. In our work, the gradient accumulation for the weights is computed in full-precision on an ARM processor.

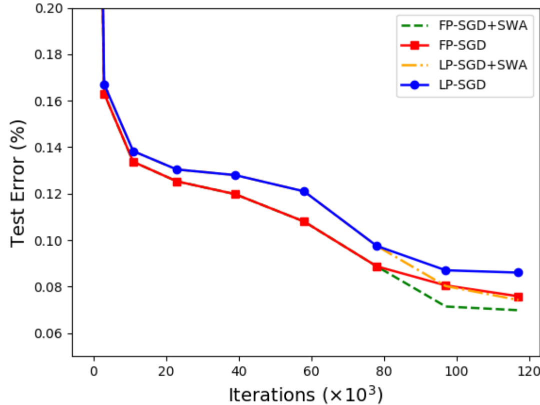

Fig. 1. Empirical results of stochastic weight averaging (SWA) with full-precision (FP) and low-precision (LP) SGD

D. Stochastic Weight Averaging for LP-SGD (SWALP)

In SWALP [8], the authors showed that simply averaging SGD iterates of the weights with a higher learning rate can recover quantisation errors and produce better low-precision training models, even when all numbers are 8 bits (including gradient accumulators). This technique is known as stochastic weight averaging (SWA) [14]. Figure 1 illustrates an example on the CIFAR10 image classification dataset. For the first $80k$ iterations standard SGD and a decaying learning rate is applied, and in the last $40k$ iterations the learning rate is modified, held constant, and the weights are averaged every epoch (approximately 400 iterations). The averaged weights are not used directly in training but rather represent the final trained model. This technique is observed to work relatively better for low-precision models. This could be because averaging cancels out errors in weights rounded up with those rounded down during quantisation. The main limitation from an implementation perspective is that a copy of the low-precision weights must be stored in high precision for the moving average. For PCIe-based accelerator cards, the high precision copy could be stored in slower host memory and the moving average calculated on the host every few hundred iterations by transferring the low-precision weights over the PCIe bus. This way, the high precision copy doesn't consume any of the fast DDR memory close to the accelerator. In embedded devices like Zynq, there is no host and the fast DDR memory is shared between the FPGA and ARM processor. So even though the high precision copy is not used regularly, it will ordinarily be saved with memory that could otherwise be used for more low-precision weights and supporting larger models.

E. Related Work

The vast majority of work on low-precision has targeted hardware acceleration for DNN inference only [15]. Examples include FINN [16], Eyeriss [17], ESE [18], RebNet [19]. In contrast, there are relatively few examples for training DNNs on FPGAs.

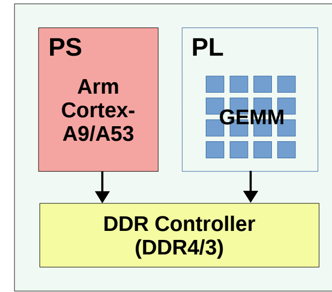

Fig. 2. High-level system overview

Early work considered the case of backpropagation through just one neuron [20], and then one multi-layer perceptron (MLP) layer [21]. Small MLP networks are also considered in [22], except training is performed with 16-bits via weight perturbations. Such a technique can reduce area by $10\times$ over backpropagation when the network can be fully pipelined. FPDeep [23] is a design framework for mapping DNN training across multiple FPGAs in a cluster. They show $3.4\times$ better energy efficiency is possible training 16-bit AlexNet, when the computation is distributed across 15 FPGAs in a pipelined manner. The authors in [5] describe an FPGA prototype which is similar in the sense that all weights and activations are stored on-chip. In general, the above mentioned works are only relevant to single FPGA implementations if the networks are very small. F-CNN [24] is more scalable and uses run-time reconfiguration to implement the different kernels used during training. The computation is based on 32-bit floating-point arithmetic, although 8-bits is now sufficient. DarkFPGA [25] is a new accelerator for 8-bit training, and is particularly optimised for convolution layers. The authors introduce several optimisations, including a new data pattern and tiling strategy, and demonstrate comparable performance with a GPU and better energy efficiency for training CIFAR10. Our work is similar in that we have also designed an accelerator for 8-bit matrix multiplication, but is sufficiently different in that we've focused on techniques for high accuracy training and reported a real application of this technology.

III. Low-precision Training Accelerator

In this section, we introduce our Low-Precision DNN training accelerator for Zynq SoC and MPSoC devices. Figure 2 provides a simple illustration of the three main components of the Zynq-based system, namely the programmable logic (PL), processing system (PS), and high bandwidth DRAM interface. We offload all GEMM instructions (which are 8-bit only) to the PL, and use the ARM processor to compute the rest of the network in floating-point. Since the majority of computation in any DNN involve GEMMs, we can achieve significant speed ups in training times over a processor-only system, while not losing any generality in terms of the type of networks we support. Our only limitation is memory, both in

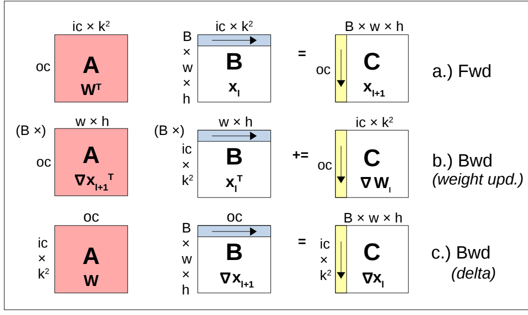

Fig. 3. Array shapes and sizes for convolution layers: A*B=C. Forward (top), backward weight gradient (middle) and backward activation gradient (bottom)

the PL and DRAM. Our designs were implemented using Xilinx SDSoC 2018.3, a predominantly software-based development flow which abstracts away much of the low level complexities associated with moving data between the PL and PS.

A. Software Overview

Our project is forked from Darknet [26] with added support for stochastic weight averaging (SWA), low-precision convolution and fully-connected layers, contiguous memory allocation on Zynq, and compilation directives for SDSoC-based FPGA acceleration. Darknet is an open source framework for DNNs written in C. It provides the backend for TinyYolo [27] and contains fast implementations for many different layers, activation functions, and DNN utility functions. We use a PYNQ v2.4 image to boot our boards [28]. Although we prefer C/C++ binaries over Python, PYNQ runs desktop Ubuntu offering a highly productive development environment for all users, not just Python ones. All our software (including bitstreams) can be installed from our repository and compiled on the board. Our repository is available online at www.github.com/sfox14/darknet-zynq.

1) Convolution: In this paper we have focused on training for convolution layers specifically. The inputs are typically images with three dimensions (w, h, ic), corresponding to the width, height and number of channels. The outputs are computed by applying a convolution kernel to the image with typically four dimensions (k_w, k_h, ic, oc), which refer to the kernel width and height, and number of input and output channels. The convolution kernel computes dot products over local regions of the input, moving over the input with an associated stride, and forming outputs with shape (w', h', oc). As mentioned previously, convolution layers can be reformulated with GEMMs by the *im2col* and *col2im* transforms. Figure 3 shows the mapping in detail for forward and backward paths. Please note that our accelerator computes convolution in channel last order, meaning the input image $\mathbf{x}$ is streamed from contiguous memory along the channel axis. For computing the weight update in the backward path, $\nabla \mathbf{x}_{l+1}$ and $\mathbf{x}_l$ must therefore be transposed in software, as shown in Figure 3.

Algorithm 1 describes the computation and hardware/software partitioning for the quantised convolution layer.

Algorithm 1: Convolution Layer

Define: layer l; time t; 8-bit weights $\bar{\boldsymbol{W}}_l^t$; input activations $\boldsymbol{x}_l^t$; deltas $\nabla \boldsymbol{x}_l^t$; weight updates $\nabla \boldsymbol{W}_l^t$; quantisation functions Q_w, Q_a, Q_e; quantisation scaling coefficients qw, $\boldsymbol{qa}$, $\boldsymbol{qe}$; gemm inputs $\boldsymbol{A}$, $\boldsymbol{B}$; gemm output $\boldsymbol{C}$; batch size K;

1. Forward:

Software:

1. $\bar{\boldsymbol{x}}_l^t, \boldsymbol{qa} = Q_a(\boldsymbol{x}_l^t)$; $\boldsymbol{B} = im2col(\bar{\boldsymbol{x}}_l^t)$;

Hardware:

2. $\boldsymbol{A} = (\bar{\boldsymbol{W}}_l^t)^{\boldsymbol{T}}$;
3. $\boldsymbol{C} = tofloat(gemm(\boldsymbol{A}, \boldsymbol{B}), qw, \boldsymbol{qa})$;

Software:

4. $\boldsymbol{x}_{l+1}^t = \boldsymbol{C}$;

2. Backward:

Software:

5. $\nabla \bar{\boldsymbol{x}}_{l+1}^t, \boldsymbol{qe} = Q_e(\nabla \boldsymbol{x}_l^t)$; $\boldsymbol{tmp} = im2col(\bar{\boldsymbol{x}}_l^{t\,\boldsymbol{T}})$;
6. **for** $i = 1, 2, \ldots, K$ **do**

   Hardware:

7. $\quad \boldsymbol{A} = \nabla \bar{\boldsymbol{x}}_l^t(i)^{\boldsymbol{T}}$; $\boldsymbol{B} = \boldsymbol{tmp}(i)$;
8. $\quad \boldsymbol{C} = tofloat(gemm(\boldsymbol{A}, \boldsymbol{B}), \boldsymbol{qe}, \boldsymbol{qa})$;

   Software:

9. $\quad \nabla \boldsymbol{W}_l^t \mathrel{+}= \boldsymbol{C}$;
10. **end**

Hardware:

11. $\boldsymbol{A} = \bar{\boldsymbol{W}}_l^t$; $\boldsymbol{B} = \nabla \bar{\boldsymbol{x}}_{l+1}^t$;
12. $\boldsymbol{C} = tofloat(gemm(\boldsymbol{A}, \boldsymbol{B}), qw, \boldsymbol{qe})$;

Software:

13. $\nabla \boldsymbol{x}_l^t = col2im(\boldsymbol{C})$;

The quantisation, *im2col* and *col2im* functions are computed in software on the ARM processor, while the GEMM (8-bit fixed-point) and fixed to float recasting function are offloaded to the FPGA. Each layer must store it's own set of 8-bit weights $\bar{\boldsymbol{W}}_l$, 8-bit input activations $\bar{\boldsymbol{x}}_l$, and FP32 quantisation scaling coefficients $(qw, \boldsymbol{qa}, \boldsymbol{qe})$. Given that SGD operates over batches of inputs, the batch size scales the size of $\bar{\boldsymbol{x}}_l$ and subsequently the memory required for the entire network.

Fully-connected layers are implemented similarly to convolution layers except *im2col* and *col2im* are not required.

2) Block floating-point Quantisation: The inputs to the GEMM are quantised to 8 bits block floating-point. We save one exponent for the weights and b exponents for the activations and deltas, where b is the batch size. This is similar to the *big-block* and *small-block* designs in SWALP [8]. The exponents are updated every iteration by setting the exponent to the maximum exponent within the block. This is the safest way of avoiding overflow but also the most inefficient. It would be worth exploring less regular exponent update algorithms as well as more fine-grained block designs, but we have deferred this for future work. The exponents are saved as floating-point scaling factors (i.e. $a = 2^E$), and are used to convert the fixed-point output from the GEMM back into a floating-point number.

3) Memory Usage: Table II shows a comparison of memory usage for different batch sizes. This example considers training for CIFAR10 on the VGG16 network. The *malloc* columns refer to non-contiguous memory while *cma* columns denote

TABLE I
VGG16 NETWORK ARCHITECTURE (CONVOLUTION LAYERS ONLY). *NOTE, $2\times$ REFERS TO THE NUMBER OF TIMES A LAYER IS REPEATED. EACH CONVOLUTION LAYER HAS STRIDE=1, AND THE INPUT IS DOWNSAMPLED USING 2×2 MAXPOOL LAYERS.

#L.	Filter size ($k \times k \times IC \times OC$)	Input size ($W \times H \times IC$)
1	$3\times3\times3\times64$	$32\times32\times3$
2	$3\times3\times64\times64$	$32\times32\times64$
3	$3\times3\times64\times128$	$16\times16\times64$
4	$3\times3\times128\times128$	$16\times16\times128$
5	$3\times3\times128\times256$	$8\times8\times128$
6 ($2\times$)	$3\times3\times256\times256$	$8\times8\times256$
7	$3\times3\times256\times512$	$4\times4\times256$
8 ($2\times$)	$3\times3\times512\times512$	$4\times4\times512$
9 ($3\times$)	$3\times3\times512\times512$	$2\times2\times512$
10 ($2\times$)	$1\times1\times512\times512$	$1\times1\times512$

TABLE II
MEMORY USAGE IN MEGABYTES (MB) FOR VARYING BATCH SIZE ON VGG16

Batch	Darknet (float)	8-bit			8-bit + SWA		
		malloc	cma	total	malloc	cma	total
1	141	4	34	38	65	34	99
32	276	110	38	148	171	38	209
128	722	438	51	489	499	51	550

the contiguous memory allocations. We allocate contiguous buffers for the inputs and outputs of the GEMM and size them for the largest layers of the network. The weights are saved in contiguous memory which avoids the overhead of copying data. The input activations are stored in non-contiguous memory and *im2col* writes them into contiguous buffers each iteration. Stochastic weight averaging (SWA) requires additional 61 MB for a floating-point copy of the low-precision weights. This table assumes a batch is processed one example at a time. If the batch is tiled by a larger factor, then the size of each direct memory access (DMA) transfer will be larger and more contiguous memory must be allocated. A larger tile size could lead to improved performance as shown in Figure 5, but this has not been implemented.

B. Hardware Design

Figure 4 shows the high level design of our accelerator architecture. The core computes matrix multiplication $AB = C$ or $A^TB = C$. The option is configurable at runtime.

1) Dataflow: Matrix A is pre-loaded into on-chip block rams (BRAMs) and mapped spatially to an $N \times N$ array of processing elements (PEs). B is streamed from DRAM in row major order (see Figure 3) forming 8-bit N-length vectors which are broadcast to each column of the PE array. Each column is responsible for computing one N-length dot product, forming $N\times$ 32-bit partial sums each cycle. The partial sums are accumulated, and the result C' is produced after an entire row of B has been processed. If A are weights

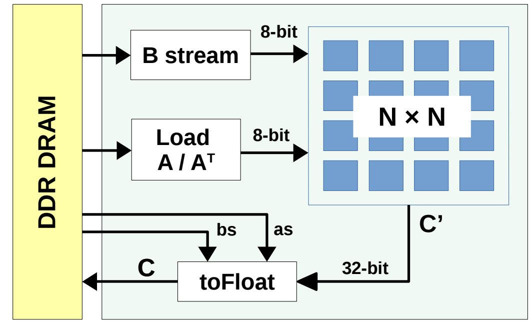

Fig. 4. Low-precision GEMM FPGA Accelerator

then this describes a weight-stationary dataflow [17].

2) Fixed to Float: The integer result C' is cast back to floating-point and re-scaled by successively multiplying the BFP scaling factors for matrix A and B. As described previously, we apply one scaling factor for the weights, and b scaling factors for the input activations and deltas, where b equals the batch size. The casting and scaling is fully pipelined, consuming moderate FPGA resources. Any other conditioning or reshaping for matrix B or C is performed by the PS.

3) Memory and Array Partitioning: Our accelerator assumes the entire matrix A can be pre-loaded into on-chip memory. Therefore, the maximum DNN size we can train is limited by the availability of on-chip memory resources, i.e. the number of BRAMs and LUT RAMs on the FPGA. Figure 3 shows the array shapes our accelerator expects for A, B and C, for forward (top) and backward paths (middle and bottom). Since we process a batch one input (image) at a time, the on-chip memory needed for A is $A_{fpga} \geq \max(oc \times ic \times k^2, oc \times w \times h)$. For VGG16 (see Table I), this means our FPGA must have at least $512 \times 512 \times 9 \times 1 = 2.35MB$ of available memory. This is mapped to BRAMs in our PEs. The PE memory is statically partitioned at compile time along both row and column axes, but the memory access pattern is run time configurable allowing data to be fetched in either row major or column major order. This is critically important for DNN training workloads specifically, where the weights W and the transposed weights W^T are both mapped to A and calculate the output activation and activation gradients in the forward and backward path respectively (as shown in Figure 3). If we don't have run time configurability then our static array partitions would have to be equal sized. This would be highly inefficient and require approximately $9\times$ more memory resources.

4) Performance Roofline: A roofline model [29] is used in Figure 5 to visually relate peak performance and off-chip memory bandwidth with the operational intensity of the largest GEMM in VGG16. The largest GEMM exists in computing

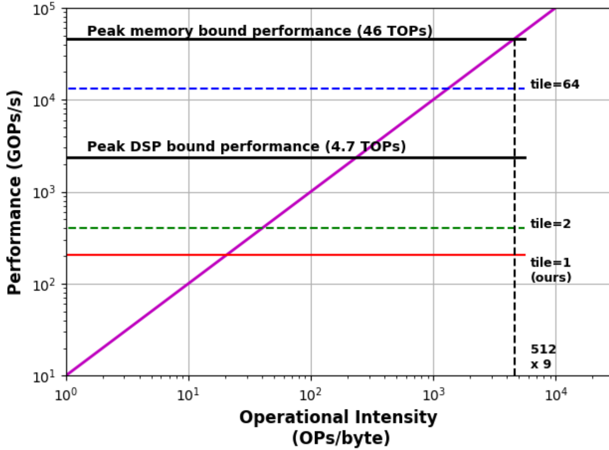

Fig. 5. Roofline model: 32x32 PE array at 200 MHz

the backward path for layers with $oc = 512$ and $ic = 512$, where 512×9 multiply accumulates (MACs) are required per input (or per byte). Figure 5 equates one operation with one MAC. We have assumed a conservative estimate of 10 GB/s for the memory bandwidth of the ZCU111 board. Therefore peak memory bound performance equals to $10 \times 512 \times 9 = 46,080$ giga-operations per second (GOPs/s), or 46 TOPs/s. The peak DSP performance is based on 4272 DSPs doing two 8-bit MAC at 550 MHz, therefore $4272 \times 2 \times 550 = 4.70$ TOPs/s.

The red line is our implementation and is based on a 32×32 PE array doing 1 MAC per cycle at 200 MHz, therefore $32 \times 32 \times 200 = 205$ GOPs/s. Clearly, large VGG layers are compute bound because the dotted line intersects the red line under the peak memory bound performance. However, even small VGG layers are compute bound. For example, the smallest GEMM in VGG has an operational intensity of 64 and peak memory bound performance of 640 GOPs/s. This is still higher than our peak compute performance. Furthermore we are also architecture bound since our peak compute performance is significantly below the peak DSP performance, indicating our chosen architecture is not utilising available compute resources efficiently.

To utilise more of the available memory bandwidth and DSP resources, we can consider interleaving multiple rows of input B and compute them in parallel. We can think of this as blocking or tiling, and the green and blue lines show performance points of different tile factors. As mentioned previously, tiling requires changes in the software backend, and requires a larger contiguous memory allocation. The authors in [25] describe tiling strategies for optimising the GEMM for training, and there are many other highly optimised GEMMs which could replace our core.

5) Implementation details: We designed our accelerator using Vivado HLS 2018.3. We encountered several limitations of using HLS which are worth mentioning here. First, arrays in HLS must ordinarily be partitioned by power of two factors otherwise large amounts of additional logic is inferred leading to problems with synthesis. Second, to implement both row major and column major reads on A, A must have the same partition factor in each dimension. Together, these issues meant our PE array can only be scaled by a minimum factor of $4\times$.

TABLE III
RESOURCE UTILISATION FOR ZCU111 BOARD

Config.	A_{max} (row/col)	BRAM 36k (1080)	LUTs/DSPs/FFs (4.2M)/(4.2k)/(850k)	Freq. (Mhz)
16x16	240/2198	149	32.3k/269/20.7k	262
16x16	336/3040	277	32.5k/269/20.8k	253
32x32	480/4160	533	69.3k/1037/25.8k	246
32x32	672/6080	1045	73.1k/1037/25.6k	181

TABLE IV
RESOURCE UTILISATION FOR PYNQ-Z1 BOARD

Config.	A_{max} (row/col)	BRAM 36k (140)	LUTs/DSPs/FFs (53k)/(220)/(106k)	Freq. (Mhz)
8x8	120/1064	45	13.9k/81/17.4k	114
8x8	168/1520	77	14.1k/81/17.4k	112

IV. Analysis and Implementation

We tested our architecture and training algorithms on two generations of Zynq SoC and MPSoC devices from Xilinx Inc. The Pynq-Z1 board, which integrates an ARM Cortex-A9 processor and small 7-series FPGA, and the ZCU111 development board (also known as RFSoC) which couples four ARM Cortex-A53 processors and a large Ultrascale+ FPGA. For comparative purposes, we have chosen to validate the training accuracy and run times on standard well known networks and datasets, such as VGG16 [30] and CIFAR10 [31]. These problems are perhaps ill-suited to the low power embedded application domain we're focused on, but they are familiar to the research community in general. In the next section, we evaluate our training accelerator on a targeted problem with real world constraints.

A. FPGA Resource Utilisation

Tables III and IV show FPGA resource utilisations for different PE array configurations on the ZCU111 and Pynq Z1 boards. As mentioned previously, our accelerator is designed to fit the entire A matrix in on-chip memory, and more specifically in block rams (BRAMs) located in PEs. If this is not possible then larger layers can be handled by breaking A into smaller blocks and calling the accelerator multiple times. Arbitrarily large layers can be solved as long as they fit in ARM memory. For now we assume the accelerator must only be called once and therefore A_{max} refers to the largest supported matrix for a specific PE and BRAM configuration. Notably, A can be at most 672×6080 and 168×1520 on the ZCU111 and Pynq-Z1 boards respectively. This refers to designs which consume 97% and

TABLE V

TEST ACCURACY (%) ON CIFAR10 AND MNIST FOR VGG16, PRERESNET-20 AND LOGISTIC REGRESSION, TRAINED WITH DIFFERENT QUANTISATION SCHEMES. THE NUMBER OF EPOCHS TAKEN TO REACH 92% ACCURACY FOR BATCH SIZE 128 ARE ALSO RECORDED.

Dataset	Model	Float		SWALP [8]		Ours (8-bit)	
		Acc.	Ep.	Acc.	Ep.	Acc.	Ep.
CIFAR10	VGG16	93.02	205	92.47	195	92.93	177
	PreResNet-20	93.29	223	93.29	225	93.72	218
MNIST	Logistic Regression	92.6	104	92.06	66	92.7	108

TABLE VI

TIME PER ITERATION (*secs*) ON CIFAR10 AND MNIST FOR VGG16 AND LOGISTIC REGRESSION, TRAINED ON PYNQ-Z1 AND ZCU111 BOARDS

Dataset	Model	Batch	Pynq-Z1		ZCU111	
			A9	FPGA	A53	FPGA
CIFAR10	VGG16	1	-	-	5.6	1.58
		32	-	-	172	9.89
		128	-	-	680	38.6
MNIST	LogReg	128	0.0401	0.0084	0.0121	0.0031
		256	0.0824	0.0149	0.0247	0.0057

55% of available BRAM memory. Due to power of two scaling we can not improve BRAM usage on the Pynq-Z1 board. Also, A_{max} is observably non-square because we are optimising the memory partitioning for GEMMs with 3×3 convolution kernels. This means, we can support convolution layers with up to 672 and 168 channels on each board respectively.

B. 8-bit Training

The key results from SWALP [8] have been reproduced in Table V for training VGG16, PreResNet-20 [32] and logistic regression networks on CIFAR10 and MNIST datasets.

All networks were trained with batch size 128, and without bias and batch normalisation. We trained for 250, 350, and 150 epochs, for VGG16, PreResNet-20 and logistic regression respectively, using starting learning rates of 0.05, 0.1 and 0.1 which are decayed linearly. Stochastic weight averaging is applied for the last 25% of epochs with a modified constant learning rate of 0.01. Importantly, Table V shows our quantisation scheme achieves virtually the same test accuracy as full-precision floating-point networks and the prior work of SWALP [8]. This is expected since our work uses a combination of 8-bit fixed-point and full-precision floating-point (for the weight gradients and updates), whereas SWALP is entirely 8-bit. Our choice for a mixed precision algorithm is predicated on the fact that Zynq has fast DDR memory shared between the PL and floating-point units in the PS. Basically, we can compute the weight updates efficiently in low-precision on the FPGA, and accumulate them in high precision on the ARM.

C. Performance

Training times per iteration are given in Table VI for varying batch sizes. As mentioned previously, our accelerator does not exploit batch level parallelism, and therefore training times increase linearly with batch size. We placed designs with 8×8 and 32×32 array configurations on the Pynq-Z1 and ZCU111 boards running at 100Mhz and 180Mhz respectively. Using low-precision and offloading all GEMMs to the FPGA, we can achieve a 17× speed-up for CIFAR10 and VGG16 on the ZCU111, and up to 5.5× and 4.3× speed-ups for MNIST on the Pynq-Z1 and ZCU111 boards respectively. This is compared to software running on an A53 (or A9) Arm processor and all computation done in full-precision floating-point. For an additional point of reference, a mid-range Tesla M40 GPU performs at roughly 16 iterations per second for batch 128 on CIFAR10 and VGG16. This corresponds to a massive 600× speedup over our ZCU111 implementation. However, the work by Luo et al. [25] has shown that this performance gap can be significantly reduced, through GEMM optimisations and also by implementing more of the training on-chip. We suggest that in applications which must be deployed stand-alone (without host communication), and which need the energy efficiency and low latency of FPGA inference engines, then system designers are likely to tolerate slower training times as long as that training can be computed with high accuracy on-chip. The low-precision techniques presented in this work do not compromise the training accuracy. Figure 6 profiles the execution time and

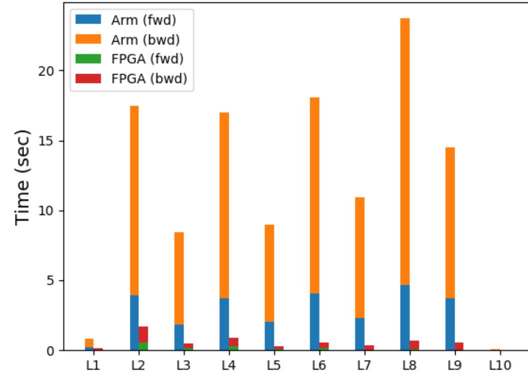

Fig. 6. VGG16 per-layer forward and backward times on ZCU111

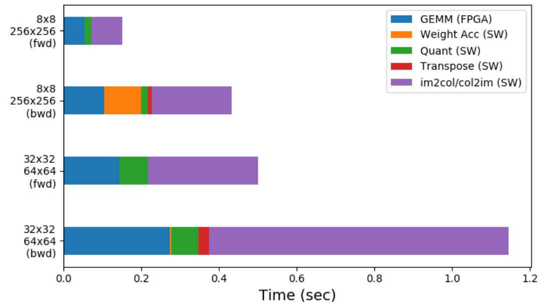

Fig. 7. Software overhead of convolution layers

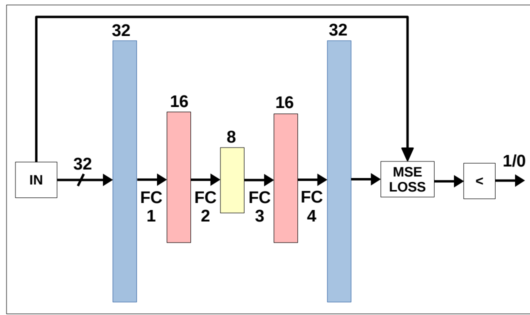

Fig. 8. Auto-encoder network architecture

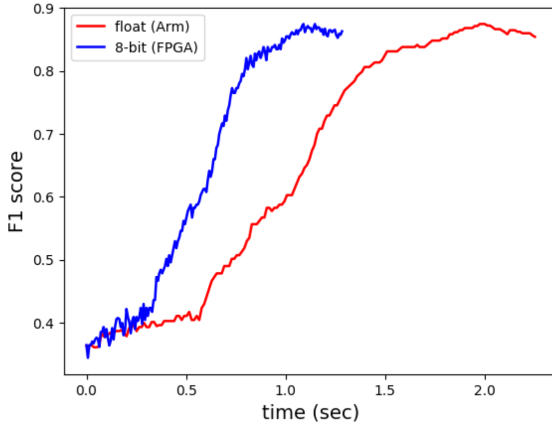

Fig. 9. Comparison of F1-scores and training times for anomaly detection

relative speed-up for forward and backward paths, and for each convolution layer in the VGG16 network (details provided in Table I). The backward path requires an extra GEMM and takes observably longer than the forward path, and layer 8 and layer 2 take the most time on the ARM and FPGA respectively. The input and output activations are particularly large in layer 2 (i.e. $32 \times 32 \times 64$), and therefore proportionately more time is spent in software on the *im2col* and quantisation functions.

D. Software Overhead

Figure 7 profiles the runtime and software overhead on two different convolution layers, for both the forward and backward paths. Clearly, *im2col* and *col2im* are the main bottlenecks, and this is especially damaging on layers with a large number of input and output activations (like layer 2, $32 \times 32 \times 64$ pixels). Future work should aim to accelerate both functions. This will require channel first ordering on the array in software, and more on-chip memory for implementing line buffers in hardware.

V. Case Study: On-chip Training of Auto-Encoders for Anomaly Detection in RF Networks

The application of neural networks to physical layer radio signals is extremely challenging due to the high data rates involved. In anomaly detection, one would like to perform training and prediction in an online manner to obtain a model of normal data, while simultaneously making normal/abnormal classifications. In this section, we train a low-precision variant of a small and simple network for detecting anomalies in frequency modulated radio signals.

Figure 8 shows the network architecture for the RF anomaly detector. The main component is the autoencoder multi-layer perceptron (MLP) network. The first two layers encode and compress the input signal by learning a dimensionality reduction transform, and the final two layers reconstruct the encoding back to the original input signal. The mean-squared error loss is computed at the output layer using the original input signal as the truth values. An anomaly is detected when the loss is higher than a preset threshold value. This is a form of unsupervised learning.

Training is still performed by offloading each GEMM to the FPGA accelerator on the ZCU111 board. This experiment uses a 16×16 PE array configuration running at 200Mhz. Our training data consists of sliding windows of complex I/Q samples. Each window has a length of 32 which represents the dimensionality of the input vectors. We perform training by collecting and then iterating low-precision SGD over a batch of 1000 inputs. To validate the performance of the anomaly detector during training, we created a test set with three different types of noise (i.e. bandpass, chirp and complex sine), and computed an F1-score for correctly detecting the known anomalies. Figure 9 shows a plot of F1-score on the test set for floating-point and 8-bit training algorithms. The anomaly detector shows convergence to an F1-score of approximately 0.87 after around 200 iterations for both curves. The 8-bit version is accelerated by the FPGA and is observably faster to converge. This reflects a $1.75\times$ speed-up over training with software-only. Much larger speed-ups are expected for wider networks that require more computation.

VI. Conclusion

In this paper, we have introduced new techniques for efficiently training DNNs with high accuracy on Zynq devices. The convolution and fully connected layers are computed on the FPGA using only 8-bit block floating-point numbers for weights, activations and activation gradients, while the rest of the network is computed in full-precision on an ARM processor. A prototype training accelerator was designed for portability across multiple Zynq boards and integration with high-level software frameworks. Results of an implementation on multiple benchmarks show up to $17\times$ speed-ups over processor only systems without any degradation in training accuracy.

REFERENCES

[1] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[2] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[3] M. Courbariaux and Y. Bengio, "Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1," *CoRR*, vol. abs/1602.02830, 2016. [Online]. Available: http://arxiv.org/abs/1602.02830

[4] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Advances in neural information processing systems*, 2018, pp. 7675–7684.

[5] M. Drumond, L. Tao, M. Jaggi, and B. Falsafi, "Training dnns with hybrid block floating point," in *Advances in Neural Information Processing Systems*, 2018, pp. 453–463.

[6] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Advances in neural information processing systems*, 2017, pp. 1742–1752.

[7] C. De Sa, M. Leszczynski, J. Zhang, A. Marzoev, C. R. Aberger, K. Olukotun, and C. Ré, "High-accuracy low-precision training," *arXiv preprint arXiv:1803.03383*, 2018.

[8] G. Yang, T. Zhang, P. Kirichenko, J. Bai, A. G. Wilson, and C. De Sa, "Swalp: Stochastic weight averaging in low precision training," in *International Conference on Machine Learning*, 2019, pp. 7015–7024.

[9] Y. Jia, "Learning semantic image representations at a large scale," Ph.D. dissertation, UC Berkeley, 2014.

[10] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015, pp. 1737–1746.

[11] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, "Scalable methods for 8-bit training of neural networks," *CoRR*, vol. abs/1805.11046, 2018. [Online]. Available: http://arxiv.org/abs/1805.11046

[12] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 7675–7684. [Online]. Available: http://papers.nips.cc/paper/7994-training-deep-neural-networks-with-8-bit-floating-point-numbers.pdf

[13] Z. Song, Z. Liu, and D. Wang, "Computation error analysis of block floating point arithmetic oriented convolution neural network accelerator design," in *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[14] P. Izmailov, D. Podoprikhin, T. Garipov, D. Vetrov, and A. G. Wilson, "Averaging weights leads to wider optima and better generalization," *arXiv preprint arXiv:1803.05407*, 2018.

[15] V. Sze, Y. Chen, J. S. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," *CoRR*, vol. abs/1612.07625, 2016. [Online]. Available: http://arxiv.org/abs/1612.07625

[16] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. A. Vissers, "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 2017, Monterey, CA, USA, February 22-24, 2017*, 2017, pp. 65–74. [Online]. Available: http://dl.acm.org/citation.cfm?id=3021744

[17] Y.-H. Chen, J. Emer, and V. Sze, "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks," in *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3. IEEE Press, 2016, pp. 367–379.

[18] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 75–84.

[19] M. Ghasemzadeh, M. Samragh, and F. Koushanfar, "Rebnet: Residual binarized neural network," in *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2018, pp. 57–64.

[20] S. Coric, I. Latinovic, and A. Pavasovic, "A neural network fpga implementation," in *Proceedings of the 5th Seminar on Neural Network Applications in Electrical Engineering. NEUREL 2000 (IEEE Cat. No. 00EX287)*. IEEE, 2000, pp. 117–120.

[21] R. G. Gironés, R. C. Palero, J. C. Boluda, and A. S. Cortés, "Fpga implementation of a pipelined on-line backpropagation," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 40, no. 2, pp. 189–213, 2005.

[22] S. Siddhartha, S. Wilton, D. Boland, B. Flower, P. Blackmore, and P. Leong, "Simultaneous inference and training using on-fpga weight perturbation techniques," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 306–309.

[23] T. Geng, T. Wang, A. Sanaullah, C. Yang, R. Patel, and M. Herbordt, "A framework for acceleration of cnn training on deeply-pipelined fpga clusters with work and weight load balancing," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018, pp. 394–3944.

[24] W. Zhao, H. Fu, W. Luk, T. Yu, S. Wang, B. Feng, Y. Ma, and G. Yang, "F-cnn: An fpga-based framework for training convolutional neural networks," in *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2016, pp. 107–114.

[25] C. Luo, M.-K. Sit, H. Fan, S. Liu, W. Luk, and C. Guo, "Towards efficient deep neural network training by fpga-based batch-level parallelism," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 45–52.

[26] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[27] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[28] "Python productivity for zynq." [Online]. Available: http://www.pynq.io/home.html

[29] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[30] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[31] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," 2009.

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," in *European conference on computer vision*. Springer, 2016, pp. 630–645.