Energy Efficient ASIC/FPGA Neural Network Accelerators

by

Shreyas Kolala Venkataramanaiah

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved February 2022 by the
Graduate Supervisory Committee:

Jae-sun Seo, Chair
Yu Cao
Chaitali Chakrabarti
Deliang Fan

ARIZONA STATE UNIVERSITY

May 2022

ABSTRACT

Convolutional neural networks(CNNs) achieve high accuracy on large datasets but requires significant computation and storage requirement for training/testing. While many applications demand low latency and energy-efficient processing of the images, deploying these complex algorithms on the hardware is a challenging task.

This dissertation first presents a compiler-based CNN training accelerator using DDR3 and HBM2 memory. An optimized RTL library is implemented to perform training-specific tasks and an RTL compiler is developed to generate FPGA-synthesizable RTL based on user-defined constraints. High Bandwidth Memory(HBM) provides efficient off-chip communication and improves the training performance. The impact of HBM2 on CNN training workloads is analyzed and compressively compared with DDR3. For training ResNet-20/VGG-like CNNs for the CIFAR-10 dataset, the proposed CNN training accelerator on Stratix-10 GX FPGA(DDR3) demonstrates 479 GOPS performance, and on Stratix-10 MX FPGA(HBM) shows 4.5/9.7 X energy-efficiency improvement compared to Tesla V100 GPU.

Next, the FPGA online learning accelerator is presented. Adopting model segmentation techniques from Progressive Segmented Training(PST), the online learning accelerator achieved a 4.2X reduction in training latency.

Furthermore, this dissertation presents an 8-bit floating-point (FP8) training processor which implements (1) Highly parallel tensor cores that maintain high PE utilization, (2) Hardware-efficient channel gating for dynamic output activation sparsity (3) Dynamic weight sparsity based on group Lasso (4) Gradient skipping based on FP prediction error. The 28nm prototype chip demonstrates significant improvements in FLOPs reduction (7.3$\times$), energy efficiency (16.4 TFLOPS/W), and overall training latency speedup (4.7$\times$) for both supervised training and self-supervised training tasks.

In addition to the training accelerators, this dissertation also presents a CNN inference accelerator on ASIC(FixyNN) and FPGA(FixyFPGA). FixyNN consists of a fixed-weight feature extractor that generates ubiquitous CNN features and a conventional programmable CNN accelerator. In the fixed-weight feature extractor, the network weights are hard-coded into hardware and used as a fixed operand for the multiplication. Experimental results demonstrate FixyNN can achieve very high energy efficiencies up to 26.6 TOPS/W, and FixyFPGA achieves 2.34$\times$ higher GOPS on ImageNet classification.

In summary, this dissertation comprehensively discusses novel architectures of high-performance and energy-efficient ASIC/FPGA CNN inference/training accelerators.

ACKNOWLEDGMENTS

I owe many thanks to my wife - Rochana Shreyas, my parents - Janaki C.V. and Venkataramanaiah K.V, and my sister - Shilpa Pradeep, for their unwavering support and love throughout my Ph.D. education. It is to them I dedicate this dissertation.

I would like to sincerely thank my advisor and committee chair, Dr. Jae-sun Seo, for developing my research skills, paper writing, time management and provide valuable guidance and encouragement throughout my Ph.D. studies. I appreciate his patience and assistance in guiding the research projects and giving his insightful recommendations and ideas at each step of the work. It has been such a pleasure and honor to be one of his students and learn from him.

I want to extend my gratitude to Dr. Yu Cao, Dr. Chaitali Chakrabarti, and Dr. Deliang Fan for their guidance in my research projects, comments and suggestions on my dissertation and for taking the time to serve as committee members. I thank Dr. Aravind Dasu and Dr. Eriko Nurvithadi from Intel Corporation, Dr. Paul Whatmough, Chuteng Zhou and Patrick Hansen from ARM research for their collaborative research opportunity and invaluable inputs during my summer research internships.

I am indebted to my colleagues, Deepak Vinayak Kadetotad, Shihui Yin, Gaurav Srivastava, Minkyu Kim, Han-Sok Suh, Jian Meng, Sai Kiran Cherupally, Jyotishman Saikia, Xiaocong Du, Zheng Li and Amitesh Sridharan for their invaluable contribution, discussions, and support throughout my study. I must also thank my graduate advisor Toni Mengert for her active help with all the administrative procedures. Thanks to my friends, Gunnam Sridhar, Shanshank T.R., Amrut Deshpande, Rahul Rao, Shreyas Ravishankar, Vinay Suresh, Dheeraj Nag, Nikhitha Prahlad, Nagarjun Aswath, Vivek Rai and many others, who were also crucial in the successful realization of this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Figure Page

Chapter 1

INTRODUCTION

1.1 CNN Training

Convolutional neural networks (CNNs) have seen exceptional success in numerous cognitive applications such as image classification and object detection. However, the immense amount of computations and parameters in state-of-the-art CNN algorithms have posed significant challenges for energy-efficient CNN hardware designs. In particular, it requires enormous memory and computes resources to perform CNN training. To support the high computation requirement, training is typically conducted in datacenters with high-end GPUs. However, the GPUs ' high power consumption and low mobility make them non-ideal for on-device learning with a limited power budget. In addition, although GPUs provide very high throughput on CNN training with large batch sizes, they suffer from low utilization/throughput for smaller batch sizes Chundi *et al.* (2019). Nowadays, training on resource-constrained platforms is also becoming more crucial for networks that need user's private data. However, executing computation-/memory-intensive training tasks on hardware platforms with power and resource constraints becomes very challenging. Low-batch training significantly reduces memory requirement and unlocks opportunities for FPGAs, which provide higher configurability for custom architecture and better energy efficiency than high-power GPUs. Training on edge FPGA devices also reduces latency overhead (due to limited data exchange with the cloud server), prevents privacy/security problems, and is well-suited to exploit new features such as low precision training, sparse weight updates, online learning, etc. They also offer a large volume of off-chip memory (DRAM) and shorter design time when compared to ASIC designs. However, training CNNs on FPGAs is a challenging task for two reasons: (1) it demands high memory bandwidth, which is the primary limiting factor in many accelerators Wissolik *et al.* (2017); Deo *et al.* (2017), (2) complexity arises in implementing a generalized flexible training accelerator supporting new advancements in CNN training algorithms and (3) compared to inference, the training phase involves a much higher number of operations (>3X) with increased complexity Choi *et al.* (2018).

To aid training on edge FPGA device, ons the algorithm side, researchers have proposed low batch trainingWu and He (2018); Masters and Luschi (2019); Ioffe (2017), low-precision training Gupta *et al.* (2015); Köster *et al.* (2017), frequency domain training Ko *et al.* (2017), and sparse weight update Sun *et al.* (2017). Techniques such as sparse weight update introduce irregular parallelism, making it more suitable for flexible FPGAs compared to GPUs Nurvitadhi *et al.* (2017). Furthermore, FPGAs are well-suited for low-precision DNN algorithms as it provides an enormous improvement in throughput and energy efficiency with low-precision arithmetic Wang *et al.* (2019). On the hardware side, training deep neural networks on FPGA platform has not been investigated comprehensively.

For CNN inference tasks, a number of FPGA accelerators have been proposed Zhang *et al.* (2015); Ma *et al.* (2017); Zhang and Li (2017); Zeng *et al.* (2018); Yang *et al.* (2019). Training accelerators for non-CNN applications were proposed in Zeng and Prasanna (2020); Liu *et al.* (2018); Gomperts *et al.* (2011); Rafael *et al.* (2005); Zhou *et al.* (2020). Only a few prior works presented a CNN training accelerator supporting all three phases of training Venkataramanaiah *et al.* (2019a); Nakahara *et al.* (2019); Luo *et al.* (2020), but these works still did not include back-propagation of either residual connections or stride-2 convolutions that are necessary for modern CNNs. Furthermore, none of the aforementioned FPGA works studied the use of high bandwidth memory, which is critical for CNN training.

For CNN training, ASIC accelerators are also receiving increasing attention because of their high energy efficiency and performance. Most of the ASIC DNN accelerators for edge or mobile applications have supported only DNN inference. Nowadays, on-chip learning processors have huge research interest in both academia and industry as they help to personalize the edge devices without sending the data to cloud. This reduces the privacy/security risks and also reduces the latency by performing computations on-chip.

The key contributions of this thesis to accelerate CNN training are

- We present a comprehensive investigation of CNN training operations and challenges in FP, BP and WU stages.
- To the best of our knowledge, we present the first FPGA accelerator for CNN training that fully utilizes high bandwidth memory (HBM2) and executes end-to-end CNN training.
- We developed a training-specific RTL module library and an RTL compiler to automatically implement CNN training accelerator with 16-bit fixed-point precision and 16-bit floating point precision.
- A configurable FPGA hardware is presented for FP, BP and WU phases of the entire CNN training process using SGD with momentum.
- Our programmable FPGA accelerator reads high-level descriptions of CNNs (similar to TensorFlow/PyTorch) including those with residual connections and stride-2 convolutions, and automatically generates RTL for synthesis.
- We analyze the impact of HBM2 on CNN training workloads, provide a compre-

hensive comparison with DDR3, and discuss the strategies to efficiently use the HBM2 features for enhanced performance.

- Our accelerator using Intel Stratix-10 (S-10) MX FPGA with HBM2 is evaluated for ResNet-20 and VGG-like CNNs for CIFAR-10 dataset, achieving up to 14X improvement in energy-efficiency, compared to Tesla V100 GPU.
- Our accelerator using Intel Stratix 10-GX FPGA is evaluated on training three different CNNs for CIFAR-10 dataset, achieving up to 479 GOPS of throughput.
- We present an 8-bit floating-point (FP8) ASIC training processor which implements (1) highly parallel tensor cores (fused multiply-add trees) that maintain high utilization throughout forward propagation (FP), backward propagation (BP), and weight update (WU) phases of the training process
- The 28nm prototype chip demonstrates large im-provements in FLOPs reduction (7.3×), energy-efficiency (16.4 TFLOPS/W), and overall training latency speedup (4.7×), for both supervised training and self-supervised training tasks

1.2 CNN Inference

Over the past few years, convolutional neural network (CNN) approaches have rapidly displaced traditional hand-crafted feature extractors, such as Haar Viola and Jones (2004) and HOG Dalal and Triggs (2005). Mobile devices exhibit constraints in the energy and silicon area that can be allocated to CV tasks, which limits the adoption of CNNs at high resolution and frame-rate (e.g. 1080p at 30 FPS). This results in a gap in energy efficiency between the requirements for real-time CV applications and the power constraints of mobile devices. In this thesis we describe **FixyNN** and **FixyFPGA** , a hardware/CNN co-design approach to CNN inference for CV on mobile devices.

FixyNN divides a CNN into two parts.The front-end layers are implemented as a heavily optimized *fixed-weight feature extractor (FFE)* hardware accelerator. The second part of the network is unique for each dataset, and hence needs to be implemented on a canonical programmable CNN hardware accelerator Nvidia (2019); Arm (2019). Following this system architecture, FixyNN diverts a significant portion of the computational load from the CNN accelerator to the highly-efficient FFE, enabling much greater performance and energy efficiency.

While FixyNN only employed an FFE for the early layers of CNNs for an ASIC design, in FixyFPGA, we employ such fixed-weight scalers for the entire CNN layers for an FPGA design. By mapping hard-coded weights in the ALMs of the FPGA, we perform CNN inference of all layers in a fully-parallel, fully-pipelined manner. Contrary to the notion that element-wise sparsity is inefficient for hardware design, one important advantage of the fixed-weight FPGA design (FixyFPGA) is that, element-wise pruning of DNNs can be seamlessly integrated with FixyFPGA design with very high efficiency. This is because pruning out weight elements is equivalent to removing the corresponding hardware operands without introducing any index overhead. This enables us to exploit the high amount of sparsity achievable by the element-wise pruning algorithms.

Overall, the main contributions of this these for CNN inference are:

- A description of a hardware accelerator architecture for the fixed-weight feature extractor (FFE), including a survey of the potential optimizations.
- An open-source tool-flow DeepFreeze (2018) for automatically generating and optimizing an FFE hardware accelerator from a TensorFlow description.

- Present results that compare FixyNN against a conventional baseline at iso-area.
- We present FixyFPGA, a fully-parallel, fully-pipelined, and pruning-friendly FPGA-based CNN accelerator design based on fixed hard-coded weights.
- We investigate implementing a number of DNN models with different widths and compression ratios with the fixed-weight scheme onto a single Intel Stratix-10 FPGA chip without any DRAM access.
- We analyze the algorithm and hardware results of DNNs for both image classification tasks (ImageNet, TinyImageNet, and CIFAR-10 datasets).

1.3 Thesis Organization

The outline of this thesis is as follows:

- **Chapter 2** presents an automatic compiler-based FPGA accelerator for CNN training. This chapter explains the RTL compiler design and hardware architecture of FPGA based CNN training accelerator using 16-bit fixed-point precision and DDR3 as the off-chip memory. The proposed accelerator is implemented on Intel Stratix-10GX FPGA. The results are comprehensively discussed and compared with prior works.
- **Chapter 3** presents an FPGA-based Low-Batch Training Accelerator for Modern CNNs Featuring High Bandwidth Memory. This chapter describes a CNN training accelerator that uses HBM as off-chip memory and supports stride-2 convolutions, residual connections. In addition, it provides the analysis of the impact of HBM2 on CNN training workloads, provides a comprehensive comparison with DDR3, and discusses the strategies to use the HBM2 features for enhanced performance efficiently.

- **Chapter 4** presents an online learning FPGA accelerator on Intel Stratix-10 MX FPGA that exploits the hardware benefits of progressive segmented training (PST).
- **Chapter 5** presents a 28nm 8-bit Floating-Point Tensor Core based CNN Training Processor with Dynamic Activation/Weight Sparsification. The training processor follows a four-core architecture supporting FP8/FP16 precision and highly optimized for energy efficient sparse CNN training.
- **Chapter 6** presents FixyNN: efficient hardware for mobile computer vision via transfer learning. This chapter details a novel design technique using a fixed-feature extractor to develop a high throughput energy-efficient CNN inference ASIC accelerator.
- **Chapter 7** presents FixyFPGA: an efficient FPGA accelerator for deep neural networks with high element-wise sparsity and without external memory access. This chapter explains the adaptation of the fixed-feature extractor technique to design a fully parallel, fully-pipelined, and pruning-friendly CNN inference FPGA accelerator without external memory access.
- **Chapter 8** concludes the dissertation.

Chapter 2

AUTOMATIC COMPILER BASED FPGA ACCELERATOR FOR CNN TRAINING

Training of convolutional neural networks (CNNs) on embedded platforms to support on-device learning is earning vital importance in recent days. Designing flexible training hardware is much more challenging than inference hardware, due to design complexity and large computation/memory requirement. In this work, we present an automatic compiler based FPGA accelerator with 16-bit fixed-point precision for complete CNN training, including Forward Pass (FP), Backward Pass (BP) and Weight Update (WU). We implemented an optimized RTL library to perform training-specific tasks, and developed an RTL compiler to automatically generate FPGA-synthesizable RTL based on user-defined constraints. We present a new cyclic weight storage/access scheme for on-chip BRAM and off-chip DRAM to efficiently implement non-transpose and transpose operations during FP and BP phases, respectively. Representative CNNs for CIFAR-10 dataset are implemented and trained on Intel Stratix 10 GX FPGA using proposed hardware architecture, demonstrating up to 479 GOPS performance.

2.1 Introduction

CNNs have shown tremendous performance in many practical tasks including computer vision Hu *et al.* (2018) and speech recognition Zhang *et al.* (2016). Deep CNNs achieve high accuracy on large datasets, but an enormous amount of computation is required for training such networks. To support the high computation requirement, training tasks have been typically performed on datacenters with high-end GPUs. Nowadays, training on resource-constrained platforms is becoming more crucial for training networks with each user's private data. However, executing computation-/memory-intensive training tasks on hardware platforms with power and resource constraints become very challenging. This gives an opportunity to map these algorithms on FPGAs, which provide high configurability and power-efficiency compared to those of GPUs. They also provide a large volume of off-chip memory (DRAM) and shorter design time when compared to ASIC designs.

For CNN inference tasks, a number of FPGA accelerators have been proposed Zhang *et al.* (2015); Ma *et al.* (2017); Zhang and Li (2017); Zeng *et al.* (2018); Yang *et al.* (2019). However, training deep neural networks on FPGA platform has not been investigated comprehensively. Compared to inference, the training phase involves a much higher number of operations (>3X) with increased complexity Choi *et al.* (2018). The training phase also involves high intermediate data volume, necessitating high memory bandwidth and large storage. GPUs have been the de-facto for training tasks to meet immense computation requirements. However, GPUs' energy-efficiency is poor Jouppi *et al.* (2017c), and they are not well-suited for on-device learning with limited power budget.

To address this issue on the algorithm side, researchers have proposed low-precision training Gupta *et al.* (2015); Köster *et al.* (2017), frequency domain training Ko *et al.* (2017), and sparse weight update Sun *et al.* (2017). Techniques such as sparse weight update introduce irregular parallelism, making it more suitable for flexible FPGAs compared to GPUs Nurvitadhi *et al.* (2017). FPGAs are well-suited for low-precision DNN algorithms as it provides large improvement in throughput and energy efficiency with low-precision arithmetic Wang *et al.* (2019). To that end, implementing configurable training hardware on FPGA becomes crucial to exploit these algorithmic advances.

On the hardware side, several prior FPGA works have implemented training of fully-connected neural networks Liu *et al.* (2018); Gomperts *et al.* (2011); Rafael *et al.* (2005). A floating-point FPGA accelerator Liu *et al.* (2017) reported training of small CNNs using an uniform computation structure with a fixed number of multiply-and-accumulate (MAC) units. F-CNN Zhao *et al.* (2016) presented a training framework where convolutions are done in FPGA and weight updates are performed in CPU. TrainWare Choi *et al.* (2018) implemented dedicated hardware for weight update using a fixed $N_{kx}{\times}N_{ky}$ MAC array as the local gradients window is reused only $N_{kx}{\times}N_{ky}$ times during weight gradient computation. However, this is not suitable for FP/BP convolutions where there exists more kernel reuse. DeepTrain Kim *et al.* (2018) presents an embedded platform for DNN training, but does not include back-propagation of pooling layers and DNN weight updates, which needs significant memory access. Overall, these works have not presented a compiler-based FPGA accelerator that supports all phases of training for various CNNs. Designing a standalone FPGA accelerator for CNN training involves managing limited memory resources to support batch operations and different CNN configurations.

In this work, we propose a flexible FPGA accelerator that performs stochastic gradient descent (SGD) based training of various CNNs. We extracted and designed training-specific operations and then developed a library based automatic RTL compiler to flexibly support training operations with different sizes of CNNs. The user provides the high-level CNN network configurations along with the design variables to characterize FPGA hardware usage to the RTL compiler. The RTL compiler generates a FPGA compatible training accelerator based on the user's requirements. The key contributions of this work are:

- We present a comprehensive investigation of CNN training operations and challenges in FP, BP and WU stages.
- We developed a training-specific RTL module library and an RTL compiler to automatically implement CNN training accelerator with 16-bit fixed-point precision.
- A configurable FPGA hardware is presented for FP, BP and WU phases of the entire CNN training process using SGD with momentum.
- Our accelerator using Intel Stratix 10-GX FPGA is evaluated on training three different CNNs for CIFAR-10 dataset, achieving up to 479 GOPS of throughput.

2.2 CNN Training Algorithm

Fig. 1 illustrates the dataflow of stochastic gradient descent (SGD) based weight update for a simple 2C-2P-1FC CNN model. The CNN design variables and naming conventions used throughout this chapter are described in Table 1. Output activation value $o^l_{x,y}$ is given by Eq. (2.1), where $w^l_{x,y}$ are kernel values and $a^{l-1}_{x,y}$ are activations

Table 1. CNN design variables

	Kernel size width/height	Output feature map width/height/depth	Input feature map width/height/depth
Convolution dimensions	N_{kx}, N_{ky}	N_{ox}, N_{oy}, N_{of}	N_{ix}, N_{iy}, N_{if}
Loop unroll factors	P_{kx}, P_{ky}	P_{ox}, P_{oy}, P_{of}	P_{ix}, P_{iy}, P_{if}

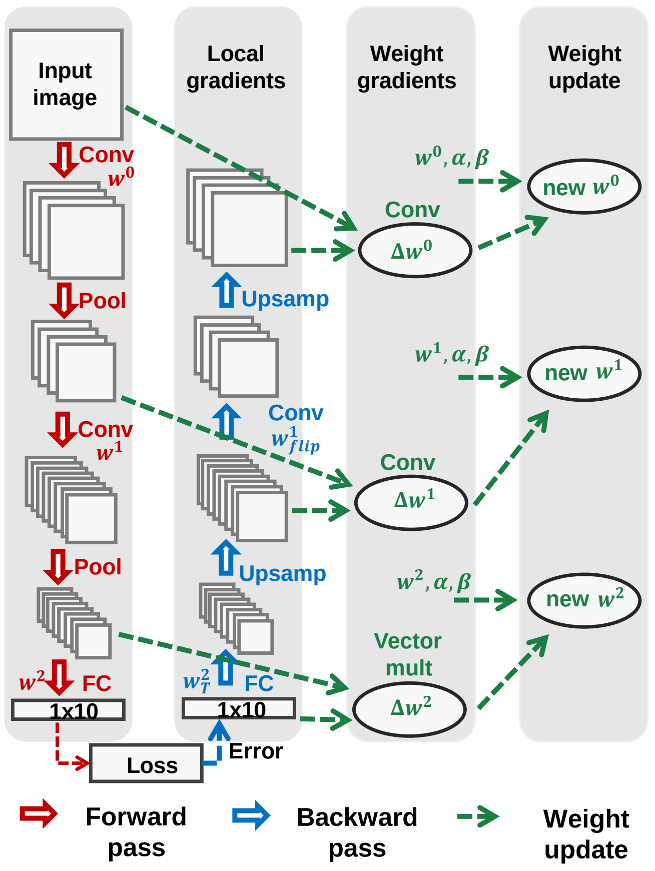


Figure 1. SGD based CNN training dataflow illustrated for a simple 2C-2P-1FC model.

from layer $l-1$.

$$o^l_{x,y} = \sum_{x'} \sum_{y'} w^l_{x,y} a^{l-1}_{(x+x'),(y+y')} \tag{2.1}$$

In supervised training, each input is associated with a label. After the completion of the forward pass, the performance of the network is estimated using a cost function. Eq. (2.2) shows a quadratic cost function of output layer L, where a_i is the obtained output value and y_i is the label. The derivative of the cost function with respect to output is also given in Eq. (2.2).

$$C = \frac{1}{2}\sum_{i}^{L}(a_i - y_i)^2, \quad \frac{\partial C}{\partial a^L_i} = (a_i - y_i) \tag{2.2}$$

Error values are back-propagated to all hidden layers and the required deviation of weight parameters to minimize the error is calculated. The derivative of the cost function with respect to weight parameters provides the required deviation for the weight parameters Δw to minimize the error. By applying the basic chain rule, weight deviation Δw can be obtained by convolving the derivative of the cost function with layer output activations, which we term as local gradients and feedforward activations. Local gradients of layer (l) can be obtained by convolving the gradients of the previous layer $(l-1)$ with its own convolution kernel.

During these backward convolutions, the original kernel tensors are flipped. The differences of BP and FP convolutions are shown in Fig. 2. Fig. 2a shows FP convolutions of input image with three input channels ($N_{if} = 3$) and two sets of kernels to obtain two output feature maps ($N_{of} = 2$). During BP, convolutions are performed using local gradients of previous layer and FP kernels, where the number of input channels and convolution depth are interchanged. In Fig. 2b, it is shown that $N_{if} = 2$ and $N_{of} = 3$. Flipped kernels are used in BP convolutions to compute the local gradients.

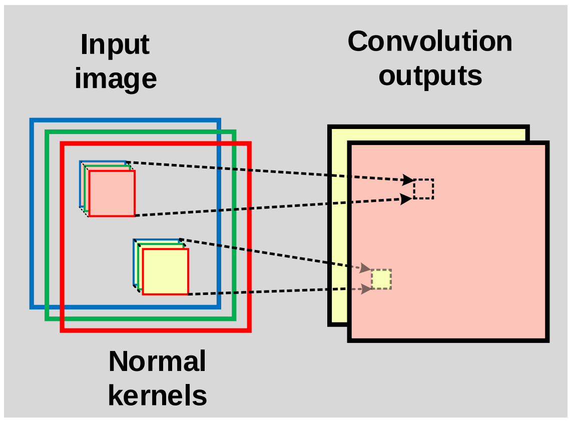
$$\delta^l_{x,y} = \varphi'_l(o^l_{x,y}) \sum_{x'} \sum_{y'} \delta^{l+1}_{x',y'} w^{l+1}_{(x-x'),(y-y')} \tag{2.3}$$

$$\Delta w_n = \frac{\partial C}{\partial w^L_{x,y}} = \sum_{x'} \sum_{y'} \delta^l_{x',y'} a^{l-1}_{(x+x'),(y+y')} \tag{2.4}$$

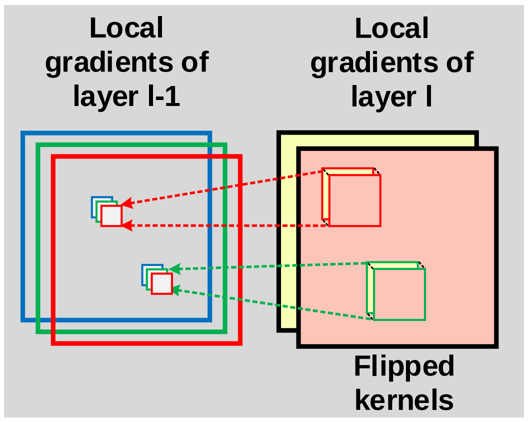
$$w^l_{i,j}(n) = -\alpha \Delta w_n + w^l_{i,j}(n-1) \tag{2.5}$$

$$w^l_{i,j}(n) = \beta \Delta w_{n-1} - \alpha \Delta w_n + w^l_{i,j}(n-1) \tag{2.6}$$

Local gradients of each layer l is computed using Eq. (2.3), where w is the flipped kernel. Eq. (2.4) is used for weight gradient computation, where l is local gradients



(a) Feedforward convolutions



(b) Backward convolutions

Figure 2. Convolution operations and changes in kernels during FP and BP Nof 2, Nif 3

of a layer and $\varphi_l^{'}(x)$ is activation gradients of layer l. The weight gradients of any layer l is obtained by the convolution of local gradient layer l and feedforward input activations of layer l. These convolutions involve large kernel sizes. One feature map of feedforward activation is convolved with one feature map of local gradients to obtain one kernel gradient (intra-tile accumulation). Hence, this weight gradient convolution results in a 4D output. These weight gradients are averaged over a batch and new weights are computed using gradient descent algorithm given by Eq. (2.5), where α represents learning rate, $w_{i,j}^{l}(n-1)$ represents weights of previous batch and Δw_n represents average weight gradients. The weight update process can be accelerated by using past weight gradients as momentum. Eq. (2.6) shows the weight update in SGD with momentum, where β is a hyper-parameter.

2.2.1 Training-Specific Computations During BP and WU

The operations during BP are different than those of FP. In backward convolutions, the inputs are scaled by activation gradients, and convolutions are performed by applying 180-degree-rotated kernels. In hardware implementation, the same kernels should be read in normal mode and transpose mode to support FP and BP operations. Similarly, fully-connected layers in BP also use transposed weight matrix to compute the local gradients. At the max-pooling node, the gradients propagate only through the selected maximum pixel location and all other pixels in the pooling window will be zero. Based on the pooling pixel index selected during FP, the gradients are upsampled and propagated back to the next layers.

During FP, we need to store not only the output activations, but also the activation gradients and max-pooling indices at all ReLU activations and max-pooling nodes. For ReLU, activation gradients are binary as the derivative of ReLU with respect to activations results in a step function. Our RTL library currently supports only ReLU activation function as it is less complex and widely used. During weight update of fully-connected layers, the weight gradients Δw are obtained by performing the outer product of the local gradient vector and the error vector. In convolution kernel updates, kernel gradient calculation involves convolution of input activations using local gradients as kernels, which are very large kernels. Each of these convolutions is considered as an FP convolution with $N_{if} = 1$ and results in N_{of} kernel gradients. To reuse FP convolution control logic, we employed an additional outer loop to iterate through the actual N_{if} local gradients.

2.2.2 CNN Training Using Fixed-Point Precision

Unlike CNN inference, CNN training usually requires higher precision. In this work, weights, activations, local gradients and weight gradients are represented with 16-bit fixed-point precision to ensure good training accuracy. Compared to floating-point representation, fixed-point training is more energy-efficient in FPGA implementation, but requires more dedicated resolution/range assignment for different variables.

2.3 CNN Training Hardware

2.3.1 RTL Compiler and Algorithm Mapping

To map various CNN algorithms with user defined hardware constraints onto FPGA, an RTL compiler for CNN training was developed. Fig. 3 shows the compiler tool flow from high-level CNN description to CNN training accelerator. According to the operations in each layer and FPGA design parameters (e.g. unroll and tiling factors), optimized handwritten Verilog modules are chosen from the RTL library to automatically generate a CNN training accelerator. The RTL library consists of Verilog modules that are specially designed to support training operations. Only the selected modules from the RTL library based on the training algorithm will be synthesized. Execution of training operations in one iteration of a batch can be scheduled sequentially similar to layer-by-layer execution of inference tasks. Each training image in a batch is processed sequentially. The scheduling of layer execution is done using the RTL compiler, and control logic parameters are generated.

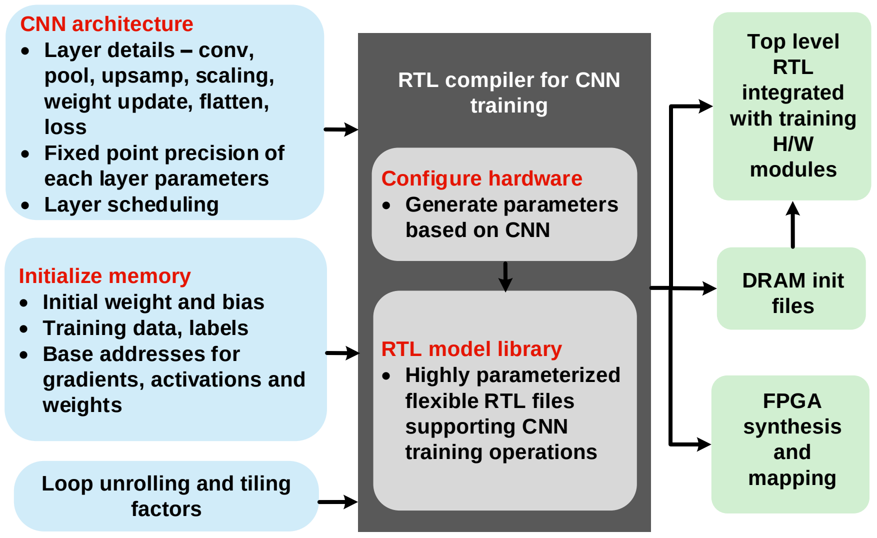

CNN architecture
Layer details – conv, pool, upsamp, scaling, weight update, flatten, loss
Fixed point precision of each layer parameters
Layer scheduling
Initialize memory
Initial weight and bias
Training data, labels
Base addresses for gradients, activations and weights
Loop unrolling and tiling factors
RTL compiler for CNN training
Configure hardware
Generate parameters based on CNN
RTL model library
Highly parameterized flexible RTL files supporting CNN training operations
Top level RTL integrated with training H/W modules
DRAM init files
FPGA synthesis and mapping

Figure 3. Proposed RTL compiler automatically generates FPGA training accelerator from high-level CNN description and FPGA design variables.

2.3.2 Training Accelerator Architecture

Fig. 4 shows the top-level diagram and dataflow of the CNN training accelerator. The global control logic governs all modules to ensure proper CNN functionalities with layer-by-layer computation, and is controlled by the parameters generated by the RTL compiler.

DRAM stores all the initial weight parameters, intermediate activations and computed weight/loss gradients using 16-bit fixed-point precision. DMA control generates the required DMA descriptors based on the layer type and tile sizes to read from and write to DRAM. Convolution, max-pooling and upsampling operations are considered as *key layers*, and ReLU, flatten, loss unit, and scaling unit are referred to as *affiliated layers*.

On-chip buffers store activation gradients and max-pooling indices. The pooling window size (e.g. 2x2) determines the bitwidth of max-pooling indices (e.g. 2-bit). After FP, loss is computed using outputs and labels. Our RTL library currently

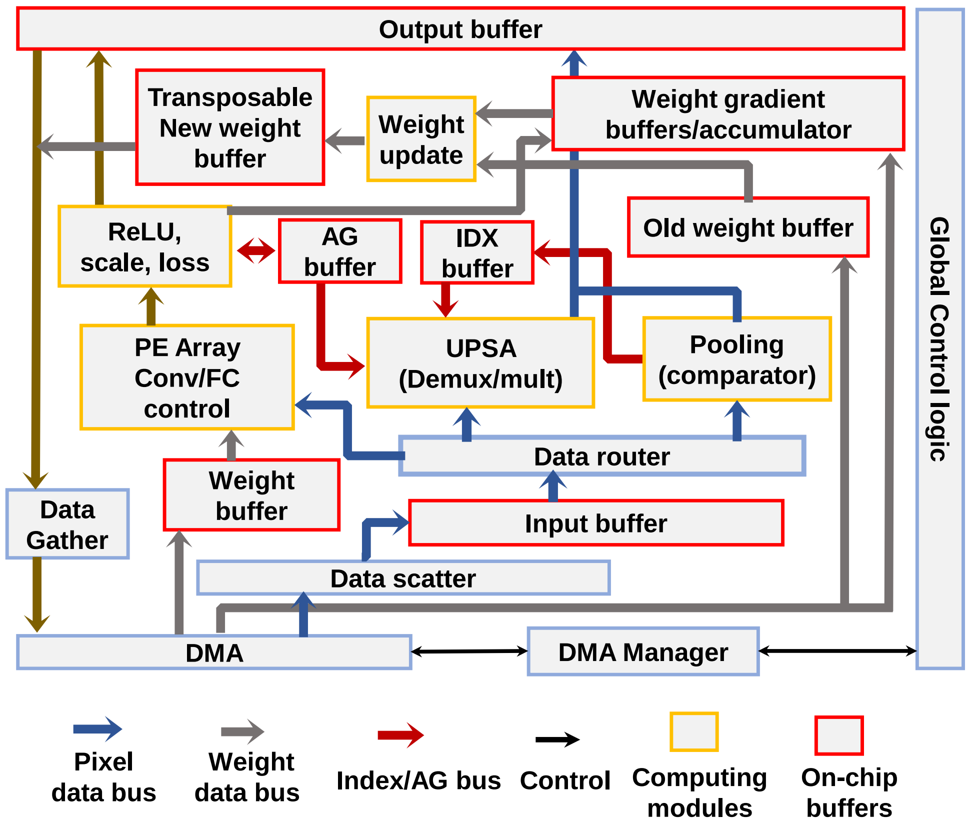
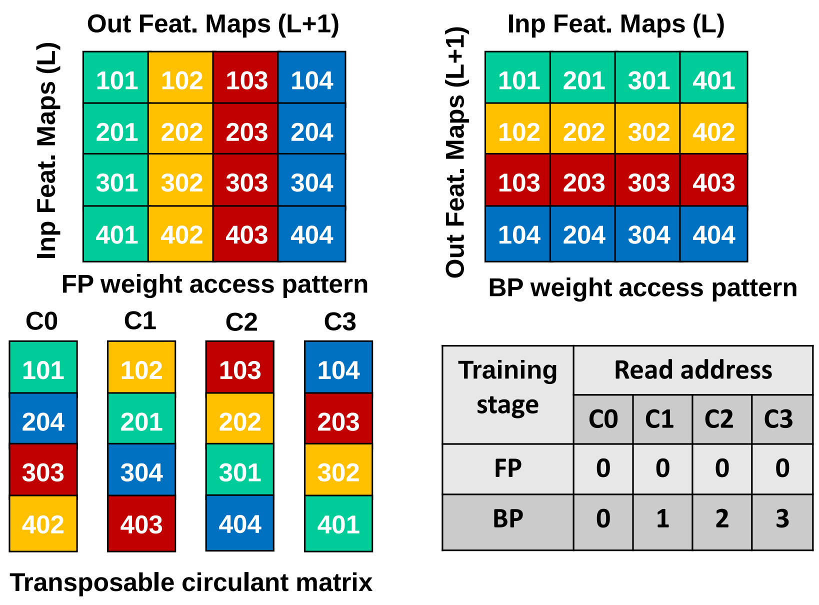


Figure 4. Top-level block diagram of CNN training accelerator.

supports square hinge loss and euclidean loss functions, and this can be easily expanded to support other loss functions. Data scatter and data gather modules are used to convert the DRAM storage pattern to on-chip buffer storage pattern and vice versa. Data router reads the data from input buffers and routes it to the selected key layer according to the array sizes. Weight update unit and weight gradient buffers are used to compute new weights based on SGD with momentum.

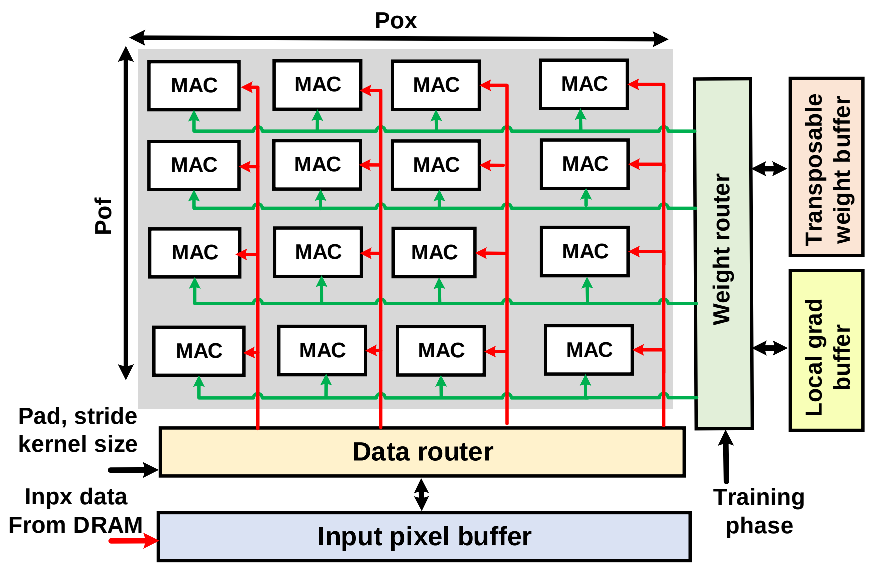


Training stage	Read address			
	C0	C1	C2	C3
FP	0	0	0	0
BP	0	1	2	3

Figure 5. Proposed transposable weight buffer stores weights in a circulant matrix, enabling both normal and transpose read.

2.3.3 MAC Array

Fig. 6 shows the 2D systolic MAC array used for the training accelerator. MAC array size is determined by the RTL compiler based on the loop unroll factors P_{ox}, P_{oy}, P_{of}. In Fig. 6, each MAC row has a different set of weights but share the same input feature map data computing P_{of} output pixels. Each column shares the same weights, but different input data computing P_{ox} or P_{oy} output pixels in parallel. Data router reads the input data and routes it to MAC units considering pad and stride sizes of the layer. Weight router distributes weights or local gradients based on the training phase. Table in Fig. 6 summarizes how the MAC array is reused with different inputs/outputs for training phases of FP, BP and WU.



Training phase	Input	Weights	Output
FP	Activations	Normal Kernels	Activations
BP	Local gradients	Flipped kernels	Local gradients
WU	Activations	Local gradients	Kernel gradients

Figure 6. Systolic MAC array is reused for training phases of FP, BP and WU, by feeding different activations/gradients/kernels.

2.3.4 Transposable Weight Buffer

BP involves convolution of flipped kernels and the local gradients. Therefore, every convolution kernel is used twice in one iteration: 1) normal weights are applied during FP, and 2) rotated weights are used in BP (Fig. 2). To achieve this without duplicating kernel storage, the kernels are stored in special transposable buffers that we propose, where data can be read both in non-transpose and transpose modes. As shown in Fig. 5, the proposed transposable buffer stores the kernels in the form of a circulant matrix using column buffers. For 2D kernels, each $N_{kx} \times N_{ky}$ kernel is considered as one block and each row has P_{of} blocks of kernels, where P_{of} represents the number of

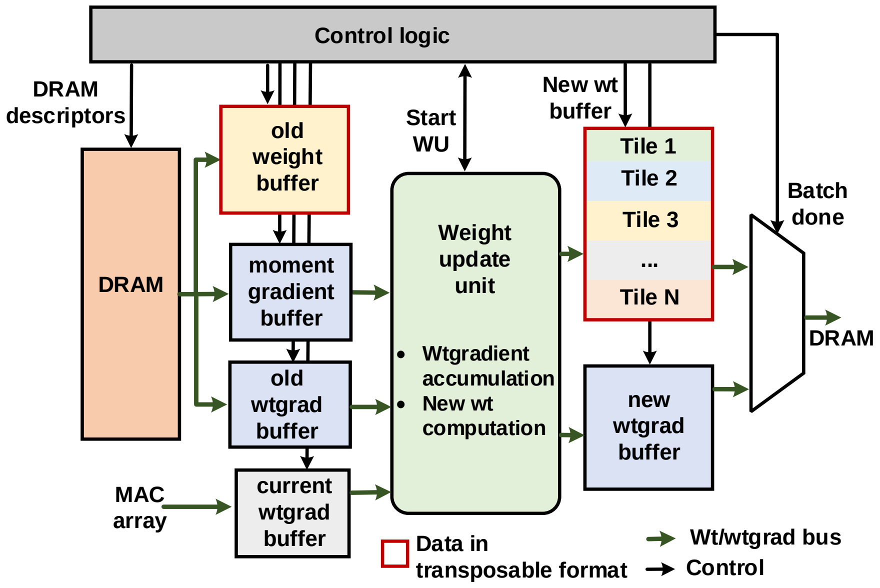


Figure 7. Block diagram of weight update unit.

output feature maps that can be computed in parallel. During backward convolution, not only the kernel is rotated by 180 degrees but also the input and output feature maps will be interchanged. In the proposed transposable buffer, every row of kernel blocks is circularly rotated and stored in the form of a circulant matrix in the single-port column buffers (Fig. 5). In the non-transpose mode, each column buffer shares the same read address, and in transpose mode, each column buffer obtains shifted addresses from the address translator unit. Address translator generates read/write addresses for column buffers for every transposable block. In each transposable block, the address vectors and the data are circularly shifted using shift registers.

2.3.5 Weight Update Unit

Weight gradients are calculated by convolving the feedforward activations with the local gradients. Convolution control logic is configurable to support tile-by-tile computation, intra-tile accumulation and large kernel sizes needed for weight gradient computation. Fig. 7 shows the dataflow after the computation of weight gradients. For every new training image in a batch, newly computed weight gradients are accumulated with old weight gradients. This accumulation is done tile-by-tile for efficient utilization of on-chip buffers. This process is repeated for the entire batch of images and accumulated gradients are stored in DRAM. At the end of the batch, while the weight gradients get accumulated, old weights and past weight gradients are also read from DRAM, and new weights are computed simultaneously, following Eq. (2.6).

Weights are initially stored in transposable format in DRAM as aforementioned. The entire transposable weights of layer l are read from DRAM to the old weight buffer. New weights are computed tile-by-tile and written back in transposable format to the new weight buffer. After completing the last tile's computation, the new weights are written back to DRAM. Control logic translates the address for transposable read/write operations, generates DRAM descriptors according to tile count and generates addresses to read newly computed weight gradients. Fully-connected weight update follows the same dataflow, but gradients are computed by outer product of local gradient vector and activation vector. Constant learning rate is applied throughout the training process, and 16-bit fixed-point precision is used for all weights and gradient computation.

2.3.6 Efficient MAC Usage in Weight Update layers

During FP and BP, convolutions the MAC array is designed to compute $P_{ox} \times P_{oy} \times P_{of}$ pixels in parallel. P_{ox}, P_{oy}, P_{of} are the loop unroll factors given by the user.

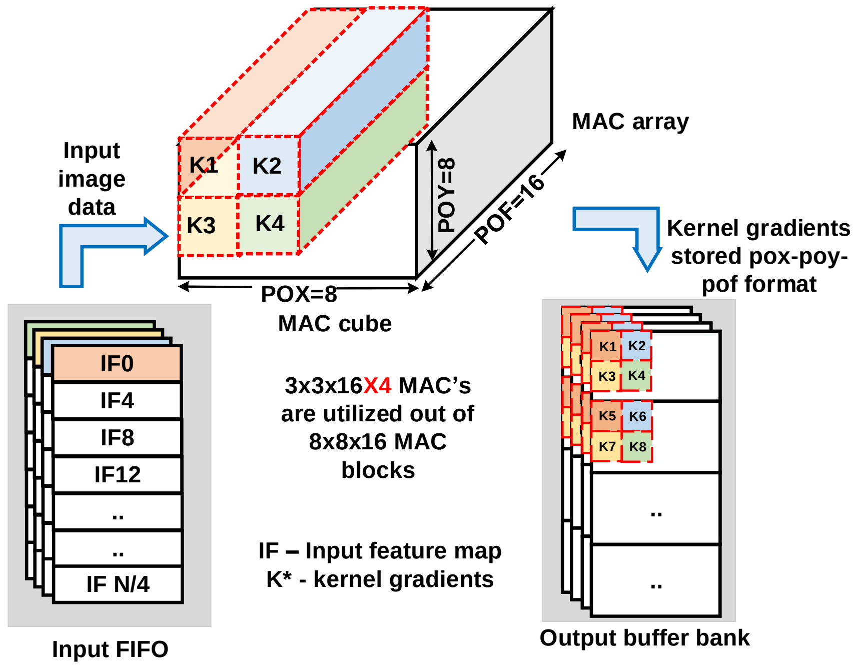


Figure 8. Operation of MAC load balancing unit during convolution weight gradient computation.

Regarding convolutions required for weight updates, however, the output feature map size N_{ox}, N_{oy} is less as the the outputs are kernel gradients. This results in inefficient usage of MAC units, since most of them will be idle. It also consumes more output buffer storage in order to store $P_{ox} \times P_{oy} \times P_{of}$ block of output data. To overcome this, MAC load balance unit was designed to utilize the idle MAC units.

The MAC load balance unit employs additional input buffers to feed the data to the MAC units in parallel. If buffer usage is critical, this optimization can be disabled using the RTL compiler. Fig. 8 shows the basic operation of MAC load balancing unit, when $P_{ox} = 8, P_{oy} = 8, P_{of} = 16$ and kernel size is $N_{ox} = 3, N_{oy} = 3, N_{of} = 16$. In this example, four kernel gradients are computed in parallel, reducing the latency by 4X without any additional MAC units. The output buffer is also efficiently used.

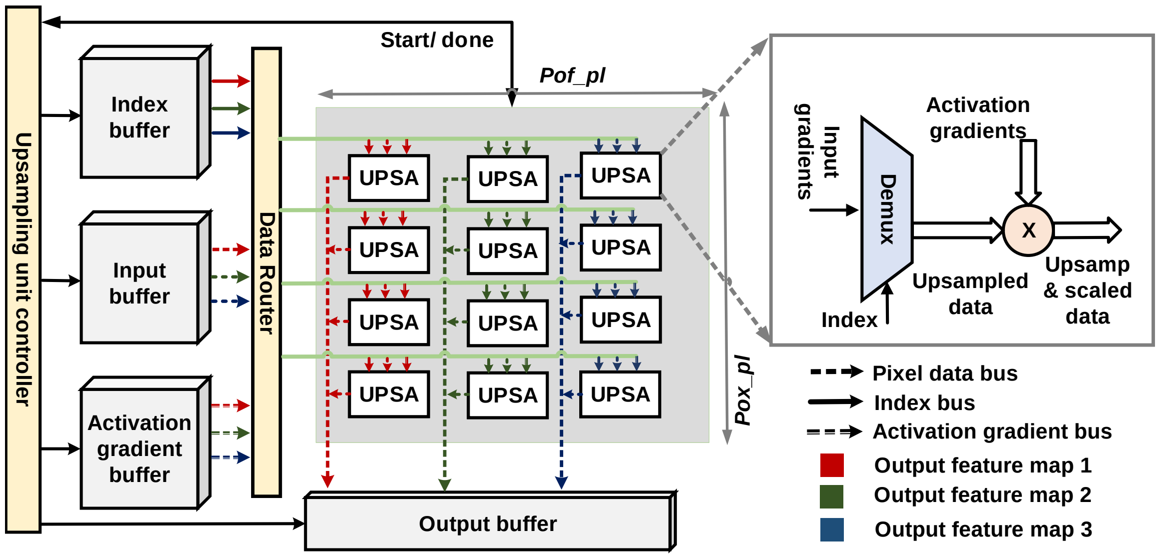


Figure 9. The upsampling unit receives input from index buffer, activation gradient buffer and input buffer.

2.3.7 Upsampling and Scaling module

During BP, the local gradient at the max-pooling node is propagated to convolution layers only through the maximum pixel position selected in FP. The gradients of unselected pixels are zero, as they do not contribute to the error. If the max-pooling unit receives the input from ReLU node, then the upsampled gradients should also be scaled by the feedforward activation gradients to compute the gradients of ReLU node.

Fig. 9 shows the upsampling unit and input/output buffers. During FP, max-pooling indices are stored tile-by-tile inside the on-chip index buffers. Each layer has its own index and activation gradient buffers. The local gradients computed in the previous iteration is read from DRAM and stored in input buffers. Data router unit rearranges the data of index, input and activation gradient buffers and sends it to the upsampling unit. Each processing element of the upsampling unit consists of a

Table 2. Evaluation of CNN training accelerator on Stratix 10 FPGA , using 16-bit fixed point precision. CIFAR10-1X refers to network structure of 16C3-16C3-P-32C3-32C3-P-64C3-64C3-P-FC, and 2X/4X designs refer to accordingly wider CNNs.

CNN network	Resource			Power (W)					Latency per epoch (s)			Throughput
	DSP	ALM	BRAM	DSP	RAM	Logic	clock	Pstatic	BS-10	BS-20	BS-40	GOPs
CIFAR-10 1X	1699 (30%)	20.8K (19%)	10.6(4.4%)	0.58	5.7	2.4	1.68	10.28	18.19	18.07	18.01	163
CIFAR-10 2X	3363 (58%)	415K (44%)	22.8(9.5%)	1.05	11.2	6.6	2.97	11	41.7	41.30	41	282
CIFAR-10 4X	5760(100%)	720K(76.2%)	54.5(22.4%)	3.48	14.6	11	4.95	16.47	98.2	96.87	96.18	479

demultiplexer and a multiplier unit. The gradient is conveyed as the demultiplexer input and the index serves as the select signal. For pooling window size of k, each processing block produces $k \times k$ pixel data corresponding to k rows of the output feature map. After each operation, k rows of activation gradients are read and the demultiplexer outputs are scaled.

2.4 Results

2.4.1 Experimental Setup

The FPGA accelerator generated by the compiler was synthesized using Intel Quartus 17.1 at 240MHz frequency. We used Stratix 10 GX FPGA as the target hardware, which includes 240 Mbits of BRAM, 5,760 DSP blocks, and 93K ALMs. The stratix-10 GX development kit is equipped with 4Gb DDR3 DRAM with 16.9Gb/s bandwidth. Weights, weight gradients, activations, and local gradients use 16-bit fixed point precision. We trained representative CNNs for CIFAR-10 dataset. '1X' CNN has the network structure of 16C3-16C3-P-32C3-32C3-P-64C3-64C3-P-FC. 2X and 4X CNN models exhibit 2X and 4X more input/output feature maps for each layer, and were also explored to achieve higher accuracy on CIFAR-10 dataset. Unroll factor of 8 was adopted for output image x and y dimensions. For output feature maps, 16,

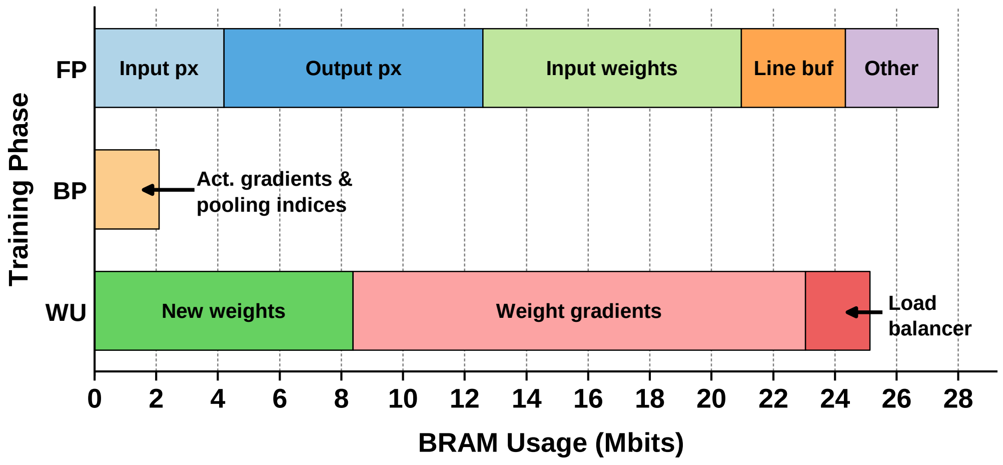


Figure 10. Buffer usage breakdown of CIFAR-10 4X CNN.

32, 64 was used as unroll factors for 1X, 2X and 4X CNNs, resulting in an 8x8x16 (1024), 8x8x32 (2048), 8x8x64 (4096) MAC arrays, respectively. Batch size (BS) of 40 and learning rate of 0.002 was used for training, towards achieving better accuracy. The software results are obtained by PyTorch Paszke *et al.* (2017) based fixed-point training model running on Nvidia Titan XP GPU.

2.4.2 Analysis of Results

Table 2 shows the comparison of CNN training performance and resource utilization for three different CNNs for the classification for CIFAR-10 dataset. The FPGA accelerator was generated from the RTL compiler using high-level description of training parameters and design variables. FPGA power numbers are obtained after routing stage from Quartus power analyzer and Intel Early Power Estimator tools using the data toggling activity from functional simulation at the junction temperature of 65C. Tiling of activations and weight gradients greatly reduces the on chip buffer usage. BRAM utilization is low because of the tiling and size of the intermediate activations and number of parameters. Training of each image in a batch is done sequentially, larger batch sizes results in less number of weight updates in one epoch resulting in improvement in latency.

To support deeper CNNs and aim good flexibility of hardware, all intermediate outputs are stored in DRAM. Fig. 11 shows the latency breakdown in each layer during different stages of training. Weight update layers will have large DRAM access latency due to access of past weight gradients, weights and storing back the updated values. 51% percent of the overall latency in one iteration of a batch is consumed in weight update layers. By sacrificing the flexibility of the hardware, this latency could be significantly reduced by using on-chip buffers for weight/gradient storage. The latency of the weight update layers depends on the parameters associated with it.

Old weight gradients are read from DRAM tile-by-tile during computation of current weight gradients. DRAM latency is hidden wherever logic latency is more than the memory access latency. The latency of weight update layers is reduced by 11% by using the compute time to read next tile data. The logic latency in weight

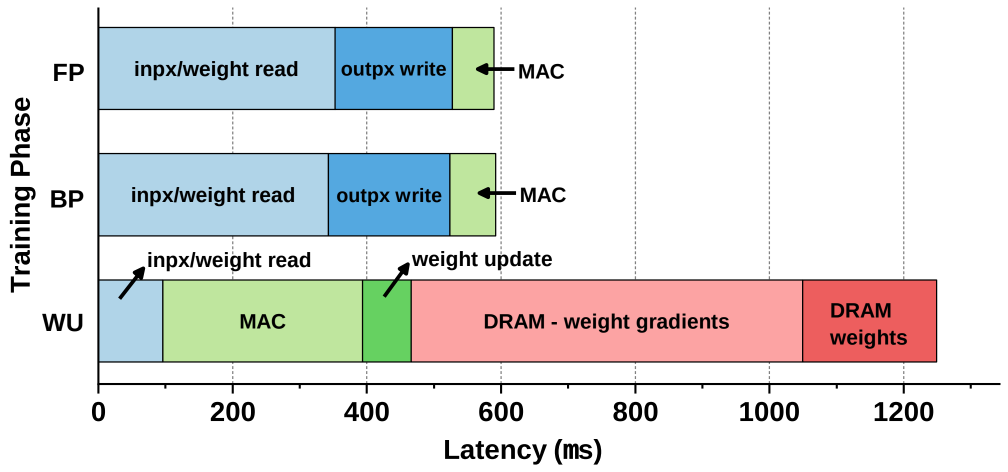


Figure 11. Latency breakdown of CIFAR-10 4X CNN for FP, BP and WU for the last iteration of a batch.

update layers is reduced by 4X, using the load balancing technique for MAC arrays. Logic in weight update layers refer to convolution operations to generate weight gradients and weight update is referred to computation of new weights. Tile sizes are carefully chosen to efficiently map compute-/memory-bounded layers. All buffers can be controlled by tile sizes apart from weight buffers, where the entire weights are read from transposable DRAM.

Fig. 10 shows the breakdown of buffer utilization for three different phases of training. The size of the weight buffer is decided by the largest layer weights. Double buffering technique is used for all other buffers, thereby hiding DRAM latency. The 1X design achieves 73% accuracy at 50 epochs with learning rate of 0.002 and batch size of 40 (similar to baseline with floating-point precision). Higher accuracy will be achievable with longer training time and deeper/wider CNNs.

2.5 Conclusion

In this chapter, we presented an automatic RTL compiler based end-to-end CNN training accelerator. CNN training operations are implemented by optimized and parameterized custom Verilog modules, and the accelerator is flexible to support various FPGA design parameters. The training performance is evaluated on Intel Stratix-10 GX FPGA for three different CNNs for CIFAR-10 dataset. The proposed training accelerator achieves throughput of up to 479 GOPS at 240MHz for CNNs with 2M parameters.

Chapter 3

FPGA-BASED LOW-BATCH TRAINING ACCELERATOR FOR MODERN CNNS FEATURING HIGH BANDWIDTH MEMORY

Training convolutional neural networks (CNNs) requires intensive computations as well as a large amount of storage and memory access. While low bandwidth off-chip memories in prior FPGA works have hindered the system-level performance, modern FPGAs offer high bandwidth memory (HBM2) that unlocks opportunities to improve the throughput/energy of FPGA-based CNN training. This chapter presents a FPGA accelerator for CNN training which (1) uses HBM2 for efficient off-chip communication, and (2) supports various training operations (e.g. residual connections, stride-2 convolutions) for modern CNNs. We analyze the impact of HBM2 on CNN training workloads, provide a comprehensive comparison with DDR3, and present the strategies to efficiently use HBM2 features for enhanced CNN training performance. For training ResNet-20/VGG-like CNNs for CIFAR-10 dataset with low batch size of 2, the proposed CNN training accelerator on Intel Stratix-10 MX FPGA demonstrates 1.4/1.7X energy-efficiency improvement compared to Stratix-10 GX FPGA with DDR3 memory, and 4.5/9.7 X energy-efficiency improvement compared to Tesla V100 GPU.

3.1 Introduction

Convolutional neural networks (CNNs) are extensively adopted in computer vision applications Krizhevsky *et al.* (2017); Long *et al.* (2015); Tateno *et al.* (2017). The training tasks of CNNs are commonly performed with GPUs using a mini-batch stochastic gradient descent (SGD) optimizer. To improve the CNN accuracy, higher batch sizes are employed for CNN training with GPUs, but this demands an excessive amount of memory and limits the capability to explore large models and tasks with high input resolution Wu and He (2018). In addition, although GPUs provide very high throughput on CNN training with large batch sizes, they suffer from low utilization/throughput for smaller batch sizes Chundi *et al.* (2019). This can be seen in Fig. 12, which reports the latency and Tesla V100 GPU utilization across different batch sizes for the task of training ResNet-20 CNN He *et al.* (2016) for CIFAR-10 Krizhevsky (2009) dataset. Recently, new CNN training algorithms that efficiently support small batch sizes (e.g. 2, 4) have been proposed Wu and He (2018); Masters and Luschi (2019); Ioffe (2017), which demonstrate on-par accuracy with state-of-the-art CNN training using large batch sizes.

Low-batch training greatly reduces memory requirement and unlocks opportunities for FPGAs, which provide higher configurability for custom architecture and better energy-efficiency than high-power GPUs. Training on edge FPGA devices also reduces latency overhead (due to limited data exchange with the cloud server), prevents privacy/security problems, and is well-suited to exploit new features such as low precision training, sparse weight updates, online learning, etc. However, training CNNs on FPGAs is a challenging task for two reasons: (1) it demands high memory bandwidth which is the primary limiting factor in many accelerators Wissolik *et al.* (2017); Deo *et al.* (2017), and (2) complexity arises in implementing a generalized flexible training accelerator supporting new advancements in CNN training algorithms.

Many prior works presented low-batch CNN inference on FPGAs and showed large improvements in storage and latency Ma *et al.* (2017); Venieris and Bouganis

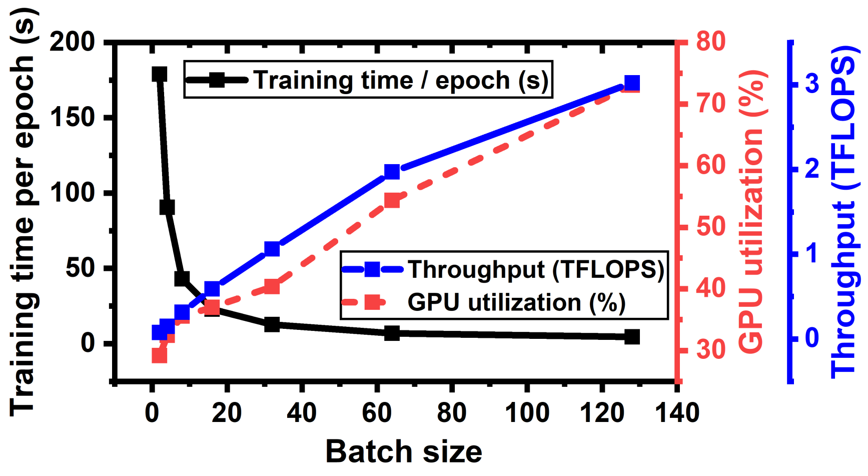


Figure 12. ResNet-20 CNN training performance for CIFAR-10 dataset on Tesla V100 GPU for different batch sizes.

(2016); Wei *et al.* (2017); Zhang *et al.* (2015); Qiu *et al.* (2016); Abdelouahab *et al.* (2018); Guo *et al.* (2017). While there are new algorithmic approaches to support low-batch training, FPGA hardware designs for CNN training tasks have been much less explored. A framework to map DNN training on FPGA clusters was presented in Geng *et al.* (2019), but an excessive amount of on-chip memory is required for training on a single FPGA platform. Several prior works Ahmad and Pasha (2020); Zhao *et al.* (2016); Choi *et al.* (2018) attempted to accelerate a part of CNN training on FPGAs, while the remaining operations were performed by the host CPU. Training accelerators for non-CNN applications were proposed in Zeng and Prasanna (2020); Liu *et al.* (2018); Gomperts *et al.* (2011); Rafael *et al.* (2005); Zhou *et al.* (2020). Only a few prior works presented a CNN training accelerator supporting all three phases of training Venkataramanaiah *et al.* (2019a); Nakahara *et al.* (2019); Luo *et al.* (2020), but these works still did not include back-propagation of either residual connections or stride-2 convolutions that are necessary for modern CNNs. Furthermore, none of the aforementioned FPGA works studied the use of high bandwidth memory, which is critical for CNN training.

In this work, we present a programmable FPGA accelerator for CNN training, which uses HBM2 for efficient off-chip communication, and supports residual connections and stride-2 convolutions for modern CNNs. The key contributions of this work are:

- To the best of our knowledge, we present the first FPGA accelerator for CNN training that fully utilizes high bandwidth memory (HBM2) and executes end-to-end CNN training.
- Our programmable FPGA accelerator reads high-level descriptions of CNNs (similar to TensorFlow/PyTorch) including those with residual connections and stride-2 convolutions, and automatically generates RTL for synthesis.
- We analyze the impact of HBM2 on CNN training workloads, provide a comprehensive comparison with DDR3, and discuss the strategies to efficiently use the HBM2 features for enhanced performance.
- Our accelerator using Intel Stratix-10 (S-10) MX FPGA with HBM2 is evaluated for ResNet-20 and VGG-like CNNs for CIFAR-10 dataset, achieving up to 14X improvement in energy-efficiency, compared to Tesla V100 GPU.

3.2 Background

Modern FPGAs, such as Intel Stratix-10 (S-10) MX Deo *et al.* (2017), are equipped with new high-speed memory technologies like high bandwidth memory (HBM2) Standard (2013). HBM2 uses 3D stacked silicon dies connected through through-silicon vias (TSVs). The main DRAM stack is placed as a top die and the base die is used

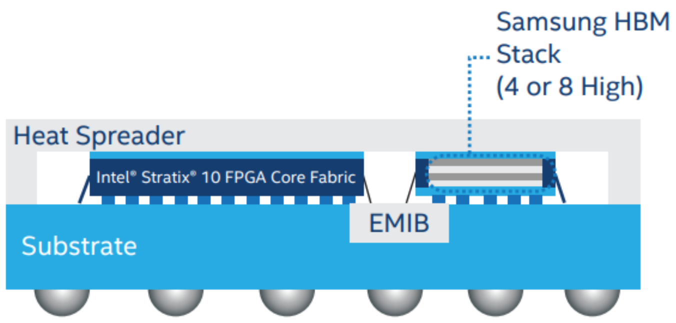

Samsung HBM Stack (4 or 8 High)
Heat Spreader
Intel® Stratix® 10 FPGA Core Fabric
EMIB
Substrate

Figure 13. Intel S-10 MX device integrated with HBM2

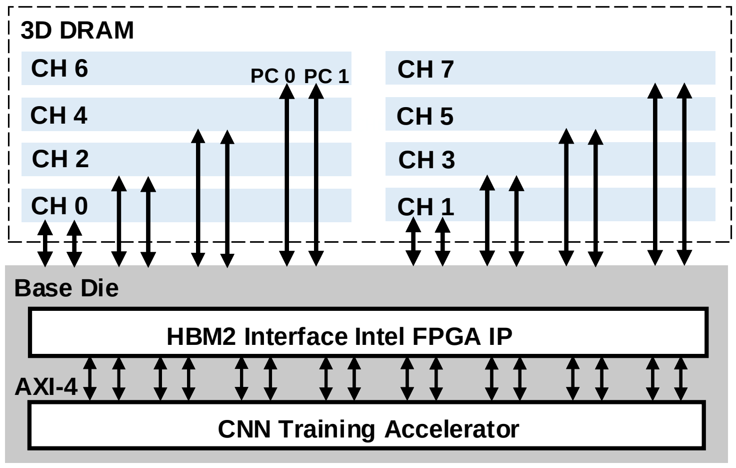


Figure 14. Eight independent physical channels (CH) and corresponding pseudo channels (PC) of HBM2 connected to CNN training accelerator on the base die using Intel HBM2 interface FPGA IP.

for I/O connections to the host device. Each die in the DRAM stack consists of two independent physical channels, which are further divided into two pseudo channels.

As shown in Fig. 13, HBM2 is integrated with the Intel S-10 MX device using the system-in-package (SiP) technology. Fig. 14 depicts the interface between the DRAM stack and the base die. All the physical channels (CH) and corresponding pseudo channels (PC) are connected to the base die using HBM2 interface Intel FPGA IP. Dedicated customizable memory controllers are provided for each physical channel of

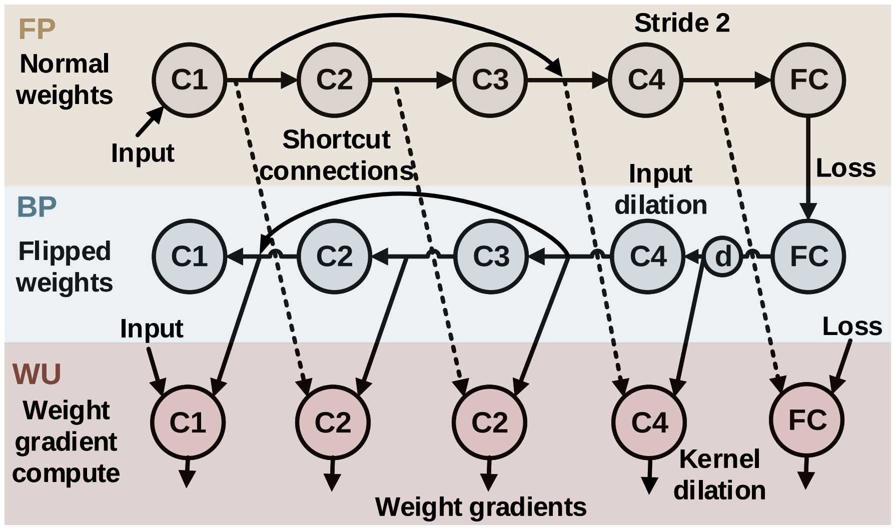


Figure 15. Training dataflow for CNNs involving stride-2 convolutions ($C4$) and shortcut connections. Ci is i^{th} convolution layer, d is the input pixel dilation, and FC is the fully-connected layer.

HBM2. Overall, HBM2 provides higher bandwidth, I/O and capacity with a small form factor, compared to traditional off-chip memories such as DDR3.

However, designing an architecture that can fully leverage high memory parallelism provided by HBM2 is challenging. Large I/O capacity of HBM2 demands unique data storage (a number of different parameters can be read in single access) and complex on-chip buffer control logic to handle the incoming data from HBM2. Accessing parallel and independent HBM2 channels requires separate memory controllers and status monitoring for all channels.

3.2.1 Modern CNN Training

CNNs are majorly trained with SGD optimizer using backpropagation algorithm, which is an iterative process used to find the best parameters of a network that

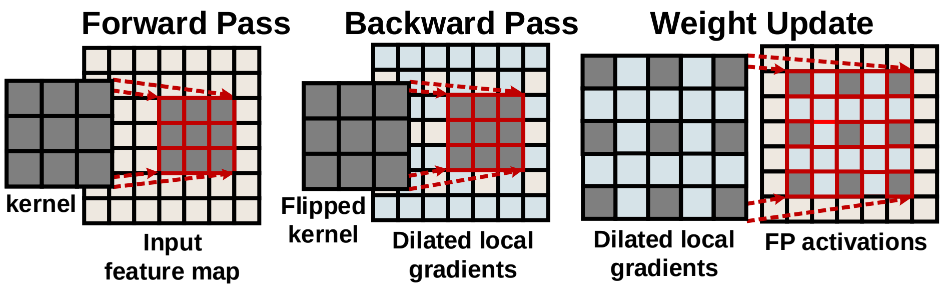


Figure 16. Stride-2 convolutions in different training phases are shown. Blue cells represent dilated positions of the activations (beige) or weights (grey).

minimizes the loss function. SGD-based training involves three phases, namely forward pass (FP), backward pass (BP) and weight update (WU). In the FP phase, the output activations are computed layer by layer in the forward direction and the FP performance is estimated using a loss function. In the BP phase, the local gradients are computed at every layer in the backward direction. During BP, convolution operations use flipped kernels and the pooling (downsampling) operations are replaced by upsampling units. In the WU phase, weight gradients are computed using the local gradients and feed-forward activations, and weight updates are computed.

Modern CNNs involve residual connections Sandler *et al.* (2018); He *et al.* (2016), and multi-stride convolutions to downsample the data and improve the storage/throughput of CNN training. Fig. 15 illustrates the overall training flow of a CNN with identity residual connections and convolutions with stride of 2. Identity shortcut operations remain the same during FP and BP, but the flow direction and the accumulation node are changed. During FP, we accumulate the shortcut connection at the output of convolution layer C3, but during BP, we accumulate the output of C2 (Fig. 15). For convolutions with stride larger than 1, local gradients are computed by performing the convolution of the horizontally and vertically dilated gradients with flipped kernels. In the WU phase, the weight gradients are computed by convolving the FP activations with dilated BP local gradients, which is similar to dilating the kernels during the convolution. Fig. 16 shows different dilations used in stride-2 convolutions.

In this work, we benchmark the training tasks of ResNet-20 CNN He *et al.* (2016) and VGG-like CNN Venkataramanaiah *et al.* (2019a) for CIFAR-10 dataset, using the proposed FPGA accelerator. ResNet-20 CNN has a convolution layer, followed by three stacks of 6 convolution layers, 9 residual identity connections, a pooling layer, and a fully-connected layer. The feature maps are downsampled at the output of every stack using stride-2 convolutions. VGG-like CNN has 6 convolution layers (C), 3 max-pooling layers (MP) and a fully-connected layer (FC), with the structure of 16C3-16C3-MP-32C3-32C3-MP-64C3-64C3-MP-FC.

3.3 CNN Training Accelerator Design

3.3.1 Overall Architecture and Operation

Fig. 17 shows the top-level block diagram of the CNN training accelerator architecture. The architecture can be mainly divided into five blocks:

1. Compute block supports various operations required for FP, BP and WU phases of training.
2. Buffer block stores input, output, weight and gradient data in on-chip buffers.
3. HBM2 configurator block generates signals to access HBM2, stores it to an on-chip buffer and write the data back to HBM2.

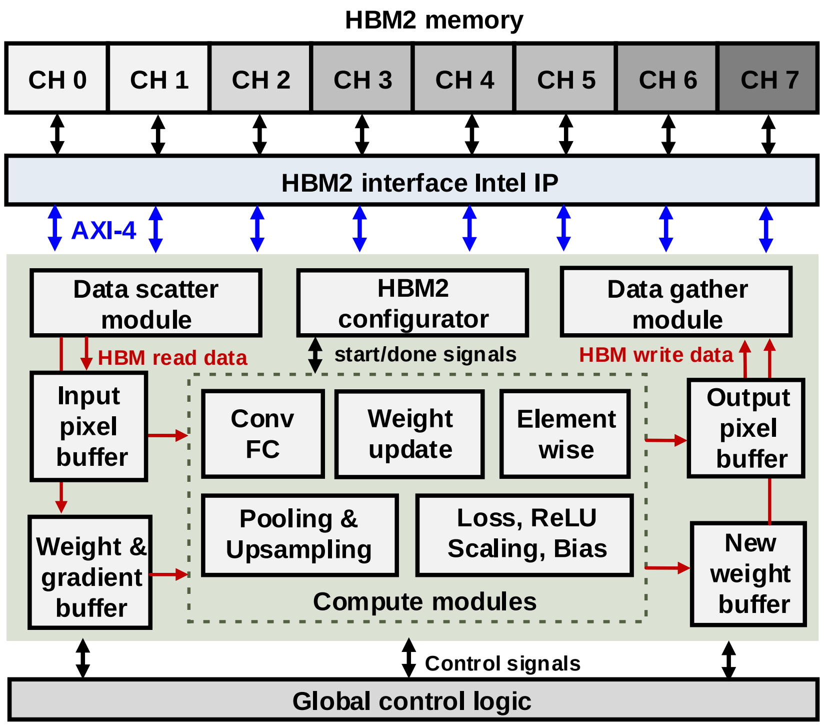


Figure 17. Top-level block diagram of the proposed CNN training hardware accelerator using HBM2 memory.

4. HBM2 memory stores all CNN training parameters, and HBM2 interface Intel IP communicates HBM2 memory and training accelerator.
5. Global control logic governs all the modules and performs layer scheduling.

The compute block consists of a systolic MAC array to support convolution and fully-connected layers. A 8x8x16 MAC array is used to compute 8x8 pixels of 16 output feature maps in parallel. MAC array size is chosen to exhibit a high utilization ratio. A MAC array of higher size, for example 32x32x16, will suffer under utilization while computing convolution of deeper layers where the output feature map size is small. Flexibility to choose the MAC array size also helps map the algorithm on different sized FPGAs. The MAC array is used to support fully-connected layers, normal convolutions during FP, transposed convolution during BP, and intra-tile accumulation in WU phases. A data router module is tightly coupled with the MAC array and distributes the parameters to the MAC array considering the padding and stride values.

Before the computed data is sent to the output buffer, it goes through a series of secondary layers including loss function, ReLU, bias and scaling unit. The scaling unit is used during BP where the derivative of a node is either 1 or 0 (e.g. ReLU, dropout layer). The secondary layers use outputs of key layers without any HBM access. In each layer, any of these secondary operations can be enabled or disabled based on the CNN structure.

Element-wise (Eltwise) module performs the element-wise addition of two input layers supporting identity shortcut connections required for ResNet CNNs He *et al.* (2016). If the volume of the input layers is different, then the smaller input layer is padded with zeros. Eltwise module is enabled once all the output data of the current layer are computed. Data from the other branch of the identity shortcut connection is read from the HBM2 to the input buffers. Finally, the accumulated data is written back to output buffers.

The pooling module is used during FP, and downsamples the input feature map by taking the maximum value within a kernel window (e.g. 2x2). During BP, the gradients will only flow through the selected pixel positions and non-selected pixel positions are padded with zeros. This operation is carried out by the upsampling unit. The compute array sizes of Eltwise, pooling, and upsampling modules are configurable.

The weight update module performs weight gradient accumulations, following the SGD algorithm. The accumulation of weight gradients is carried out for all training images in a batch. New weights are computed using the final weight gradient value and is written back to HBM2. Data scatter/gather module rearranges the data for HBM communication.

The global control logic governs the layer scheduling, enabling the secondary operations and configures the modules as required for the given network. The global control logic reads the detailed CNN structure through configuration registers. An RTL compiler is developed to generate these configurations, where the CNN structure, MAC array sizes and other control parameters are inputs to the compiler framework. The compiler framework reads the high-level inputs and translates the layer by layer execution schedule as parameters for the configuration registers, which is read by the global control logic in run-time. The RTL compiler consists of a highly parameterized hand-written RTL library which is optimized for CNN training. The overall accelerator consisting of configurable modules is shown in Fig. 17. The RTL compiler only compiles the required modules based on each CNN structure, without including any unused modules.

CNN training involves various parameters such as activations, weights, weight gradients, local gradients, momentum gradients, etc. The parameters required for a given layer is read from the HBM2 and stored in on-chip buffers (Fig. 17). Input/output pixel buffers are used to store the inputs/outputs of the compute blocks. Weight buffer is designed to support efficient weight access in both non-transpose and transpose directions (for FP and BP phases, respectively), following the schemes proposed in Venkataramanaiah *et al.* (2019a); Yin and Seo (2020). Weight gradient buffers are used during the WU phase to read the old gradients and momentum gradients. All the parameters required for the entire CNN training are stored in HBM2 memory.

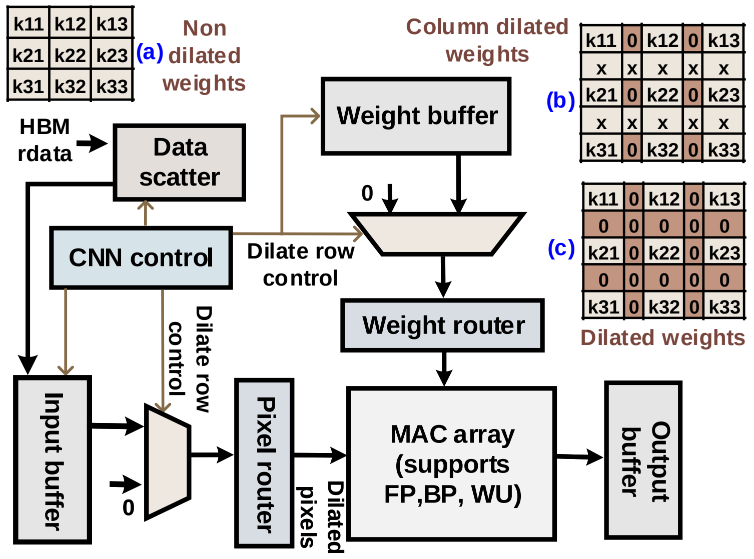
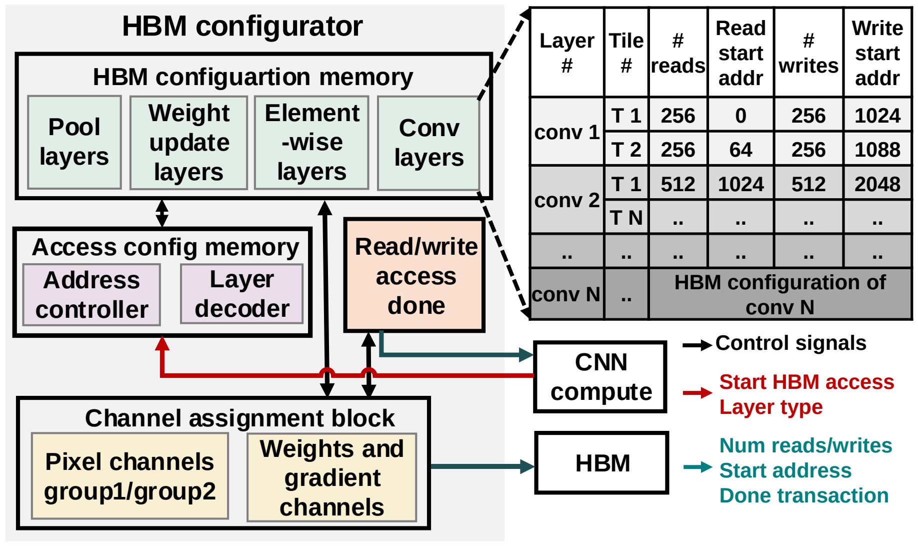


Figure 18. Flexible MAC unit with a dilation control block for both weights and activations. The non-dilated weights/activations from the HBM2 is rearranged by the dilation control block and the data scatter unit.

3.3.2 Dilated Convolutions

The BP and WU phases of stride-2 convolutions require dilations in both weights and input feature maps, (Fig. 16). Fig. 18 shows the design of control logic to support BP and WU of stride-2 convolutions. The non-dilated data (a) is loaded from HBM2 to on-chip buffers. Storage of dilated images/kernels in HBM2 is avoided to reduce latency. The non-dilated data is rearranged by the scatter unit according to the on-chip buffer storage pattern requirement. During this data rearrangement, the data scatter unit dilates the data (pixels or weights) in the x-dimension (b). Dilations in y-dimension is performed by the address control logic by skipping the writes of every dilated row. While reading the data to the convolution engine, every dilated row is detected and padded with zeros (c). This dataflow is replicated for both weights and input feature maps, and can be configured as needed using global control logic.



Layer #	Tile #	# reads	Read start addr	# writes	Write start addr
conv 1	T 1	256	0	256	1024
	T 2	256	64	256	1088
conv 2	T 1	512	1024	512	2048
	T N	..	..	..	..
..	..	..	..	..	..
conv N	..	HBM configuration of conv N			

Figure 19. HBM2 configurator module generates HBM2 read and write configuration signals based on the layer type.

3.3.3 HBM2 Configurator Module

Fig. 19 shows the HBM2 configurator, which generates HBM2 read/ write transaction details. The HBM2 configurator consists of a configuration memory that is preloaded with the information of every transaction. The information in the configuration memory includes the number of read/write transactions, and the read/write start addresses. Each layer has its own configuration memory as depicted in Fig. 19. Given the current layer details and tile count, the address controller generates the read address for configuration memory selected by the layer decoder. Once the transaction information is read from the memory, it is assigned to channels in the channel assignment block.

16 pseudo channels of HBM2 provide a high number of I/O data pins. To effectively utilize this parallelism provided by HBM2, proper channel allocation and organized

Table 3. HBM channel allocation for training parameters. 16 pseudo channels of the HBM2 is divided into four groups, for which input/output activations (in./out act), local gradients (LG), transposable weights and weight gradients (Wt gradients) are assigned.

Phase	**Layer**	**Activations & local gradients**		**Weights**	**Wt gradients**
		channel 0-3	channel 4-7	channel 8-11	channel 12-15
FP	C	in./out. act	in./out. act	transposable weights	NA
	P,EW	in./out. act	in./out. act	NA	NA
	FC	NA	NA	transposable weights	NA
BP	C	in./out. LG	in./out. LG	transposable weights	NA
	P,EW	in./out. LG	in./out. LG	NA	NA
	FC	NA	NA	transposable weights	NA
WU	C	in. act	in. LG	old/new weights	old, new moment gradients
	FC	NA	NA		

parameter storage become a necessity. For our application, 16 pseudo channels of HBM2 are divided into four groups of four channels. Each CNN training parameter that is stored in HBM2 is assigned with one of the four-channel groups. Table 3 provides the details of channel group allocation and the channel groups used in each phase of training. Channels 0-3 (group 1) and channels 4-7 (group 2) are used to store the local gradients and activations, channels 8-11 (group 3) are used to store the weights, and channels 12-15 (group 4) are used to store the weight gradients (both current weight gradients and momentum gradients). This channel allocation is done to reduce the off-chip latency of the WU phase.

In the WU phase, we need to read both activations and the local gradients to compute the weight gradients. To maximize the channel utilization, the local gradients and activations are stored in the two channel groups in a ping-pong manner. For example, if convolution layer 1 outputs are stored in channel group 2, then its corresponding local gradients are stored in channel group 1, and during the WU phase,

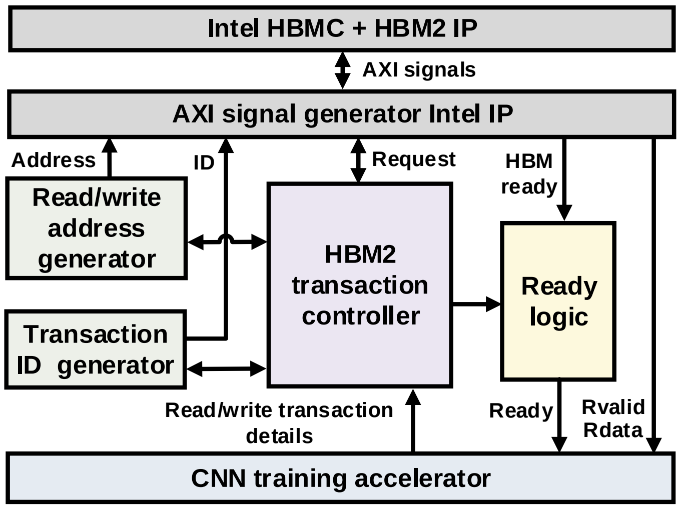
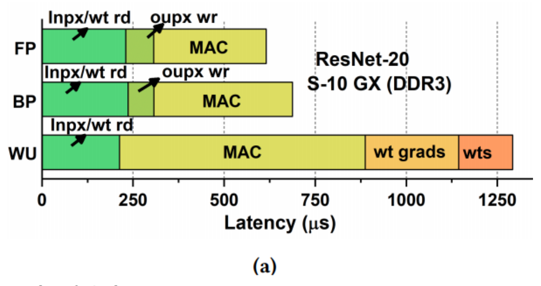


Figure 20. CNN training accelerator integrated with HBM2, following AMBA AXI4 protocol.

we read channel groups 1 and 2 simultaneously. Using this channel allocation, all 16 channels will be active during the WU phase. The channel assignment block assigns the transaction information read from the configuration memory to one of the channel groups based on the request from the CNN compute module. The done logic module monitors transactions of every channel and HBM2 status signals, and generates a 'done' signal when the transaction is complete.

3.3.4 HBM2 Integration

HBM2 communication uses the Intel HBM2 controller (HBMC) following the AMBA AXI-4 protocol. HBMC provides independent AXI ports for each channel. The read/write transaction information obtained from the HBM2 configurator is sent to the HBM transaction controller (HTC). Fig. 20 shows the integration of HTC and other modules with Intel HBM IPs, enabling successful HBM2 communication.



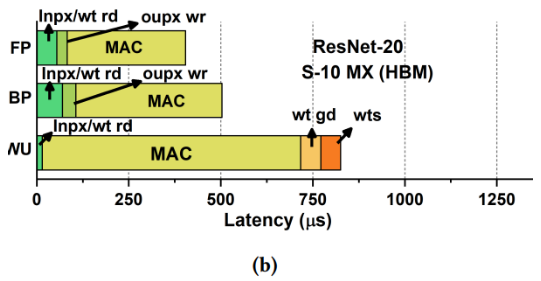
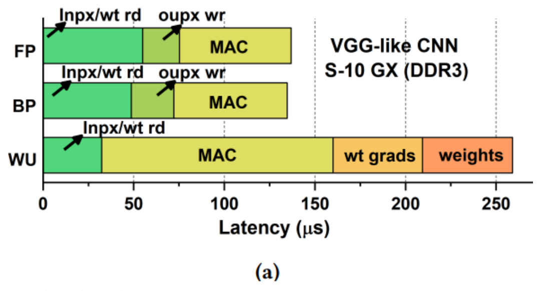


Figure 21. Training latency breakdown of ResNet-20 CNN for (a) S-10 GX device with DDR3 and (b) S-10 MX device with HBM2, both running at 185 MHz.



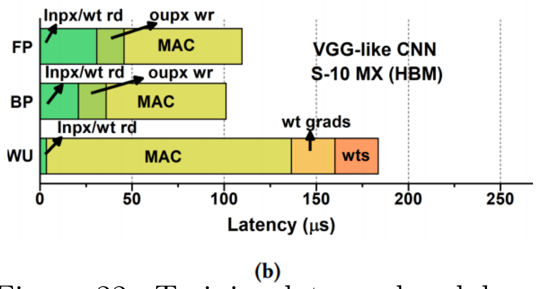


Figure 22. Training latency breakdown of VGG-like CNN for (a) S-10 GX device with DDR3 and (b) S-10 MX device with HBM2, both running at 185 MHz.

HTC monitors the request from the CNN training accelerator and the status of HBM2 memory. Based on the read/write request from the training accelerator, HTC enables corresponding address/transaction ID tag generators. The generated address and transaction IDs are converted to AXI signals using the AXI signal generator. Ready logic monitors the status of HTC, address generators, and HBM2 and indicates whether the HBM is ready to accept the next transaction.

3.3.5 Data Scatter/Gather Unit

HBM2 demands complex and flexible data collection/gathering units as more data is streamed in one cycle. To achieve this, customized data scatter/gather units were designed which can collect/send the data based on the channel allocation. The storage pattern of the parameters on on-chip buffers depends on the layer and parameter

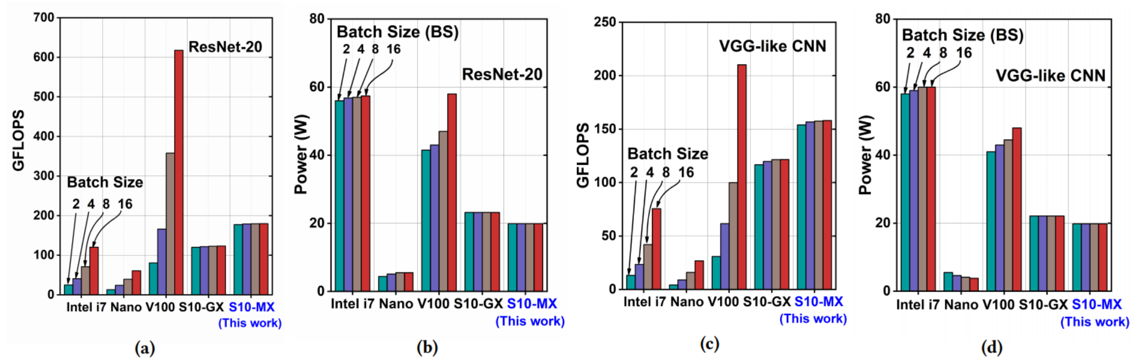


Figure 23. Throughput and power comparison of training tasks using Intel i7-9800X CPU, Tesla V100 GPU, Jetson Nano, S-10 GX FPGA with DDR3, and S-10 MX FPGA with HBM2. (a) Throughput and (b) power for ResNet-20 CNN training and (c) throughput and (d) power of VGG-like CNN training are shown

types. The continuous data stream from the HBM2 channels are collected by the data scatter unit where the data is rearranged and distributed to the on-chip buffers. The scatter unit also separates channels based on the parameter channel allocation and processes all channel groups in parallel. The data gather unit collects the data from output buffers (or new weight and weight gradient buffers in WU phase) and reorganizes the data before sending it to HBM2 channels. Data scatter/gather unit considers the channel allocations of different parameters, data precision, unroll factors and layer type.

3.3.6 HBM2 Initialization

The HBM initialization module loads the HBM with training data and other initial parameters. To initialize HBM with the training data, the data is loaded from the host PC to the on-chip memory (M20K) of the FPGA. The on-chip memory (M20K) works as an intermediate buffer for each pseudo-channel. Due to the limited on-chip memory resources, we used small buffers, and these buffers will be used multiple times to load a large amount of data to HBM. The HBM configurator and HBM control modules of the CNN training accelerator is reused to perform the HBM initialization. After loading all required training data, the CNN training accelerator is enabled. The HBM initialization is controlled by the host system.

3.4 Experimental Results

3.4.1 Experimental Setup

We evaluate our FPGA accelerator on two CNNs (ResNet-20 and VGG-like CNN) with CIFAR-10 as the training dataset. The control logic is also configurable to support large dataset (ImageNet) training, but consumes more FPGA resources to store and process larger input images. The initial weight parameters and configuration register values of benchmark CNNs are generated from our RTL compiler framework developed in Matlab. Intel S-10 MX (1SM21BHU2F53E2VGS1) and S-10 GX(1SG280LU3F50E3VGS1) were used as the target FPGA hardware. S-10 MX is equipped with 133 Mbits of M20K, 3,960 DSP blocks, 702K ALMs and 8GB HBM2 memory providing peak memory bandwidth of up to 512 GBps and S-10 GX includes 5,760 DSP blocks, 933K ALMs and 240 Mbits of M20K and 4GB DDR3 with 16.9GB/s bandwidth. Identical design optimizations has been performed on both S-10 MX and S-10 GX design for fair comparison.

All parameters use 16-bit floating-point precision to reduce the memory footprint compared to 32-bit floating-point precision. Since the DSP units of Intel S-10 GX/MX FPGAs only support 32-bit floating-point precision, 16-bit (32-bit) to 32-bit (16-bit) floating-point precision converters are used before (after) DSP computation to utilize the existing DSP blocks in S-10 FPGAs. The latency was measured using the functional simulation of the training accelerator. Using Intel Quartus 19.4 FPGA software, the accelerator was synthesized, placed/routed and the bitstream was uploaded to the FPGA board. Intel(R) Core (TM) i7-9800X CPU is used as the host system. For comprehensive comparison among FPGA, CPU, and GPU hardware for the same training tasks, we measured the power of each actual hardware system. FPGA board power consumption is measured using the Intel board test system (BTS) power monitor. The Intel BTS power monitor reported junction temperature of 47°C. Intel(R) Core (TM) i7-9800X CPU power is measured with the powerstat command using RAPL domains. To evaluate the performance of GPUs, we developed a floating point CIFAR-10 training model using PyTorch Paszke *et al.* (2017). Tesla V100 GPU power measurements are done using CUDA nvidia-smi API and Jetson Nano power measurements are done using Nvidia tegrastat utility. To minimize measurement inaccuracy, 20 samples of power measurements are averaged over the duration of one epoch training.

3.4.2 Results and Analysis

Table 4 shows the resource utilization of two CNN benchmarks (ResNet-20 and VGG-like CNN) for the CIFAR-10 dataset. All the training images in a given batch are processed sequentially. This greatly reduces the on-chip memory requirements as we only read the data required to process one training image at a time from HBM2. We achieved maximum operation frequency of 185 MHz for S-10 MX and GX implementations.

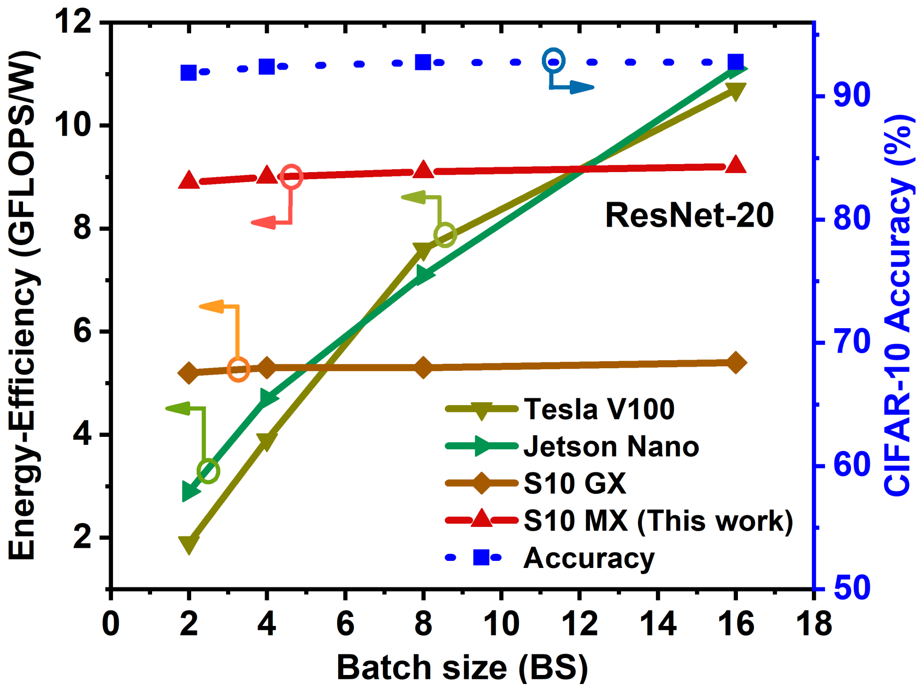


Figure 24. Energy and accuracy comparison of low-batch CNN training on Tesla V100 GPU, Jetson Nano, S-10 MX and GX devices.

For S-10 GX (with DDR3) implementation, Fig. 21(a) and Fig. 22(a) show the latency breakdown of the proposed accelerator for three training phases (FP, BP, and WU) of the last training image of a batch (involving actual weight updates) for

Table 4. Resource utilization for training tasks of ResNet-20 and VGG-like CNNs on Intel Stratix-10 MX and Stratix-10 GX FPGA.

CNN	FPGA	DSP	ALM	M20Ks	Registers	Freq.
ResNet-20	S10-MX	1040 (26%)	239k (34%)	2558 (13.9M)	390k	185 MHz
VGG-like	S10-MX	1046 (26%)	221k (31%)	2998 (11.4M)	353k	185 MHz
ResNet-20	S10-GX	1043 (18%)	148k (16%)	1779 (14M)	385k	185 MHz
VGG-like	S10-GX	1044 (18%)	97k (10.4%)	1297 (11M)	167k	185 MHz

ResNet-20 CNN and VGG-like CNN, respectively. The latency breakdown includes the reading of input pixels and weights from off-chip memory (Inpx/wt rd), computing the convolution outputs (MAC), writing the output pixels (oupx wr), wt gradients (wt grads) and new weights (wts) back to HBM. In the overall training time, the off-chip DDR3 memory consumes 47% of latency and logic consumes 53%. For memory-bound CNNs, even with high hardware parallelism, the low bandwidth of DDR3 memory will limit the performance Venkataramanaiah *et al.* (2019a). This critical memory bandwidth bottleneck can be addressed using HBM2.

Fig. 21(b) and Fig. 22(b) provide the latency breakdown of the proposed FPGA accelerator implemented on the S-10 MX device using HBM2. WU phase consumes longer latency than FP/BP phases, as it involves weight gradient computation, gradient accumulation and computation of new weights. The high off-chip memory bandwidth provided by HBM2 significantly reduces the latency consumed to read/write the activations and weights from/to the off-chip memory. As a result, the logic latency dominates the total latency, compared to S-10 GX implementation with DDR3 in all three phases of training. Further latency improvement could be achieved by increasing the number of parallel MAC arrays or by increasing the operating frequency. Using the proposed channel allocation scheme and HBM2 for the S-10 MX implementation, we achieved ∼4X reduction in off-chip memory latency and ∼1.5X reduction in system-level CNN training time, compared to those of the S-10 GX implementation with DDR3.

Fig. 23(a) and Fig. 23(c) provide the low-batch training throughput of two CNN benchmarks (ResNet-20 and VGG-like CNN) on Intel i7-9800X CPU, Jetson Nano embedded platform, Tesla V100 GPU, S-10 GX FPGA and S-10 MX FPGA. The overall training time of GPUs significantly increases with lower batch sizes. The proposed FPGA training accelerator achieves better throughput compared all other hardware platforms on both the benchmarks for small batch sizes of 2 and 4. Tesla V100 provides better throughput for higher batch sizes (8 and 16) but at the cost of high power consumption. The power consumption of all hardware platforms for different batch sizes are shown in Fig. 23(b) and Fig. 23(d). The FPGA power consumption is low because of less utilization ($\sim$30%) of FPGA resources, operating frequency of 185MHz and junction temperature of 47°C reported by Intel BTS tool. Compared to S-10 MX FPGA, Jetson Nano consumes less power ($\sim$5W) but suffers from long training latency. For ResNet-20/VGG-like CNNs, our FPGA implementation of CNN training using S-10 MX with HBM2 is $\sim$4.5-9.7X more energy-efficient compared to Tesla V100 GPU, $\sim$3-7X more energy-efficient compared to low-power Jetson Nano embedded platform, and $\sim$1.7X more energy-efficient compared to implementation on S-10 GX with DDR3. Our proposed S-10 MX design with HBM2 addresses the critical memory bottleneck problem for CNN training and the custom architecture enables efficient low-batch training.

Fig. 24 shows the overall energy-efficiency and accuracy comparison of Tesla V100 GPU, Jetson Nano, S-10 MX, S-10 GX devices for ResNet-20 training across different batch sizes. It can be seen that the low-batch training accuracy has minimal degradation compared to high-batch training accuracy Masters and Luschi (2019). At the same frequency and MAC array size, S-10 MX design provides 1.7X improvement in energy-efficiency compared to S-10 GX design by greatly reducing the off-chip communication latency.

3.5 Conclusion

This chapter presents a flexible CNN training accelerator on FPGA using HBM2, which performs end-to-end training of modern CNNs involving residual connections and stride-2 convolutions. The FPGA accelerator is implemented on Intel S-10 MX (with HBM2) and S-10 GX (with DDR3) devices, demonstrating system-level benefits of HBM2 over conventional DDR3 off-chip memory. The proposed accelerator achieves 4.5-9.7X energy-efficiency improvement compared to Tesla V100 GPU and 7-11X improvement in throughput compared to that of Intel i7-9800X CPU for low-batch training tasks of ResNet-20/VGG-like CNNs.

Chapter 4

EFFICIENT AND MODULARIZED TRAINING ON FPGA FOR REAL-TIME APPLICATIONS

Training of deep Convolution Neural Networks (CNNs) requires a tremendous amount of computation and memory and thus, GPUs are widely used to meet the computation demands of these complex training tasks. However, lacking the flexibility to exploit architectural optimizations, GPUs have poor energy efficiency of GPUs and are hard to be deployed on energy-constrained platforms. FPGAs are highly suitable for training, such as real-time learning at the edge, as they provide higher energy efficiency and better flexibility to support algorithmic evolution. This chapter first develops a training accelerator on FPGA, with 16-bit fixed-point computing and various training modules. Furthermore, leveraging model segmentation techniques from Progressive Segmented Training, the newly developed FPGA accelerator is applied to online learning, achieving much lower computation cost. We demonstrate the performance of representative CNNs trained for CIFAR-10 on Intel Stratix-10 MX FPGA, evaluating both the conventional training procedure and the online learning algorithm. The demo is available at https://github.com/dxc33linger/PSTonFPGA_demo.

4.1 Introduction

The recent development of machine learning algorithms and computing hardware has enabled many modern edge applications, such as autonomous vehicles, surveillance drones, and robots. Training of these ML-edge applications is typically performed on cloud servers because of their high computing capability. Sending the data to the cloud incur large latency overhead and raises privacy/security concerns. Training at the edge enables limited data exchange with the cloud and helps in personalizing, improving energy efficiency and protecting the private data. The edge devices are also preferred to handle the learning from a data stream over time locally and in real-time, *i.e.* online learning.

In order to enable online learning at the edge for real-time applications, several major challenges need to be solved: (1) When new data arrives in a stream, there is very limited or even no access to previously learned data. Yet the learned knowledge (*i.e.* network parameters) from previous data should not be forgotten (*i.e.* overwritten or deteriorated due to the learning of new observations) Kirkpatrick *et al.* (2017); Chaudhry *et al.* (2018); Li and Hoiem (2017); Rebuffi *et al.* (2017b). (2) The network should be able to update its parameters according to the incoming data stream. It is preferred that such adaption is completed locally and in real-time for an edge device Du *et al.* (2019a); Venkataramanaiah *et al.* (2019b). (3) Although GPUs provide remarkably high parallelism and throughput making it a viable option for real-time learning, they are not suitable for power constrained platforms. Hardware design for flexible and energy efficient training at the edge is challenging due to design complexity, large computation/memory/power requirement and other resource budges Han *et al.* (2016b, 2015); Du *et al.* (2019b); Liu *et al.* (2015); Li *et al.* (2015).

FPGAs are well suited to exploit these algorithmic advances and tackle the above-mentioned challenges as they provide high energy efficiency, good flexibility, and large on-chip and off-chip memories. Several FPGA based training/inference accelerators have been proposed Liu *et al.* (2018); Gomperts *et al.* (2011); Rafael *et al.* (2005); Liu *et al.* (2017); Zhao *et al.* (2016); Choi *et al.* (2018); Guo *et al.* (2019) but they fail to show end-to-end training capability. Venkataramanaiah *et al.* (2019b) proposes an FPGA based fixed-point training accelerator capable of demonstrating end-to-end training. An RTL generator is used to generate the architecture according to the network structure and design requirements. The proposed accelerator can also support novel training methodologies like PST and provides great flexibility to exploit optimizations.

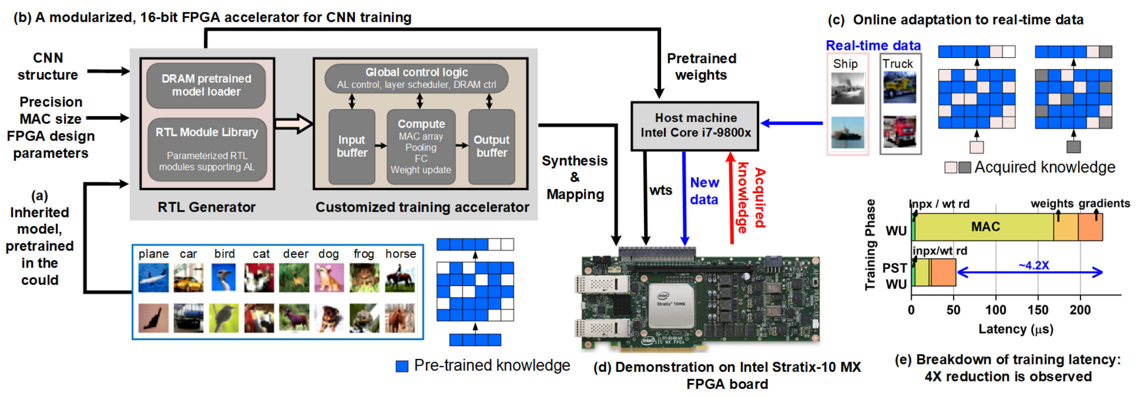

(b) A modularized, 16-bit FPGA accelerator for CNN training
CNN structure
Precision
MAC size
FPGA design parameters
DRAM pretrained model loader
RTL Module Library
Parameterized RTL modules supporting AL
RTL Generator
Global control logic
AL control, layer scheduler, DRAM ctrl
Input buffer
Compute
MAC array
Pooling
FC
Weight update
Output buffer
Customized training accelerator
(a) Inherited model, pretrained in the could
plane car bird cat deer dog frog horse
Pre-trained knowledge
Synthesis & Mapping
Pretrained weights
Host machine
Intel Core i7-9800x
wts
New data
Acquired knowledge
(d) Demonstration on Intel Stratix-10 MX FPGA board
(c) Online adaptation to real-time data
Real-time data
Ship
Truck
Acquired knowledge
Training Phase
WU
PST WU
inpx / wt rd
weights gradients
MAC
inpx/wt rd
~4.2X
0
50
100
150
200
Latency (µs)
(e) Breakdown of training latency: 4X reduction is observed

Figure 25. The demonstration system consists of Intel Straix-10 FPGA initialized with pre-trainined model parameters. The new data is streamed to the FPGA and learned locally in real-time using PST algorithm.

In this work, we demonstrate online CIFAR-10 CNN learning on an FPGA based 16-bit fixed-point training accelerator Venkataramanaiah *et al.* (2019b) on Intel Stratix-10 MX FPGA Deo *et al.* (2016). The proposed accelerator is augmented to support PST Du *et al.* (2019a) which further improves the performance of online CNN training. We also demonstrate the PST algorithm by deploying the pretrained, segmented model (*i.e.* selected weights are frozen in the network) on the FPGA and training the network with new real-time data.

4.2 System Overview

4.2.1 Demo System

Figure 25 depicts the overall system setup to demonstrate training of CNNs using PST algorithm. First, a large amount of knowledge is pretrained and important model parameters are frozen in the network (Figure 25a) following the process described in Du *et al.* (2019a). The pretrained model is sent to RTL generator which generates the customized training accelerator and HBM2 memory initialization files (Figure 25b). The generated training accelerator uses the frozen weights stored in HBM2 and performs the inference. This forms an inherited model, which is used to acquire new knowledge; the model is then exposed to a new unlearned data stream and the network parameters are updated accordingly in real-time on the FPGA (Figure 25c). The entire system is demonstrated on Intel Stratix-10 MX FPGA board (Figure 25d). Benefiting from the model inheritance, the online training of new observations requires much less computation cost and lower latency, as compared to traditional continual learning scheme that learns from scratch. PST greatly aids in improving the computation cost by updating only the required weights instead of updating all the network parameters in the traditional training schemes. Latency breakdown graph (Figure 25e) shows the latency benefit of using PST compared to conventional training in the weight update (WU) phase.

4.2.2 CNN Training Hardware

The RTL generator generates the CNN training hardware using the high-level network details given by the user. It uses a highly parameterized handwritten RTL module library designed to support various layers of CNN training. The user can also reconfigure the architecture by changing the FPGA design parameters such as precision, MAC array size, tiling, and layer scheduling. To support novel training algorithms like PST, the RTL generator is designed to read the pretrainied CNN model and generate the HBM2 initialization files to load the frozen weights.

The CNN training hardware is flexible to support forward pass (FP), backward pass (BP) and weight update (WU) phases of training. The hardware consists of a global control logic that governs all the modules and enables layer by layer execution by using the parameters generated by the RTL generator. The HBM2 stores all the initial weight parameters (or weights from a pretrained model), activations and computed weight gradients/new weights. The input/output on-chip buffer is used to store the input/output parameters required for a given layer. For example, while computing a convolution layer the input buffers stores the input activations, weights and output buffers store the convolved outputs.

The core compute blocks reads the data from the input buffers and perform the computation based on the layer type and the outputs are sent to output buffers. The convolution block uses a 2D systolic MAC array flexible to support all three phases of the training. The weight update block computes and accumulates the weight gradients. At the end of the batch, the accumulated weight gradients are scaled and new weights are computed using the stochastic gradient descent algorithm. To support PST where we need to only update the selected weights, the control logic was augmented to skip the HBM2 access if the frozen weights thereby reducing the off-chip communication. The weight updates and weight gradient computation was performed only for the selected weights.

4.2.3 Demonstration Setup

We showcase our system with CIFAR-10 Krizhevsky *et al.* (2009) dataset. The CIFAR-10 dataset consists of 60,000 32×32 color images in 10 classes, with 5,000 training images and 1,000 testing images per class. The classes include common objects such as plane, bird, truck, etc. We demonstrate online learning on FPGA with a CNN structure of 16C3-16C3-MP-32C3-32C3-MP-64C3-64C3-MP-FC, where 'NCk' represents convolution layer with 'N' output feature maps and kernel size of 'k', 'MP' represents max pooling layer and 'FC' represents a fully connected layer. The accelerator was synthesized by Intel Quartus 19.2 at 150 MHz frequency. We used Stratix-10 MX equipped with HBM2 as the target hardware and Intel(R) Core(TM) i7-9800X as a host machine. All the parameters used 16-bit fixed point precision.

Chapter 5

FIXYNN: EFFICIENT HARDWARE FOR MOBILE COMPUTER VISION VIA TRANSFER LEARNING

The computational demands of computer vision tasks based on state-of-the-art Convolutional Neural Network (CNN) image classification far exceed the energy budgets of mobile devices. This chapter proposes FixyNN, which consists of a fixed-weight feature extractor that generates ubiquitous CNN features, and a conventional programmable CNN accelerator which aprocesses a dataset-specific CNN. Image classification models for FixyNN are trained end-to-end via transfer learning, with the common feature extractor representing the transfered part, and the programmable part being learnt on the target dataset. Experimental results demonstrate FixyNN hardware can achieve very high energy efficiencies up to 26.6 TOPS/W ($4.81\times$ better than iso-area programmable accelerator). Over a suite of six datasets we trained models via transfer learning with an accuracy loss of $< 1\%$ resulting in up to 11.2 TOPS/W – nearly $2\times$ more efficient than a conventional programmable CNN accelerator of the same area.

5.1 Introduction

Real-time computer vision (CV) tasks such as image classification, object detection/tracking and semantic segmentation are key enabling technologies for a diverse range of mobile computing applications, including augmented reality, mixed reality, autonomous drones and automotive advanced driver assistance systems (ADAS). Over

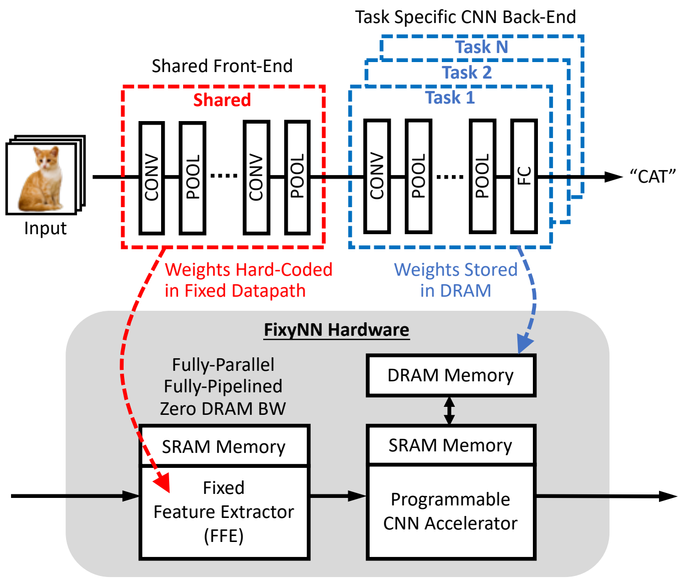

Task Specific CNN Back-End
Task N
Task 2
Task 1
Shared Front-End
Shared
Input
CONV
POOL
CONV
POOL
CONV
POOL
POOL
FC
"CAT"
Weights Hard-Coded in Fixed Datapath
Weights Stored in DRAM
FixyNN Hardware
Fully-Parallel Fully-Pipelined Zero DRAM BW
DRAM Memory
SRAM Memory
SRAM Memory
Fixed Feature Extractor (FFE)
Programmable CNN Accelerator

Figure 26. FixyNN proposes to split a deep CNN into two parts, which are implemented in hardware using a (shared) fixed-weight feature extractor (FFE) hardware accelerator for the shared front-end and a canonical programmable accelerator for the task-specific back-end.

the past few years, convolutional neural network (CNN) approaches have rapidly displaced traditional hand-crafted feature extractors, such as Haar Viola and Jones (2004) and HOG Dalal and Triggs (2005). This shift in focus is motivated by a marked increase in accuracy on key CV tasks such as image classification Simonyan and Zisserman (2014). However, this highly desirable improvement in accuracy comes at the cost of a vast increase in computation and storage Suleiman *et al.* (2017), which must be met by the hardware platform. Mobile devices exhibit constraints in the energy and silicon area that can be allocated to CV tasks, which limits the adoption of CNNs at high resolution and frame-rate (e.g. 1080p at 30 FPS). This results in a gap in energy efficiency between the requirements for real-time CV applications and the power constraints of mobile devices.

Two key trends that have recently emerged are starting to close this energy efficiency gap: more efficient CNN architectures and more efficient hardware. The first is the design of more compact CNN architectures. *MobileNetV1* Howard *et al.* (2017a) was an early and prominent example of this trend, where the CNN topology is designed to minimize both the number of multiply-and-accumulate (MAC) operations and the number of parameters, which is essentially the compute and storage required of the hardware platform. MobileNetV1 similar accuracy to VGG (top-5 ImageNet 89.9% vs. 92.7%), with only ∼3% of the total parameters and MACs. The second trend is the emergence of specialized hardware accelerators tailored specifically to CNN workloads. Typical optimizations applied to CPU, GPU and accelerators include: provision for small floating-point and fixed-point data types, use of optimized statically-scheduled scratchpad memories (as opposed to cache memories), and an emphasis on wide dot-product and matrix multiplication datapaths.

In this chapter we describe **FixyNN**, which builds upon both of these trends, by means of a hardware/CNN co-design approach to CNN inference for CV on mobile devices. Our approach (Figure 26) divides a CNN into two parts. The first part of the network implements a set of layers that are common for all CV tasks, essentially producing a set of universal low-level CNN features that are shared for multiple different tasks or datasets. The second part of the network provides a task-specific CNN back-end. These two CNN parts are then processed on different customized hardware. The front-end layers are implemented as a heavily optimized *fixed-weight feature extractor (FFE)* hardware accelerator. The second part of the network is unique for each dataset, and hence needs to be implemented on a canonical programmable CNN hardware accelerator Nvidia (2019); Arm (2019). Following this system architecture, FixyNN diverts a significant portion of the computational load from the CNN accelerator to the highly-efficient FFE, enabling much greater performance and energy efficiency.

The use of highly aggressive hardware specialization in the FFE makes FixyNN a significant step forward towards closing the energy efficiency gap on mobile devices. At the same time, by leveraging transfer learning concepts, we are able to exploit aggressively optimized specialized hardware without sacrificing generalization.

This chapter describes and evaluates **FixyNN**; the main contributions are listed below:

- A description of a hardware accelerator architecture for the fixed-weight feature extractor (FFE), including a survey of the potential optimizations.
- An open-source tool-flow DeepFreeze (2018) for automatically generating and optimizing an FFE hardware accelerator from a TensorFlow description.
- Demonstration of the use of *transfer learning* to generalize a single common FFE to train a number of different back-end models for different datasets.
- Present results that compare **FixyNN** against a conventional baseline at iso-area.

5.2 Related Work

CNN Hardware Accelerators. There is currently huge research interest in the design of high-performance and energy-efficient neural network hardware accelerators, both in academia and industry Barry *et al.* (2015); Arm (2019); Nvidia (2019); Reagen *et al.* (2017a). Some of the key topics that have been studied to date include dataflows Chen *et al.* (2016b); Samajdar *et al.* (2018), optimized data precision Reagen *et al.* (2016), systolic arrays Jouppi *et al.* (2017a), sparse data compression and compute Han *et al.* (2016a); Albericio *et al.* (2016); Parashar *et al.* (2017); Yu *et al.* (2017); Ding *et al.* (2017); Whatmough *et al.* (2018), bit-serial arithmetic Judd *et al.* (2016), and analog/mixed-signal hardware Chen *et al.* (2016a); LiKamWa *et al.* (2016);

Shafiee *et al.* (2016); Chi *et al.* (2016); Kim *et al.* (2016); Song *et al.* (2017). There is also published work on hardware accelerators optimized for image classification for real-time CV Buckler *et al.* (2018); Riera *et al.* (2018); Zhu *et al.* (2018), along with simulation tools SCALE-Sim (2019).

Image Processing Hardware Accelerators. The hardware design of the fixed feature extractor in FixyNN is reminiscent of image signal processing hardware accelerators. In particular, the use of native convolution and line-buffering have been explored in prior works including Ragan-Kelley *et al.* (2013); Hegarty *et al.* (2016, 2014); Lee and Messerschmitt (1987); Horstmannshoff *et al.* (1997).

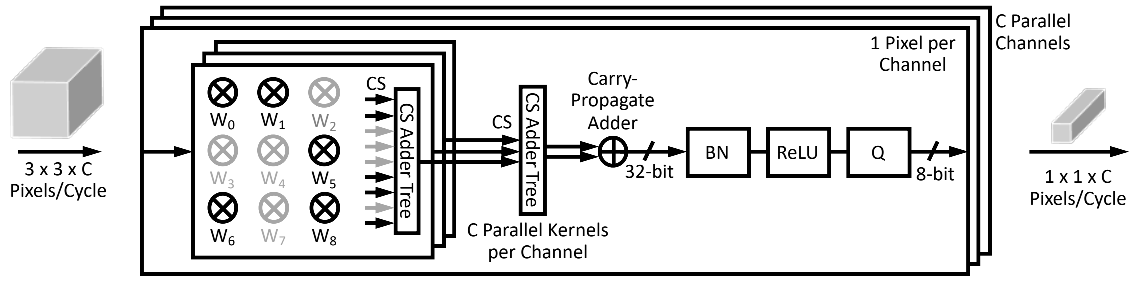


Figure 27. A fully-parallel fixed-weight native convolution hardware datapath stage for a 3×3 CONV layer. Other CNN layer shapes are implemented in an identical fashion, but with different dimensions. “CS” denotes carry-save arithmetic representation. “BN” denotes batch normalization and incorporates the bias term. “Q” denotes a programmable quantization function that converts from 32-bit to 8-bit. The multiplier symbols actually represent fixed-weight shift-add scalers with a single input operand. Grey multipliers and signals denote hardware removed due to pruned zero or small non-zero weights.

Transfer Learning and Domain Adaptation. In FixyNN, we use transfer learning techniques to share an optimized fixed feature extractor amongst multiple different back-end CNN models. Yosinski *et al.* (2014) first established the transferability of features in a deep CNN, outlining that the early layers of a CNN learn generic features that can be transferred to a wide range of related tasks. Fine-tuning the model on the new task yields better performance Yosinski *et al.* (2014) than training from scratch. Transfer learning has subsequently found a wide range of applications. For example, a deep CNN trained on the **ImageNet** dataset Russakovsky *et al.* (2015) was successfully transferred to detect pavement distress in roads Gopalakrishnan *et al.* (2017). Interestingly, more recent work demonstrated it is also possible to fix the last fully-connected layer in a CNN as a Hadamard matrix Hoffer *et al.* (2018).

Domain adaptation Tzeng *et al.* (2015) is a concept closely related to transfer learning. It refers to learning adaptive models that work on different visual domains (e.g. hand-written digits versus printed street numbers). The residual adapter architecture Rebuffi *et al.* (2017a, 2018) marks the recent progress in this field to efficiently learn parametrized models for several tasks and domains simultaneously. FixyNN can benefit from future advances in transfer learning and domain adaptation techniques.

Hardware Generators for CNN Accelerators. A number of previous works have proposed solutions to automatically generate optimized hardware accelerator designs Venieris *et al.* (2018); Mahajan *et al.* (2016); Sharma *et al.* (2016); Hernández-Lobato *et al.* (2016); Reagen *et al.* (2017b). There are also some relevant contributions from the image processing domain Ragan-Kelley *et al.* (2013); Hegarty *et al.* (2014) that similarly generate high-performance convolution hardware. The DeepFreeze tool we developed in this work was a necessity in order to explore fixed-weight feature extractors, as hand-written Verilog modules containing millions of parameters would have been impractical otherwise. We did not explore applying FixyNN on FPGAs Umuroglu *et al.* (2017) in this chapter, but plan to look at this in future work. We are also planning to explore heavily-constrained Internet-of-Things (IoT) applications Kodali *et al.* (2017) in future work.

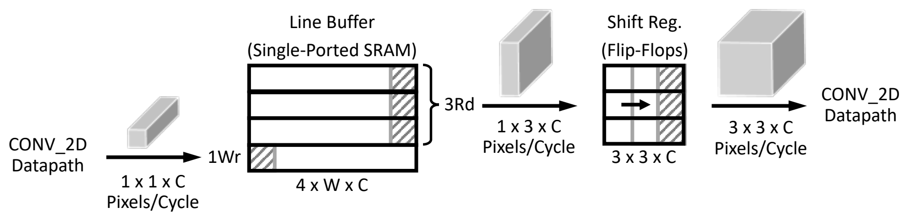


Figure 28. Overview of the fully-pipelined feature map buffering micro-architecture between consecutive layers of fixed-weight fully-parallel CNN layers. This example illustrates the case for two consecutive CNN layers with 3×3 kernels.

5.3 Fixed-Weight Feature Extractor Hardware Design

FixyNN combines two specialized hardware accelerators: a heavily-optimized *fixed-weight* feature extractor (FFE), and a more conventional *programmable* CNN accelerator. This combination provides very high energy efficiency without sacrificing generalization across a range of datasets. Fixing the weights of a convolution (CONV) layer in a fully-parallel, fully-pipelined FFE accelerator enables a number of aggressive hardware optimizations in the FFE, and therefore results in significantly improved throughput and energy efficiency, which cannot be matched by a programmable accelerator. We emphasize five major optimizations stemming from fixing weights in the hardware.

- **Fixed Shift-Add Scalers.** Hardware weight multipliers, which ordinarily have two input operands, are transformed into simple fixed scalers with a single input operand. Fixed scalers are formed by simply adding a series of hard-coded bit-wise shifts of the input operands and are very cheap in hardware. The number of bit-shifts and additions required per fixed multiplier is determined by the number of non-zero bits in the binary representation of the weight (i.e. Hamming weight). This represents a very significant *strength reduction* and results in substantial reduction in power consumption, logic delay and silicon area Cooper *et al.* (2001).

- **Zero-Overhead Weight Pruning.** Weights with a zero or small non-zero value are redundant and can be explicitly removed from the datapath hardware. This results in a reduction in datapath area and power, linearly proportional to the weight sparsity for the layer. In a programmable CNN accelerator, there is overhead in exploiting sparsity, due to the requirement to encode the position in the matrix of non-zero weights Parashar *et al.* (2017).
- **Optimized Intermediate Precision** The precision used for multipliers and accumulators are typically set to the worst-case values in a programmable accelerator. However, in the FFE, we know the weights and their magnitude a-priori, and can therefore perform static analysis to optimize the product and accumulator bit-widths, which further reduces the hardware cost.
- **Zero DRAM Bandwidth.** The weights for the CONV layers implemented in the FFE are hard-coded in the datapath logic and do not need to be stored in memory. Hence, unlike a programmable accelerator, there is no need to access expensive off-chip DRAM when using the FFE.
- **Minimal Activation Storage.** By using native convolution that does not incur storage overheads for *IM2COL* expansion Warden (2015), and also implementing fully-pipelined hardware, we can reduce storage of activation feature maps to a minimum. This is in contrast to programmable accelerators, which typically process layers in a serial fashion, to maximize weight reuse, and therefore must buffer the entire output feature map for each layer at once.

In the remainder of this section, we describe the hardware design of the FFE. We first describe the arithmeric datapath stage, followed by the buffering stage, and finally the tool flow to automatically implement and optimize the FFE from a high-level model description.

5.3.1 Fully-Parallel Fixed-Weight CNN Datapath

The computation for each CONV layer is implemented as a flat, fully-parallel, pruned fixed-weight arithmetic logic stage (Figure 27). The fixed scalars that replace the multipliers are generated by the synthesis tool, as the weights are embedded as literals in the Verilog hardware description language (HDL). These fixed scalars are also subsequently optimized by the synthesis tool to reduce gate-count, using techniques such as Booth recoding Booth (1951), canonical signed-digit encoding and other well-known datapath optimizations Zimmermann (2009). The adder trees following the multipliers are combined by the synthesis tool into a wide carry-save (CS) addition tree with a single carry-propagate adder Zimmermann (2009). Following the convolutions, there are operations in each layer for batch normalization (BN) [1], which scale and shift activations (and integrates the bias term), rectified linear unit (ReLU) activation function and a quantization step to convert from the wider precision of the accumulator node back to the narrow representation for activation data. As we will describe in Section 5.6.2, the BN parameters are important for transfer learning, so we keep these programmable, using dedicated registers. This is not a big overhead as there are a very small number of BN parameters. Simple max pooling layers are also supported.

[1] A widely-adopted technique to improve performance and stability by ensuring layer outputs have zero mean and unit variance Ioffe and Szegedy (2015).

5.3.2 Fully-Pipelined CNN Buffering

In contrast to programmable CNN accelerators that typically convert convolution into Generic Matrix Multiplication (GEMM), computing the CNN in a serial fashion, the FFE implements native convolution with *fully-pipelined* CONV layers. However, buffering is required between consecutive datapath stages, because a typical $3 \times 3 \times C$ CONV kernel, where C is the number of channels, consumes a $3 \times 3 \times C$ input pixel tensor per cycle, but generates only a single small $1 \times 1 \times C$ output tensor, where C is the number of output channels. Hence, we must buffer several $1 \times 1 \times C$ outputs into a larger $3 \times 3 \times C$ input for the next layer.

This buffering function is achieved using the common approach of a *line buffer*, which stores activations of each layer row by row until the required tensor size has been built up. Figure 28 gives an overview of the arrangement for a simple CNN layer with a 3×3 kernel shape. In this case, due to the discrepancy in input/output tensor dimensions, we need to buffer three full rows before we can start to generate the larger tensors we need for the following layer. We implement the line buffer using simple single-port SRAMs, and therefore actually require four independent SRAM banks, such that we can write a single-row patch to one bank per cycle, and read the three-row patch from three banks per cycle, concurrently. After reading/writing the last pixel in a row, the four banks are rotated to overwrite the data associated with the oldest row (double-buffer). This arrangement can be further optimized Hegarty *et al.* (2014, 2016); Ragan-Kelley *et al.* (2013), for example, by using dual-port SRAMs, which were not available to us in our process technology.

Following the SRAM line buffer, a flip-flop based shift-register is implemented such that the convolution window moves efficiently over the feature map, without

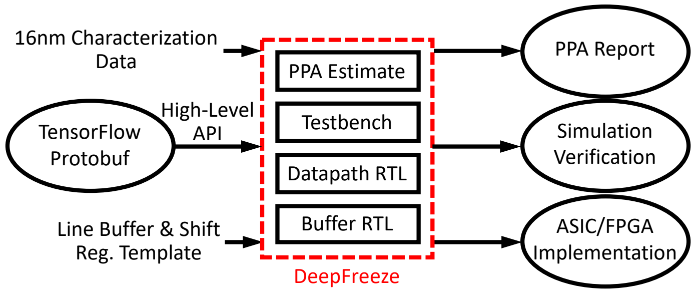


Figure 29. The DeepFreeze tool flow automatically generates Verilog HDL for optimized fixed feature extractors from a high-level description of the model in a software framework such as TensorFlow.

re-reading data. The shift-register consumes $1 \times 3 \times C$ pixels per cycle from the SRAM line buffer and outputs a $3 \times 3 \times C$ pixel volume per cycle. The advantage of the shift-register stage is an SRAM bandwidth reduction of $3\times$. Larger CNN kernels, such as $5 \times 5 \times C$ and $7 \times 7 \times C$ are arranged in a similar fashion, with dimensions scaled appropriately. Strides of more than one are also supported. We also make a provision to allow the activation data to be optionally streamed from any intermediate buffer stage, to allow a smaller number of fixed layers to be utilized for models that are more difficult to train via transfer learning.

5.3.3 DeepFreeze Tool Flow

To facilitate implementing FFE accelerators with possibly millions of hard-coded weights, we developed an open-source tool called *DeepFreeze*. DeepFreeze generates fixed CNN hardware accelerator designs for a specified set of layers from a model described in a standard machine learning software framework, such as TensorFlow.

DeepFreeze first parses the network from a given framework into an internal representation of that model. It then generates a fixed datapath from the model description using a direct code generation step, which reads the model weights and emits Verilog source code with the weights embedded as immediate values. Zero weights are automatically removed entirely from the hardware (pruning is assumed to be performed outside of the DeepFreeze tool-flow). During the datapath generation, the bit-widths of the fixed scalars are optimized individually based on the scalar value. The precision for the intermediate activations is specified as a hardware parameter, along with the accumulator width. The final Verilog is constructed by connecting consecutive combinational datapath stages with buffer stages, which are instantiated from a parameterized Verilog template. The generated Verilog can be directly read in by any synthesis tool for ASIC or FPGA implementation. DeepFreeze also generates a validation suite with testbench for simulation. Finally, the tool generates an estimate of power, performance and area (PPA) for the high-level model provided. This estimate uses simple extrapolations from data derived from implementation experiments, and is useful for rapid design space exploration.

5.4 Transfer Learning with a Fixed Feature Extractor

In the previous section, we described the hardware design of a fixed feature extractor accelerator that offers substantially better throughput/latency and energy compared to programmable CNN accelerators. However, we do not propose to fix the whole network for two reasons. Firstly, for large models, the silicon area of the fixed hardware accelerator would be prohibitive in most applications. Secondly, fixing the whole network would make it impossible to change the task or dataset; it would essentially result in a single-function hardware accelerator. Therefore, in FixyNN we propose to fix only a portion of the front-end of the network, and use a canonical programmable accelerator to process the remainder (Figure 26). The fixed portion provides a set of more universal CNN features specific to the application domain of vision tasks, whereas the programmable portion of the network specific to a given a dataset. In this section, we briefly outline how to train arbitrary CNN vision models that incorporate a fixed feature extractor implemented a-priori.

Transfer learning is a concept that we introduced in Section 5.2. Here, we highlight transfer learning as a concept that suggests it is perfectly feasible to train a new model that incorporates a fixed feature extractor, at least within the same application domain of CV. As previously motivated, the central advantage is that the performance and power efficiency of the fixed feature extractor are significantly superior. In addition, there are a number of auxiliary advantages, such as a significantly smaller model to store, maintain and update.

The CNN model architecture we use in this work is MobileNetV1 Howard *et al.* (2017a), which is an efficient model designed for mobile computer vision. MobileNet exploits the efficient depth-wise separable convolution layer, which is composed of M $3 \times 3 \times 1$ depth-wise convolution filters (M is the number of input channels) and N $1 \times 1 \times M$ point-wise convolution filters (N is the number of output channels). A depth-wise separable convolution layer costs between $8\times$ to $9\times$ less computation than a traditional 3×3 kernel. Additionally, MobileNet is a suitable architecture for FixyNN because the FFE can directly concatenate the depth-wise and point-wise kernels without any buffering, as the output dimensions of the depth-wise layer are the same as the input dimensions of the point-wise layer. MobileNet has 13 CONV layers in total, with a fully connected layer for final classification. The first CONV layer is a traditional convolution layer and the remaining 13 CONV layers are depth-wise separable layers. A width multiplication factor α Howard *et al.* (2017a) is introduced to explore different size models with the same basic architecture. For a given layer in the baseline MobileNet that has M input channels and N output channels, the same layer in MobileNet-α has αM input channels and αN output channels. The width multiplier value of α reduces the computational cost and parameters by roughly α^2.

The procedure for training an image classification model on a given dataset is as follows. We start by assuming the fixed feature extractor has already been defined, using the MobileNet architecture trained on the **ImageNet** data. The early-layer weights are fixed for the feature extractor, while the remainder of the network is fine-tuned on the target dataset. Further details of the training procedure can be found in Section 5.5.2.

As discussed in Section 5.3, fixing the weights in the feature extractor leads to a number of optimizations that cannot be as easily exploited in a programmable accelerator. We may gain further benefits in latency, energy and silicon area through more aggressive optimization of the CNN layers for the fixed feature extractor by forcing more sparsity and Hamming weight reduction during training and fine-tuning.

5.5 Experimental Methodology

To evaluate FixyNN, we conduct experiments in both hardware modeling and transfer learning. The hardware modeling experiments compare FixyNN against state-of-the-art hardware accelerator designs. The transfer learning experiments evaluate generalization of a fixed feature extractor across a set of tasks.

5.5.1 Hardware Modeling

FixyNN consists of two hardware components: the FFE, and a programmable CNN accelerator. The FFE is generated using our DeepFreeze tool (Section 5.3.3). We use 8-bit precision for weights and activation data, and 32-bit for accumulators. For ASIC implementation experiments, we use Synopsys Design Compiler with TSMC 16nm FinFET process technology to characterize silicon area. Timing analysis for throughput/latency is performed with Synposys PrimeTime. All simulations use a clock frequency of 810 MHz. Power characterization is performed using Synopsys PrimeTime PX with switching activity annotated from simulation trace data.

The programmable accelerator is based on published results for the NVIDIA Deep Learning Accelerator (NVDLA) Nvidia (2019). NVDLA is a state-of-the-art open-source neural network accelerator, with Verilog RTL for hardware implementation and a TLM SystemC simulation model that can be used for software development, system integration, and testing. NVDLA is configurable in terms of hardware resources. Table 5 summarizes the published performance of NVDLA in six nominal configurations.

Config.	#MACs	Buffer (KB)	16nm Area (mm^2)	TOPS	TOPS/W
A	64	128	0.55	0.056	2.0
B	128	256	0.84	0.156	3.8
C	256	256	1.00	0.358	5.6
D	512	256	1.40	0.728	6.8
E	1024	256	1.80	1.166	6.3
F	2048	512	3.30	2.095	5.4

Table 5. Published NVDLA configurations, reproduced from NVDLA

To explore the final FixyNN design space (Section 5.6.1), we combine PPA models of an FFE containing the first N layers of the network, along with the NVDLA programmable accelerator drawn from the published configurations. DeepFreeze is used to model the PPA of the fixed feature extractor. Since the hardware performance of the FFE is heavily dependent on the sparsity of the network, we assume a cautious 50% sparsity across the model for simplicity. Prior work has demonstrated that 50% of weights can be pruned from MobileNet with minimal accuracy loss Zhu and Gupta (2017). The hardware modeling of NVDLA is from published data. Because the latency of the FFE is much lower than that of the programmable NVDLA in the configurations we tested, we assume perfect clock gating in FixyNN to eliminate FFE power when idle. Finally, we do not model FC layers as they are heavily memory bound and we would never be able to fix them anyway due to the huge number of parameters.

5.5.2 Transfer Learning

The fixed feature extractor is constrained not only by silicon area considerations, but also by the achievable model accuracy. The foundational work on transfer learning showed that as more layers are transfered, the accuracy becomes limited due to change in representational power and the later layers are more task specific than the early layers Yosinski *et al.* (2014). In previous work, transfer learning is typically applied on big models such as AlexNet, which is prohibitively expensive from a hardware implementation point of view. Furthermore, it is arguably easier to perform transfer learning when the model capacity is very high as more parameters are available to fit the new dataset. In this chapter, we perform a set of transfer learning experiments showing good performance with fixed weights on MobileNet, a much more constrained model.

Inspired by the *visual decathlon challenge* Rebuffi *et al.* (2017a) introduced to explore multiple-domain learning for image recognition, we choose seven different image recognition tasks to design our experiments: **ImageNet** Russakovsky *et al.* (2015), **CIFAR-100** Krizhevsky (2009), **CIFAR-10** Krizhevsky (2009), **Street View House Numbers (SVHN)** Netzer *et al.* (2011), **Flowers102 (Flwr)** Nilsback and Zisserman (2008), **FGVC-Aircraft (Airc) Benchmark** Maji *et al.* (2013), and **The German Traffic Sign Recognition (GTSR) Benchmark** Stallkamp *et al.* (2012). These datasets vary in number of images, resolution and granularity. For example, **ImageNet** and **CIFAR-100** are diverse datasets with a wide range of objects, while **Flwr** and **Airc** are fine-grained recognition tasks for specific vision domains of flowers and aircrafts respectively.

For the first set of experiments, we use MobileNet-0.25, an efficient model with only 41 million MACs and 0.47 million parameters. The model is first trained on **ImageNet** to an accuracy of 49.8% (state-of-the-art for this small MobileNet model) and then transfered to the other six vision tasks. The baseline results are obtained by performing full-fledged fine-tuning, where all the parameters of the model are updated during fine-tuning on the new dataset. This is used as the baseline case for a model running on a programmable DLA. Six different FixyNN topologies are explored in these experiments, with different number of layers being fixed. In some topologies, all batch normalization layer scaling and bias parameters in the model are retrained on the new dataset. We call this configuration Adaptive Batch-Normalization (BN).

Stochastic gradient descent with an initial learning rate of 0.01 and momentum of 0.9 is used to perform fine-tuning (except for GTSR dataset, where an initial learning rate of 0.001 is used for better convergence). The learning rate is decayed $10\times$ every 100 epochs (200 epochs for GTSR). A batch size of 128 is used. The seven datasets come with different resolutions. For the purpose of standardization, all images are resized to 224×224 using bilinear interpolation. Data augmentation preproccessing is applied to all datasets. Random color distortion, flipping and cropping are applied. Horizontal left-right flipping is turned off for **SVHN** and **GTSR**, cropping ratio is also increased as these two datasets are street number and traffic sign photos. MobileNet-0.25 is a limited capacity model so little regularization is required. Weight decay of 4×10^{-5} is used in fine-tuning (4×10^{-4} for GTSR).

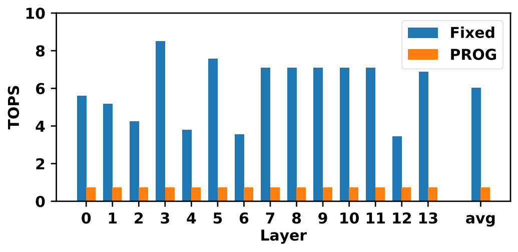
To demonstrate generalization of this approach, a second set of experiments are carried out using MobileNet-1.0. MobileNet-1.0 has 569 million MACs and 4.24 million parameters, which is about $10\times$ bigger than MobileNet-0.25. It is trained on **ImageNet** to an accuracy of 70.9%. We only transfer this model to **CIFAR100** to showcase the similar trend of transfer learning performance for a bigger model.

5.6 Experimental Results

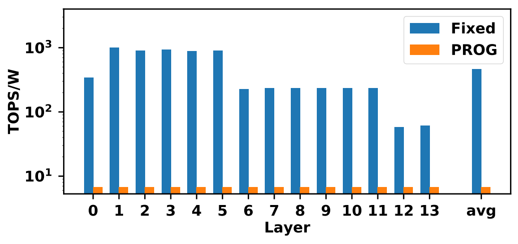
In this section, we first describe the hardware performance of FixyNN, then explore the CNN generalization performance and finally draw the two together with a discussion.

5.6.1 Hardware

To demonstrate the advantages of incorporating a FFE into a system, we begin by comparing the two hardware components of FixyNN. Figure 30 compares the throughput (TOPS) and energy efficiency (TOPS/W) for the FFE and programmable NVDLA accelerators over each of the 13 layers of MobileNet-0.25. Clearly, FFE outperforms NVDLA in all regards, showing an average improvement in TOPS and



(a) Throughput



(b) Energy Efficiency

Figure 30. Per-layer throughput and energy efficiency of a fixed-weight feature extractor vs programmable NVDLA on MobileNet-0.25.

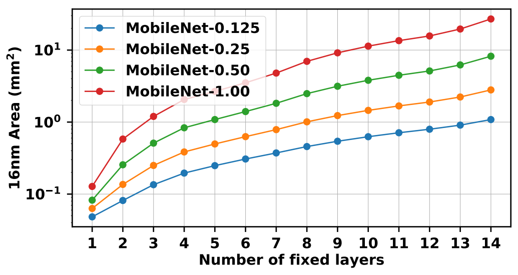
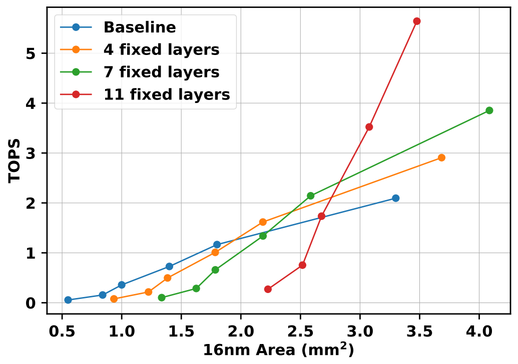
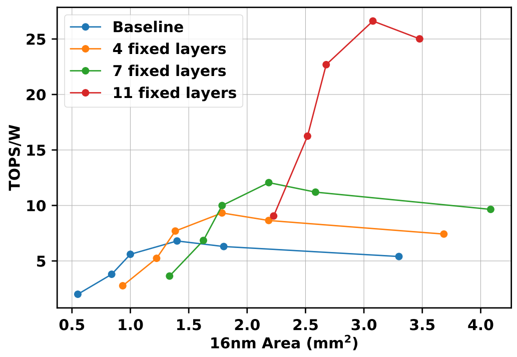


Figure 31. Cumulative area of a fixed feature extractor for MobileNets of varying width.

TOPS/W of 8.3× and 68.5×, respectively. This healthy improvement is essentially the motivation for exploring the fixed feature extractor. However, the silicon area required by the FFE is a practical limitation on the number of layers we can reasonably fix in the FFE. Figure 31 demonstrates how the area of the FFE scales with the number of fixed layers for several different size MobileNet networks. In FixyNN, we want to balance the distribution of layers between the FFE and the programmable accelerators



(a) Throughput



(b) Energy Efficiency

Figure 32. Performance and energy efficiency of different FixyNN topologies. Each line corresponds to a single size feature extractor being used with different sized programmable accelerators.

Design Parameters		FixyNN					Baseline		**Improvement**	
Priority	Area budget (mm^2)	Fixed layers	NVDLA Config.	Total Area (mm^2)	TOPS	TOPS/W	TOPS	TOPS/W	**TOPS**	**TOPS/W**
Throughput	2	None	E	1.80	1.17	6.30	1.17	6.30	**1.00×**	1.00×
	3	7	E	2.59	2.14	11.20	1.66	5.83	**1.29×**	1.92×
	4	11	E	3.48	5.64	25.01	2.21	5.29	**2.55×**	4.73×
Efficiency	2	7	C	1.79	0.66	9.99	1.15	6.31	0.57×	**1.58×**
	3	11	C	2.68	1.73	22.69	1.71	5.77	1.01×	**3.93×**
	4	11	D	3.08	3.52	26.62	1.96	5.53	1.80×	**4.81×**

Table 6. Pareto-optimal FixyNN configurations for a given area budget, with throughput and efficiency priority. "Improvement" is relative to an NVDLA configuration of comparable silicon area. All results shown are modeled in 16nm CMOS technology.

to maximize energy efficiency and generalization (Section 5.6.2), given silicon area constraints.

Having demonstrated the advantages of the fixed feature extractor on single individual layers, we now demonstrate a practical FixyNN system. We define a search space of potential FixyNN systems by combining a fixed feature extractor of a given size, and a programmable DLA of a given configuration (Table 5). The design space is given in Figure 32 for throughput and energy efficiency. Each line in these plots is a different number of fixed layers, while each marker on each line is a different configuration of the programmable accelerator (Table 5). Our baseline for comparison is a fully programmable NVDLA accelerator with no fixed layers, which represents the current state-of-the-art.

In terms of throughput (Figure 32a), all configurations scale approximately linearly with area. At small area budgets, the fully programmable baseline outperforms FixyNN, because the FFE is heavily bottlenecked by the programmable NVDLA, resulting in little benefit from the extra area consumed by the FFE. However at higher area budgets, FixyNN can afford to fix more layers, resulting in reduced load on the programmable DLA and large gains in throughput. In terms of energy efficiency (Figure 32b), the baseline NVDLA scales well with area initially, due to an increase in data re-use and other amortizations, however it saturates (and even falls off) as limitations on utilization or memory bandwidth prohibit further gains. Due to the exceptional energy efficiency of the FFE, as the load diverted from the NVDLA to the FFE increases, so too does the energy efficiency. This becomes significant at area budgets greater than 1mm^2, at which point it becomes more efficient to utilize silicon area to fix more layers of the network than it is to scaling up the programmable accelerator.

An additional advantage of the FFE is the fact that it does not require access to expensive off-chip DRAM memory for either weights or activations, since weights are fixed in the datapath and activations are minimally pipelined in efficient and compact line buffers on-chip. This saves power, and also sidesteps an important system-level constraint; NVDLA rapidly becomes bottlenecked on DRAM bandwidth as the accelerator is scaled up.

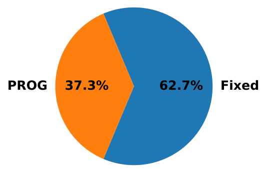
Table 6 gives pareto-optimal FixyNN configurations from the design space in Figure 32, given different design constraints. In general, this table shows it is more effective to implement a larger FFE at higher area budgets (above 1mm^2), as scaling the programmable NVDLA provides diminishing benefits beyond $\sim$1mm^2. With an area budget of 4mm^2, FixyNN provides up to 2.55× and 5.84× improvement in TOPS and TOPS/W respectively, at iso-area for MobileNet-0.25.

We chose to investigate the optimal configuration for energy efficiency at an area budget of 2-3mm^2 (11 fixed layers with NVDLA configuration C). Figure 33 shows a breakdown of the PPA between the FFE and the programmable DLA. This figure demonstrates how even though the fixed datapath performs a large majority of the operations in the network, it only takes a small fraction of the energy and latency that the programmable NVDLA requires.

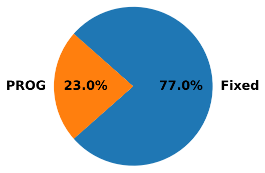
The optimal configurations of FixyNN are dependent on the size of the model. We repeated the experiment above, but using the larger MobileNet-1.00. FixyNN now provides benfits at area budgets greater than 3mm^2, compared to the 1mm^2 break-even point for MobileNet-0.25. At an area budget of 4mm^2, fixing the first 4 layers of the network provides a 1.28× improvement in energy efficiency. This improvement is even greater at larger areas. The published results for NVDLA do not include any configuration larger than 3.3mm^2, and therefore it is difficult to make a fair evaluation at larger area budgets. Nonetheless, we expect that as NVDLA scales up, memory bandwidth will bottleneck the system, resulting in reduced throughput and energy benefit. FixyNN solves this problem by reducing the load on DRAM.

5.6.2 Model Accuracy

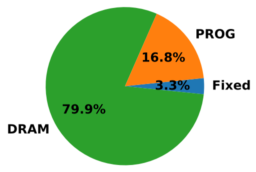
Table 7 summarizes the accuracies for the first set of transfer learning experiments with MobileNet-0.25, where the first row shows the baseline accuracy. As we go down the table, a higher percentage of the network is fixed, hence a bigger FFE is used.



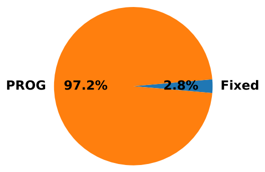
(a) Area



(b) Operations



(c) Energy



(d) Latency

Figure 33. PPA breakdown of FixyNN for MobileNet-0.25 with 7 fixed layers and a 1.00mm^2 NVDLA.

Adaptive Batch Normalization helps a transferred model to achieve better accuracy with a relatively small hardware cost. Images in different datasets come from different visual domains and have therefore very different statistical distributions, adaptive BN helps the model better adapt to the new domain.

Our experiments show that for datasets **CIFAR-100**, **CIFAR-10**, **SVHN** and **Flwr**, we can fix 77% of the network while suffering less than 2% loss in model accuracy. For datasets **Airc** and **GTSR**, similar accuracy performance relative to the baseline requires fixing a smaller percentage of the network in FFE (between 27% and 44%).

Transfer learning models are trained in floating-point datatype without forcing sparsity. Pruning and quantization are orthogonal to transfer learning and will affect model accuracy equally regardless of being transferred or not. Our observation for accuracy loss will hold even after further pruning and quantization of the model.

In Table 8, we report transfer learning accuracies for MobileNet-1.0. Only results on **CIFAR-100** are shown here. Similar trend in transfer learning accuracy loss is observed. Overall accuracies are improved as MobileNet-1.0 has a bigger model capacity. Fixing the first 11 convolution layers of the network with adaptive BN results in 1.6% accuracy drop.

Model			Accuracy on datasets (%)						
Fixed layers	Adaptive BN	Fixed Ops (%)	ImageNet	CIFAR100	CIFAR10	SVHN	Flwr	Airc	GTSR
0	N	0.0	49.8	72.8	93.5	95.8	88.1	67.7	97.7
4	Y	27.1	49.8	72.5	93.3	95.7	88.3	66.7	97.8
7	Y	44.3	49.8	72.0	92.7	95.8	87.5	64.0	95.0
7	N	46.6	49.8	69.4	91.7	94.7	85.2	63.2	93.5
11	Y	77.0	49.8	71.1	91.7	94.6	86.9	56.7	89.2
14	Y	97.0	49.8	68.5	85.3	91.0	82.8	41.9	59.3
14	N	100.0	49.8	54.5	77.0	48.0	77.8	30.5	46.1

Table 7. Transfer learning results for MobileNet-0.25 with fixed feature extractor, the model is trained on ImageNet and transferred to six different vision tasks.

Model			Accuracy (%)	
Fixed layers	Adaptive BN	Fixed Ops(%)	ImageNet	CIFAR100
0	N	0.0	70.9	81.7
4	Y	21.4	70.9	81.2
7	Y	39.9	70.9	80.7
7	N	40.6	70.9	80.2
11	Y	76.4	70.9	80.1
14	Y	99.1	70.9	76.7
14	N	100	70.9	61.6

Table 8. Transfer learning results for MobileNet-1.0 with fixed feature extractor. The model is trained on ImageNet and transferred to CIFAR-100.

5.6.3 Discussion

Having presented the experimental results, we finally draw together some conclusions regarding the design of FixyNN systems. Summarizing Section 5.6.1, we found that the hardware throughput and energy-efficiency gains of FixyNN outpaces the baseline of an iso-area programmable NVDLA accelerator at the same silicon area cost when we fix 7 or more layers of Mobilenet-0.25. The hardware throughput and energy efficiency of FixyNN reach as high as 5.64 TOPS (2.55$\times$ better than the iso-area NVDLA baseline) and 26.62 TOPS/W (4.81$\times$ better than the iso-area NVDLA baseline) respectively, at an area budget of $< 4\text{mm}^2$. On the other hand, Section 5.6.2 demonstrates experimentally that as we fix more layers in the FFE, the task of training a new network incorporating the FFE on a different dataset becomes more challenging, and will generally incur an accuracy loss which depends on the dataset. Therefore, in practice, the system designer must balance the requirements of throughput/energy-efficiency and accuracy across a variety of datasets. While this is obviously a nuanced trade-off, we offer a straightforward analysis to help emphasize the potential benefit of the FixyNN.

We consider an arbitrary constraint that the maximum tolerable degradation in accuracy is no greater than 2% on the suite of six transfered datasets we examined in Section 5.6.2. We also specify a $<3\text{mm}^2$ silicon area budget for accelerating CV workloads. A FixyNN system that fixes 4 layers (27.1% Ops) with adaptive BN, a 0.38mm^2 FFE and NVDLA config. *E*, can meet this specification, with a total area of 2.18mm^2. Over all six datasets we studied, this FixyNN configuration achieves a maximum accuracy degradation of no more than 1.0%, with the most challenging being **Airc**. If we compare this design to a baseline consisting of a larger NVDLA of the same silicon area as the total FixyNN design (2.18mm^2, we achieve an improvement in throughput of 1.15$\times$ and in energy efficiency of 1.42$\times$.

As discussed in Section 5.6.2, two of the six datasets are significantly less tolerant to a large number of fixed layers, which limits the improvement we demonstrate in the previous scenario. Therefore, to prioritize *average* performance across all datasets while otherwise still meeting the same constraints, we modify the FixyNN design so that the datasets with high accuracy degradation only use a portion of a larger FFE. This allows us to define a FixyNN system that fixes 7 layers with adaptive BN (44.3% Ops / 0.79mm^2 FFE) and uses NVDLA config. *E*, for a total area of 2.59mm^2. With this configuration, four of the six datasets utilize the entire FFE as before, resulting in an improvement in throughput of 1.29× (2.14 TOPS) and in energy-efficiency of 1.92× (11.19 TOPS/W) over a baseline design of the same area. The two datasets with high accuracy degradation may opt to use only 4 layers of the FFE, resulting in 0.98× and 1.48× in throughput and energy-efficiency, respectively.

5.7 Conclusion

Real-time computer vision workloads on mobile devices demand extremely high energy-efficiency for CNN computations, which can only be achieved with specialized hardware. This chapter evaluates FixyNN as a solution derived from closer integration of computer systems and machine learning. FixyNN achieves an optimal balance of energy-efficiency from processing part of the network with heavily customized hardware for CNN feature extraction, and generalization to different CV tasks by means of a programmable portion that is trained using transfer learning. Our experimental evaluation demonstrates that FixyNN hardware can achieve very high energy efficiency of up to 26.6 TOPS/W (4.81× better than iso-area programmable accelerator). We considered a suite of six image classification problems, and found we can train models using transfer learning with an accuracy loss of $< 1\%$, and achieving up to 11.2 TOPS/W, which is nearly 2× more efficient than a conventional programmable CNN accelerator of the same area.

Chapter 6

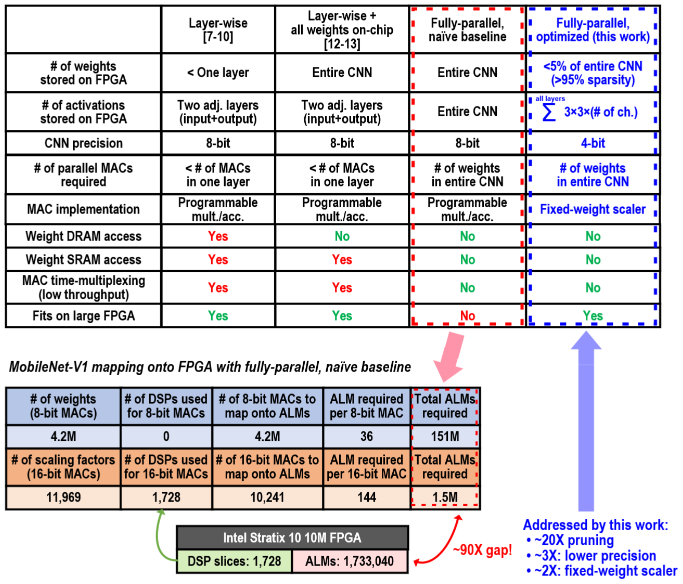
FIXYFPGA: EFFICIENT FPGA ACCELERATOR FOR DEEP NEURAL NETWORKS WITH HIGH ELEMENT-WISE SPARSITY AND WITHOUT EXTERNAL MEMORY ACCESS

Convolutional neural networks (CNNs) have become very popular in real-time computer vision systems. CNNs involve a large amount of computation and storage and typically demand a highly efficient computing platform. Researchers have explored a diverse range of software and hardware optimizations to accelerate CNN inference in recent years. The high power consumption of GPUs and the lack of flexibility with ASIC has promoted interest in FPGAs as a promising platform to efficiently accelerate these CNN inference tasks. Various FPGA-based CNN accelerators have been proposed to low precision weights and high-sparsity in various forms. However, most of the previous work requires off-chip DDR memory to store the parameters and expensive DSP blocks to perform the computation. In this work, we propose the FixyFPGA, a fully on-chip CNN inference accelerator that naturally supports high-sparsity and low-precision computation. In our design, the weights of the trained CNN network are hard-coded into hardware and used as fixed operand for the multiplication. Convolution is performed by streaming the input images to the compute engine in a fully-paralleled, fully-pipelined manner. We analyzed the performance of the proposed scheme with both image classification tasks and object detection tasks based on the low precision, sparse compact CNN models. Compared to prior works, our design achieved 2.34× higher GOPS on ImageNet classification.

6.1 Introduction

Convolutional neural networks (CNNs) have been successful in many practical applications including image classification, object detection and segmentation, and various algorithms and architectures have been proposed in a very fast pace Simonyan and Zisserman (2015); He *et al.* (2016); Howard *et al.* (2017b); Tan and Le (2019). GPUs are the *de facto* hardware platform for DNN training workloads, aided by the highly-parallel computing with a massive number of processing cores. However, due to the high price and the lack of reconfigurability, GPU is usually not an ideal solution for DNN inference acceleration, especially for models with high sparsity or customized architectures. ASICs such as the Google TPU Jouppi *et al.* (2017b) typically have the highest energy-efficiency, but their limited configurability can introduce a significant risk of premature obsolescence, as the model architectures evolve over time. With DNN algorithms evolving at a fast pace, ASIC designs will always lag behind the cutting edge due to the long design cycle. To that end, FPGAs have a unique advantage with potentially higher throughput and efficiency than GPUs, while offering faster time-to-market and potentially longer useful life than ASIC solutions.

Figure 34 (top) shows the categorization of different FPGA-based CNN accelerator schemes. Most of the conventional FPGA-based CNN accelerators Ma *et al.* (2020); Wu *et al.* (2019); Yu *et al.* (2020); Ye *et al.* (2020) in the literature use off-chip DRAM to store the weights, and the FPGA accelerator performs computation for a single-layer (or a subset of a single-layer) in a time-multiplexed manner. However, the throughput of such designs is often limited by the DRAM bandwidth and the number of multipliers constructed by DSPs. Furthermore, frequently accessing the off-chip memory also introduces high energy consumption Horowitz (2014).



	Layer-wise [7-10]	Layer-wise + all weights on-chip [12-13]	Fully-parallel, naïve baseline	Fully-parallel, optimized (this work)
# of weights stored on FPGA	< One layer	Entire CNN	Entire CNN	<5% of entire CNN (>95% sparsity)
# of activations stored on FPGA	Two adj. layers (input+output)	Two adj. layers (input+output)	Entire CNN	$\sum^{\text{all layers}}$ 3×3×(# of ch.)
CNN precision	8-bit	8-bit	8-bit	4-bit
# of parallel MACs required	< # of MACs in one layer	< # of MACs in one layer	# of weights in entire CNN	# of weights in entire CNN
MAC implementation	Programmable mult./acc.	Programmable mult./acc.	Programmable mult./acc.	Fixed-weight scaler
Weight DRAM access	Yes	No	No	No
Weight SRAM access	Yes	Yes	No	No
MAC time-multiplexing (low throughput)	Yes	Yes	No	No
Fits on large FPGA	Yes	Yes	No	Yes

MobileNet-V1 mapping onto FPGA with fully-parallel, naïve baseline

# of weights (8-bit MACs)	# of DSPs used for 8-bit MACs	# of 8-bit MACs to map onto ALMs	ALM required per 8-bit MAC	Total ALMs required
4.2M	0	4.2M	36	151M
# of scaling factors (16-bit MACs)	# of DSPs used for 16-bit MACs	# of 16-bit MACs to map onto ALMs	ALM required per 16-bit MAC	Total ALMs required
11,969	1,728	10,241	144	1.5M

Intel Stratix 10 10M FPGA	
DSP slices: 1,728	ALMs: 1,733,040

Figure 34. (Top) Categorization of DNN accelerators on FPGAs. (Bottom) Mapping the entire MobileNet-V1 CNN onto FPGA requires a number of techniques employed collectively in this work.

To eliminate DRAM access for DNN inference using a single FPGA, the entire DNN model including weights and activations must be mapped onto the on-chip memory on the FPGA. One of the most well-known compact DNN models for the ImageNet dataset is MobileNet Howard *et al.* (2017b), which achieves a similar accuracy compared to the conventional VGG-16 Simonyan and Zisserman (2015) (138M weights) or ResNet-18 He *et al.* (2016) (11M weights) architectures with significantly less parameters (4.2M weights) and MAC operations. A few prior FPGA designs have fully mapped the compact MobileNet-V1 CNN to a single FPGA without DRAM access Zhao *et al.* (2019); Hall and Betz (2020). This is possible because recent large-scale FPGAs such as Intel Stratix-10 GX2800 or 10M Intel (2021) integrates up to >200Mb of on-chip memory (M20K), which can comfortably hold all MobileNet-V1 weights (4.2M) either in 8-bit or 16-bit precision. Both works Zhao *et al.* (2019); Hall and Betz (2020) store MobileNet-V1 weights in on-chip M20K memory, and load the weights into time-multiplexed multiply-and-accumulate (MAC) units to perform layer-by-layer inference in a pipelined manner.

To fully map MobileNet-V1 onto existing FPGAs and maximize throughput, one MAC unit per each weight (i.e. 4.2M MAC units) will be required. Typically DSP blocks are employed for parallel MAC computation, but only thousands of DSP slices exist in large FPGAs (e.g. 1,728 in Intel Stratix-10 10M), and all of these could be used up for the high-precision channel-wise scaling factor computation (Section **??**), which is necessary in 8-bit or lower precision CNNs for better gradient estimation and lower quantization error. This means that all 4.2M MAC units need to be implemented with ALMs. Since the mapping of one 8-bit MAC needs 36 ALMs, a total of 151M ALMs are needed for the fully-parallel baseline, which represents a $\sim 90\times$ gap with the FPGA that has the largest number of ALMs (1.73M), as shown in Fig. 34 (bottom).

To bridge this gap, lower precision quantization or pruning can be performed, but previous work Zhao *et al.* (2019); Hall and Betz (2020) did not consider pruning and only lowered the activation/weight precision down to 8-bit. For compact models such as MobileNet, it has been difficult to quantize the activation/weight precision below 8-bit without considerable accuracy loss. A recent algorithm work Park and Yoo (2020) presented new quantization techniques that lower the precision of MobileNets to 4-bit with minimal accuracy degradation, but did not integrate pruning.

With respect to pruning, element-wise pruning achieves higher sparsity, but the irregular memory access and the index storage overheads, especially for low-precision DNNs, have hindered efficient hardware implementation Han *et al.* (2016c); Lee *et al.* (2021). Structured pruning schemes Wen *et al.* (2016); Srivastava *et al.* (2019); Yang *et al.* (2020) generate sparsity in a hardware-friendly manner, by removing a group of parameters in row-/column-wise, block-wise, filter-wise, or channel-wise manner.

This leads to efficient hardware acceleration, but the amount of sparsity in structured pruning schemes is typically much lower than element-wise pruning schemes Mao *et al.* (2017).

On the other hand, FixyNN Whatmough *et al.* (2019) proposed a fixed-weight feature extractor (FFE) design, where the weights are hard-coded in the datapath logic and do not need to be stored in memory. LogicNets Umuroglu *et al.* (2020) also proposed a similar technique to implement neural networks with look-up tables in FPGAs, but the hardware design is only benchmarked for small neural networks with unconventional datasets for jet substructure classification and network intrusion detection.

While FixyNN Whatmough *et al.* (2019) only employed an FFE for the early layers of CNNs for an ASIC design, in this work, we employ such fixed-weight scalers for the entire CNN layers for an FPGA design. By mapping hard-coded weights in the ALMs of the FPGA, we perform CNN inference of all layers in a fully-parallel, fully-pipelined manner. Contrary to the notion that element-wise sparsity is inefficient for hardware design, one important advantage of the fixed-weight FPGA design (FixyFPGA) is that, element-wise pruning of DNNs can be seamlessly integrated with FixyFPGA design with very high efficiency. This is because pruning out weight elements is equivalent to removing the corresponding hardware operands without introducing any index overhead. This enables us to exploit the high amount of sparsity achievable by the element-wise pruning algorithms Lee *et al.* (2021).

Overall, the main contributions of this work are:

- We present FixyFPGA, a fully-parallel, fully-pipelined, and pruning-friendly FPGA-based CNN accelerator design based on fixed hard-coded weights.
- We investigate implementing a number of DNN models with different widths and

compression ratios with the fixed-weight scheme onto a single Intel Stratix-10 FPGA chip without any DRAM access.

- We analyze the algorithm and hardware results of DNNs for both image classification tasks (ImageNet, TinyImageNet, and CIFAR-10 datasets).

6.2 Fixed Weight Accelerator Design

FixyFPGA implements CNN models in a layer-parallel fashion, where every non-zero parameter is encoded in the hardware design as a fixed-weight multiplier (scaler). This layer-parallel approach leads to very high gains in latency and energy, by 1) removing the energy and BW limitations of DRAM, and 2) increasing the number of MACs that can be implemented on an FPGA by $\sim 1.7\times$ using fixed-weight scalers.

6.2.1 Fixed-Weight CNN Datapath

Fixed-weight scalers are significantly smaller, faster and lower energy than full multipliers. Fixed scalers are implemented with a series of hardwired shifts, which are essentially free in hardware, and an adder. The hardware cost is essentially a function of the input operand (activation) bitwidth, and the Hamming weight (i.e. the number of non-zero bits) of the multiplier (weight), which determines the number of partial products. The adder tree needed to process the flattened output feature map is highly pipelined to achieve high clock frequency. The fixed-weight datapaths are implemented in RTL by embedding the weights into the Verilog as literals. The synthesis tool then generates highly optimized deep sum-of-product datapaths using techniques such as Booth encoding and carry-save addition Zimmermann (2009). Zero weights are simply ignored and do not generate any hardware.

Based on our actual implementation of 4-bit MobileNet-V1 on Intel Stratix 10 FPGA (Section 6.4), the FixyFPGA scheme consumes 5.87 ALMs per 4-bit scaler on average (i.e. total ALM usage divided by the number of non-zero weights). In comparison, by mapping a 4-bit MAC unit with real multipliers and accumulaters onto the same FPGA, we found that one single non-fixed-weight 4-bit MAC consumes 10.0 ALMs. Therefore, this shows that the fixed design can at least achieve 1.7× reduction in ALMs for each MAC implementation.

6.2.2 Fully-Pipelined Activation Buffering

Implementing direct convolution in a layer-parallel configuration requires buffering of activation data flowing through the datapaths. A typical 3×3×C convolution, where C is the number of channels, consumes a 3×3×C input pixel tensor per cycle, but generates only a single small 1×1×C output tensor. Hence, we must buffer several 1×1×C outputs into a larger 3×3×C input for the next layer. Figure 28 shows how this is implemented using a *line buffer* to store activations at each layer row by row until the required tensor size has been buffered up. Due to the mismatch in input and output tensor dimensions, we need to buffer three full rows before we can start generating the larger output tensors for the following layer. The line buffer itself is implemented on FPGA using on-chip M20K memory and ALMs, and therefore actually requires four independent SRAM banks, so that we can write a single-row patch to one bank per cycle, and read the three-row patch from three banks per cycle, concurrently. After reading/writing the last pixel in a row, the four banks are rotated to overwrite the data associated with the oldest row. Following the SRAM line buffer, a shift-register shifts the convolution window over the feature map, without re-reading data. The shift-register consumes 1×3×C pixels per cycle from the SRAM line buffer and outputs a 3×3×C volume.

6.2.3 Deep Freeze Tool Flow

We implemented a tool called *Deep Freeze* to automatically generate a fixed-weight CNN accelerator in Verilog RTL directly from a simple model description in a high-level API. This tool is available as an open source project, which is not linked here for the blind review process. A direct code generation step reads the integer model weights and emits Verilog HDL logic with the weights embedded as immediate constants. Zero weights are skipped entirely. The precision for the intermediate activations is specified as a hardware parameter, along with the accumulator width. The final Verilog is constructed by connecting consecutive combinational datapath stages with buffer stages, which are instantiated from a parameterized Verilog template. The generated Verilog can be directly read in by any synthesis tool for ASIC or FPGA implementation. The generated code is optimized for size and helps reduce compile time, which can be long for such a dense datapath dominated design. During the datapath generation, the bit-widths of the fixed scalers are optimized individually. Deep Freeze also generates a validation suite with testbench for simulation.

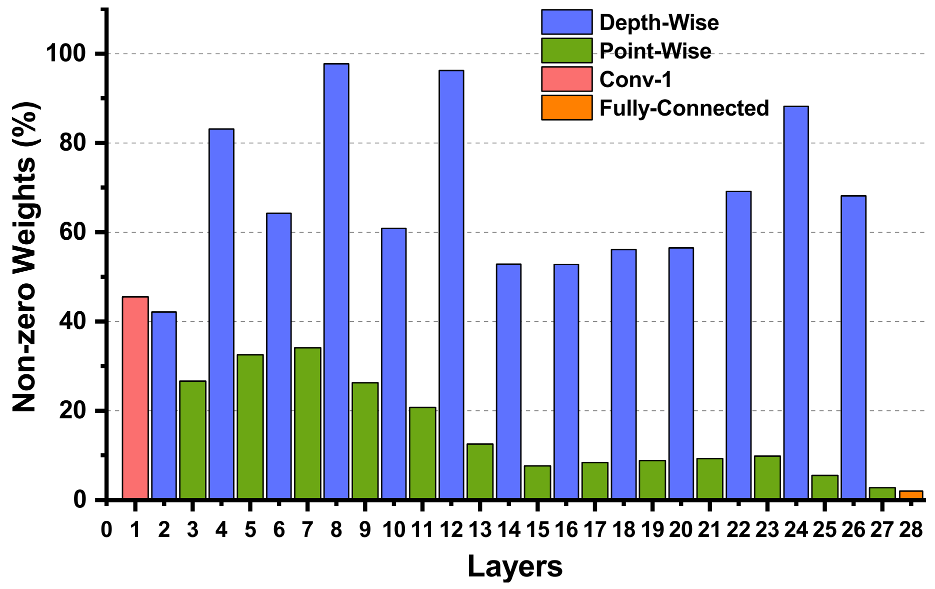


Figure 35. Percentage of non-zero weights in each layer of 4-bit MobileNet-V1 0.75 after element-wise pruning (total number of non-zero weights is 161K).

6.3 Experiment Results

6.4 Experiment Results

Table 9. Evaluation of CNN accelerators on Stratix 10 10M FPGA with various CNN models and datasets.

Models	# of Params	Top-5 Acc. (%)	Input Size	DSP	ALM	M20K	Freq. (MHz)	FPS	TOPS	Power (W)
MobileNet-V1 1.0 Width [1]	165K	73.32	224×224 [2]	1.73K (100%)	1335.9K (77%)	1.39K (21%)	132.85	2.65K	3.01	30.36
		71.59	64×64 [2]	1.73K (100%)	1015.4K (59%)	1.32K (20%)	163.11	39.8K	3.74	27.30
MobileNet-V1 0.75 Width [1]	161K	72.87	224×224 [2]	1.73K (100%)	1099.1K (63%)	1.11K (17%)	172.92	3.45K	2.27	27.43
		68.41	64×64 [2]	1.73K (100%)	1024.6K (59%)	1.11K (17%)	177.43	43.3K	2.32	26.62
MobileNet-V1 0.5 Width [1]	161K	71.47	224×224 [2]	1.73K (100%)	824.43K (48%)	0.75K (12%)	163.91	3.27K	1.24	26.90
		68.26	64×64 [2]	1.73K (100%)	802.51K (46%)	0.75K (12%)	169.41	41.4K	1.02	26.10
VGG7-C	198K	99.58 (CIFAR-10)	32×32	0.36K (21%)	814.98K (47%)	0.29K (4%)	137.29	134.07K	90.95	22.03

[1]Widths of 1.0/0.75/0.5 represent that the number of channels in MobileNet-V1 models are scaled accordingly.
[2]Input image size of 224×224 is for ImageNet dataset, and 64×64 is for TinyImageNet dataset.

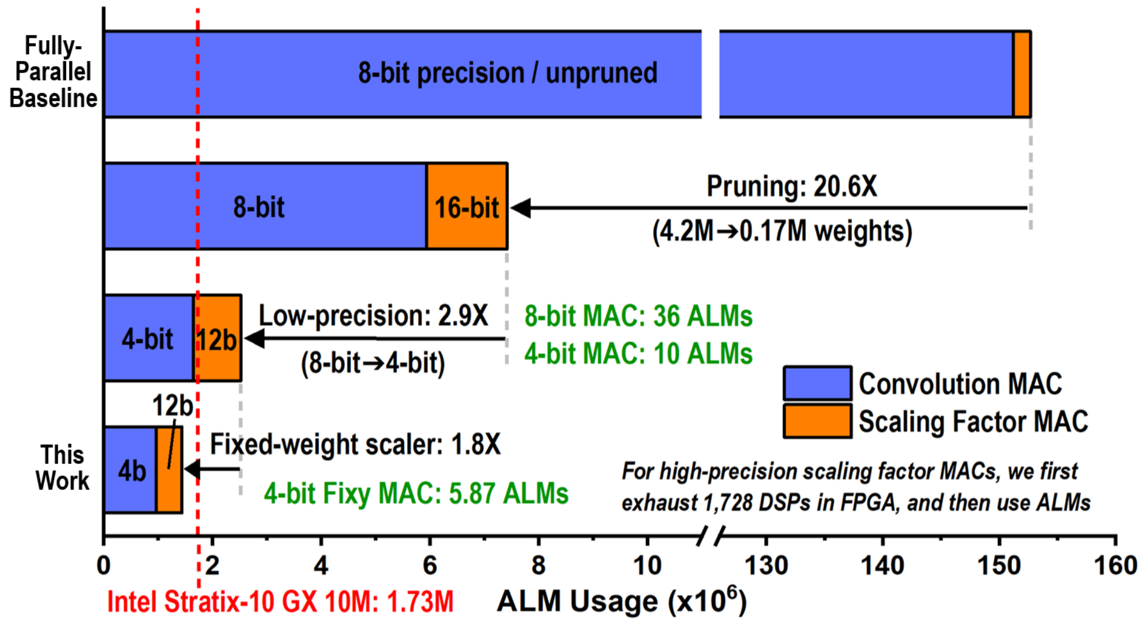


Figure 36. For fully-parallel implementation of MobileNet-V1 on FPGA, ALM usage is reduced by 107× collectively by pruning, low-precision quantization, and fixed-weight scalers.

6.4.1 Experiment Setup

All algorithm experiments are completed with Pytorch API, and the FPGA accelerator generated by the compiler was synthesized using Intel Quartus 20.3. We used Stratix 10 GX 10M FPGA as the target FPGA device, which includes 132 Mb of M20K memory, 1,728 DSP blocks, and 1.73M ALMs. Given the pre-trained 4-bit sparse PyTorch-trained model, we first extract the fixed-point parameters and then generate the corresponding RTL files with the Deep Freeze tool.

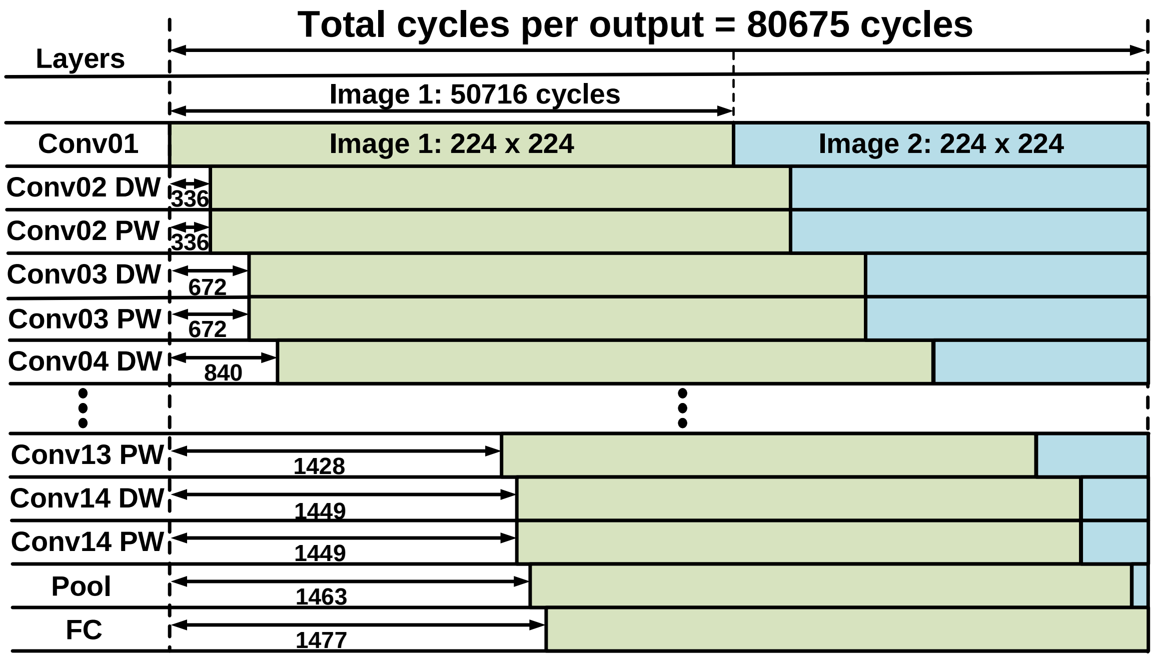


Figure 37. Layer-wise timing analysis of MobileNet-V1 generated by RTL simulation with 224×224×3 input image for ImageNet.

Table 10. Comparison to different FPGA accelerators for MobileNets for ImageNet.

Implementations	Model	W / A	Platform	DRAM	Frequency (MHz)	Latency (ms)	GOPS	Frame rate (fps)	Sparsity
DPU Wu *et al.* (2019)	MobileNet-V2	8 / 8	Xilinx Zynq US+	Yes	333	1.23	922	430	Dense
HPIPE Hall and Betz (2020)	MobileNet-V1	16 / 16	Intel Stratix 10	No	430	0.65	-	5157	Dense
TuRF Zhao *et al.* (2018b)	MobileNet-V1	8 / 8	Intel Stratix V	No	150	4.33	264	231	Dense
TuRF Zhao *et al.* (2018a)	MobileNet-V1	16 / 16	Intel Stratix V	No	200	0.88	1287	1131	Dense
Tomato Zhao *et al.* (2019)	MobileNet-V1	Mixed / 8	Intel Stratix 10	No	156	0.32	3536	3109	Dense
This work	MobileNet-V1	4 / 4	Intel Stratix 10	No	133	0.37	3013	2648	96%

6.4.2 FPGA Implementation Results and Analysis

6.4.2.1 Image classification

Figure 36 shows the how the ∼90× gap pointed out in Section 6.1 is addressed in this work, by a series of techniques including pruning with high sparsity (∼20×), 4-bit quantization, and the usage of fixed-point scalers for the entire MobileNet-V1 model. After applying all techniques, the total ALM usage of the proposed FixyFPGA design for MobileNet-V1 model falls under the 1.73M available ALMs in the target FPGA device. The elimination of the DRAM communication also leads to large savings in energy and latency.

Figure 37 shows the timing diagram for the overall MobileNet-V1 inference, based on RTL simulation. The fully-pipelined activation buffering maximized the computation efficiency by sending the basic $3 \times 3 \times C$ volume to the next layer rather than waiting for the previous computation to complete.

Table 9 summarizes the resource utilization, throughput, operating frequency, and power consumption with the various CNN models that are trained for different datasets. Every layer of all implemented CNNs are quantized down to 4-bit precision and fully hard-coded into the data logic on FPGA, without any DRAM access. Using the Intel Early Power Estimator, we obtained the power consumption at the junction temperature of 75°C. With the fully-pipelined and fully-parallel FixyFPGA scheme, the latency per image was computed as: $T = \frac{X_H \times X_W}{f}$, where X_H and X_W represents the height and width of the input image. Given the different input sizes of various datasets, the proposed FixyFPGA achieves 3.01, 3.74, and 90.95 GOPS for ImageNet, TinyImageNet, and CIFAR-10 classifications, respectively.

6.4.2.2 Comparison to Prior Works

We compared the hardware performance with previous fully on-chip FPGA-based CNN accelerators with regards to operating frequency, latency, throughput, etc. Unlike the prior works that used on-chip memory to store the weights, our design fully embedded the CNN parameters onto the logic units, which enables us to apply the element-wise pruning without any sparsity index. Therefore, compared with the previous memory-based 8-bitZhao *et al.* (2019); Zhao *et al.* (2018b) or 16-bit Zhao *et al.* (2018a) implementations, our design with 4-bit precision and high sparsity will have a large potential for energy-efficiency improvements. Table 10 shows the comparison results between our design and and other recent works. TuRF Zhao *et al.* (2018a) performed the computation in a layer-by-layer fashion, where the next layer has to wait until the current layer's computation completes. In contrast, with the fully-pipelined and fully-parallel design, our FixyFPGA achieved 3.01 TOPS, which is 2.34× higher than TuRF Zhao *et al.* (2018a) along with 2.37× latency improvements. Similar to TuRF Zhao *et al.* (2018a), Tomato Zhao *et al.* (2019) stores the power-of-two (POT) weights inside the on-chip memory, streams into the compute engines in a pipelined manner then keeps rolling the output channel to perform the BN multiplications with the given factor. To support such computation, the MAC units should be time-multiplexed. Also, restricting the weights to POT values can lead to considerable accuracy loss in general, due to the rigid resolution of POT quantization Li *et al.* (2020). Our proposed design achieved similar hardware performance in a fully-parallel manner, which could be more beneficial to the practical scenarios with high throughput and low-power demands.

In addition to the highly-efficient hardware design, our deployed model is also highly sparse, and such high sparsity will improve the power efficiency even further. On the other hand, it is true that such aggressive compression scheme will improve the hardware efficiency with the cost of accuracy degradation. We will address such tradeoff to alleviate the accuracy degradation in the future work.

6.5 Conclusion

In this chapter, we presented FixyFPGA, a fully-parallel and fully pipelined FPGA-based CNN accelerator design with the objective of compact and high-throughput hardware acceleration. For MobileNet and VGG models for ImageNet, TinyImageNet, and CIFAR-10we performed low-precision quantization down to 4-bit, together with high sparsity of >95%, towards mapping the entire CNN models onto the target FPGA device and eliminating DRAM access. We achieved 3.01 TOPS for ImageNet classification with a low end-to-end latency of 0.37ms. Compared to prior works, our design achieved 2.34× higher GOPS on ImageNet classification.

Chapter 7

CONCLUSION

In this dissertation, we presented an automatic RTL compiler-based end-to-end CNN training accelerator. Optimized and parameterized custom Verilog modules implement CNN training operations, and the accelerator is flexible to support various FPGA design parameters. The training performance is evaluated on Intel Stratix-10 GX FPGA for three different CNNs for the CIFAR-10 dataset. The proposed training accelerator achieves a throughput of up to 479 GOPS at 240MHz for CNNs with 2M parameters.

Furthermore, we presented a flexible CNN training accelerator on FPGA using HBM2, which performs end-to-end training of modern CNNs involving residual connections and stride-2 convolutions. The FPGA accelerator is implemented on Intel S-10 MX (with HBM2) and S-10 GX (with DDR3) devices, demonstrating system-level benefits of HBM2 over conventional DDR3 off-chip memory. The proposed accelerator achieves 4.5-9.7X energy-efficiency improvement compared to Tesla V100 GPU and 7-11X improvement in throughput compared to that of Intel i7-9800X CPU for low-batch training tasks of ResNet-20/VGG-like CNNs. This dissertation further presents an online learning FPGA accelerator and demonstrated online learning on FPGA with a CNN structure of 16C3-16C3-MP-32C3-32C3-MP-64C3-64C3-MP-FC, where 'NCk' represents convolution layer with 'N' output feature maps and a kernel size of 'k', 'MP' represents max-pooling layer and 'FC' represents a fully connected layer. The accelerator was synthesized at 150 MHz frequency. Adapting model segmentation techniques from Progressive Segmented Training(PST), the online learning accelerator achieved a 4.2X reduction in training latency. The CIFAR-10 dataset consists of 60,000 32×32 color images in 10 classes, with 5,000 training images and 1,000 testing images per class. The classes include common objects such as planes, birds, trucks, etc. We demonstrate online learning on FPGA with a CNN structure of 16C3-16C3-MP-32C3-32C3-MP-64C3-64C3-MP-FC, where 'NCk' represents convolution layer with 'N' output feature maps and a kernel size of 'k', 'MP' represents max-pooling layer and 'FC' represents a fully connected layer. The accelerator was synthesized by Intel Quartus 19.2 at 150 MHz frequency. We used Stratix-10 MX equipped with HBM2 as the target hardware and Intel(R) Core(TM) i7-9800X as a host machine. All the parameters used 16-bit fixed-point precision.

Next, we presented a prototype chip was fabricated in 28nm CMOS. Including the skipped operations, peak throughput of 3.76 TFLOPS was achieved at 1.1V, and peak energy-efficiency of 16.4 TFLOPS/W was achieved at 0.6V. We evaluated a number of DNNs for both supervised training and self-supervised training tasks, and achieved high FLOPs reduction (up to 7.3X) and training speedup (up to 4.7X) compared to the dense models. The training speedup is >3.5X better than the speedup reported in the state-of-the-art sparse training processor.

Finally, in addition to the CNN training accelerators, an ASIC (FixyNN) and FPGA (FixyFPGA) CNN inference accelerator adopting fixed-feature extractors is presented. FixyNN achieved an optimal balance of energy efficiency from the processing part of the network with heavily customized hardware for CNN feature extraction and generalization to different CV tasks using a programmable portion trained using transfer learning. Our experimental evaluation demonstrated that FixyNN hardware could achieve a very high energy efficiency of up to 26.6 TOPS/W ($4.81\times$ better than iso-area programmable accelerator). We considered a suite of six image classification problems. We found we can train models using transfer learning with an accuracy loss of $< 1\%$, and achieving up to 11.2 TOPS/W, which is nearly $2\times$ more efficient than a conventional programmable CNN accelerator of the same area.

Next, the FixyFPGA, a fully parallel and fully pipelined FPGA-based CNN accelerator design with the objective of compact and high-throughput hardware acceleration, is presented. MobileNet and VGG models for ImageNet, TinyImageNet, and CIFAR-10 performed low-precision quantization down to 4-bit, together with high sparsity of $>95\%$, towards mapping the entire CNN models onto the target FPGA device and eliminating DRAM access. We achieved 3.01 TOPS for ImageNet classification with a low end-to-end latency of 0.37ms. Compared to prior works, our design achieved $2.34\times$ higher GOPS on ImageNet classification.

In summary, this dissertation comprehensively discusses novel architectures of high-performance and energy-efficient ASIC/FPGA CNN inference/training accelerators.

REFERENCES

Abdelouahab, K., M. Pelcat, J. Serot and F. Berry, “Accelerating cnn inference on fpgas: A survey”, arXiv preprint arXiv:1806.01683 (2018).

Ahmad, A. and M. A. Pasha, “Optimizing hardware accelerated general matrix-matrix multiplication for cnns on fpgas”, IEEE Transactions on Circuits and Systems II: Express Briefs (2020).

Albericio, J., P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger and A. Moshovos, “Cnvlutin: Ineffectual-neuron-free Deep Neural Network Computing”, in “Proc. of ISCA”, (2016).

Arm, “Arm Machine Learning Processor”, URL https://developer.arm.com/products/processors/machine-learning/arm-ml-processor (2019).

Barry, B., C. Brick, F. Connor, D. Donohoe, D. Moloney, R. Richmond, M. O’Riordan and V. Toma, “Always-on Vision Processing Unit for Mobile Applications”, IEEE Micro (2015).

Booth, A., “A Signed Binary Multiplication Technique”, Quarterly Journal of Mechanics and Applied Mathematics **4**, 2, 236–240 (1951).

Buckler, M., P. Bedoukian, S. Jayasuriya and A. Sampson, “Eva2: Exploiting temporal redundancy in live computer vision”, in “Proceedings of the 45th Annual International Symposium on Computer Architecture”, ISCA ’18, pp. 533–546 (IEEE Press, Piscataway, NJ, USA, 2018), URL https://doi.org/10.1109/ISCA.2018.00051.

Chaudhry, A., M. Ranzato, M. Rohrbach and M. Elhoseiny, “Efficient lifelong learning with a-gem”, arXiv preprint arXiv:1812.00420 (2018).

Chen, H. G., S. Jayasuriya, J. Yang, J. Stephen, S. Sivaramakrishnan, A. Veeraraghavan and A. C. Molnar, “ASP vision: Optically computing the first layer of convolutional neural networks using angle sensitive pixels”, CoRR **abs/1605.03621**, URL http://arxiv.org/abs/1605.03621 (2016a).

Chen, Y.-H., J. Emer and V. Sze, “Eyeriss: A Spatial Architecture for Energy-efficient Dataflow for Convolutional Neural Networks”, in “Proc. of ISCA”, (2016b).

Chi, P., S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang and Y. Xie, “PRIME: A Novel Processing-in-Memory Architecture for Neural Network Computation in ReRAM-Based Main Memory”, in “Proc. of ISCA”, (2016).

Choi, S., J. Sim, M. Kang and L.-S. Kim, "TrainWare: A memory optimized weight update architecture for on-device convolutional neural network training", in "Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)", (2018).

Chundi, P. K., P. Liu, S. Park, S. Lee and M. Seok, "FPGA-based Acceleration of Binary Neural Network Training with Minimized Off-Chip Memory Access", in "IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)", pp. 1–6 (2019).

Cooper, K. D., L. T. Simpson and C. A. Vick, "Operator strength reduction", ACM Trans. Program. Lang. Syst. **23**, 5, 603–625, URL http://doi.acm.org/10.1145/504709.504710 (2001).

Dalal, N. and B. Triggs, "Histograms of Oriented Gradients for Human Detection", in "Proc. of CVPR", (2005).

DeepFreeze, "RTL generation tool for CNNs", URL https://github.com/ARM-software/DeepFreeze (2018).

Deo, M., J. Schulz and L. Brown, "Intel stratix 10 mx devices solve the memory bandwidth challenge", Intel White Paper (2016).

Deo, M., J. Schulz and L. Brown, "Intel Stratix 10 MX Devices Solve the Memory Bandwidth Challenge", Intel Whitepaper (2017).

Ding, C., S. Liao, Y. Wang, Z. Li, N. Liu, Y. Zhuo, C. Wang, X. Qian, Y. Bai, G. Yuan *et al.*, "CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-Circulant Weight Matrices", in "Proc. of MICRO", (2017).

Du, X., G. Charan, F. Liu and Y. Cao, "Single-net continual learning with progressive segmented training", in "2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)", pp. 1629–1636 (2019a).

Du, X., Z. Li, Y. Ma and Y. Cao, "Efficient network construction through structural plasticity", IEEE Journal on Emerging and Selected Topics in Circuits and Systems **9**, 3, 453–464 (2019b).

Geng, T., T. Wang, A. Li, X. Jin and M. Herbordt, "A Scalable Framework for Acceleration of CNN Training on Deeply-Pipelined FPGA Clusters with Weight and Workload Balancing", arXiv preprint arXiv:1901.01007 (2019).

Gomperts, A., A. Ukil and F. Zurfluh, "Development and implementation of parameterized FPGA-based general purpose neural networks for online applications", IEEE Transactions on Industrial Informatics **7**, 1, 78–89 (2011).

Gopalakrishnan, K., S. K. Khaitan, A. Choudhary and A. Agrawal, “Deep convolutional neural networks with transfer learning for computer vision-based data-driven pavement distress detection”, Construction and Building Materials **157**, 322–330 (2017).

Guo, K., S. Liang, J. Yu, X. Ning, W. Li, Y. Wang and H. Yang, “Compressed cnn training with fpga-based accelerator”, in “Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays”, pp. 189–189 (2019).

Guo, K., S. Zeng, J. Yu, Y. Wang and H. Yang, “A survey of fpga-based neural network accelerator”, arXiv preprint arXiv:1712.08934 (2017).

Gupta, S., A. Agrawal, K. Gopalakrishnan and P. Narayanan, “Deep learning with limited numerical precision”, in “Proceedings of the International Conference on Machine Learning (ICML)”, pp. 1737–1746 (2015).

Hall, M. and V. Betz, “HPIPE: Heterogeneous Layer-Pipelined and Sparse-Aware CNN Inference for FPGAs”, in “ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)”, (2020).

Han, S., X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz and W. Dally, “EIE: Efficient Inference Engine on Compressed Deep Neural Network”, in “Proc. of ISCA”, (2016a).

Han, S., X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz and W. J. Dally, “Eie: efficient inference engine on compressed deep neural network”, ACM SIGARCH Computer Architecture News **44**, 3, 243–254 (2016b).

Han, S., 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).

Han, S., H. Mao and W. J. Dally, “Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding”, in “International Conference on Learning Representations (ICLR)”, (2016c).

He, K., X. Zhang, S. Ren and J. Sun, “Deep residual learning for image recognition”, in “IEEE Conference on Computer Vision and Pattern Recognition (CVPR)”, pp. 770–778 (2016).

Hegarty, J., J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz and P. Hanrahan, “Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines”, in “Proc. of SIGGRAPH”, (2014).

Hegarty, J., R. Daly, Z. DeVito, J. Ragan-Kelley, M. Horowitz and P. Hanrahan, “Rigel: Flexible Multi-Rate Image Processing Hardware”, in “Proc. of SIGGRAPH”, (2016).

Hernández-Lobato, J. M., M. A. Gelbart, B. Reagen, R. Adolf, D. Hernández-Lobato, P. N. Whatmough, D. Brooks, G.-Y. Wei and R. P. Adams, "Designing neural network hardware accelerators with decoupled objective evaluations", in "NIPS workshop on Bayesian Optimization", (2016).

Hoffer, E., I. Hubara and D. Soudry, "Fix your classifier: the marginal value of training the last weight layer", CoRR **abs/1801.04540**, URL http://arxiv.org/abs/1801.04540 (2018).

Horowitz, M., "Computing's energy problem (and what we can do about it)", in "IEEE International Solid-State Circuits Conference (ISSCC)", pp. 10–14 (2014).

Horstmannshoff, J., T. Grotker and H. Meyr, "Mapping multirate dataflow to complex rt level hardware models", in "Proceedings IEEE International Conference on Application-Specific Systems, Architectures and Processors", pp. 283–292 (1997).

Howard, A. G., M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications", CoRR **abs/1704.04861**, URL http://arxiv.org/abs/1704.04861 (2017a).

Howard, A. G., M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto and H. Adam, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications", arXiv preprint arXiv:1704.04861 (2017b).

Hu, J., L. Shen and G. Sun, "Squeeze-and-excitation networks", in "Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)", pp. 7132–7141 (2018).

Intel, "Intel Stratix 10 GX/SX Product Table", URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/pt/stratix-10-product-table.pdf (2021).

Ioffe, S., "Batch renormalization: Towards reducing minibatch dependence in batch-normalized models", in "Advances in neural information processing systems", pp. 1945–1953 (2017).

Ioffe, S. and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift", CoRR **abs/1502.03167**, URL http://arxiv.org/abs/1502.03167 (2015).

Jouppi, N. P., C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, R. C. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz,

A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit", in "Proc. of ISCA", (2017a).

Jouppi, N. P., C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit", in "ACM/IEEE International Symposium on Computer Architecture (ISCA)", p. 1–12 (2017b).

Jouppi, N. P., C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit", in "Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)", pp. 1–12 (2017c).

Judd, P., J. Albericio, T. Hetherington, T. M. Aamodt and A. Moshovos, "Stripes: Bit-serial Deep Neural Network Computing", in "Proc. of MICRO", (2016).

Kim, D., J. Kung, S. Chai, S. Yalamanchili and S. Mukhopadhyay, "Neurocube: A Programmable Digital Neuromorphic Architecture with High-Density 3D Memory", in "Proc. of ISCA", (2016).

Kim, D., T. Na, S. Yalamanchili and S. Mukhopadhyay, "Deeptrain: A programmable embedded platform for training deep neural networks", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **37**, 11, 2360–2370 (2018).

Kirkpatrick, J., R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska *et al.*, "Overcoming catastrophic forgetting in neural networks", Proceedings of the national academy of sciences **114**, 13, 3521–3526 (2017).

Ko, J. H., B. Mudassar, T. Na and S. Mukhopadhyay, "Design of an energy-efficient accelerator for training of convolutional neural networks using frequency-domain computation", in "Proceedings of the ACM/EDAC/IEEE Design Automation Conference (DAC)", pp. 1–6 (2017).

Kodali, S., P. Hansen, N. Mulholland, P. Whatmough, D. Brooks and G. Wei, "Applications of deep neural networks for ultra low power iot", in "2017 IEEE International Conference on Computer Design (ICCD)", pp. 589–592 (2017).

Köster, U., T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof, A. Khosrowshahi, C. Kloss, R. J. Pai and N. Rao, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks", in "Advances in Neural Information Processing Systems", pp. 1742–1752 (2017).

Krizhevsky, A., "Learning multiple layers of features from tiny images", Master's thesis, University of Tront (2009).

Krizhevsky, A., G. Hinton *et al.*, "Learning multiple layers of features from tiny images", (2009).

Krizhevsky, A., I. Sutskever and G. E. Hinton, "Imagenet classification with deep convolutional neural networks", Commun. ACM **60**, 6, 84–90, URL https://doi.org/10.1145/3065386 (2017).

Lee, E. A. and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing", IEEE Transactions on Computers **C-36**, 1, 24–35 (1987).

Lee, J., S. Park, S. Mo, S. Ahn and J. Shin, "A Deeper Look at the Layerwise Sparsity of Magnitude-based Pruning", in "International Conference on Learning Representations (ICLR)", (2021).

Li, Y., X. Dong and W. Wang, "Additive Powers-of-Two Quantization: A Non-uniform Discretization for Neural Networks", in "International Conference on Learning Representations (ICLR)", (2020).

Li, Z. and D. Hoiem, "Learning without forgetting", IEEE transactions on pattern analysis and machine intelligence **40**, 12, 2935–2947 (2017).

Li, Z., C. Liu, Y. Wang, B. Yan, C. Yang, J. Yang and H. Li, "An overview on memristor crossabr based neuromorphic circuit and architecture", in "2015 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)", pp. 52–56 (IEEE, 2015).

LiKamWa, R., Y. Hou, J. Gao, M. Polansky and L. Zhong, “RedEye: Analog ConvNet Image Sensor Architecture for Continuous Mobile Vision”, in “Proc. of ISCA”, (2016).

Liu, C., B. Yan, C. Yang, L. Song, Z. Li, B. Liu, Y. Chen, H. Li, Q. Wu and H. Jiang, “A spiking neuromorphic design with resistive crossbar”, in “2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)”, pp. 1–6 (IEEE, 2015).

Liu, Q., J. Liu, R. Sang, J. Li, T. Zhang and Q. Zhang, “Fast neural network training on FPGA using quasi-newton optimization method”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems **26**, 8, 1575–1579 (2018).

Liu, Z., Y. Dou, J. Jiang, Q. Wang and P. Chow, “An FPGA-based processor for training convolutional neural networks”, in “Proceedings of the International Conference on Field Programmable Technology (ICFPT)”, pp. 207–210 (2017).

Long, J., E. Shelhamer and T. Darrell, “Fully convolutional networks for semantic segmentation”, in “IEEE Conference on Computer Vision and Pattern Recognition (CVPR)”, pp. 3431–3440 (2015).

Luo, C., M.-K. Sit, H. Fan, S. Liu, W. Luk and C. Guo, “Towards efficient deep neural network training by fpga-based batch-level parallelism”, Journal of Semiconductors **41**, 2, 022403 (2020).

Ma, Y., Y. Cao, S. Vrudhula and J. Seo, “An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks”, in “Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)”, pp. 1–8 (2017).

Ma, Y., Y. Cao, S. Vrudhula and J. Seo, “Automatic Compilation of Diverse CNNs Onto High-Performance FPGA Accelerators”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems **39**, 2, 424–437 (2020).

Mahajan, D., J. Park, E. Amaro, H. Sharma, A. Yazdanbakhsh, J. K. Kim and H. Esmaeilzadeh, “TABLA: A Unified Template-based Framework for Accelerating Statistical Machine Learning”, in “Proc. of HPCA”, (2016).

Maji, S., E. Rahtu, J. Kannala, M. Blaschko and A. Vedaldi, “Fine-grained visual classification of aircraft”, arXiv preprint arXiv:1306.5151 (2013).

Mao, H., S. Han, J. Pool, W. Li, X. Liu, Y. Wang and W. J. Dally, “Exploring the Granularity of Sparsity in Convolutional Neural Networks”, in “IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR) Workshops”, (2017).

Masters, D. and C. Luschi, “Revisiting small batch training for deep neural networks”, arXiv preprint arXiv:1804.07612 (2019).

Nakahara, H., Y. Sada, M. Shimoda, K. Sayama, A. Jinguji and S. Sato, "FPGA-Based Training Accelerator Utilizing Sparseness of Convolutional Neural Network", in "IEEE International Conference on Field Programmable Logic and Applications (FPL)", pp. 180–186 (2019).

Netzer, Y., T. Wang, A. Coates, A. Bissacco, B. Wu and A. Y. Ng, "Reading digits in natural images with unsupervised feature learning", in "NIPS workshop on deep learning and unsupervised feature learning", vol. 2011, p. 5 (2011).

Nilsback, M.-E. and A. Zisserman, "Automated flower classification over a large number of classes", in "Computer Vision, Graphics & Image Processing, 2008. ICVGIP'08. Sixth Indian Conference on", pp. 722–729 (IEEE, 2008).

Nurvitadhi, E., G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra *et al.*, "Can FPGAs beat GPUs in accelerating next-generation deep neural networks?", in "Proceedings of ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)", pp. 5–14 (2017).

Nvidia, "Nvidia Deep Learning Accelerator (NVDLA)", URL http://nvdla.org/primer.html (2019).

Parashar, A., M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler and W. J. Dally, "SCNN: An Accelerator for Compressed-sparse Convolutional Neural Networks", in "Proc. of ISCA", (2017).

Park, E. and S. Yoo, "PROFIT: A Novel Training Method for sub-4-bit MobileNet Models", in "European Conference on Computer Vision (ECCV)", pp. 430–446 (2020).

Paszke, A., S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga and A. Lerer, "Automatic differentiation in PyTorch", in "NIPS 2017 Autodiff Workshop", (2017).

Qiu, J., J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *et al.*, "Going deeper with embedded fpga platform for convolutional neural network", in "ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)", pp. 26–35 (2016).

Rafael, G., C. Ricardo, C. Joaquín, C. Angel and W. M. Maeda, "FPGA implementation of a pipelined on-line backpropagation", Journal of VLSI Signal Processing **40**, 2, 189–213 (2005).

Ragan-Kelley, J., C. Barnes, A. Adams, S. Paris, F. Durand and S. Amarasinghe, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines", in "Proc. of PLDI", (2013).

Reagen, B., R. Adolf and P. Whatmough, *Deep Learning for Computer Architects* (Morgan & Claypool Publishers, 2017a).

Reagen, B., J. M. Hernández-Lobato, R. Adolf, M. Gelbart, P. Whatmough, G. Wei and D. Brooks, “A case for efficient accelerator design space exploration via bayesian optimization”, in “2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)”, pp. 1–6 (2017b).

Reagen, B., P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei and D. Brooks, “Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators”, in “Proc. of ISCA”, (2016).

Rebuffi, S., H. Bilen and A. Vedaldi, “Learning multiple visual domains with residual adapters”, CoRR **abs/1705.08045**, URL http://arxiv.org/abs/1705.08045 (2017a).

Rebuffi, S.-A., H. Bilen and A. Vedaldi, “Efficient parametrization of multi-domain deep neural networks”, in “Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition”, pp. 8119–8127 (2018).

Rebuffi, S.-A., A. Kolesnikov, G. Sperl and C. H. Lampert, “icarl: Incremental classifier and representation learning”, in “Proceedings of the IEEE conference on Computer Vision and Pattern Recognition”, pp. 2001–2010 (2017b).

Riera, M., J. Arnau and A. Gonzalez, “Computation reuse in dnns by exploiting input similarity”, in “2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)”, pp. 57–68 (2018).

Russakovsky, O., J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, “Imagenet large scale visual recognition challenge”, International Journal of Computer Vision **115**, 3, 211–252 (2015).

Samajdar, A., Y. Zhu, P. N. Whatmough, M. Mattina and T. Krishna, “Scale-sim: Systolic CNN accelerator”, CoRR **abs/1811.02883**, URL http://arxiv.org/abs/1811.02883 (2018).

Sandler, M., A. Howard, M. Zhu, A. Zhmoginov and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks”, in “Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)”, pp. 4510–4520 (2018).

SCALE-Sim, “Arm CNN accelerator simulator”, URL https://github.com/ARM-software/SCALE-Sim (2019).

Shafiee, A., A. Nag, N. Muralimanohar, R. Balasubramonian, J. Strachan, M. Hu, R. S. Williams and V. Srikumar, “ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars”, in “Proc. of ISCA”, (2016).

Sharma, H., J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra and H. Esmaeilzadeh, "From High-Level Deep Neural Models to FPGAs", in "Proc. of MICRO", (2016).

Simonyan, K. and A. Zisserman, "Very deep convolutional networks for large-scale image recognition", arXiv preprint arXiv:1409.1556 (2014).

Simonyan, K. and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition", in "International Conference on Learning Representations (ICLR)", (2015).

Song, L., X. Qian, H. Li and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning", in "Proc. of HPCA", (2017).

Srivastava, G., D. Kadetotad, S. Yin, V. Berisha, C. Chakrabarti and J. Seo, "Joint Optimization of Quantization and Structured Sparsity for Compressed Deep Neural Networks", in "IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)", pp. 1393–1397 (2019).

Stallkamp, J., M. Schlipsing, J. Salmen and C. Igel, "Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition", Neural networks **32**, 323–332 (2012).

Standard, J., "High bandwidth memory (HBM) DRAM", JESD235 (2013).

Suleiman, A., Y.-H. Chen, J. Emer and V. Sze, "Towards Closing the Energy Gap Between HOG and CNN Features for Embedded Vision", in "Proc. of ISCAS", (2017).

Sun, X., X. Ren, S. Ma and H. Wang, "meProp: sparsified back propagation for accelerated deep learning with reduced overfitting", in "Proceedings of the International Conference on Machine Learning (ICML)", pp. 3299–3308 (2017).

Tan, M. and Q. Le, "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks", in "International Conference on Machine Learning (ICML)", pp. 6105–6114 (2019).

Tateno, K., F. Tombari, I. Laina and N. Navab, "CNN-SLAM: Real-time dense monocular SLAM with learned depth prediction", in "IEEE Conference on Computer Vision and Pattern Recognition (CVPR)", pp. 6243–6252 (2017).

Tzeng, E., J. Hoffman, T. Darrell and K. Saenko, "Simultaneous deep transfer across domains and tasks", in "Proceedings of the IEEE International Conference on Computer Vision", pp. 4068–4076 (2015).

Umuroglu, Y., Y. Akhauri, N. J. Fraser and M. Blott, "LogicNets: Co-Designed Neural Networks and Circuits for Extreme-Throughput Applications", in "IEEE International Conference on Field-Programmable Logic and Applications (FPL)", pp. 291–297 (2020).

Umuroglu, Y., 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", FPGA '17, pp. 65–74 (ACM, New York, NY, USA, 2017), URL http://doi.acm.org/10.1145/3020078.3021744.

Venieris, S. I. and C.-S. Bouganis, "fpgaConvNet: A framework for mapping convolutional neural networks on FPGAs", in "IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)", pp. 40–47 (2016).

Venieris, S. I., A. Kouris and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions", ACM Comput. Surv. **51**, 3, 56:1–56:39, URL http://doi.acm.org/10.1145/3186332 (2018).

Venkataramanaiah, S. K., Y. Ma, S. Yin, E. Nurvithadhi, A. Dasu, Y. Cao and J.-s. Seo, "Automatic compiler based fpga accelerator for cnn training", in "2019 29th International Conference on Field Programmable Logic and Applications (FPL)", pp. 166–172 (IEEE, 2019a).

Venkataramanaiah, S. K., Y. Ma, S. Yin, E. Nurvithadhi, A. Dasu, Y. Cao and J.-s. Seo, "Automatic compiler based fpga accelerator for cnn training", in "2019 29th International Conference on Field Programmable Logic and Applications (FPL)", pp. 166–172 (IEEE, 2019b).

Viola, P. and M. J. Jones, "Robust Real-time Object Detection", IJCV (2004).

Wang, E., J. J. Davis, R. Zhao, H.-C. Ng, X. Niu, W. Luk, P. Y. Cheung and G. A. Constantinides, "Deep neural network approximation for custom hardware: Where we've been, where we're going", arXiv preprint arXiv:1901.06955 (2019).

Warden, P., "Why GEMM is at the heard of deep learning", URL https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/ (2015).

Wei, X., C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang and J. Cong, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs", in "IEEE/ACM Design Automation Conference (DAC)", pp. 1–6 (2017).

Wen, W., C. Wu, Y. Wang, Y. Chen and H. Li, "Learning Structured Sparsity in Deep Neural Networks", in "Advances in Neural Information Processing Systems (NeurIPS)", (2016).

Whatmough, P. N., S. K. Lee, D. Brooks and G. Wei, “Dnn engine: A 28-nm timing-error tolerant sparse deep neural network processor for iot applications”, IEEE Journal of Solid-State Circuits **53**, 9, 2722–2731 (2018).

Whatmough, P. N., C. Zhou, P. Hansen, S. K. Venkataramanaiah, J. Seo and M. Mattina, “FixyNN: Efficient Hardware for Mobile Computer Vision via Transfer Learning”, in “Conference on Machine Learning and Systems (MLSys)”, (2019).

Wissolik, M., D. Zacher, A. Torza and B. Da, “Virtex UltraScale+ HBM FPGA: A revolutionary increase in memory performance”, Xilinx Whitepaper (2017).

Wu, D., Y. Zhang, X. Jia, L. Tian, T. Li, L. Sui, D. Xie and Y. Shan, “A High-Performance CNN Processor Based on FPGA for MobileNets”, in “IEEE International Conference on Field Programmable Logic and Applications (FPL)”, pp. 136–143 (2019).

Wu, Y. and K. He, “Group normalization”, in “European Conference on Computer Vision (ECCV)”, pp. 3–19 (2018).

Yang, L., Z. He and D. Fan, “Harmonious Coexistence of Structured Weight Pruning and Ternarization for Deep Neural Networks”, AAAI Conference on Artificial Intelligence **34**, 04, 6623–6630 (2020).

Yang, Y., 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 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)”, pp. 23–32 (2019).

Ye, H., X. Zhang, Z. Huang, G. Chen and D. Chen, “HybridDNN: A Framework for High-Performance Hybrid DNN Accelerator Design and Implementation”, in “ACM/IEEE Design Automation Conference (DAC)”, pp. 1–6 (2020).

Yin, S. and J. Seo, “A 2.6 TOPS/W 16-bit Fixed-Point Convolutional Neural Network Learning Processor in 65nm CMOS”, IEEE Solid-State Circuits Letters **3**, 13–16 (2020).

Yosinski, J., J. Clune, Y. Bengio and H. Lipson, “How transferable are features in deep neural networks?”, in “Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2”, NIPS’14, pp. 3320–3328 (MIT Press, Cambridge, MA, USA, 2014), URL http://dl.acm.org/citation.cfm?id=2969033.2969197.

Yu, J., A. Lukefahr, D. Palframan, G. Dasika, R. Das and S. Mahlke, “Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism”, in “Proc. of ISCA”, (2017).

Yu, Y., T. Zhao, K. Wang and L. He, “Light-OPU: An FPGA-Based Overlay Processor for Lightweight Convolutional Neural Networks”, in “ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)”, p. 122–132 (2020).

Zeng, H., R. Chen, C. Zhang and V. Prasanna, “A framework for generating high throughput CNN implementations on FPGAs”, in “Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)”, pp. 117–126 (2018).

Zeng, H. and V. Prasanna, “GraphACT: Accelerating GCN Training on CPU-FPGA Heterogeneous Platforms”, in “ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)”, pp. 255–265 (2020).

Zhang, C., 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 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays”, pp. 161–170 (2015).

Zhang, J. and J. Li, “Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network”, in “Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)”, pp. 25–34 (2017).

Zhang, Y., M. Pezeshki, P. Brakel, S. Zhang, C. Laurent, Y. Bengio and A. Courville, “Towards end-to-end speech recognition with deep convolutional neural networks”, in “INTERSPEECH”, (2016).

Zhao, R., H.-C. Ng, W. Luk and X. Niu, “Towards efficient convolutional neural network for domain-specific applications on fpga”, in “IEEE International Conference on Field Programmable Logic and Applications (FPL)”, pp. 147–1477 (2018a).

Zhao, R., X. Niu and W. Luk, “Automatic Optimising CNN with Depthwise Separable Convolution on FPGA: (Abstact Only)”, in “ACM/SIGDA International Symposium on Field-Programmable Gate Arrays”, pp. 285–285 (2018b).

Zhao, W., 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 “Proceedings of the IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP)”, pp. 107–114 (2016).

Zhao, Y., X. Gao, X. Guo, J. Liu, E. Wang, R. Mullins, P. Y. K. Cheung, G. Constantinides and C. Xu, “Automatic Generation of Multi-Precision Multi-Arithmetic CNN Accelerators for FPGAs”, in “IEEE International Conference on Field-Programmable Technology (ICFPT)”, pp. 45–53 (2019).

Zhou, S., R. Kannan and V. K. Prasanna, “Accelerating Stochastic Gradient Descent Based Matrix Factorization on FPGA”, IEEE Transactions on Parallel and Distributed Systems **31**, 8, 1897 – 1911 (2020).

Zhu, M. and S. Gupta, “To prune, or not to prune: exploring the efficacy of pruning for model compression”, ArXiv e-prints (2017).

Zhu, Y., A. Samajdar, M. Mattina and P. Whatmough, “Euphrates: Algorithm-soc co-design for low-power mobile continuous vision”, in “Proceedings of the 45th Annual International Symposium on Computer Architecture”, ISCA ’18, pp. 547–560 (IEEE Press, Piscataway, NJ, USA, 2018), URL https://doi.org/10.1109/ISCA.2018.00052.

Zimmermann, R., “Datapath synthesis for standard-cell design”, in “2009 19th IEEE Symposium on Computer Arithmetic”, pp. 207–211 (2009).