Customizing Low-Precision Deep Neural Networks For FPGAs

Julian Faraone* Giulio Gambardella# David Boland* Nicholas Fraser# Michaela Blott# Philip H.W. Leong*

The University Of Sydney*
Xilinx Research Labs#
(julian.faraone, david.boland, philip.leong)@sydney.edu.au (giuliog, nfraser, mblott)@xilinx.com

***Abstract*—In this paper, we argue that instead of solely focusing on developing efficient architectures to accelerate well-known low-precision CNNs, we should also seek to modify the network to suit the FPGA. We develop a fully automative toolflow which focuses on modifying the network through filter pruning, such that it efficiently utilizes the FPGA hardware whilst satisfying a predefined accuracy threshold. Although fewer weights are removed in comparison to traditional pruning techniques designed for software implementations, the overall model complexity and feature map storage is greatly reduced. We implement the AlexNet and TinyYolo networks on the large-scale ImageNet and PascalVOC datasets, to demonstrate up to roughly $2\times$ speedup in frames per second and $2\times$ reduction in resource requirements over the original network, with equal or improved accuracy.**

I. Introduction

Deep Convolutional Neural Networks (CNNs) are typically designed for optimal prediction capabilities without considerations for practical implementations and hardware costs. As a result, modern CNNs are introducing increasing numbers of layers and high-dimensional filters in order to improve prediction capabilities on challenging Image Classification and Object Detection tasks. Unfortunately, this growth results in increased computational and memory requirements. Quantized Neural Networks (QNNs) are well suited to customisable hardware, such as FPGAs and ASICs, because it is possible to save silicon area with reduced precision, as well as to exploit the reduced memory bandwidth requirement. In particular, FPGA implementations of extreme forms of QNNs, such as Binary and Ternary Neural Networks (BNNs and TNNs) have been shown to achieve superior throughput and power efficiency over CPU and GPU platforms [1]. As accuracies for these networks become increasingly better with new training methods [2], network customizations are important to improving their amenability to FPGAs. Pruning methods seek to remove parameters from the network, reducing the on-chip memory, memory bandwidth and computational requirements, whilst minimizing any loss in accuracy. Coarse-grained pruning methods such as filter pruning, result in a structured sparse representation that maintains the regular data access patterns of the original network. Generally this can make use of existing optimized hardware implementations. One of the limitations of existing filter pruning methods is that they do not take into account the underlying FPGA accelerator architecture.

Altogether, our work makes the following contributions:

- We propose a novel quantization error pruning heuristic which minimizes the error in the weights rather than in the output feature maps.
- We propose a resource-aware method for customizing low precision CNNs to underlying FPGA dataflow architectures.
- We achieve the highest reported frames per second (FPS), FPS/kLUT and FPS/BRAM on the popular AlexNet.

II. Redundancies In CNNs

CNNs typically consist of convolutional and fully-connected layers. For each convolutional layer, filters are applied to all pixels of the Input Feature Maps (IFMs), with the result passed into an element-wise activation function f to compute Output Feature Maps (OFMs). The OFMs then become the IFMs for the next layer. For each fully-connected layer, all pixels of the IFMs are multiplied by weights to generate each pixel of the OFM. In this paper we use the weight and activation quantization as proposed in [2]. During training both the full precision and quantized weight values are needed; for inference only the quantized weights are required. While quantization methods reduce the number of bits for weights and arithmetic operations, pruning methods reduce the total number of weights that must be stored and arithmetic operations that must be performed. Filter pruning chooses the least important filter/s to keep and prunes out both those filters and their corresponding OFM. Filters are considered important if their removal results in accuracy degradation. However, while filter pruning reduces the total number of operations, on hardware designs, it is typically the feature map memory or available resources for processing elements that limits performance.

III. CNN Acceleration

We use the following notation to describe our CNNs. For layer j in a CNN, we assume there are I_j IFMs of dimensions $F_j \times F_j$ and N_j filters of dimensions $K_j \times K_j$ (we assume feature maps and filters of squared dimensions). The OFMs from layer j are the IFMs to layer $j+1$, and can be represented as I_{j+1}.

Various research studies have explored how to take advantage of these quantization methods in order to create FPGA-based CNN accelerators with high throughput and low power [3]. A Framework For Fast, Scalable Binarized Neural Network (FINN) [4], demonstrated the advantages of fitting models in on-chip memory. The methods introduced in this paper could be applied to improve the performances of many of these implementations for minimal or no accuracy loss. We implement a pruning strategy which focuses on customizing the pruning process for FPGA architectures. FPGAs pose unique considerations over other hardware platforms as the amount of layer unrolling is set by the designer and resources can be arbitrarily allocated. Furthermore, since the highest performance implementations store intermediate FMs in on-chip BRAM, only considering the model-size is misleading to the overall hardware savings from pruning.

IV. Hardware-Aware Pruning

A. Layer Selection

To fit a design on the FPGA, it becomes necessary to re-use the same PEs. To enable this, we must add data buffers to hold IFMs and OFMs between layers, as shown in Figure 1. The total number of operations for layer j is given by (1). We assume that a layer has PE_j processing elements that can be re-used to perform these operations. The total number of times that these PEs are reused is given by (2). Note that for maximum efficiency we must match throughput between layers, to do this, the reuse should be the same for all layers. In a fully pipelined implementation $Reuse = 1$, in this case there is no need to use any memory: OFMs from one layer can go directly to the SWUs of the next layer. If $Reuse = 2$, then half of the time, OFMs must be stored, half of the time they can go directly to the Sliding Window Units (SWUs) of the next layer. As we increase the reuse, we reduce the PE requirements, but increase the RAM requirements for intermediate buffers. We can model the cost of BRAMs for OFMS per layer using (3), where B_w denotes the number of words stored in a BRAM respectively.

$$OPS_j = K_j \times K_j \times I_j \times N_j \times F_{j+1} \times F_{j+1} \quad (1)$$

$$Reuse = \lceil \frac{OPS_j}{PE_j} \rceil \quad (2)$$

$$BRAM_{IFM_{j+1}} = \lceil \frac{I_j \times F_{in} \times F_{in} \times (Reuse - 1)}{Reuse \times B_w} \rceil \quad (3)$$

To help us create a working hardware design we can utilize these equations, along with a model of the resources cost per PE, the resource cost per SWU and the BRAM cost to store all network weights. We then follow a basic heuristic for hardware-optimized pruning, given in Algorithm 1. This heuristic helps us to determine which layer to prune. We begin with a fully pipelined design ($Reuse = 1$). If this does not fit on the FPGA, the design is constrained by the resources for PEs. We therefore have the option of pruning the layer utilizing the most resources for PEs, or increasing $Reuse$. After this, if the design is constrained by the resources for PEs, we have the same options; if the design is limited by BRAMs, we can prune the layer utilizing the most BRAMs (FM + Weight memory).

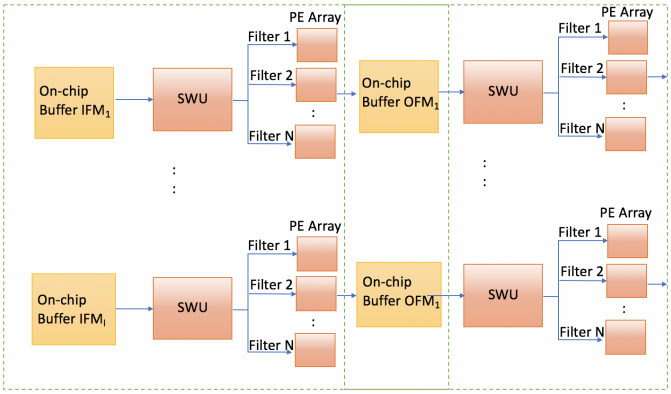

Fig. 1. Advantage of reading from on-chip BRAMs

Algorithm 1 Pruning Process For FPGA implementation

1. Initialize:
Choose PE size such that reuse = 1.
2. Iteratively prune filters:
while accuracy change $\leq \eta$ **do**
  Prune filters by 10% using (4)
  Re-train
  Save Model
end while
3. Reuse:
while design exceeds available FPGA resources **do**
  Increase hardware reuse
end while
4. Model Finetuning:
Add/Remove filters and ensure they are a multiple of the number of PEs
Re-train
5. Check:
if Accuracy satisfies η **then**
  continue
else
  Go to Step 3 using saved model from previous iteration.
end if
6. Deploy

B. Model-finetuning

Pruning layers involves the removal of feature maps. To ensure efficient use of the underlying hardware, we prune layers so that the remaining number of feature maps in a layer is a multiple of the number of PEs assigned to that layer, i.e. $N\%PE = 0$. In addition, we also attempt to *increase* the number of feature maps in other resource-inexpensive layers such that $N\%PE = 0$. This is only the case if it doesn't impinge on the desired performance of the design. Once again, this is to ensure maximum efficiency. This can translate into a significant increase in the accuracy of our design, or recover some of the accuracy that is lost by pruning the most expensive layer. But importantly, comes with a minimal increased cost in terms of resources: there is only a slight increase in weight

memory/resources. There is also a slight increase in power requirements as more of the circuit will be active. Note that on other architectures such as GPUs or CPUs, this decision is unlikely to be taken as it would increase the workload; instead the ideal course of action is to simply prune the overall network to reduce the total number of operations.

C. Quantization Error Pruning

Once the layers to prune have been chosen, we must choose which filters to prune in a given layer. Pruning approaches attempt to remove weights or filters with minimal reduction in accuracy. With traditional floating point network pruning, error in the OFMs is what impinges overall accuracy. It follows that a typical pruning technique is to select weights or filters which have the lowest magnitudes and hence contribute to the smallest activation outputs. Unfortunately, this technique does not easily extend to extremely low-precision QNNs. If two weights have the same sign, they have the same quantized value and hence equal contributions to the OFMs. We hypothesize that for highly quantized representations, such as BNNs and TNNs, it is the quantization error in the quantized weight values that impinges overall accuracy. As such, we rank the importance of filters based on the highest accumulated mean-squared quantization error (MSQE) for each filter, as described by (4), where $n = K_j \times K_j \times I_j$.

$$MSQE_N = \frac{1}{n}\sum_{z=1}^{n}(q_z - w_z)^2 \tag{4}$$

D. Data Fine-tuning

To avoid excessive training times, and ensure minimal accuracy loss, after filters are selected to prune, we remove them to reconstruct the network with its new customized configuration by initializing the weights with values from the saved filters. We then retrain using a quarter of the initial training epochs to fine-tune weight values and recover the accuracy for the new CNN configuration. As such, we do this pruning process iteratively for some predefined percentage of filters in each iteration. In our work, $\approx 10\%$ of filters are pruned per iteration. In the last pruning iteration we ensure $N\%PE = 0$, as discussed in Section IV-B.

V. Experimental Setup

A. Networks

We evaluate our methods on 2 networks. Firstly, we use an AlexNet-variant inspired by DoReFa-Net [2] which has 1-bit weights and 2-bit activations. This is used for classifying the ImageNet dataset which has 224×224 input image sizes. Secondly, we consider a fully-convolutional network, TinyYolo [5] and binarize the network weights and use 3-bits for activations for this second benchmark. This network is used on the Pascal VOC dataset for Object Detection which has a 418×418 input image size. The TinyYolo model is much more compact in terms of weight memory footprint than AlexNet, although has a larger total number of operations and input image size. For both networks, we quantize the first and last layers to 8-bit representations and the activations bit widths for all layers is the same. Finally, we also quantize the inputs to 8-bit values, for no accuracy loss.

B. Computing Core

To investigate the effectiveness of our pruning method, we use the FINN [4] hardware library in Vivado HLS. The FINN architecture uses SWUs to feed so-called Matrix-Vector Threshold Units (MVTU) for CONV or FC layers. SWUs can also feed Pooling Units (PU). There are three key parameters for FINN: PEs, SIMD lanes and Matrix-Multiple Vector (MMV) length. PEs refer to the number of OFMs evaluated in parallel, SIMD refers to the number of parallel IFMs processed in each PE, MMV controls to the amount of output pixels evaluated in parallel. We extend the PEs in the MVTU to arbitrary precision activations for the middle layers, which replaces the XNOR-popcount from the PEs in FINN by either an addition or subtraction (ADD/SUB) depending on the sign of the weight. For the 8-bit input vector and 8-bit weights in the first layer, DSPs are used for the MAC before being fed into the thresholding unit. The overall system designs for the architectures discussed in this work use our MVTU extensions to compute the CONV layers.

VI. Results

A. Streaming Dataflow

1) System Design: For dataflow architectures, FINN allows the selection of parallelism ($P = SIMD \times PE \times MMV$) for each layer depending on the number of operations. The overall throughput of the network is dependent on the layer which requires the largest number of cycles to compute. For the AlexNet implementation, a large proportion of total operations is done in the 2nd and 3rd layers. Additionally, the first layer requires more resources per MAC operation, as it is computed with 8-bit inputs and weights, hence requiring DSPs for implementation and more memory resources for weight storage. Hence, the first three layers are chosen to be pruned equally for resource and throughput improvements. This is done iteratively for a total of 40% of filters. For the first layer, the number of IFMs is 3, meaning $SIMD \leq 3$. The number of PEs and MMV is restricted by the number of OFMs and OFM dimension respectively, hence $MMV \leq 54$ and $PE \leq 60$ (as the OFMs are reduced to 60 after pruning and model fine-tuning). We set $SIMD = 3, PE = 60$ & $MMV = 18$ as the throughput mustn't exceed the latency of the SWU to construct the image matrix. To achieve load balancing for all succeeding layers, the resources are allocated such that the estimated cycles matches this first layer. In our system design, the last CONV layer writes to the host memory and FC layers are computed on the host CPU. This is due to their large BRAM and low operations requirement. A large proportion of the operations and weight + FM memory in the original TinyYolo network is in the last two quantized layers. Hence, we prune these layers to improve the load-balancing and also reduce the number of BRAMs. We also double the number of second layer filters, as increasing the number of PEs in that

TABLE I
ALEXNET AND TINYYOLO IMPLEMENTATION RESOURCE USAGE

AlexNet	LUTs	DSPs	BRAMs	Freq.	FPS	Acc.
Original	375,037 57%	2,693 49%	1,527 35%	159	3,265	50.1
Pruned-30%	228,104 34%	1,938 35%	1,057 25%	172	3,530	50.3
Pruned-40%	188,924 25%	1,698 25%	955 20%	185	3,797	50.1
TinyYolo						
Original	179,265 27%	500 9%	2,731 63%	233	1,189	47.8
Pruned-50%	234,883 35%	189 3%	1,930 45%	240	1,226	48.5
Available	663,360	5,520	4,320			

TABLE II
SCALING UP PARALLELISM FOR PRUNED ALEXNET

	Original	Pruned-30%	Pruned-40%
Freq (MHz)	159	130	150
LUTs	375,037	410,073	302,325
BRAMs	1,527	1,333	1,156
DSPs	2,693	3,772	2,693
FPS	3,265	5,359	6,172
FPS/kLUT	8.70	13.14	20.44
FPS/BRAM	2.14	4.02	5.39

layer from 32 to 64 imposes minimal resource impact and helps preserve accuracy during re-training.

2) Performance: We measure the performance of both pruned and non-pruned versions of AlexNet by implementing the dataflow architecture on the Xilinx KU115 board. For the pruned topologies, there are two alternatives for improving our hardware; 1) By fixing the model's parallelism, we can reduce concurrency and save computational resources. This translates to a greater ability to fit the model on low-cost FPGAs and achieve higher frequencies. 2) Increasing the parallelism of our original architecture, which reduces latency and improves our FPS. Following 1), we maintain the same parallelism for pruning AlexNet at 30% & 40% as displayed in Table I. As the frequency increases, the frame rate also improves whilst resources are significantly reduced. For TinyYolo, there is a slight increase in FPS whilst resources are again reduced. We also note our pruning strategy improves the accuracy of TinyYolo by 0.7mAP and AlexNet-30% by 0.2%. For 2) we scale-up the parallelism our implementation. We maintain similar frequencies to the original AlexNet network and achieve roughly a $2\times$ speedup over the original network as displayed in Table II with a significant reduction in resources and the same accuracy. Scaling for the original network was unable to achieve the FPS reported for the pruned topologies due to device resource constraints.

VII. RELATED WORK

The W/Act Precision for AlexNet in Table III metric is for the weight and activation precision. To calculate our FPS/BRAM against previous implementations, we assume a direct conversion of M20k to BRAM(18k). Also, to calculate our FPS/kLUT we assume 8-input ALMs and 6-input LUT units are equivalent. It is evident our design achieves a highly superior $3.8\times$ improvement in FPS, $4.9\times$ in FPS/kLUT and $7.5\times$ in FPS/BRAM over previous state-of-the-art implementations. This demonstrates the largely efficient use of resources in our design as our pruning strategy is able to remove redundant filters which don't contribute much to the overall accuracy.

TABLE III
COMPARISON TO PREVIOUS ALEXNET IMPLEMENTATIONS

	Li16 [6]	Aydonat17 [7]	Moss17 [3]	**Ours**
Device	VC709	Aria 10	Aria 10	KU115
Freq	156	303	312.5	150
LUTs	273,805	246k (ALM)	427.2k (ALM)	302,325
BRAMs	1,913	2487 (M20k)	2000 (M20k)	1,156
DSPs	2,144	1,476	1518	2,693
W/Act.	16/16-bit	16/16-bit	1/1-bit	1/2-bit
Acc.	-	56.0	44.2[1]	**50.1**
FPS	391	1,020	1,610	**6,172**
FPS/kLUT	1.43	4.14	3.77	**20.44**
FPS/BRAM	0.20	0.37	0.72	**5.39**

VIII. CONCLUSIONS

We presented a hardware-aware filter pruning framework for customizing low precision CNNs to underlying FPGA architectures. Following the selection of resource-heavy layers, an effective heuristic was established which utlizes the information from the reduced precision weights to measure the relative importance of each filter. Reducing computational requirements whilst maintaining the regular data access patterns of dense matrices was demonstrated by applying it to significantly improve the state-of-the-art in terms of FPS, FPS/kLUT and FPS/BRAM for the AlexNet and TinyYolo networks.

IX. ACKNOWLEDGMENTS

This research was supported under the Australian Research Councils Linkage Projects funding scheme (project number LP130101034) and Zomojo Pty Ltd.

REFERENCES

[1] E. Nurvitadhi, D. Sheffield, J. Sim, A. K. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC," *Int. Conf. on Field-Programmable Technology*, pp. 77–84, 2016.

[2] 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.

[3] D. J. M. Moss, E. Nurvitadhi, J. Sim, A. K. Mishra, D. Marr, S. Subhaschandra, and P. H. W. Leong, "High performance binary neural networks on the Xeon+FPGA^{TM} platform," in *FPL*. IEEE, 2017, pp. 1–4.

[4] 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," *CoRR*, vol. abs/1612.07119, 2016. [Online]. Available: http://arxiv.org/abs/1612.07119

[5] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016.

[6] H. Li, X. Fan, L. Jiao, W. Cao, X. Zhou, and L. Wang, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," *Proc. Int. Conf. on Field Programmable Logic and Applications*, pp. 1–9, 2016.

[7] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An opencl(tm) deep learning accelerator on arria 10," *CoRR*, vol. abs/1701.03534, 2017.

[1] Accuracy obtained from 1/1-bit AlexNet in [**?**]