AddNet: Deep Neural Networks using FPGA-Optimized Multipliers

Julian Faraone, Martin Kumm *Member, IEEE*, Martin Hardieck, Peter Zipf *Member, IEEE*, Xueyuan Liu, David Boland, Philip H.W. Leong *Senior Member, IEEE*

***Abstract*—Low-precision arithmetic to accelerate deep learning applications on Field-Programmable Gate Arrays (FPGAs) has been studied extensively because it offers the potential to save silicon area, or increase throughput. However, these benefits come at the cost of a decrease in accuracy. In this paper, we demonstrate that reconfigurable constant coefficient multipliers (RCCMs) offer a better alternative for saving silicon area than utilizing low precision arithmetic. RCCMs multiply input values by a restricted choice of coefficients using only adders, subtractors, bit shifts and multiplexers, meaning they can be heavily optimised for FPGAs. We propose a family of RCCMs tailored to FPGA logic elements to ensure their efficient utilization. To minimize information loss from quantization, we then develop novel training techniques which map the possible coefficient representations of the RCCMs to neural network weight parameter distributions. This enables usage of the RCCMs in hardware, while maintaining high accuracy. We demonstrate the benefits of these techniques using AlexNet, ResNet-18 and ResNet-50 networks. The resulting implementations achieve up to 50% resource savings over traditional 8-bit quantized networks, translating to significant speedups and power savings. Our RCCM with the lowest resource requirements exceeds 6-bit fixed point accuracy, while all other implementations with RCCMs achieve at least similar accuracy to an 8-bit uniformly quantized design, while achieving significant resource savings.**

I. Introduction

Convolutional Neural Networks (CNNs) have been widely adopted in recent computer vision applications due to their superior prediction capabilities, with researchers gravitating towards larger networks with higher computational complexity and memory requirements [1]. Field Programmable Gate Array (FPGA) implementations have demonstrated improved latency and power efficiency compared with central processing unit (CPU) and graphics processing unit (GPU) technologies, e.g. [2], [3]. In contrast to CPU/GPU technologies, they allow customized data paths, enabling improved parallelism and less data movement. This design flexibility poses an opportunity to optimize system performance through custom hardware tailored to the application.

Optimizations via compression, quantization and neural network layer explorations have been utilized to reduce complexity and boost performance, e.g. [4], [5]. In particular, quantizing inference networks to very low precision, such as constraining weight representations to binary or ternary values, both reduces memory requirements and enables multiplications to be replaced with the exclusive NOR operation [3], [6]. However, the disadvantage of extreme quantization is that the networks typically incur significant accuracy degradation for very low precisions, especially for complex problems.

One limitation with traditional fixed point quantization is that it has a uniform distribution. However, it has been demonstrated that a non-uniform distribution with the same number of potential weights can result in better accuracy, provided the distribution appropriately matches the desired full-precision neural network weight distribution [7], [8]. It follows, that reducing precision may not be the best method to save silicon area. Reconfigurable constant coefficient multiplications (RCCMs) are an alternative method to reduce FPGA resources through time-multiplexing and resource sharing [9]. They are usually realized using additions, subtractions, bit shifts and multiplexers, meaning multiplies are implemented without requiring digital signal processing (DSP) blocks on an FPGA. However, RCCMs are restricted to a given number of target coefficients; this has restricted their use to digital signal processing application domains including digital filtering and linear transformations, e.g., [10]. We propose a method, AddNet, to design RCCMs with coefficient sets that approximate the desired distribution of neural network weights. Furthermore, we develop a method to train neural networks to take advantage of RCCMs. In doing so, we demonstrate that using AddNet to optimize neural networks outperforms low precision arithmetic in terms of accuracy for a given silicon area budget.

AddNet consists of the following stages. First, we design a family of RCCMs which are customized to the underlying logic elements on the FPGA. These exhibit very low resource usage and have varying coefficient sets. The RCCM coefficient set whose distribution best replicates the weight distribution of a pre-trained network is chosen and the network is re-trained with weights restricted to these coefficients. This allows the optimizer to update network weight parameters during training while incorporating information about the underlying hardware. This study does not consider the embedded multipliers present in all modern FPGAs; in practical implementations, we envisage different CNN layers using embedded multipliers or our RCCM, depending on resource and throughput requirements.

The trained network is able to learn a representation com-

J. Faraone, X. Liu, D. Boland and P. Leong are with the School of Electrical and Information Engineering, The University of Sydney, Sydney, NSW 2006, Australia (e-mail: julian.faraone@sydney.edu.au, david.boland@sydney.edu.au, philip.leong@sydney.edu.au). M. Kumm is with the Applied Computer Science faculty of the Fulda University of Applied Sciences, 36037 Fulda, Germany (e-mail: martin.kumm@cs.hs-fulda.de), M. Hardieck and P. Zipf are with the Digital Technology Group, University of Kassel, 34121 Kassel, Germany (e-mail: hardieck@uni-kassel.de, zipf@uni-kassel.de).

Manuscript received February xx, 2019; revised April xx, 2019.

patible with the underlying optimized RCCM, achieving both high performance and accuracy. This allows a significant reduction in resource usage for a given throughput, making our designs suitable for resource-constrained implementations. Additionally, we can scale the parallelism of the design to achieve much higher frame rates for similar resource usage. Specifically, our work makes the following contributions:

- A novel family of arithmetic RCCM circuits tailored to the FPGA fabric for neural network applications which significantly reduces resource requirements.
- A distribution matching technique which allows a specific RCCM to be selected based on the required distribution of weights in a CNN, and a training algorithm which finds solutions compatible with the selected RCCM.
- We demonstrate our method achieves significant improvement in accuracy over low-precision (1-6 bit) implementations, and significant reductions in Look-up Table (LUT) usage over 8-bit fixed point precision with no loss in accuracy for state-of-the-art networks such as ResNet [11] implemented in fixed-point. Moreover, weight storage requirements are reduced through implicit weight sharing.

The remainder of the paper is structured as follows: Section II provides a background to training CNNs and constant coefficient multipliers. In Section III, recent state-of-the-art research on quantization training and hardware architectures for the implementation of CNNs is reviewed. Our methodology for designing our RCCMs is described in Section IV. Our training techniques and selection of RCCM is presented in Section V. The hardware architecture used for evaluating the effects of our methods is described in Section VI, followed by training and resource usage results in Section VII. Finally, we conclude the article in Section VIII.

II. Background

In this section, we discuss the basic computation of Convolutional Neural Networks (CNNs), introduce fixed-point training and softcore multiplier optimizations.

A. Convolutional Neural Networks (CNNs)

CNNs are biologically inspired networks that process input tensors (multidimensional arrays) of (w, h, d) dimensions in a translationally invariant manner [1]. Typically, w, and h are spatial dimensions and d is the number of channels. Processing operations such as convolution, pooling and activation functions are applied in a series of layers, each of which transforms the input tensors from dimension $(w, h, d) \rightarrow (w', h', d')$. A convolutional layer produces d' output channels, each formed through a convolution of the input tensor with a K_l convolutional kernel window, where typically $K_l = 1, 3, 5$ so it operates on local input regions. The output of convolutional layer l, takes as input S_l images of spatial dimensions w_l and h_l and d is the number of neurons. The pixel $y_{l;d;w;h}$ at location (w, h) for the dth neuron is calculated as

$$y_{l;w';h';d'} = g\left(\sum_{s=0}^{S_l} \sum_{j=0}^{J_l} \sum_{k=0}^{K_l} w_{l;d';s;j;k}\, y_{l-1;d;w+j;h+k}\right); \quad (1)$$

where g is the elementwise activation function (such as the ReLU function $g(x) = \max(0, x)$). The outputs of a layer are used as inputs to the next layer. A 2-dimensional (2D) convolutional layer can be described as matrix multiplication, followed by the elementwise activation function. Similarly, the output of a fully-connected layer can be described as

$$y = g(W^T x); \quad (2)$$

where $x \in \mathbb{R}^{d \times w \times h}$ is the input, $y \in \mathbb{R}^{d' \times w' \times h'}$ is the output, and $W \in \mathbb{R}^{d' \times d \times w \times h}$ are the weights of a linear transformation. Convolutional layers form the bottleneck for CNN implementations and this tensor form allows efficient matrix-multiplication libraries to be applied.

Pooling layers are downsamplers of 2D images. Max pooling layers provide a spatial maximum function which divides an input image into small sub-tiles of a given window size and then replaces these with the maximum value in the sub-tile. An average pooling layer is similar, however it finds the average in the sub-tile rather than the maximum.

B. Fixed-point Training

Deep neural network (DNN) training is an iterative process which has a feedforward path to compute the output and a backpropagation path for learning, which involves calculating gradients and update the network weights. Training of low-precision networks typically involves maintaining a set of single or double precision floating point weights W which are quantized to a representation $Q(W)$ prior to inference, e.g. [6]. As the quantization functions employed are piecewise and constant, the gradients of quantized weights can be calculated and applied to update their corresponding full-precision weights [12]. A quantization function which reduces mismatch in forward and backward paths is crucial for high accuracy.

To further improve accuracy, an alternative to low-precision networks is to use a weight sharing approach [13], [14]. Weight sharing involves choosing a finite set of full-precision weights indexed by a codebook. Typically, these weights are chosen to match the desired distribution to reduce information loss, unlike traditional fixed-point quantization where weights are uniformly distributed. Keeping the number of different weights in the codebook small reduces the word size of the indices leading to a small memory footprint. However, weight sharing is normally not applied in FPGA implementations as the weight mapping process introduces additional delays in the critical path of the circuit and requires extra hardware. Furthermore, higher precision arithmetic units also consume more area. For the proposed RCCM, an implicit weight sharing is utilized, reducing coefficient memory without requiring any mapping hardware. Meanwhile our RCCMs are optimized for FPGA hardware meaning they consume less area than fixed-point equivalents.

C. Small Softcore Multipliers

Due to the low precision requirements of neural networks, efficient implementations of small multipliers recently have gained growing interest [15], [16]. As FPGAs provide embedded multipliers it seems natural to use them. For small

multiplications, there is a way to perform two multiplications up to 8 8 bit in a single DSP of typically 18 bit [16]. In case the embedded multipliers are not sufficient, efficient logic-based (i.e. softcore) multiplier implementations are necessary. The use of radix-4 Booth encoding together with an FPGA mapping that maps both Booth encoder and decoders in the same LUT showed to be the most efficient way to implement softcore multipliers leading to up to 50% resource reductions [17], [18] on Xilinx FPGAs. Unfortunately, they are only this efficient for large word sizes of 16 bit and above. For lower word sizes, Xilinx Coregen showed the best results [18]. An optimization which is particularly suited for small multipliers by re-structuring common multiplier algorithms was recently proposed in [15]. They were optimized for Intel Stratix 10 showing the smallest resources and latency. The optimizations described in our work add further constraints designing multipliers which do not allow arbitrary fixed-point number support. This is achieved by applying concepts from reconfigurable multipliers.

III. Related Work

Many quantization methods for neural networks have been explored with the aim of achieving efficient inference in hardware. An efficient way of training networks with different forward and backward function was introduced in [12]. This led to new derivations of uniform quantization functions for low-precsion neural networks in [19], [20]. SYQ [21] further explored the importance of initializations and designing a quantization function which reduces the forward and backward mismatch. They achieved state-of-the-art accuracies under low-precision weights and activations. This inspired the derivation of the distribution matching initialization method for efficient quantization. Effective non-uniform quantization forms were also explored in the form of log representations [7]. This form can also compute multiplierless multiply-accumulates (MACs), however the distribution of the representations is restricted to the log domain.

There have been several accelerator architecture designs for low-precision CNNs with uniform quantization arithmetic. Recent literature includes commercial architectures [22], [23] and also academic approaches [24]–[28]. The benefits, in terms of power and throughput, of fitting a design on-chip was described in [3]. Other FPGA architectures have been implemented to utilize the highly amenable nature of CNNs which constrain weight parameters to be only binary or ternary representations [29], [30]. With restrictions in the efficiency of both software and hardware implementations of neural networks, software-hardware co-design is considered an effective approach to achieve optimal performance [31], [32]. A method for designing a quantization function for both increasing accuracy of binarized CNNs while maintaining efficient multiplierless hardware was proposed in [33]. Additionally, an efficient LSTM implementation in [34] utilized load-balance-aware pruning to achieve both network compression and high hardware utilization. Similarly, training highly sparse ternary networks and designing efficient CNN hardware for exploitation was described in [30]. To the best of our knowledge, AddNet is the first quantization scheme which embeds reconfigurability directly into its representations.

Fig. 1: Example of a reconfigurable multiplier with the coefficient set $\{12305, 20746\}$

IV. AddNet Reconfigurable Multipliers

In this section, we introduce reconfigurable multipliers and describe their design in AddNet.

A. Reconfigurable Multipliers

A constant coefficient multiplier is a circuit which computes $y = cx$, using only additions, subtractions and bit shifts, where c is some pre-defined number. For example, to compute $y = 6x$ in terms of additions and shifts, we can use

$$(x << 2) + (x << 1) = 6x \quad (3)$$

The "<<" operator represents an arithmetic left shift.

An RCCM is a circuit which computes $y = c_s x$ where c_s is an element from a discrete coefficient set $C = \{c_0, c_1, \ldots, c_{N-1}\}$, chosen from a $\lceil \log_2(N) \rceil$ bit select signal s [35]. RCCMs are usually realized using additions, subtractions, bit shifts and multiplexers (MUXes). Previous work has shown potential for reducing resource usage compared to a generic multiplier, especially for small values of N [9], [35]–[37].

Fig. 1 shows an example of an RCCM with coefficient set $C = \{12305, 20746\}$. In this example, there is one 2:1 multiplexer for each adder, each having s as the select line input. The three adders sum various shifted versions of x. Each adder is assigned a coefficient set where the value of each row corresponds to the multiple for each configuration. For instance, the top-most adder computes:

$$\begin{cases} x + (x << 1) = 3x & \text{if } s = 0 \\ x + (x << 2) = 5x & \text{if } s = 1 \end{cases}$$

The bottom-most adder outputs the final output y with coefficient set $C = \{c_0, c_1\} = \{12305, 20746\}$ multiplier-less by:

$$y = \begin{cases} x + ((x << 3) + ((x + x << 1) << 11) << 1) = 12305x & \text{if } s = 0 \\ (x << 8) + ((x + x << 2) + ((x + x << 2) << 11) << 1) = 20746x & \text{if } s = 1 \end{cases}$$

By utilizing multiplexers in this way, the computation of $c_1 = 12305$ is able to reuse the adders from computing $c_2 = 20746$, and vice-versa.

To date, prior research with RCCMs has focused on the design of an RCCM for a predefined set of target constants (e.g., obtained from a digital filter design). This design using minimal resources is an NP-complete optimization problem [36]. However, we want to use RCCMs in neural networks where the coefficients (weights) are not known in advance. As a result, we invert the RCCM design, and instead of searching for an RCCM circuit for a given coefficient set, this work aims to find one with very low resource usage and a maximum of "useful" coefficients. This low-cost RCCM then replaces multipliers in a conventional CNN implementation. Instead of storing the coefficients, the corresponding select values are stored, which also has the side-effect that it requires fewer bits of storage than the direct coefficient value.

B. FPGA Multiplier Mapping

We searched for building blocks that efficiently map to the logic fabric of an FPGA. Our designs are optimized for the latest Xilinx FPGAs (Virtex 5+6, the 7th generation FPGAs and UltraScale/UltraScale+ FPGAs), but similar circuits can be found for other FPGAs. For these devices, a slice provides either 6-input LUTs with a single output (used in Topology A) or two 5-input LUTs with shared inputs (used in Topology B). As such, we designed our base topologies to ensure the MUXes fit into the same LUTs that are required for the adders.

Fig. 2 shows the two base topologies used to build the RCCM units in this work. Each of these consists of an adder with at least one input being the output of a MUX. These topologies allow operations of the form $A_p \quad B_q$. For Topology A, A_p can consist of up to four different input values ($p \in 1, \ldots, 4$) with $A_4 = 0$ and B_q can only take one value, ($q = 1$). For Topology B, $p \in 1, \ldots, 3$ with $A_3 = 0$ and $q \in 1, 2$, with $B_2 = 0$. The sign and source signals are selected using a 2-bit input signal s. Since there are more possibilities than MUX inputs, a function (s) is used to choose the actual operation, where (s) is determined at design time, but may be different for each individual RCCM. Note that there is another possibility to map more input sources to the adder as described in [38], however to ensure the topology fits into a single LUT, this comes at a cost of less select inputs. Through our experimentation, we found that the chosen topologies were sufficient for creating RCCMs with a desired coefficient set to simplify the training process. This is further described in Section V.

All contemporary FPGA devices are similar in that their logic blocks consist of LUTs followed by a fast carry chain.

(a) Topology A (b) Topology B

Fig. 2: Base topologies used to build reconfigurable multipliers

TABLE I: Properties of the evaluation of the proposed RCCM units with maximum possible set size $S = 2^{w_s}$

RCCM	w_s	#unique coefficient sets
2-Add	4	1145
3-Add	6	44198
4-Add	8	4040952

Hence, a simple adder can be extended by multiplexers with no additional cost for certain multiplexer sizes when carefully selected for the target device. The detailed slice mappings of our base topologies are shown in Fig. 3, highlighting how our design consumes exactly the same silicon area as a traditional ripple-carry adder with the same word size on that FPGA (which would only implement the XOR gate to complete the carry logic to a full adder).

C. Architectures Considered

The base topologies described above can be combined in many ways to design RCCM units. Topology A has the advantage of a potentially larger coefficient set as it allows three different sources at input A_p. On the other hand, Topology B has the property that input B_q can be negated or zeroed, which provides symmetric coefficients around zero (as $A_p \quad B_1 = (A_p + B_1)$). We designed three different RCCM architectures from these topologies shown in Fig. 4. These consist of one to three elements of Topology A in the early stages and Topology B at the output stage to ensure symmetric coefficients. Symmetric coefficients improve the ability to match the distribution of the coefficient sets to the pre-trained neural network weights which are typically also approximately symmetric around zero. The benefits of this are further discussed in Section V. Note also that these designs can be trivially pipelined.

As can be seen in Fig. 4, the A_p inputs to the topologies are all connected to left shift operations s_{ij}, which are all hard-wired since these do not require any LUT resources. It follows that the supported coefficient set depends on the operation mapping function(s) and the fixed bit shifts s_{ij}. As mentioned in Section IV-B, each instance of base Topology A or B consumes the same area as a traditional ripple-carry adder. Hence, as the RCCMs of Fig. 4 consists of 2, 3 and 4 base topologies, they are, respecively, called 2-Add, 3-Add and 4-Add RCCMs in the following.

(a) Topology A

(b) Topology B

Fig. 3: Bit level FPGA slice mapping of base topologies of Fig. 2. This is applicable to any FPGA using 6-input LUTs, including Xilinx Ultrascale and Intel Stratix X devices

.

The obtained RCCM architectures can multiply with up to 2^{w_s} different coef cients where w_s denotes the total number of bits used for the select signal. For the 2-Add, 3-Add and 4-Add RCCMs, this translates to $w_s = 4; 6$ and 8 respectively, as can be seen in Fig. 4. We chose to evaluate coef cient sets where Topology A had 4 different mapping functions and Topology B a single one. In addition all maximum bit shifts were set to $'_{max} = 3$. This limits the total number of unique combinations as shown in Table I. With these coef cient sets, an exhaustive enumeration of possible coef cient combinations is feasible with a few minutes of computation time. This allows us to then nd the desired coef cient set based on its similarity to the pre-trained neural network weight distribution. We note that it may be possible to improve on our results by exploring more mapping functions, which would generate a larger number of unique coef cient sets, but at the cost of longer execution time.

V. AddNet Training

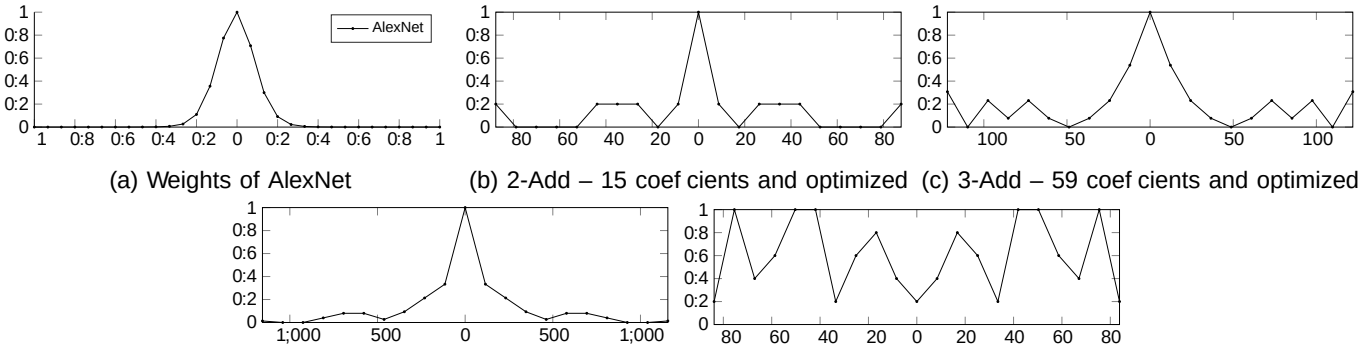
The previous section described a family of optimized multipliers. In this section, we now address the issue of nding the best coef cient set for a given neural network. Neural networks can typically tolerate a certain amount of regularization for their weight representations before the accuracy is impinged upon. Thus our strategy is to utilize this knowledge and select an RCCM coef cient set which exhibits a distribution similar to the distributions of the neural network weights and re-train the network to learn the representation of the coef cient set.

A. Distribution Matching

To achieve high accuracy in quantized neural network training, it is important to reduce quantization error by using a function which can ef ciently map its representations to the full-precision values. This is important to minimize information loss and to achieve a good initialization for training [21]. Fixed-point representations using quantization typically uniformly partition the weight parameter space. However, representations using RCCM coef cient sets discussed in Section IV are non-uniformly partitioned and can vary in size, range and the nature of the distribution. Thus, for ef cient training, we choose an RCCM with a coef cient set to match the distribution of a pre-trained model. We use the Kullback-Leibler divergence [39] as a measure of the similarity of two

(a) 2-Add RCCM (b) 3-Add RCCM (c) 4-Add RCCM

Fig. 4: Selected RCCM circuits



(a) Weights of AlexNet (b) 2-Add – 15 coef cients and optimized (c) 3-Add – 59 coef cients and optimized

(d) 4-Add – 207 coef cients and optimized (e) 3-Add – 63 coef cients and unoptimized

Fig. 5: Distribution for CNN weights and constant multiplier coef cients

distributions. Let R denote the distribution of the coef cient set of the RCCM, and P is the reference distribution of the pre-trained model weights and N the total number of weights. The Kullback-Leibler divergence D_{KL} is de ned as

$$D_{KL}(P\|R) = \sum_{i=0}^{N-1} P(i) \log \frac{P(i)}{R(i)} : \quad (4)$$

Thus, for each enumeration of the coef cient sets, we measured the divergence D_{KL} to the pre-trained network weights and selected the top-5 sets with the smallest divergence. We call this technique *distribution matching*. From the top-5 sets, we selected the set with the largest number of coef cients, to maximize the number of representable states for the weights during re-training. As a secondary criteria, we only selected coef cient sets that include zero. Note that a zero weight could alternatively also be realized by resetting the output ip- op in a pipelined implementation. Since this leads to an additional select-bit, that has to be stored in the coef cient memory or a separate decoder, this was not further investigated.

To give an example, the weight distribution (using 31 bins) from AlexNet on ImageNet is given in Fig. 5(a). As shown, the weight parameters in this example follow a distribution similar to a Gaussian distribution, meaning that small weight values near zero occur much more often than large values. The coef cient sets of the RCCM circuits of Fig. 4 with the best distribution matching are given in Table II. Their distributions are shown in gures 5(b), 5(c) and 5(d) which are similar to the pre-trained model. We call these optimized 2-Add, 3-Add and 4-Add RCCM circuits.

The exhaustive search for coef cient combinations yields distributions of different nature, meaning this method would

TABLE II: Optimized RCCM coefficients

arch.	#coeff	Coefficient set ()
2-Add	15	0 1 2 8 28 36 44 92
3-Add	59	0 1 2 3 4 5 6 7 9 10 12 13 14 16 23 29 30 32 63 69 70 72 87 93 94 96 119 125 126 128
4-Add	207	0 1 2 4 5 7 8 9 11 13 14 15 16 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 36 37 38 39 40 46 48 54 58 64 69 70 71 74 75 76 78 80 81 82 84 85 87 94 96 102 114 118 126 134 142 150 166 174 182 190 194 198 206 214 222 230 238 246 258 262 270 278 286 302 310 318 326 334 382 398 446 450 526 566 574 582 614 622 654 662 670 686 694 710 766 782 830 1214

TABLE III: Configuration parameters of the the RCCM units

RCCM	type	s_1 s_0 00	01	10	11	shifts $'_{i1}$	$'_{i2}$	$'_{i3}$	$'_{i4}$
2-Add	A_I	A1+B1	A2+B1	A3+B1	B1	0	1	3	2
	B	-A1+B1	-A2+B1	A1-B1	A2-B1	0	3	2	–
3-Add	A_I	A1+B1	A2+B1	A3+B1	-A2+B1	0	2	3	3
	A_{II}	A1+B1	A2+B1	A3+B1	-A1+B1	0	1	3	0
	B	-A1+B1	-A2+B1	A1-B1	A2-B1	0	3	0	–
4-Add	A_I	A1+B1	A2+B1	A3+B1	-A3+B1	0	1	3	0
	A_{II}	A1+B1	A2+B1	A3+B1	-A2+B1	0	1	3	1
	A_{III}	A1+B1	A2+B1	A3+B1	-A1+B1	0	1	3	3
	B	-A1+B1	-A2+B1	A1-B1	A2-B1	0	3	1	–

most likely be able to efficiently map to other potential network weight distributions. To further justify our approach of distribution matching, we also study an RCCM with an unoptimized choice of coefficient set with differing distribution nature. Fig. 5(e) shows the distribution with the worst (i.e., largest D_{KL}) divergence score for 63 coefficients. It is not obvious that the corresponding coefficient set, C_t = {0 8 12 14 16 18 20 21 23 24 36 38 40 42 44 45 47 49 51 52 54 56 58 60 68 70 72 74 76 77 79 88}, would lead to poor CNN inference accuracy. However, as shown in Table IV, when used with AlexNet [40] in the 2-Add case, Top-1/Top-5 accuracy is 53.8%/76.9% (results are presented as a Top-k percentage, where a classification is considered correct if the actual class is among the highest k probabilities). With distribution matching, the accuracy is 55.8%/79.8%, which is significantly better than the unoptimized set and equivalent to full-precision floating point accuracy.

TABLE IV: Accuracy change from optimized distribution matching on AlexNet for the 2-Add case

	Unoptimized	Distribution Matching	32bit Float.
Top-1	53.8%	55.8%	55.1%
Top-5	76.9%	79.8%	79.2%

B. Weight Quantization

To both exploit our RCCM and achieve high accuracy, our network should be trained to match the underlying inference hardware. During our fixed-point training, for each layer, w_l, we firstly clip the weights so that $w_l \in (-M, M)$, where M is a range hyperparameter, at each inference step and then quantize them to fixed-point representations. As discussed in Section II, our multiplier consists of a fixed point input and a value from C. During AddNet training, we introduce a function whereby every floating point weight is quantized according to

$$q(w_l) = \arg\min_{c_i \in C'} |c_i - |w_l|| \quad (5)$$

where $c_i \in C'$ represents the possible positive coefficients of C scaled by γ_l. Here, (5) aims to minimize the quantization error between the quantized weight values and the representations in our coefficient set. The scaling with γ_l is done so that $q_l \in (-M, M)$ where M is initially the range of the pre-trained model. By using distribution matching we minimize this quantization error to achieve an efficient initialization. We then re-train the network using the straight through estimator (STE) approach as described in [12]. This approach allows a non-differentiable function defined in the forward path to use a non-zero surrogate derivative function in the backward path gradient calculations. Thus, in our case, we allow:

$$\frac{\partial L}{\partial q} = \frac{\partial L}{\partial w} \quad (6)$$

where L is the loss function. The quantized weights $q(w_l)$ are used for inference in the forward path and the floating point weights w_l are updated in the backward path. During training, γ_l becomes a parameter which is also updated during backpropagation. By using a representation compatible with the multiplier in the forward path, the network learns a representation both high in accuracy and hardware efficiency. After training, the floating point weights are discarded and $q(w_l)$ is used for hardware deployment.

C. Activation Quantization

As initial training results did not show accuracy degradations compared to activations larger than 8-bit two's complement, we first uniformly quantize the activations to 8-bit. We also selected the input word size of the RCCM accordingly. In the forward path, we approximate the function in (2) with G, which uniformly quantizes a real number $x \in [0, m]$ to a k-bit number:

$$G(x) = \frac{1}{2^f} \lfloor 2^f x + \frac{1}{2} \rfloor \quad (7)$$

where $\lfloor \cdot \rfloor$ returns the greatest integer less than or equal to the argument and m is the upper bound. m itself is bounded by its arbitrary unsigned two's complement fixed point representation where f is the number of fractional bits and hence $m = 2^{k-f} - 2^{-f}$. A summary of the training process is given in Algorithm 1, which is similar to references [3], [6], with the addition of distribution matching and incorporating the quantization scheme of (5).

Algorithm 1 Training a CNN using AddNet representations

Initialize: Pre-train model
Set adder size
c = DistributionMatching($\sigma(s)$) using (4)
Inputs: Minibatch of inputs & targets (I, Y), Loss function $L(Y, \hat{Y})$, current weights W_t and learning rate, η_t
Outputs: Updated W_{t+1}, γ_{t+1} and η_{t+1}

Forward propagation:
for l=1 to L do
  Q_l = Quantize(W_l) using (5) and (7)
end for
$\hat{Y}$ = ForwardPropagation (I, Y, Q_l) using (7)

Backward Propagation:
$\frac{\partial \hat{L}}{\partial Q_l}$ = WeightBackward(Q_l; $\frac{\partial \hat{L}}{\partial \hat{Y}}$)
$\frac{\partial \hat{L}}{\partial \gamma_l}$ = ScalarBackward($\frac{\partial \hat{L}}{\partial Q_l}$; γ_l; $\frac{\partial \hat{L}}{\partial \hat{Y}}$)
W_{t+1} = UpdateWeights(W_t; $\frac{\partial \hat{L}}{\partial Q_l}$; η_t)
γ_{t+1} = UpdateScalars(γ_t; $\frac{\partial \hat{L}}{\partial \gamma_l}$; η_t)
η_{t+1} = UpdateLearningRate(η_t; t)

VI. Experimental Setup

In this section, we present the system used to evaluate the benefits of our AddNet optimizations. We implemented the circuits in figures 2, 3 and 4 in the hardware description language (HDL) VHDL, as they were more naturally described in a HDL than using other high level synthesis tools. We then chose to integrate it into the open source FPGA CNN Library by Alpha Data in VHDL [41], which provides basic neural network layers for generating custom 8-bit fixed point CNN implementations. We instantiate the multiplier in replace of the traditional VHDL fixed point multiplication used in the original Alpha Data source code. This is used as our hardware library.

The bitstream generation workflow is illustrated in Fig. 6. After defining the CNN architecture and pre-training the network, the RCCM coefficients are calculated using distribution matching. The user provides this information to our AddNet tensorflow software library which then trains the network for a specified adder size. Once trained, the weights are written to a file. These weights, along with the parallelism factors and architectural preferences are provided to the hardware library. The bitstream is then generated using Vivado 2018.1 with both the Peripheral Component Interconnect Express (PCIe) interface and Network Accelerator Core which are downloaded onto the FPGA. We tested both our tensorflow inference and hardware accelerator output to ensure correctness of the design.

A. System Overview

The Network Accelerator core is integrated with a PCIe interface as illustrated in Fig. 7(a) which uses a streaming approach to direct memory access (DMA) data at an efficient rate across the PCIe bus and back. The design is targeted to the Alpha Data ADM-PCIe-8K5 board with a Xilinx KU115

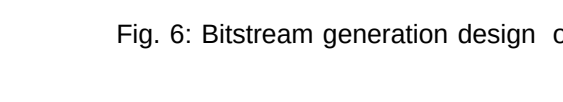


Fig. 6: Bitstream generation design flow

FPGA, which consists of 2160 BRAMs (36K), 5520 DSPs and 663,000 LUTs. A board-specific PCIe Alpha Data IP core is used to interface between PCIe and our network accelerator core. This IP core can be configured to provide and consume AXI4 DMA streams of width 256 bits at a clock rate of 250 MHz in response to API function calls from the host. This stream width is reduced to match the buffer sizes for the inputs (24 bits) and weights $(4:::8$ bits) which control data ingress. The weight data is sent in contiguous bursts to each layer in the Network Accelerator core to match the expected input behavior. DMA channel 0 is used to provide the input data for layer 0 and weights are initialized on DMA channel 1. The layer output is sent back over PCIe using a separate DMA channel. Additionally, a memory mapped direct slave port is used to access a bank of registers which can be read by the host to measure performance.

B. Network Layer Accelerator Core

The Network Layer Accelerator Core performs the MAC operations in parallel to compute the convolution as in equation (1). The core receives input from the feature and weight buffers and writes to the output buffer. The feature and weight buffers stream data into a serial to parallel converter to fan-out the data to n parallel Processing Elements (PEs). This is shown in Fig. 7(b). Once all data reaches the PE, up to n multiplications between features and weights are performed in parallel, and the results accumulated. Here, we replace the standard 8-bit multiplier with our AddNet constant coefficient multiplier described in Section 4 to reduce the cost per MAC over fixed point implementations. The data is then accumulated before being fed into a ReLU activation function. The output then fans-in via a parallel to serial converter before being streamed out of the current layer and into the subsequent layer. After fanning in, the feature stream data is multiplied by an 8-bit scaling constant γ_l, which is pre-computed. The number of multipliers is significantly larger than the number of layers in neural network designs. Hence, although we add one additional scale operation per layer, it only constitutes a tiny proportion of the overall area in comparison to high precision architectures.

C. Architectures

To quantify the benefits of the AddNet optimizations, we use two different architectures with 2-Add, 3-Add and 4-

(a) AlphaData Library System Design (b) Network Layer Accelerator Core including AddNet multipliers

Fig. 7: Hardware Accelerator System design

Add RCCMs, as well as traditional 8-bit fixed point. The architectures we study are a single layer CNN accelerator and a full AlexNet-variant network [40] which reduces the filter size in the first layer to 7x7 and changes the stride of the first and second layers to 2. The single layer implementation represents a loopback architecture. To implement a full network using this architecture, data is sent between the host and FPGA after each layer is computed sequentially. To minimize the amount of loopback iterations, we instantiate 2048 PEs as this equates to the number of neurons in the largest layer for all our networks. As such, we can compute all layer output feature maps for any of our networks during each loopback iteration. The AlexNet implementation represents a full data flow where all convolutional layers are processed on the FPGA. For all AlexNet implementations using RCCMs, the first layer uses the 4-Add RCCM. This was because a higher number of coefficients was required in the first layer to achieve higher accuracy.

Both architectures were developed by AlphaData, we have not added any optimizations aside from our arithmetic operators. This enables us to focus on the benefits of our optimizations; we believe AddNet could improve on any 8-bit fixed-point deep learning circuit.

D. Memory Use

One important advantage of the RCCM designs is the reduction in the number of coefficients required for storage. Instead of storing the coefficients, c_s, only the indexes has to be stored. This is similar to the weight sharing approach. However, no decoder circuit is necessary to realize the codebook as this is implicitly done by the proposed RCCM. While the coefficients in Table II would require 8, 10 and 12 bit to represent in two's complement, storing the index requires only 4, 6 and 8 bits for the 2-Add, 3-Add and 4-Add, respectively. So, significant savings in storage and memory bandwidth are possible for the 2-Add and 3-Add cases. For the Kintex Ultrascale devices, BRAMs can be a 36K unit or 18K units. As the number of 36K BRAMs are reported, the weight buffers at each PE are calculated as 0.5 to represent 18k BRAMs or 1 to represent 36K BRAMs.

VII. Results

We now display various hardware utilization and accuracy results to demonstrate applicability in neural network computation. The hardware results were obtained after place and route (PAR) using the Vivado 2018.1 design tool.

A. Reconfigurable Multiplier Resources

First, we made a comparison of the resource usage of our proposed RCCM compared to a generic multiplier. As generic multiplier, we selected the native Xilinx multiplier as it showed the best results for low word sizes [18]. Fig. 8 shows the LUT resources for varying input (activation) word sizes w_{in} from 3 to 16 bits. While the generic multiplier grows at about 7.4 LUTs/bit, the proposed 2-Add, 3-Add and 4-Add RCCMs only grow at 2, 3 and 4 LUTs/bit, respectively. It can be seen that the 2-Add and 3-Add RCCMs always outperform the generic multiplier for $w_{in} > 4$ bit while the 4-Add RCCM is only interesting for larger word sizes of $w_{in} > 9$ bits. For the considered 9 bit activation, 55.2% and 32.8% of the LUTs can be saved by using the 2-Add and 3-Add RCCMs. This improves further as we increase the activation precision, suggesting that this multiplier and quantization method can be very effective for CNN inference applications and potentially on-chip neural network training, which both benefit from higher activations precision. For the multipliers used in this experiment, the pipelined RCCMs can

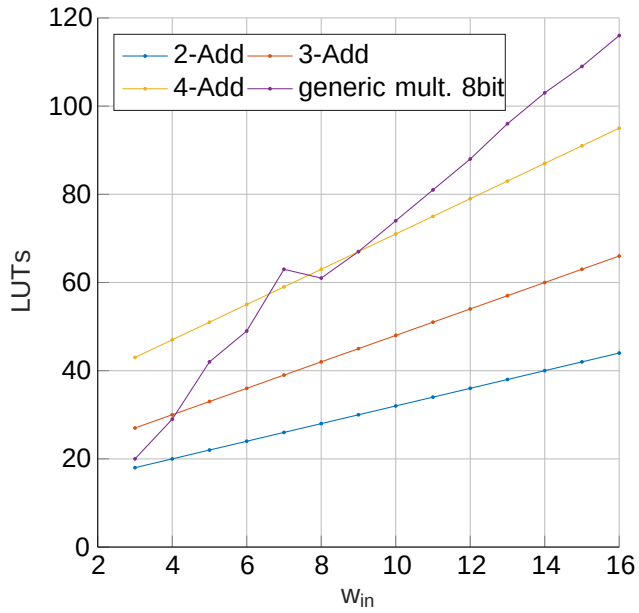


Fig. 8: LUT results from synthesis for the proposed RCCMs and a generic 8 w_{in} multiplier

TABLE V: PAR result comparison one layer with 10 neurons (with and without DSP mapping enabled)

Method	LUTs	FFs	DSPs	BRAM	Freq. [MHz]
8-bit, with DSPs	238	1015	10	5	446.63
8-bit, no DSPs	1407	1515	0	5	355.11
2-Add	818	1421	0	5	464.11
3-Add	928	1365	0	5	415.15
4-Add	1179	1385	0	5	342.68

operate between 350 MHz (4 bit) and 250 MHz (16 bit) while the generic multiplier can be clocked at between 200 MHz (4 bit) and 150 MHz (16 bit). Here, it is expected that the generic multiplier can be faster for faster timing constraints at the cost of additional resources.

B. Architecture Resource Utilization

In this section, we ran PAR experiments to compare our RCCMs against conventional 8-bit multipliers in a single CNN layer. Table V shows the resource utilization as well as the obtained speed.

The first row uses the fewest LUTs as multiplication is done in the DSPs. When DSPs are disabled, LUT usage dramatically increases. Our 2-Add design achieves the highest frequency at a significantly reduced LUT count compared to the 8-bit DSP disabled implementation. However, we note that the LUT usage could be reduced if implemented with tree-structured optimizations as in [15]. The 3-Add and 4-Add designs have more flexibility compared with the 2-Add, but require slightly more LUTs and operate with reduced frequency.

Table VI shows resource utilization for the two architectures, using the 2,3,4-Add and 8-bit DSP disabled designs. The 2,3,4-Add cases all achieve significant LUT savings. The PCIe interface uses 48 DSPs and 100 BRAMs. Weight storage memory reduction is also apparent in the form of a decrease

TABLE VI: Summary Of PAR Utilization on the Xilinx KU115 for all arithmetic types with PCIe interface included.

Xilinx KU115	Architecture	2-Add	3-Add	4-Add	8-bit disab. [41].	8-bit enab. [41]
BRAM (2160)	Conv Layer	1154	1154	1154	1154	170
	AlexNet	1365	1557	1557	1557	1229
DSP (5520)	Conv Layer	48	48	48	48	96
	AlexNet	48	48	48	48	3760
LUTs (663K)	Conv Layer	187.0	205.6	255.8	383.0	36.2
	AlexNet	331.7	372.8	430.7	467.1	128.8
Estim. Power	Conv Layer	7.6W	7.6W	7.8W	7.5W	7.2W
	AlexNet	39W	44W	48W	52W	29W

TABLE VII: Per Layer BRAM usage. p represents the parallelism of the PE and b represents the bits required to store each coefficient

	p	PE	b		BRAMs		Memory (MB)	
			2-Add	8-bit	2-Add	8-bit	2-Add	8-bit
Conv1	4	96	8	8	96	96	0.04	0.04
Conv2	4	256	4	8	256	256	0.15	0.31
Conv3	1	384	4	8	192	384	0.44	0.88
Conv4	2	384	4	8	384	384	0.33	0.66
Conv5	2	256	4	8	256	256	0.44	0.88

in BRAM utilization from 1557 in the 8-bit model to 1365 for 2-Add. This is because the reduction in bits per weight by using 2-Adders results in a 50% savings in BRAMs in the 3rd convolutional layer, as highlighted in Table VII. This large savings is due to the discrete size of Xilinx BRAMs. Xilinx BRAMs can be configured to have a data width of 1, 2, 4, 9 and 18 bits. The wider the data width, the fewer number of words that can be stored per BRAM. The required data width for each PE is given by the bits stored per weight, b, multiplied by the parallelism of the PE, p. In the case of Conv3, where $p = 1$, by reducing b to 4-bits, it is possible to store all the required weights for a PE using only a single 18K BRAM, in contrast to a 36K BRAM for the 8-bit case. In other layers where p is higher, reduced memory use of 2-adders does not result in fewer BRAMs used due to their discrete sizes.

In Table VI, we also present the estimated overall board power. Evidently there is an increase in board power when we disable DSPs. This is due to the increase of LUT resources which typically leads to more switching. However, comparing the 8-bit with DSPs disabled with AddNet implementations, we see reductions in power. Again, this is largely due to the reduction in LUTs.

The silicon area savings could also be used to scale-up the parallelism to improve throughput, reduce latency or fit the design on a smaller FPGA. For example, while the AlexNet and Conv Layer implementations already have one PE per output feature map, we can increase p and compute more output feature map pixels in parallel to reduce the number of PE iterations required to compute a layer. This trivial optimization could be applied when accelerating most large neural networks. Typically such networks have lots of in-

TABLE VIII: PAR frequencies for pipelined versions of the RCCMs

Type	2-Add	3-Add	4-Add
Original	447.43 MHz	483.09 MHz	342.82 MHz
Pipelined	770.42 MHz	578.03 MHz	623.83 MHz

herent parallelism and high computational requirements, with FPGA accelerators implementing some form of layer folding due to resource restrictions. This is especially the case for higher precision implementations when accuracy preservation is paramount. Thus, the AddNet multiplier is a very widely applicable tool for improving the parallelism of existing FPGA DNN architectures. Alternatively, these area reductions allow the current design to t on a smaller device, such as the Xilinx VU3P, which would lead to expected reductions in power consumption as less hardware is being used.

C. Frequency

The AlphaData CNN Library can operate conservatively at 250 MHz [41] and the critical path lies in the PCIe interface. Therefore, since our RCCMs can operate at a higher frequency, it does not translate to an increase in operating frequency of the overall system. However, as mentioned in Section IV-C, the RCCMs designed can also be trivially pipelined to improve their operating frequencies. As the multipliers were additionally implemented without the PCIe interface as both standalone components and within a CNN layer, we explored the frequencies of pipelined versions. The post PAR frequencies with a clock constraint of 250 Mhz (more aggressive time constraints will lead to higher frequencies than what is reported) for the standalone multipliers are shown in Table VIII. Signi cant frequency improvements are demonstrated from the pipelined versions which would lead to designs achieving higher frequencies when the multiplier lies in the critical path of the system. As the frequency is maintained at 250 MHz for all our implementations, the throughput remains constant.

D. Effect of Layer Size

We now explore how the resources usage scales with parallelism for a single layer convolutional core without the PCIe interface. This is an important metric for data ow implementations as we want to instantiate a higher number of PEs in layers with the most operations to achieve load balancing with less operationally intensive layers. Fig. 9 shows LUT usage from PAR where the number of parallel PEs is equal to typical neuron layer sizes used in our trained networks. Using such sizes allows us to simulate computing all output feature maps of a layer in parallel. As expected, all implementations scale linearly with the number of PEs. However, for the AddNet multipliers, as we increase the number of PEs we see smaller increase in LUTs in comparison to the 8-bit version. This is ampli ed further with the smaller multiplier implementations which demonstrate smaller gradients to the 4-Add version. For example, with 2048 PEs instantiated, we achieve a substantial 52% LUT reduction. Typically neural network implementations are constrained by the number of PEs we can instantiate per layer due to resource scaling.

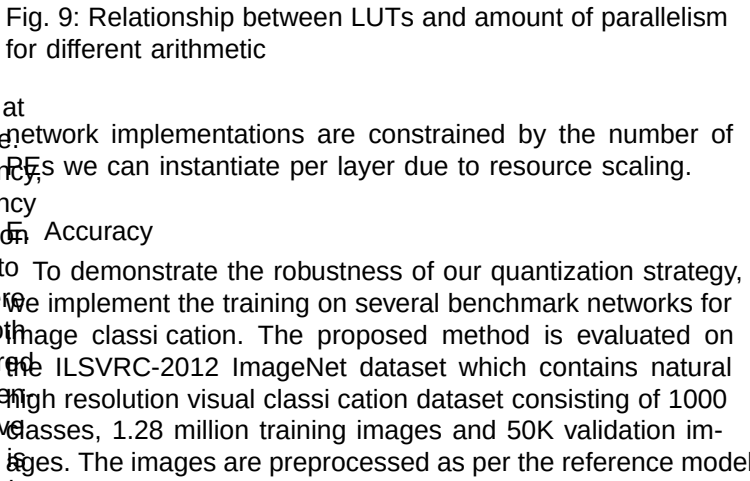


Fig. 9: Relationship between LUTs and amount of parallelism for different arithmetic

E. Accuracy

To demonstrate the robustness of our quantization strategy, we implement the training on several benchmark networks for image classi cation. The proposed method is evaluated on the ILSVRC-2012 ImageNet dataset which contains natural high resolution visual classi cation dataset consisting of 1000 classes, 1.28 million training images and 50K validation images. The images are preprocessed as per the reference models by resizing the inputs to 256256 before being randomly cropped to 224 224. We report our single-crop performance evaluation results using Top-1 and Top-5 accuracy, where the cross-entropy loss of the predicted classi cation against the actual classi cation is minimized during training. The AlexNet network consists of 5 convolutional and 3 fully-connected layers. ResNet networks consist of blocks of two or three convolutional layers and a residual connection [11]. Two models are explored with varying depths of these blocks.

In Table IX, we display the accuracies of quantizing for different multiplier sizes and compare them to xed point retraining and oating point network accuracies. All results were trained with the 4-Add RCCM in the rst and last layers to preserve accuracy and were trained for a xed number of epochs. For all these networks we achieve at least 8-bit accuracy with resource savings through our multiplier. This demonstrates the effectiveness of AddNet. In particular, we can achieve equivalent to oating point accuracy for AlexNet with only 2-Add multipliers which translates to large resource savings. In some instances, the accuracy is improved and this is due to the regularization effect of the quantization which improves the generalization of the network.

F. Accuracy vs Area

Fundamentally, our goal is to achieve the highest possible accuracy while consuming the smallest amount of resources.

TABLE IX: Accuracy results [%] for AddNet, oating-point (32 bit) and xed-point training over various ImageNet models

Model		2-Add	3-Add	4-Add	oat.	8-bit	6-bit	4-bit	Ternary	Binary
AlexNet	Top-1	55.8	55.8	55.9	55.1	55.5	54.7	53.9	53.2	52.0
	Top-5	79.8	79.8	80.0	79.2	78.6	78.5	78.3	78.1	76.9
ResNet-18	Top-1	65.1	66.0	66.4	68.6	66.0	63.5	62.0	61.6	57.5
	Top-5	86.4	87.6	87.8	88.2	87.5	85.9	85.4	84.2	81.2
ResNet-50	Top-1	72.1	72.7	73.3	76.0	72.5	69.6	68.4	67.0	65.0
	Top-5	91.2	91.5	92.0	92.9	91.6	89.5	89.1	88.7	86.5

TABLE X: Accuraciy results from changing activation bit width for 2-Add ResNet-18

	2-bit	4-bit	8-bit	12-bit	16-bit
Top-1	54.3%	57.7%	58.2%	58.2%	58.2%
Top-5	79.8%	81.4%	82.0%	82.0%	82.0%

Thus, it is important to evaluate the accuracy achieved against the amount of resources used. To do this, we have analyzed the area consumed for different precisions of traditional xed-point training against AddNet training. Fig. 10 shows these evaluations for each network. The closer data points are to the top left corner of the graphs, the more optimized and more ef cient the method. We see that both the 2-Add and 3-Add cases show improvements over the traditional quantization methods. This demonstrates the effectiveness of our training methodology. The 4-Add case achieves the same or greater accuracy than the 8-bit but with signi cantly less resources. Additionally, for all three networks, the 2-Add and 3-Add case signi cantly improve accuracy and area over 6-bit implementations and the 2-Add case signi cantly improves accuracy and area over the 4-bit implementations. This is a very important contribution of this work: instead of reducing precision, which is a standard approach to save silicon area, our method gets better area savings and much better accuracy for all networks.

After investigating the effect of weight precision, we also analyze the effect of activation precision on both accuracy and area. Table X shows the accuracy against different sizes for w_{in} using the 2-Add multiplier coef cients for ResNet-18. We particularly analyze 2-Add ResNet-18 as it has the highest discrepancy from our 8-bit and full-precision models. As shown, increasing the activation to 16-bits does not close the accuracy gap. Observing Fig. 8, the area of the 2-Add with $w_{in} = 16$ is roughly equivalent to the 3-Add with $w_{in} = 8$. Thus, in this case, it is much more effective to use the 3-Add with $w_{in} = 8$ as the accuracy is improved.

VIII. Conclusion

In this paper we explored recon gurable constant coef cient multipliers for the inference problem in convolutional neural networks. A novel distribution matching scheme which restricts the allowable coef cient values in a computationally tractable manner is proposed, as well as a training algorithm. Our results show that this approach achieves better accuracy than networks which constrain weight parameters to have binary or ternary values, while allowing the expensive multipliers usually used in xed-point implementations to be replaced by shifts, adds and small multiplexers. Furthermore, the restricted number of possible coef cient values allows an encoding scheme to signi cantly reduce weight storage. Overall, our approach reduces mismatch between CNN computation and existing FPGA device architectures, making more ef cient implementations possible.

While we have demonstrated the bene ts of these techniques on CNNs, we expect training any type of neural network to make use of RCCMs using our approach should be explored as an alternative to simply studying the use of reduced precision arithmetic. Our technique introduces a new dimension for optimization of neural networks in which the arithmetic is directly customized, and is orthogonal to matrix decomposition and sparsity-inducing approaches. Future work will explore combining these techniques.

Acknowledgement

This work was partially funded by the AustraliaGermany Joint Research Cooperation Scheme and the German Academic Exchange Service (DAAD) under grant no. 57388068.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," Nature, vol. 521, pp. 436 EP –, 05 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14539

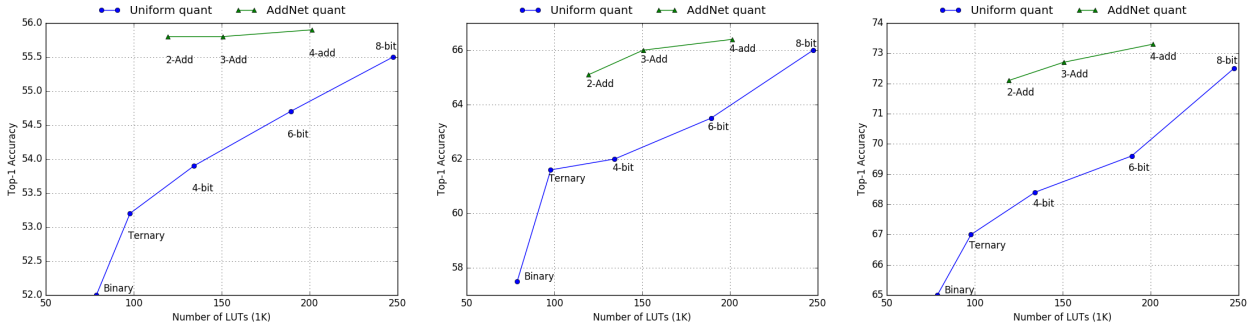
[2] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '15. New York, NY, USA: ACM, 2015, pp. 161–170. [Online]. Available: http://doi.acm.org/10.1145/2684746.2689060

[3] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 65–74. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021744

[4] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding," in 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings, 2016. [Online]. Available: http://arxiv.org/abs/1510.00149

[5] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "MobileNets: Ef cient Convolutional Neural Networks for Mobile Vision Applications," CoRR, vol. abs/1704.04861, 2017. [Online]. Available: http://arxiv.org/abs/1704.04861

[6] 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



(a) AlexNet (b) ResNet-18 (c) ResNet-50

Fig. 10: Accuracy-Area comparison of uniform and AddNet quantization for AlexNet and ResNet

[7] D. Miyashita, E. H. Lee, and B. Murmann, "Convolutional Neural Networks using Logarithmic Data Representation," *CoRR*, vol. abs/1603.01025, 2016. [Online]. Available: http://arxiv.org/abs/1603.01025

[8] D. D. Lin, S. S. Talathi, and V. S. Annapureddy, "Fixed Point Quantization of Deep Convolutional Networks," in *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ser. ICML'16. JMLR.org, 2016, pp. 2849–2858. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045390.3045690

[9] S. S. Demirsoy, A. Dempster, and I. Kale, "Design guidelines for reconfigurable multiplier blocks," in *Circuits and Systems, 2003. ISCAS '03. Proceedings of the 2003 International Symposium on*, 2003.

[10] R. I. Hartley, "Subexpression sharing in filters using canonic signed digit multipliers," *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, vol. 43, no. 10, pp. 677–688, Oct 1996.

[11] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *CoRR*, vol. abs/1512.03385, 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[12] Y. Bengio, N. Léonard, and A. C. Courville, "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation," *CoRR*, vol. abs/1308.3432, 2013. [Online]. Available: http://arxiv.org/abs/1308.3432

[13] W. Chen, J. T. Wilson, S. Tyree, K. Q. Weinberger, and Y. Chen, "Compressing Neural Networks with the Hashing Trick," in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 2285–2294. [Online]. Available: http://dl.acm.org/citation.cfm?id=3045118.3045361

[14] Y. W. Q. H. Jiaxiang Wu, Cong Leng and J. Cheng, "Quantized Convolutional Neural Networks for Mobile Devices," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[15] M. Langhammer and G. Baeckler, "High Density and Performance Multiplication for FPGA," in *IEEE Symposium on Computer Arithmetic*, 2018.

[16] Y. Fu, E. Wu, A. Sirasao, S. Attia, K. Khan, and R. Wittig, "Deep Learning with INT8 Optimization on Xilinx Devices (White Paper)," Xilinx, Inc., Tech. Rep., 2017.

[17] E. G. Walters, "Partial-Product Generation and Addition for Multiplication in FPGAs with 6-Input LUTs," *Asilomar Conference on Signals, Systems and Computers*, pp. 1247–1251, 2014.

[18] M. Kumm, S. Abbas, and P. Zipf, "An Efficient Softcore Multiplier Architecture for Xilinx FPGAs," in *IEEE Symposium on Computer Arithmetic (ARITH)*, 2015, pp. 18–25.

[19] M. Courbariaux, Y. Bengio, and J.-P. David, "BinaryConnect: Training Deep Neural Networks with Binary Weights During Propagations," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15. Cambridge, MA, USA: MIT Press, 2015, pp. 3123–3131. [Online]. Available: http://dl.acm.org/citation.cfm?id=2969442.2969588

[20] 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*, vol. abs/1606.06160, 2016. [Online]. Available: http://arxiv.org/abs/1606.06160

[21] J. Faraone, N. Fraser, M. Blott, and P. H. Leong, "SYQ: Learning Symmetric Quantization for Efficient Deep Neural Networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2018.

[22] M. S. Abdelfattah, D. Han, A. Bitar, R. DiCecco, S. O'Connell, N. Shanker, J. Chu, I. Prins, J. Fender, A. C. Ling, and G. R. Chiu, "DLA: Compiler and FPGA Overlay for Neural Network Inference Acceleration," *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 411–4117, 2018.

[23] "Xilinx DPU," accessed: 2019-05-09. [Online]. Available: https://www.xilinx.com/products/intellectual-property/dpu.html

[24] C. Ding, S. Wang, N. Liu, K. Xu, Y. Wang, and Y. Liang, "REQ-YOLO: A resource-aware, efficient quantization framework for object detection on FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: ACM, 2019, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/3289602.3293904

[25] M. Hardieck, M. Kumm, K. Möller, and P. Zipf, "Reconfigurable convolutional kernels for neural networks on FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*, 2019, pp. 43–52. [Online]. Available: https://doi.org/10.1145/3289602.3293905

[26] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions," *ACM Comput. Surv.*, vol. 51, no. 3, pp. 56:1–56:39, Jun. 2018. [Online]. Available: http://doi.acm.org/10.1145/3186332

[27] K. Abdelouahab, M. Pelcat, J. Sérot, and F. Berry, "Accelerating CNN inference on FPGAs: A Survey," *CoRR*, vol. abs/1806.01683, 2018. [Online]. Available: http://arxiv.org/abs/1806.01683

[28] K. Guo, S. Zeng, J. Yu, Y. Wang, and H. Yang, "A survey of FPGA based Neural Network Accelerator," *CoRR*, vol. abs/1712.08934, 2017. [Online]. Available: http://arxiv.org/abs/1712.08934

[29] D. J. Moss, S. Krishnan, E. Nurvitadhi, P. Ratuszniak, C. Johnson, J. Sim, A. Mishra, D. Marr, S. Subhaschandra, and P. H. Leong, "A Customizable Matrix Multiplication Framework for the Intel HARPv2 Xeon+FPGA Platform: A Deep Learning Case Study," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: ACM, 2018, pp. 107–116. [Online]. Available: http://doi.acm.org/10.1145/3174243.3174258

[30] G. Venkatesh, E. Nurvitadhi, and D. Marr, "Accelerating Deep Convolutional Networks using low-precision and sparsity," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2017, New Orleans, LA, USA, March 5-9, 2017*, 2017, pp. 2861–2865. [Online]. Available: https://doi.org/10.1109/ICASSP.2017.7952679

[31] K. Kwon, A. Amid, A. Gholami, B. Wu, K. Asanovic, and K. Keutzer, "Co-design of Deep Neural Nets and Neural Net Accelerators for Embedded Vision Applications," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: ACM, 2018, pp. 148:1–148:6. [Online]. Available: http://doi.acm.org/10.1145/3195970.3199849

[32] Y. Yang, Q. Huang, B. Wu, T. Zhang, L. Ma, G. Gambardella, M. Blott, L. Lavagno, K. Vissers, J. Wawrzynek, and K. Keutzer, "Synetgy: Algorithm-hardware Co-design for ConvNet Accelerators on Embedded FPGAs," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser.