Compressed CNN Training with FPGA-based Accelerator

ABSTRACT

Training convolutional neural network (CNN) usually requires large amount of computation resource, time and power. Researchers and cloud service providers in this region needs fast and efficient training system. GPU is currently the best candidate for CNN training. But FPGAs have already shown good performance and energy efficiency as CNN inference accelerators. In this work, we design a compressed training process together with an FPGA-based accelerator for energy efficient CNN training. We adopt two of the widely used model compression methods, quantization and pruning, to accelerate CNN training process.

The difference between inference and training brought challenges to apply the two methods in training. First, training requires higher data precision. We use the gradient accumulation buffer to achieve low operation complexity while keeping gradient descent precision. Second, sparse network results in different types of functions in forward and back-propagation phases. We design a novel architecture to utilize both inference and back-propagation sparsity. Experimental results show that the proposed training process achieves similar accuracy compared with traditional training process with floating point data. The proposed accelerator achieves 641GOP/s equivalent performance and 2.86x better energy efficiency compared with GPU.

KEYWORDS

FPGA, Convolutional Neural Network, Training

1 INTRODUCTION

Convolutional neural networks (CNN) have made significant performance improvement in computer vision tasks [10, 20]. However, the accuracy improvement comes at significant cost of both inference and training computation. A wide range of work have been proposed [6, 19] to achieve fast and energy efficient inference, showing outstanding performance over GPU. Few work focuses on the hardware accelerator design for training phase.

Training neural networks requires large amount of computation resources and can take days or weeks. Fast and energy efficient training is important for researchers and cloud service providers. GPU is currently the most suitable platform for CNN training. Recently, a number of cloud service providers including Microsoft, Amazon, Alibaba, and Huawei have deployed large FPGA clusters in their data centers. This provides the environment for FPGA-based neural network training acceleration.

The training of neural network requires similar operations as inference. Previous work on inference accelerator design gives hint on training acceleration. Model pruning and quantization have proved to be effective to reduce the requirements of computation, bandwidth and memory footprint, and have almost no effect on the performance (accuracy metric) of neural network inference [8]. A series of the accelerator designs [6, 19] follow these ideas with customized architecture. Compared with inference, applying pruning and quantization to training are facing more challenges.

The first challenge lies in the optimization of training. Existing work [8] applies pruning and quantization to training but only in the fine-tune phase after the model converges. The benefit of accelerating the fine-tune phase is quite limited. Using quantization and pruning in early stage of training is risky. Different from inference, training neural networks needs small steps of gradient descent to optimize the model parameters. Using low precision data can lead to convergence failure or accuracy decline. Pruning on the network before convergence can also limit the parameter space of the model and hurt the final training accuracy.

The second challenge lies in hardware design for processing sparse network training. Existing accelerator designs implements sparse matrix-vector multiplication kernels to skip the zero multipliers in sparse models. We refer to this kind of sparsity as operator-sparse pattern. In the back-propagation phase of training, utilizing sparsity means to avoid computing gradients for those already pruned parameters, which introduce the result-sparse pattern. Error propagation also requires the transposition of a sparse parameter matrix, bringing difficulty in accessing data efficiently from external memory. No existing designs support result-sparse pattern and sparse matrix transposition.

The third challenge lies in the great variety between the loop dimensions of forward and backward process of CNNs. Traditional fixed loop unrolling strategy may suffer from severe performance loss.

In this work, we address the above two challenges with the following contributions:

- We propose a hardware friendly training process with advanced quantization and pruning. Specially, we explore the effect of when to apply quantization and pruning with real tasks.
- We design dedicated processing elements (PEs) on FPGA to support both operator-sparse and result-sparse patterns. The sparse matrix transposition function is supported by specific scheduling method with a novel data organization in external memory.
- We design configurable loop mapping strategy for both forward and backward CNN computation.
- We evaluate a prototype system on FPGA to show the effectiveness of the design. Experimental results show that the proposed accelerator achieves 641GOP/s equivalent performance and 2.86x better energy efficiency compared with GPU.

The rest of this paper is organized as follows. Section 2 introduces the background of training of a CNN. Section 3 reviews previous work on software and hardware level CNN acceleration. Section 4 and section 5 introduces the proposed training process and hardware platform respectively. Experimental results are shown in section 6. Section 7 concludes this paper.

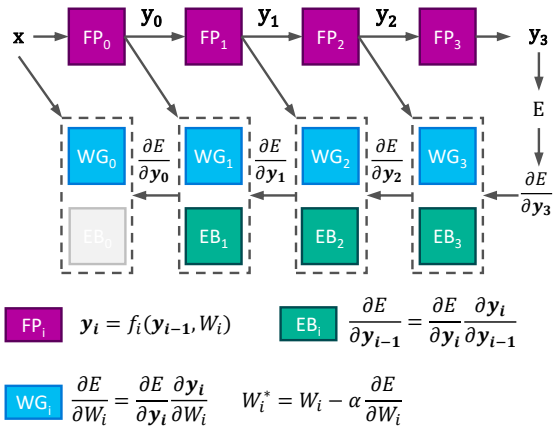

Figure 1: An example flow of training a 4 layer neural network. f_i and W_i denote the function and weight of each layer respectively.

2 PRELIMINARY

Before introducing the process of CNN training, we first claim the symbols used in this section in Table 1.

Table 1: Symbols used to describe training process

Symbol	Description
x	network input
y_i	output of layer i
f_i	function of layer i, can be 2d convolution, matrix vector multiplication, ReLU, pooling, etc.
W_i	weights of layer i
α	learning rate
E	error of the network output

2.1 Training Flow of CNN

Training of neural network is usually done with stochastic gradient descent. An example flow is shown in Figure 1 The training process consists of two alternate phases: forward phase(FP) and back-propagation(BP). A training process conducts these two phases iteratively until the model converges. The forward phase randomly select a batch of training inputs and calculates their inference results with the current model. The back-propagation phase calculates the gradient of error to the weights in each layer according to the chain rule and update the weights by gradient descent. We further separate this phase into 2 phases: error back-propagation (EB) phase calculates the gradients of error E to the activations of each layer; weight gradient (WG) phase calculates the gradient of weights $\partial E/\partial W_i$ in each layer and update the weights.

2.2 Computation Pattern for BP

The computation pattern for training CNN is similar to inference. For convolution layers, EB phase conducts convolution on the output error with the rotated kernels as shown in equation 1, where M and N denote the number of input and output channels. WG phase conducts the convolution on the input feature map with the output error as shown in equation 2.

$$\frac{\partial E}{\partial \mathbf{y_{i-1}}(m)} = \sum_{n=0}^{N-1} \text{conv2d}(\frac{\partial E}{\partial \mathbf{y_i}}, rot180(\mathbf{W_i}(m,n))) \tag{1}$$

$$\frac{\partial E}{\partial \mathbf{W_i}(m,n)} = \text{conv2d}(\mathbf{y_{i-1}}(m), \frac{\partial E}{\partial \mathbf{y_i}(n)}) \tag{2}$$

For fully connected layers, the inference phase is simply matrix vector multiplication. So the EB and WG phases can be expressed in equation 3 and 4 respectively.

$$\frac{\partial E}{\partial \mathbf{y_{i-1}}} = \mathbf{W_i}^\top \frac{\partial E}{\partial \mathbf{y_i}} \tag{3}$$

$$\frac{\partial E}{\partial \mathbf{W_i}} = \left(\frac{\partial E}{\partial \mathbf{y_i}}\right)^\top \mathbf{y_{i-1}} \tag{4}$$

3 RELATED WORK

In this section, we introduce the related work on neural network training and hardware accelerator design respectively. For training, we focus on data quantization and network pruning. For hardware accelerator, we introduce both inference and training accelerators.

3.1 Training with Fixed Point Data

We usually adopt 32-bit floating point data in both inference and training of neural networks. An effective way for neural network inference acceleration is to use fewer bits for data in NN models to reduce both memory and computation cost. Previous researches [7, 19] show that 8 or even fewer bits can be used to achieve significant acceleration on customized hardware architectures with negligible accuracy loss.

Other studies have tried to use fixed-point gradients [11, 12, 27] in neural network training. Although these work adopts extremely narrow bit-width for weights, activations and gradients, they all require storing floating-point scale factors with floating-point operations for multiplication and gradient distribution statistics. Floating point operations are still of high cost. For example, Zhou et al. [27] quantize the weights to 1 bit, activations to 2 bit, and the gradients of activations to 6 bit. The quantization method is to uniformly quantize each blob between [−absmax, absmax], and a floating-point absmax value must be stored for each blob. So the quantization process introduces more floating point operations.

3.2 Neural Network Pruning

Pruning is an efficient method to reduce both the model size and the number of operations by removing insignificant parameters and fine-tune the model, which helps achieve neural network acceleration. Han et al. [9] introduce a direct method by iteratively pruning parameters below a threshold. This method reduces network parameters to $1/9 - 1/13$. However, the pruning is done with each single parameter, referred to as 0-dimension pruning. The irregularity of the pruned model brings a challenge for executing the network efficiently on hardware. Higher dimensional pruning helps regularize the shape of the network and benefits hardware

design. Some researches focus on exploring group-wise sparsity (1-dimension or 2-dimension pruning)[13, 22, 26] of the convolution filters. Recently, filter-wise pruning (3-dimension pruning)[14, 18] is proposed to reduce model size while totally keeping the model regularity. As suggested by Mao et al.[17], 0 to 2-dimension pruning does negligible harm to the network accuracy while 3-dimension pruning incurs great accuracy loss. All the researches above prunes the network after the training process converges. Thus the benefit to accelerate the fine-tune process after pruning is limited.

3.3 CNN Inference Accelerator

Many hardware designs have been proposed to accelerate dense CNN inference. As suggested by [16], for a single layer, a CNN inference accelerator design involves three aspects: loop unrolling, loop tiling, and loop interchange. Loop unrolling strategy decides the parallelism or the peak performance of the hardware. [3, 19, 23] explore different unroll dimensions and hardware designs respectively. Loop tiling and interchange strategy decides how the computation of a layer is scheduled. When the data needs to be loaded from external memory, a good loop tiling and interchange strategy helps minimize bandwidth requirement and maximize the utilization of hardware computation power. For CNN training, the back propagation also consists of nested loops. Thus the same design methods can be applied.

With the research progress in neural network model pruning, more and more work focus on accelerating sparse CNN inference. One kind of work focus on utilizing the sparsity of network parameters [6, 24]. Albericio, et al.[1] utilize the data sparsity brought by ReLU function. Han et al. [7] utilizes both the sparsity of data and parameters. We refer to these kind of designs as operator-sparse type, which means the one or both the operators of MAC functions are sparse. For CNN training with sparse parameters, a new type of sparse computation, result-sparse is needed. When calculating the gradient of network parameters, the operators are activations of one layer and the gradients of the activations of the next layer, which can both be dense. But we do not need to compute the gradients of the pruned parameters. So the results are sparse.

3.4 CNN Training Accelerator

Compared with inference, less studies focus on training acceleration. Previous accelerator designs [15, 25] use 32-bit floating point for computation. Both the performance and energy efficiency of the accelerator are not competitive compared with GPU or state-of-the-art inference accelerators. Geng et al. propose the FPDeep [4] framework to map the training process on multiple FPGAs, which offers a high performance solution. Compared with previous work, FPDeep uses 16-bit fixed point data for training and designs independent modules for FP, EB and WG phases respectively. But the hardware still does not utilize sparsity in training. In this paper, we propose a hardware architecture that utilize both operator-sparsity and result-sparsity in CNN training.

4 HARDWARE FRIENDLY TRAINING

In this section, we introduce our hardware-friendly training method using fixed point data and advanced pruning. The whole training process includes the following steps:

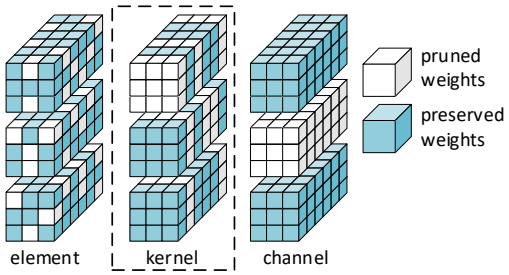

Figure 2: Illustration of the kernel-wise pruning

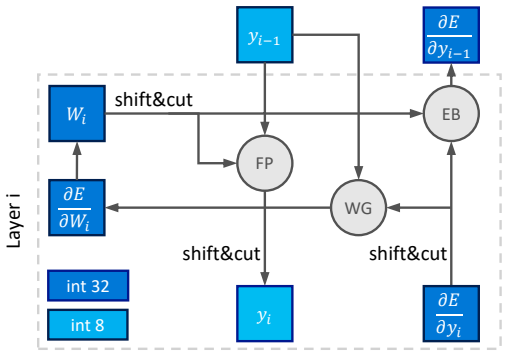

Figure 3: Proposed hardware-friendly training process with fixed-point data. The weights are stored with long bit-width (32-bit in the figure) to make valid accumulation. The computations are processed with short bit-width (8-bit in the figure).

- Train the model with full-precision activations, weights, gradients.
- Prune the model.
- Apply quantization to activations, weights, gradients according to the last epoch of training.
- Continue training the pruned model with fixed-point activations, weights, gradients.

We will focus on pruning and data quantization in this section.

4.1 Data Quantization in Training

4.2 Advanced Pruning

Pruning brings acceleration potential to training. By forcing part of the weights to zero, we reduce not only the computation in FP and EB phases, but also the computation in WG phases because the gradients to the zero weights are not needed. Traditional pruning is usually applied when the model is well trained. To increase the potential of acceleration, we choose to prune the model before the training process converges. We also use a structured pruning method for convolution layers to simplify the hardware design. We denote shape of weights as (N, C, H, W). N represents output channel, C represents input channel, H represents height, and W represents width. The pruning method masks each $H \times W$ kernel as a whole if the L2-norm of the kernel is small.

In this work, the pruning process is automatically done by pruning the kernels with the smallest L2-norms as long as the accuracy drop of the model is within a given range. To reduce the time for accuracy test, pruning is done layer by layer and with a step of 10% weights of each layer.

Traditional CNN training relies on full-precision data, i.e. 32-bit floating point data, to guarantee a good training accuracy. However, using fixed-point data in training process can help increase the energy efficiency. For CNN inference, various accelerators have been proposed with fixed point operations to increase energy efficiency. For training, using fixed point data usually suffers great model accuracy loss. Recent work [27] use narrow bit-width only for data storage in training process but have to convert the data to floating point to process addition and multiplication. In this paper, we propose a training process using both fixed point format for data storage and computation.

In the proposed training process, every fixed-point number is represented with low bit-width (e.g. 8 bit in our implementation) together with a scaling factor. We keep a common scale for each fixed-point blob, where a blob can be the activation, weights, gradient or error of a layer. Directly using this data format will induce two problems.

The first problem is how to convert the original floating point data to the fixed point version. One choice is to use the dynamic range of each blob as the scaling factor. This strategy keeps the data precision to the best degree, but brings extra statistic and normalization operations for each data blob in each iteration. In this work, we execute floating-point training iterations to analyze the dynamic scale of each blob and keep the scale through the rest of training process. We choose the nearest 2^n as the scaling factor which means data normalization can be implemented with shift operations on fixed-point data.

The second problem is the trade-off between bit-width and training accuracy. Low bit-width simplifies operations and reduce the storage consumption, but also reduce the model accuracy. Consider the small learning rate and gradient vanishing, the updated value in each iteration is small, the scaling factor of weights can be much larger than that of gradients. In this work, we use a large bit-width buffer (i.e. 32-bit) to store the weights but only use the MSBs of weights in FP and EB phases. Thus the accumulation of small gradients can be guaranteed while the computation is simplified. Figure 3 shows the proposed training example. All the MAC operations in FP, WG, and EB phases are executed with 8-bit fixed point data.

5 HARDWARE DESIGN

In this section, we introduce the hardware design for training CNN with sparse parameters. Especially, we will show architecture support for sparse operations and how this design differ from inference accelerators.

5.1 Overall Architecture

Figure 4(a) shows the overall architecture of the hardware design. When the accelerator works, the host CPU sends a mini-batch of training data to the accelerator to process an iteration of training. The accelerator first processes inference on the mini-batch and sends back the result vector to CPU to calculate the loss. After that, the gradient vector of the last layer is given back to the accelerator to do back propagation.

Memory. Compared with inference, the FP results of each layer of the mini-batch should be kept until BP is done. Though we use fixed-point data for training, the training on a mini-batch still requires GB level of memory, which is impractical to be stored on-chip. We use the on-board external memory to meet the large storage requirement of training. A set of on-chip buffers are implemented to explore the spatial locality of convolution operation and allows fast irregular data access for sparse operations. Details of each PE will be introduced in section 5.2.

Computation Kernel. The convolution and fully connected layers are executed on the hardware layer by layer with the PE array. Two stream processor: Pre-Stream and Post-Stream are attached to the PE arrays. These two processors execute the workload for ReLU and pooling layers as a pipeline stage before/after the PE array as shown in Figure 4(a). As pooling and ReLU layers has little workload, the stream processors are implemented as common modules for all the PEs and can be bypassed if necessary.

5.2 PE for Sparse Computation

To fully take advantage of the sparse weights in training, the hardware should support the following two types of sparse operations:

- **Operator-sparse**: For FP and EB steps, the activations or errors are multiplied with the sparse weights to get the result feature map or errors. So one of the two operators for multiplication is sparse.
- **Result-sparse**: For WG steps, the activations are multiplied with the back propagated errors to get the weight gradients. Both the operators of multiplication are dense but the result is sparse.

We choose to process the sparse computation in a block-wise manner. In this manner, the hardware can support both of the sparse operation types efficiently. An example of a 4×4 block of fully connected layer executed in a PE is shown in Figure 5. Each weight matrix element is stored with its relative 2-D position in the block. Each time a pair of index (x, y) is fetched from the index buffer. For inference, as shown in Figure 5(a), x is used to access the activations in data buffer and y is used to access the results. To calculate the gradients of the sparse parameters, data buffer and parameter buffer store the activation of two adjacent layers. x and y are used to access the buffers respectively. Different PEs work independently on different blocks for higher parallelism.

Note that for the case in Figure 5(a), changing the order of indexes will not change the result, as long as the order of indexes corresponds to that of the parameters. This property is also true for Figure 5(b). So in EB step, when we need to use a transpose format of the weight matrix as suggested in Section 2, we only need to exchange x and y with a hardware multiplexer and do not need to change the data format in external memory.

For CONV layers, as the sparsity is 2-d kernel level structured, the same sparsity is also supplied. If we replace the entries in the matrix with 2-d convolution kernels, each element in vector as a feature map, and scalar multiplication with 2-d convolution, the above example becomes a CONV layer example.

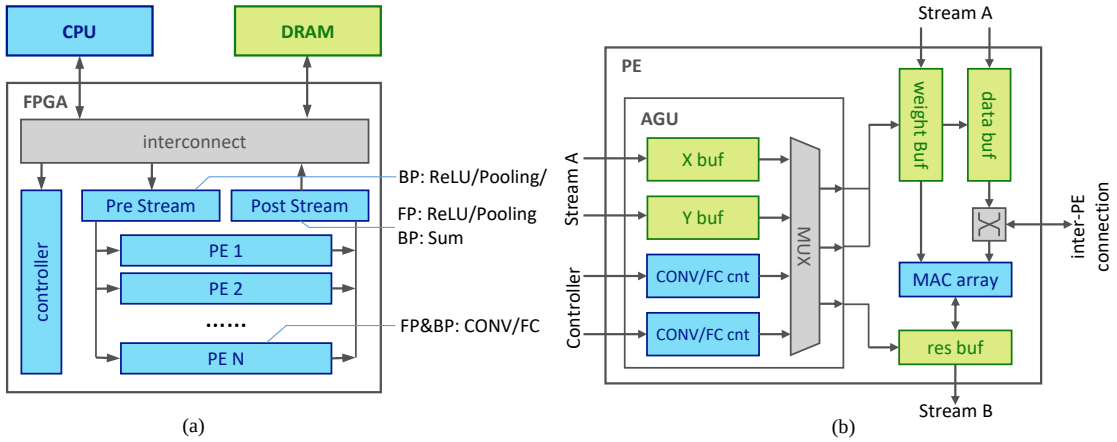

Figure 4: The CNN training accelerator architecture. (a) The overall system architecture. (b) The structure of a single PE.

The structure of each PE is shown in Figure 4(b). Similar to the example in Figure 5, we implement **data buf** for activations, **weight buf** for weights(convolution kernels) and **res buf** for activations with on-chip RAM. **data buf** and **weight buf** are in simple dual port mode and works in a ping-pong manner by splitting the address space. **res buf** is implemented with two simple dual port RAMs because accumulation function requires extra RD/WR port. We implement the address generation unit(AGU) to support the computation pattern. AGU implements **XBuf** and **YBuf** to store the relative index of each weights or convolution kernel within a processing block. For the buffers to be randomly accessed, the X and Y index read from index buffer directly serve as the high bits of RAM addresses, which help to select the channel or neuron to use. An index independent counter helps generate the address sequence to carry out 2-d convolution for CONV layer or scaler multiplication for FC layer. For the buffer to be sequentially accessed, another counter is used generate address sequence.

5.3 Loop Unrolling Strategy

There is already much discussions on how to unroll the loops on hardware for CNN inference with dense models [17, 23]. For CNN training, the loops of the back propagation phase is similar to that of the inference phase. But the loop dimension varies greatly between the FP and BP phases. For example, if a convolutional layer does 3×3 convolution on 224×224 feature maps with 1 padding, computing the gradient of these convolution kernels needs 224×224 convolutions on 224×224 feature maps with 1 padding. according to equation 2. This causes great varieties in convolution kernel sizes and convolution result sizes and thus limits the loop unrolling design. Introducing sparsity further limits the choice of loop unrolling parameters. We analyze different unroll dimensions in this section to decide the hardware design strategy.

Batch. Parallelism in batch dimension will not affect hardware design compared with inference accelerator. For training, usually a mini-batch is used in each iteration. So parallelism in batch is always preferred.

Layer. Parallelize the process of different layers means pipeline different layers. For CNN training, the result of processing one mini-batch is necessary for the next one. So implementing a long pipeline will cause large overhead between the process of adjacent mini-batches.

Input/output channel. The unroll parameter in these two dimensions are limited by network sparsity. This is caused by workload imbalance. For example, for a CONV layer, unroll the output channel with M means splitting the workload to M different hardware units to process in parallel. Because the network is kernel-level sparse, the number of 2-d convolution kernels for each hardware unit to process may different. We do experiments to show how the parallelism affects the hardware utilization ratio as shown in Figure 6. In Figure 6 (a), we suggest that the parameters are uniformly distributed with the density of 30% and varies the size of parameter. It is clear to see that to keep the utilization ratio, we should choose smaller parallelism for a smaller parameter size. In Figure 6 (b), we keep the parameter size and changes the sparsity. For parameters with less non-zero values, we need to use a smaller parallelism to get the same hardware utilization ratio.

Feature map. Unrolling in feature map means computing multiple output feature map pixels in parallel. If the output of 2-d convolution is comparable or even smaller than the unroll factor, there will be hardware overhead. For training of CONV layers, the WG steps can be expressed as the 2-d convolution of the input feature map with the gradient of the output feature map. The convolution output size is the same as the convolution kernel, which can be as small as 3×3. So the unroll factor should be carefully chosen.

Convolution kernel. Unrolling convolution kernel dimension is faced with the same problem as that of unrolling feature map, not in WG steps, but in FP and EB steps. So the unroll factor should be carefully chosen.

So we choose to unroll batch and not implement layer pipeline. Each PE implements b MAC units to process b inputs in parallel. To reduce the channel level parallelism, we explore the possibility of unrolling feature map and convolution kernel dimension in CONV

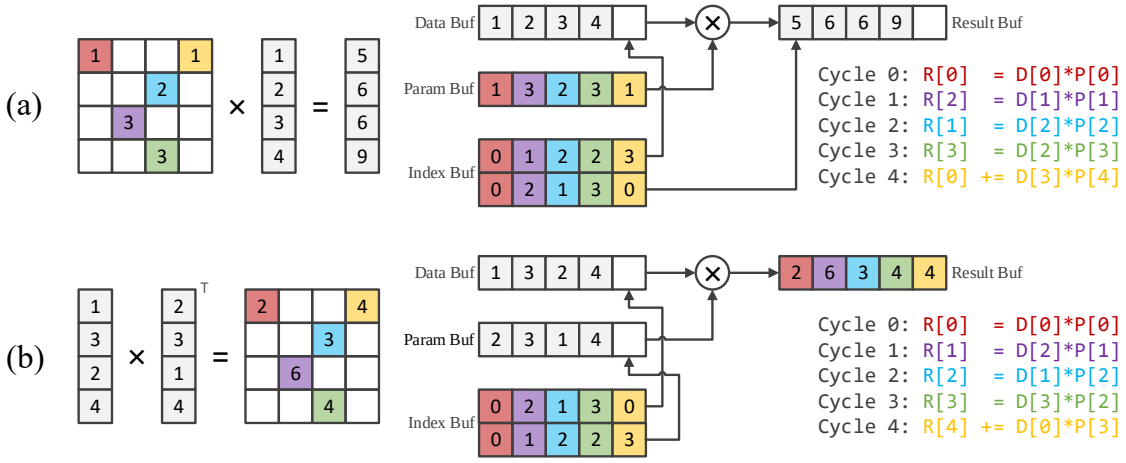

Figure 5: The hardware behavior for sparse network process. (a) Sparse matrix vector multiplication for inference and calculation of gradient of activations. (b) Vector vector multiplication to calculate the gradient of sparse parameters.

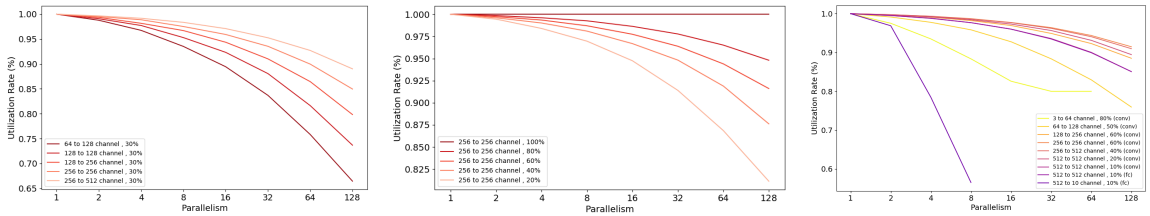

Figure 6: Hardware utilization ratio under different sizes of sparse parameters, sparsity, and PE numbers. (a) Estimation with a fixed sparsity and different parameter scales. (b) Estimation with a fixed parameter size and different sparsities. (c) Simulation on a real pruned VGG-11 model with different PE numbers.

layers. We propose a configurable unroll method between these two dimension.

For FP and EB phases of CONV layers, we let each m PEs compute m output pixels of a same feature map in parallel. To save on-chip buffer, we split the feature map to m PEs and enables data sharing among them with multiplexers. An example of a 2×2 convolution on a 3×3 feature map with 4PEs are shown in Figure 7(a). In each cycle, all the PEs fetch the same convolution kernel, multiplies it with a pixel and locally accumulate the result. After 4 cycles, the convolution is done. Note that except for the first cycle, no PE fetches data from its own buffer, which is enabled by data sharing. We simulate the effect of PE groups size on a pruned VGG-11 model, which is shown in Table 2. As can be seen, 2×2 group already increase the theoretical utilization to 93%, which means larger group sizes can bring limited performance improvement.

For WG step, we let each m PEs fetches m different convolution kernels (in this step is the feature map gradient) and accumulate the result for a same output pixel in parallel. This example is shown in Figure 7(b). The result are added together when data is written back to external memory.

We simulate the workload imbalance effect on designs with different number of PEs as shown in Figure 6 (c). We simulate on a real pruned VGG-11 network. As can be seen from the curves, except for the first and last layer of the network, most layers still shows over 90% utilization ratio with 32 PEs.

Table 2: Comparison of PE utilization with different group sizes on a network

Layer	Feature size	Workload (MOP)	Utilization		
			Single	2x2	Ideal
conv1	32×32	3.54	80	88.5	100
conv2	16×16	151	88.6	95.8	100
conv3_1	8×8	37.7	95.1	98.2	100
conv3_2	8×8	75.5	96.4	98.8	100
conv4_1	4×4	37.7	96.3	98.7	100
conv4_2	4×4	75.5	95.6	98.4	100
conv5_1	2×2	18.87	93.5	97.8	100
conv5_2	2×2	18.87	93.5	97.8	100
Normalized Speed			0.928	0.974	1

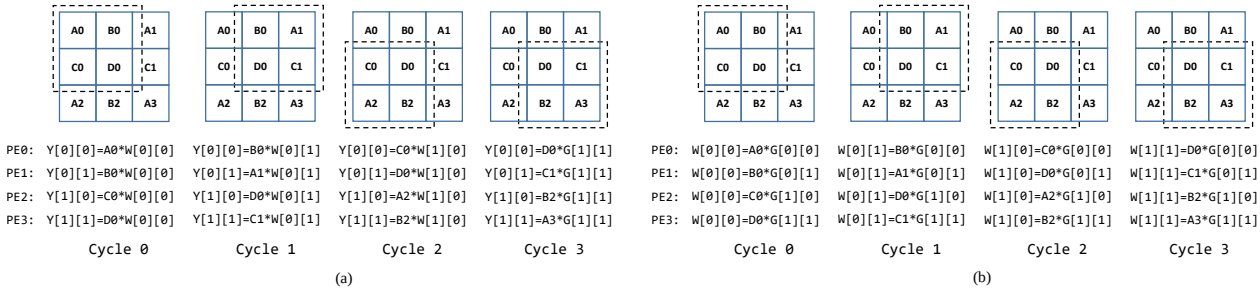

Figure 7: Example of 2×2 convolution on 3×3 feature map with 4 PEs. The letter in each pixel denotes the PE it belongs to. The number denotes the address it stores in the buffer. The feature map needed for each cycle is marked with the dashed line box. (a) convolution in FF step. (b) convolution in WG step.

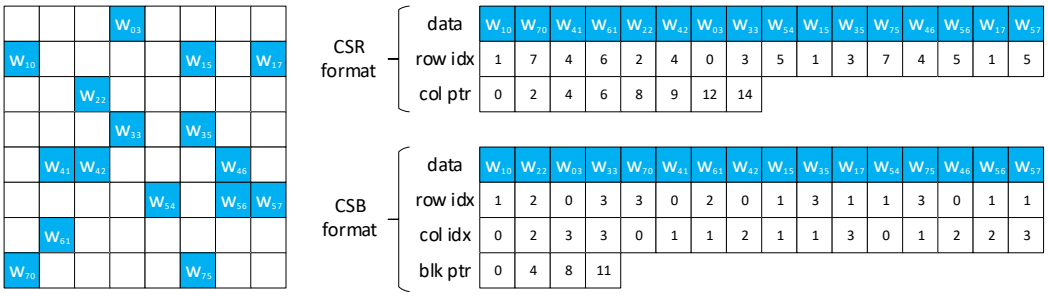

Figure 8: Example of different sparse matrix store formats for structured sparse convolution kernels. (a) Compressed Sparse Column(CSC) (b) Compressed Sparse Block(CSB)

5.4 Data Organization

First, we use the compressed sparse block (CSB) data format [2] for the sparse weight storage. The sparse weight For fully connected layer, the inference phase and back propagation phase needs to access the parameter matrix in the original version and a transposed version. For convolutional layer, this is similar if we treat the weights as a 2-D matrix of 2-D convolution kernels. The commonly adopted CSC format, as shown in Figure 8(a) stores data column by column, thus leads to the difficulty in row major access. The CSB format stores the elements block by block. In this format, the access direction can be controlled by changing the access order of blocks. This format also fit with the hardware's block-wise process behavior as introduced in section 5.2. Since the transpose of a single block is achieved in PE level, we can access the sparse weights in both original and transposed format.

Second, we increase the access continuity of feature map by using a $(CHWC_mN)$ storage format. As introduced above, we process the channels in a block manner, and we always process the batch in parallel. So we store a batch of a same pixel continuously. Then, we store each d channels of this pixel together. d is chosen the same as the corresponding convolution kernel block size of this layer. This requires that the output channel block size of one layer should be the same as the input channel block size of the next. So the memory access burst can reach $d * N * p$ where p is the width to be processed each time, limited by on-chip buffer size and image width.

5.5 Scheduling Strategy

As introduced above, each PE will process corresponding computations on a weight block each time. In our scheduling strategy, all the PEs will work with the same input channels/neurons, on different output channels/neurons. They first accumulate the result in ResBuf untill all the input channels are processed, then move on to the next set of output channels. For CONV layers, when all the (input, output) channel pairs are processed, the PEs will move on to the next part of feature map until the whole feature map is processed. Mainly 4 types of operations are involved in the scheduling strategy.

- **Common Load**: Load common data from external memory to all the PEs. In FP and WG phases, this is the common feature maps. In EB phase, this is the common feature map gradient.
- **Local Load**: Load local data to a PE or a group of PE. In FP and EB phases, this is the network weights and index. In the WG phase, this also includes weights buffer.
- **Calculation**: Run a single or group of PE to process a weight block.
- **Save**: Save the result of a single or group of PE to external memory.

Load operation, each PE(PE group), and save operation can work in parallel. Figure 9(a) shows an computation bounded time line example of a system with 4 PE groups. As long as $T_{calc} > T_{load\_common} + T_{load\_local} \times N_{pe}$, the system is computation

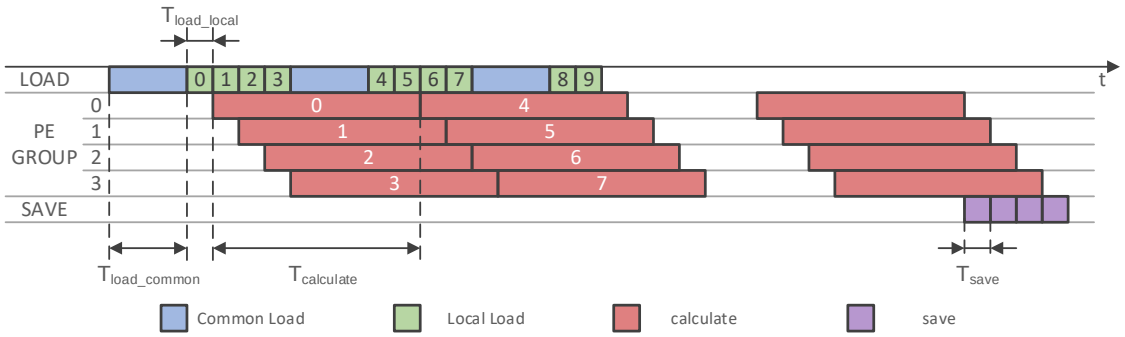

Figure 9: Example scheduling time line. The number for load local and calculate denote the data dependency

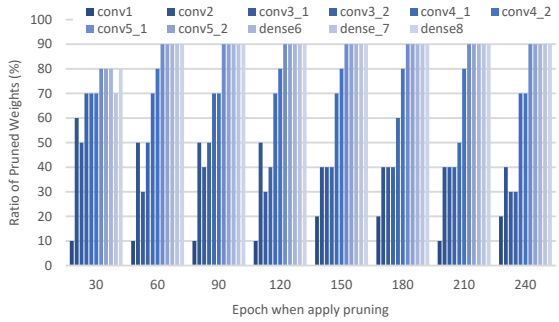

Figure 10: Ratio of pruned weights after different numbers of normal training epochs.

bounded. Note that in real cases, T_{calc} varies between different PEs and along time as the sparsity is random. T_save can usually be ignored because it appears much less than load.

6 EXPERIMENT

In this section, we first evaluate the proposed training process on the real dataset. Then we show the hardware experimental results.

6.1 Evaluation of Training

We first evaluate the proposed training method with a software version implemented with Tensorflow. We use the VGG-11 model [21] on CIFAR-10 dataset for evaluation. The model consists of 8 convolution layers and 3 fully connected layers. We first test pruning the model after different normal training epochs. We train the model with floating point data and saves checkpoints after every 30 epochs. The saved checkpoints are pruned using the method in section 4.2. The results are shown in Figure 10. The ratio of pruned weights does not vary greatly through the process of normal training. In general, most of the layers can be pruned to 40% or less of the original size. This brings great potential of acceleration if hardware supports sparsity well.

Then we test training the pruned models with different weight buffer sizes. We force the total training epochs to be 300. The model accuracy curves are shown in Figure 11. When using 32-bit weight buffer, we see that training still works when gradients are quantized.

Table 3: Prototype design resource utilization

Resource	LUT	Reg	Block RAM	DSP
Available	663360	1326720	2160	5520
Utilization	199111	249122	1060	1030
Ratio	30%	19%	49%	19%

But when to apply pruning and quantization matters. Starting as early as 30 epochs or as late as 24 epochs drops the final model accuracy from 90.62% to around 88%. When using 24-bit weight buffer, start quantization and pruning as late as 240 epochs can even lead to worse result: the accuracy does not improve in the rest of training process. Choose somewhere in the middle seems to bring good results but more experiments are needed to verify this conclusion. In this experiment, start quantization and pruning after only 60 epochs still brings 90.65% accuracy at the end of training. When using 16-bit weight buffer, all the training stops after 120 epochs of training because the learning rate is too small while the bit-width is too short.

6.2 Hardware Performance

A prototype design is implemented on the Xilinx KCU1500 development board, with 4 independent DDR. In this system, feature map, neuron and their gradients are all of 8-bit and stored in DDR0. Network weights, each of which includes 8-bit weight for computation and extra 16-bit buffer, are stored in DDR1. Corresponding index for sparse representation are stored in DDR0 using {y[3:0], x[3:0]} format. This means the the maximum weights block size can be 16×16. The system operates at 250MHz. Each 2×2 PEs are grouped together for CONV layers. The resource consumption after implementation is shown in Table 3. The power cost of the design is estimated to be 26.8W by Vivado 2018.1.

We use the model with 24-bit weight buffer after training 60 epochs for evaluation. We simulate the performance of each layer in each training step with the DDR model and controller from Xilinx IP. Detailed running time and performance are shown in Table 4. The peak performance of the hardware is $250MHz \times 1024DSP \times 2 = 500GOP/s$. For FF and NG steps, the proposed hardware achieves 900GOP/s overall performance which achieves at least 1.8× speedup over a dense accelerator with the same peak performance. From

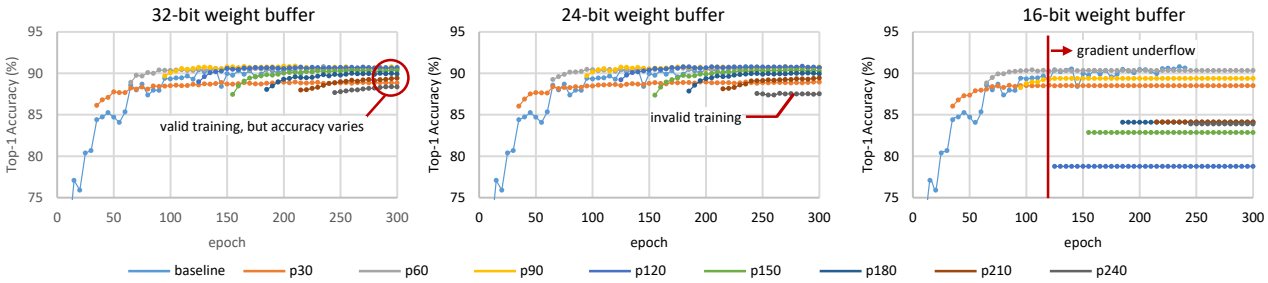

Figure 11: Accuracy curve of different training evaluation when quantization and pruning are applied at different stages. The number for each curve denotes the number of normal training epochs and baseline denotes a training without quantization and pruning.

Table 4: The performance of each layer. *Comp.* indicates the complexity of each layer. *Perf.* indicates the performance of running each layer on our hardware. *Bound type* indicates the performance of each layer is bounded with bandwidth (B) or computation (C).

layer		Forward Pass (FP)				Error Backpropagation (EB)				Weight Gradient (WG)			
	Comp. (GOP)	Time (us)	Perf. (GOP/s)	bound type	Utilize rate	Time (us)	Perf. (GOP/s)	bound type	Utilize rate	Time (us)	Perf. (GOP/s)	bound type	Utilize rate
conv1	0.11	733	154.4	B	27%	-	-	-	-	4158	27.2	B	5%
conv2	1.21	1487	812.2	C	95%	1487	812.5	C	95%	2804	430.8	B	50%
conv3_1	1.21	1696	712.4	C	97%	1694	713.0	C	97%	2477	487.7	B	67%
conv3_2	2.42	2877	839.7	C	98%	2876	840.1	C	98%	4398	549.3	B	64%
conv4_1	1.21	1217	992.3	C	97%	1217	992.9	C	97%	2686	449.8	B	44%
conv4_2	2.42	1457	1658.4	C	97%	1456	1659.2	C	97%	3933	614.2	B	36%
conv5_1	0.60	366	1651.7	B	32%	358	1687.7	B	33%	915	659.8	B	13%
conv5_2	0.60	365	1652.7	B	32%	358	1688.7	B	33%	915	660.0	B	13%
dense6	0.02	329	51.0	B	1%	328	51.1	B	1%	717	23.4	B	0.5%
dense7	0.02	329	51.0	B	1%	328	51.1	B	1%	717	23.4	B	0.5%
dense8	0.003	75	4.4	B	0.1%	74	4.4	B	0.1%	272	1.2	B	0.02%
total	9.81	10931	897.5	-	-	10988	892.8	-	-	23993	408.9	-	-

the bound type column, we see that most of the CONV layers with heavy workload are computation bounded. This shows that the proposed accelerator can handle large network well.

Besides that, those layers which are bandwidth bounded gives insights to hardware design methods. The first layer suffers from the bandwidth problem because the channel number for this layer is small. We only cut 3×8 block for this layer. A small block size increases the ratio between the necessary feature maps and the necessary convolution kernels. Compared with the workload imbalance result in 5.3, we see that small layers suffer more on bandwidth rather than workload imbalance. Besides reducing the number PEs, increase the buffer size in PEs can help further explore the data locality of 2-d convolution and improves performance.

The last few layers also suffers greatly from a limited bandwidth. On the one hand, FC layers and convolution layers with small feature maps are of high bandwidth cost for network parameters. On the other hand, split the parameters into small blocks decreases the memory access efficiency. When upgrading the weights of each layer, the weight buffer consumes more bandwidth and causes the performance loss. Reduce the weight buffer size should also be a future research topic.

A performance and energy efficiency comparison is with state-of-the-art neural network inference/training accelerators and GPU training result is shown in Table 5. The GPU used for comparison is GTX Titan X GM200. Although GPU achieves about 2x speed compared with FPGA, the FPGA part only consumes 1/5 power compared with GPU and achieves 2.86x energy efficiency. The proposed accelerator achieves much higher energy efficiency compared with the designs [15, 25] using floating point data.

The performance of the proposed design is still not as good as FPDeep [4]. This may be caused by two reasons. The first reason is that FPDeep use stores all the weights of convolution layers on-chip by using multiple FPGAs. The proposed design in this paper is limited by memory system as can be seen from Table 2 and Table 4. The second reason is that FPDeep designs computation kernels for each layer independently, which leads to high utilization ratio. This shows that scaling down FPDeep is hard. We will also explore ways to scale up the proposed design in the future.

Table 5: Performance comparison between the proposed accelerator and existing CNN inference/training accelerators and GPU

Platform	[5]	[16]	[6]	[25]		[15]	[4]	Proposed		Titan X
	XC7Z020	GX1150	KU060	Maxeler MPC-X		ZU19EG	VC709	KCU1500		GM200
Function	Inference	Inference	Inference	Inference	Training	Training	Training	Inference	Training	Training
Quantization	fixed 8	fixed 8/16	fixed 12	float 32	float 32	float 32	fixed 16	fixed 8	fixed 8/24	float32
Sparsity	No	No	Yes	No	No	No	No	Yes	Yes	No
Performance (GOP/s)	84.3	645.25	2516	62.06	7.01	86.12	1022 (Per FPGA)	897.5	641.1	1252
Power (W)	3.5	21.2	41	N.A.	27.3	14.2	32	26.8		150
Energy Eff. (GOP/s/W)	24.1	30.43	61.4	N.A.	0.27	6.05	31.97	33.61	24.01	8.4

7 CONCLUSION

In this paper, we accelerate neural network training with both software optimization and hardware design. We propose a hardware friendly training process where the fine-tune stage are totally based on fixed-point data computation and using a 2 dimension pruning method to produce regular model sparsity. Specialized processing units are designed to utilize model sparsity in both the inference and back propagation phases for acceleration. A hardware accelerator is designed to adapt to the great loop dimension variety between CNN inference phase and back propagation phase and cover bandwidth cost with certain data arrangement and schedule strategy. Proposed hardware achieves 641GOP/s equivalent performance and 2.63x better energy efficiency compared with GPU.

REFERENCES

[1] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: ineffectual-neuron-free deep neural network computing. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*. IEEE, 1–13.

[2] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 233–244.

[3] Zidong Du, Robert Fasthuber, Tianshi Chen, et al. 2015. ShiDianNao: shifting vision processing closer to the sensor. In *ISCA*. ACM, 92–104.

[4] Tong Geng, Tianqi Wang, Ahmed Sanaullah, Chen Yang, Rui Xu, Rushi Patel, and Martin Herbordt. 2018. FPDeep: Acceleration and Load Balancing of CNN Training on FPGA Clusters. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 81–84.

[5] Kaiyuan Guo, Lingzhi Sui, Jiantao Qiu, Jincheng Yu, Junbin Wang, Song Yao, Song Han, Yu Wang, and Huazhong Yang. 2017. Angel-Eye: A Complete Design Flow for Mapping CNN onto Embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2017).

[6] Song Han, Junlong Kang, Huizi Mao, Yiming Hu, Xin Li, Yubin Li, Dongliang Xie, Hong Luo, Song Yao, Yu Wang, et al. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA.. In *FPGA*. 75–84.

[7] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. 2016. EIE: efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*. IEEE Press, 243–254.

[8] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149* (2015).

[9] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both Weights and Connections for Efficient Neural Network. In *Advances in Neural Information Processing Systems*. 1135–1143.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[11] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks. In *Advances in neural information processing systems*. 4107–4115.

[12] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations. *Journal of Machine Learning Research* 18 (2017), 187–1.

[13] Vadim Lebedev and Victor Lempitsky. 2016. Fast convnets using group-wise brain damage. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2554–2564.

[14] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, and Hans Peter Graf. 2016. Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710* (2016).

[15] Zhiqiang Liu, Yong Dou, Jingfei Jiang, Qiang Wang, and Paul Chow. 2017. An FPGA-based processor for training convolutional neural networks. In *Field Programmable Technology (ICFPT), 2017 International Conference on*. IEEE, 207–210.

[16] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 45–54.

[17] Huizi Mao, Song Han, Jeff Pool, Wenshuo Li, Xingyu Liu, Yu Wang, and William J. Dally. 2017. Exploring the Granularity of Sparsity in Convolutional Neural Networks. In *Computer Vision and Pattern Recognition Workshops*. 1927–1934.

[18] Pavlo Molchanov, Stephen Tyree, Tero Karras, Timo Aila, and Jan Kautz. 2016. Pruning convolutional neural networks for resource efficient transfer learning. *arXiv preprint arXiv:1611.06440* (2016).

[19] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, et al. 2016. Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 26–35.

[20] E Shelhamer, J. Long, and T Darrell. 2017. Fully Convolutional Networks for Semantic Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39, 4 (2017), 640.

[21] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[22] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. 2016. Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems*. 2074–2082.

[23] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 161–170.

[24] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An accelerator for sparse neural networks. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 20.

[25] Wenlai Zhao, Haohuan Fu, Wayne Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang. 2016. F-CNN: An FPGA-based framework for training convolutional neural networks. In *Application-specific Systems, Architectures and Processors (ASAP), 2016 IEEE 27th International Conference on*. IEEE, 107–114.

[26] Hao Zhou, Jose M Alvarez, and Fatih Porikli. 2016. Less is more: Towards compact cnns. In *European Conference on Computer Vision*. Springer, 662–677.

[27] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients. *arXiv preprint arXiv:1606.06160* (2016).