Compressed CNN Training with FPGA-based Accelerator

ABSTRACT

Training convolutional neural network (CNN) usually requires large amount of computation resources, time and power. Researchers and cloud service providers in this region needs fast and efficient training system. GPU is currently the best candidate for DNN 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. 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 3x better energy efficiency compared with GPU.

KEYWORDS

FPGA, Convolutional Neural Network, Training

ACM Reference Format:

. 2018. Compressed CNN Training with FPGA-based Accelerator. In *Proceedings of ACM Conference (FPGA'19).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Convolutional neural networks (CNN) has made significant performance improvement in computer vision tasks [7, 16]. However, the accuracy improvement comes at significant cost of both inference and training computation. A wide range of work have bee proposed [3, 15] to achieve fast and energy efficient inference, showing outstanding performance over GPU. But the training phase of neural networks was not fully focused on.

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 [5]. A series of the accelerator designs [3, 15] follow these ideas with specific hardware. But applying pruning and quantization to training are faced with challenges.

The first challenge lies in the optimization of training. Existing work [5] 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 process fine steps of gradient descent to improve the model accuracy. 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.

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 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 3x 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 optimization.

FPGA'19, February 2019, Monterey, CA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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.

2 PRELIMINARY

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

Table 1: Symbol used to describe training

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). The forward phase randomly select a batch of training inputs and calculates their inference result 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 gradient of error, $\partial E/\partial y_i$ to the output of each layer; weight gradient (WG) phase calculates the gradient of weights $\partial E/\partial W_i$ in each layer and update the weights. One trains a CNN by iteratively conduct FP and BP until the weights converge.

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 channel. WG phase conducts the convolution on the input feature map with the output error as shown in equation 2.

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

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

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.

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

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

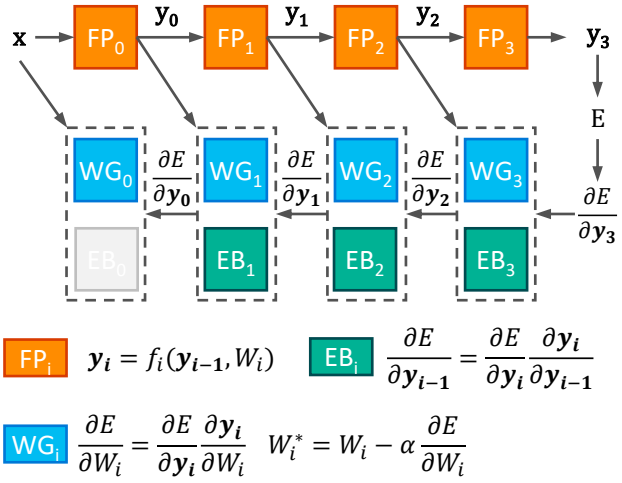

Figure 1: An example flow of training a 4 layer neural network. f_i and W_i denote the function and weight of each layer respectively.

3 RELATED WORK

In this section, we will introduce related work on software and hardware respectively. For software, we will focus on neural network training with fixed-point data and pruned parameters. For hardware, as few previous work targets at training acceleration, we will focus on inference accelerators.

3.1 Training with Fixed Point Data

Usually, neural network models are trained and used with 32-bit floating point data on GPU or CPU. An effective way for neural network acceleration is to use fewer bits for data in NN models to reduce memory cost and simplify each of the operation.

For inference, previous research [4, 15] show that 8-bit or even fewer bits can be used to achieve significant acceleration on customized hardware architectures with negligible accuracy loss. Courbariaux et al.[8] demonstrate how to train a binary NN model which brings relatively small accuracy loss on the CIFAR-10 dataset. Tang et al. [18] focus on how to enable better training of a BNN(binarized NN) on larger datasets using strategies such as small learning rate, bipolar regularization, etc.

Some work also studies how to perform the training process with fixed-point gradients [8, 9, 22], in these work, not only the weights and the activations, the gradients in the training process are also quantized. Zhou et al. [22] quantize the weights to 1 bit, activations to 2 bit, and the gradients of activations to 6 bit. Their quantization method is to uniformly quantize each blob between [−absmax, absmax], and a floating-point absmax value must be stored for each blob. To do quantization, they need global information of each blob (calculate the floating-point number of each blob). Although the bit-width of data blobs is low in these work, 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 while total fixed-point based training process still needs research.**

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, which helps achieve neural network acceleration. Han et al. [6] introduced a direct method by iteratively pruning parameters below the threshold and fine-tuning. Such element-wise pruning (0-dimension pruning) is powerful for compression and reduces storage 9x-13x. However, the irregularity of the pruned model brings a challenge for executing the network efficiently on hardware. Some researches focus on exploring group-wise sparsity (1-dimension or 2-dimension pruning)[10, 19, 21] of the convolution filters. Recently, pruning filters (3-dimension pruning)[11, 14] is proposed to reduce model size while totally keeping the model regularity. As suggested by Mao et al.[13], 0 to 2-dimension pruning does negligible harm to the network accuracy while 3-dimension pruning incurs great accuracy loss. In this work, we choose 2-dimension pruning which keeps acceptable model accuracy and relatively good regularity.

3.3 Dense CNN Inference Accelerator

Many hardware designs have been proposed to accelerate dense CNN inference. As suggested by [12], 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. [2, 15, 20] explore different unroll dimensions and hardware designs respectively. Loop tiling and loop interchange strategy decide how the computation of this layer is scheduled. When the data needs to be loaded from external memory, a good loop tiling and loop 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.

For CNN training, as suggested by section 2, the loops in back propagation phase is similar to that in inference phase. But the loop dimension varies greatly between the feed forward and back propagation 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. **This causes great variety in convolution kernels and convolution result sizes and thus limits the loop unroll design.**

3.4 Sparse CNN Inference Accelerator

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 [3**?**], which is denoted by $S \times D = D$ type. [1] utilizes the data sparsity brought by ReLU function, which is denoted by $D \times S = D$. [4] utilizes both the sparsity of data and parameters. For CNN training with sparse parameters, a new type of sparse computation, $D \times D = S$ is needed. When calculating the gradient of network parameters, the operators are activation of one layer and the gradient of the activation of the next layer, which can both be dense. But we do not need to compute the gradient of the pruned parameters. So the result is sparse. In this work, we propose an architecture to support both $S \times D = D$ and $D \times D = S$ computation.

4 HARDWARE FRIENDLY TRAINING

We first introduce two hardware-friendly training methods: fixed-point training and structured sparsity, followed by validating these methods on the real dataset.

4.1 Fixed-point Data Based Training

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 of training. For CNN inference, varies 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 [22] 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 of a layer, the weights of a layer, or the gradients of the weights/activations of a layer. Using this data format naively will induce two problems.

The first problem is how to convert the original floating point data to the fixed point version. This means to decide the scaling factor for each data blob. One choice is to use the dynamic range of each blob as the scaling factor can keep the data precision to the best degree, but this brings extra statistic and normalization operations for each data blob in each iteration. In this work, we focus on fine-tuning a pruned network. So we execute floating-point training iterations to analyze the dynamic scale of each blob and keep the scale through the rest of training process. Furthermore, 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. For CNN training, the update step in each iteration is usually small because of small learning rate and gradient vanishing. Thus the update step is easy to underflow if using fixed point data with narrow bit-width. In this work, we use narrow bit-width for activations and gradients. Because the scaling factor of gradient blob can be much smaller than that of weights, we use a larger bit-width to avoid gradient underflow when the gradients and weights are to be aligned and added together. To reduce the hardware cost for each operation, we only use the MSBs of weights to execute feed forward and back propagation.

An example of the proposed fixed-point based training process is shown in Figure 2. The right part of the figure shows the feed forward phase of the network from top to bottom while the left part shows the back propagation phase from bottom to top. The learning rate is merged with the scaling factor and converted to shift operation before computing ΔW and ΔB. Currently, weight decay and momentum are not supported in our hardware design. So we also omit these two functions in our software experiments.

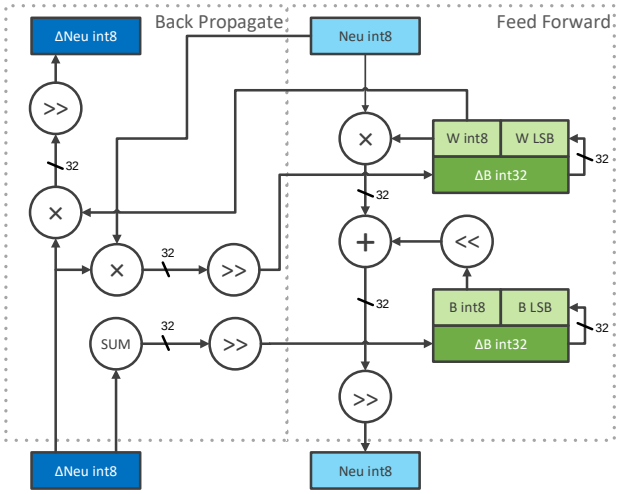

Figure 2: An example of the proposed training process with 8-bit for neuron and weight MSB and 24bit for weight LSB buffer.

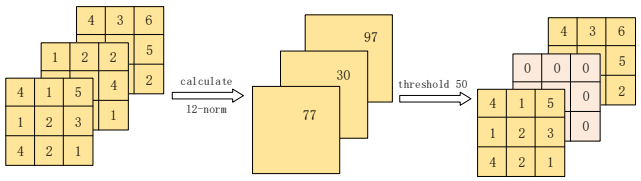

Figure 3: Illustration of the group-wise pruning

These two functions will not greatly affect hardware design and are to be supported in the future.

4.2 Structured Pruning

We denote shape of weights as (N, C, H, W). N represents output channel, C represents input channel, H represents height, and W represents weight. Compared to element-wise pruning, group-wise pruning is more efficient for hardware to realize. We choose kernel-dim pruning, which means the atomic elements during pruning is $H \times W$. L2-norm of the atomic elements is calculated, and atomic elements with small l2-norm are pruned. Mask matrix is used to label where is pruned and has the same shape as weights. If the weight is pruned, its mask becomes 0. Then we get sparse weights and mask matrix.

4.3 Software Validation

The whole training process used in this work is as follows:

- Training a fixed-point model with fixed-point weights and activations using full-precision gradients on software.
- Pruning weights and getting the mask matrix.
- Training a fixed-point model on hardware using fixed-point gradients while keeping pruned weights zero.

Table 2: Structure of the neural network for the experiment. Sparsity denotes the ration of zeros element or convolution kernel in a layer.

Layer	Type	Dim (NxCxHxW)	Sparsity
conv1	conv	64x3x3x3	0.1
pool1	max pool	-	-
conv2	conv	128x64x3x3	0.4
pool2	max pool	-	-
conv3_1	conv	256x128x3x3	0.3
conv3_2	conv	256x256x3x3	0.4
pool3	max pool	-	-
conv4_1	conv	512x256x3x3	0.5
conv4_2	conv	512x512x3x3	0.7
pool4	max pool	-	-
conv5_1	conv	512x512x3x3	0.9
conv5_2	conv	512x512x3x3	0.9
pool5	max pool	-	-
dense6	fc	512x512	0.9
dense7	fc	512x512	0.9
dense8	fc	512x10	0.9

The first two steps are implemented with software which use CPU and GPU as the computation platform. The third step can be executed with the proposed hardware architecture. To test the performance of the fixed-point based training method, we implemented a software version of the fixed-point data based CONV, ReLU, and pooling layers on TensorFlow.

In our implementation, the bit width of the weight buffer is set to be 24bit. The fixed scale of every gradient accumulation buffer is decided using the weight scale, the gradient scale and the learning rate after the first step. In this fine-tune stage, the fixed scales for every weight/activation/gradient blobs are fixed, and no momentum or weight decay is used. This ensures that every detail will be the same as the hardware implementation.

We perform the experiment using VGG-11[17] on CIFAR-10[**?**] dataset. While training in the first step, the learning rate is set to 0.05 and decayed by 0.5 every 30 epochs; weight decay is set to 5×10^{-4} and momentum is set to 0.9. We pruned model to the same sparsity in the whole experiments shown in Table 2. In the third training step with fixed point data, we compare the result of training with and without momentum. The accuracy is nearly the same(90.54 vs. 90.53).

Furthermore, we also evaluate where is a good point to stop floating point training and starts fixed point training with pruned weights. Usually, we prune model when it is perfectly trained and achieves the best accuracy. But if we want to save the training time, we do not need to prune model until it perfectly trained. Table 3 shows experimental results for pruning at different training stages.

The result in Table 3 shows that, if we start pruning at half of training, it may cost less epochs without harming accuracy, even with small accuracy improvement. The best accuracy occurred having 130 epochs of dense training and 200 epochs of sparse training. The stopping criteria is when the learning curve starts to flatten. In this training task, 200 out of 330 epochs can be accelerated with hardware.

Table 3: Comparison of training result with different number of initial epochs before training

Initial accuracy	Initial epochs	Fine-tune Accuracy	Fine-tune epoches	Total epoches
90.98	220	90.81	100	320
90.05	130	91.11	200	330
89.58	100	91.01	195	295
88.06	65	90.54	130	195
85.50	45	90.05	115	160

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. This process is iteratively executed until training converges.

External memory are implemented to meet the large storage requirement of training. Compared with inference, the feed-forward results of each layer of the mini-batch should be kept until back propagation 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. 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.

The CNN layers are executed on the hardware layer by layer. For pooling and ReLU layers, the FF, NG and WG stages are all implemented as a pipeline stage after or before PE. Because pooling and ReLU layers has little workload, the functions are implemented as common modules for all the PEs and can be bypassed if necessary. For each layer, the data is first loaded from DDR and is sent to the Upsample/Mask unit. In inference phase, the data is directly issued to corresponding PEs. In back propagation phase, upsampling and mask function is applied on the activation data. Then in each PE, the calculation of the nested loops are processed. Then the results are sent to Pooling/ReLU/Merge unit where the pooling and ReLU function is applied as traditional CNN inference accelerator design. The merge function adds gradient result from different PEs during back propagation phase.

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:

- $D \times S = D$: For FF and NG steps, the neurons or neuron gradients are multiplied with the sparse weights to get the neuron of next layer or the gradient of the previous layer. So one of the two operators for multiplication is sparse.
- $D \times D = S$: For wG steps, the neurons are multiplied with the back propagated neuron gradients 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 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 6. 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 6(a), x is used to access the activations in data buffer and y is used to access the result. To calculate the gradient 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 6(a), changing the order of indexes in index buffer 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 6(b). So in NG step, when we need to use a transpose format of the weight matrix, we only need to exchange x and y with a hardware multiplexer and do not need to change the data format in external memory as suggested in Section **??**.

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.

The structure of each PE is shown in Figure 4(b). Similar to the example in Figure 6, we implement **DataBuf** for neuron(feature map), **ParamBuf** for weights(convolution kernels) and **ResBuf** for neuron(feature map) with on-chip RAM. **DataBuf** and **ParamBuf** are in simple dual port mode and works in a ping-pong manner by spliting the address space. **ResBuf** is implemented with two simple dual port RAMs because accumulation function requires extra RD/WR port.

The detail of the address generation unit(AGU) is shown in Figure 5. AGU implements **XBuf** and **YBuf** to store the relative index of each weights or convolution kernel within a processing block. We use a multiplexer on the write port of the buffer to support transpose function. 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 enough discussions on how to unroll the loops on hardware for CNN inference with dense models [13, 20]. But for back propagation and sparse model, the strategy should be different. Here, we analyze different unroll dimensions one by one.

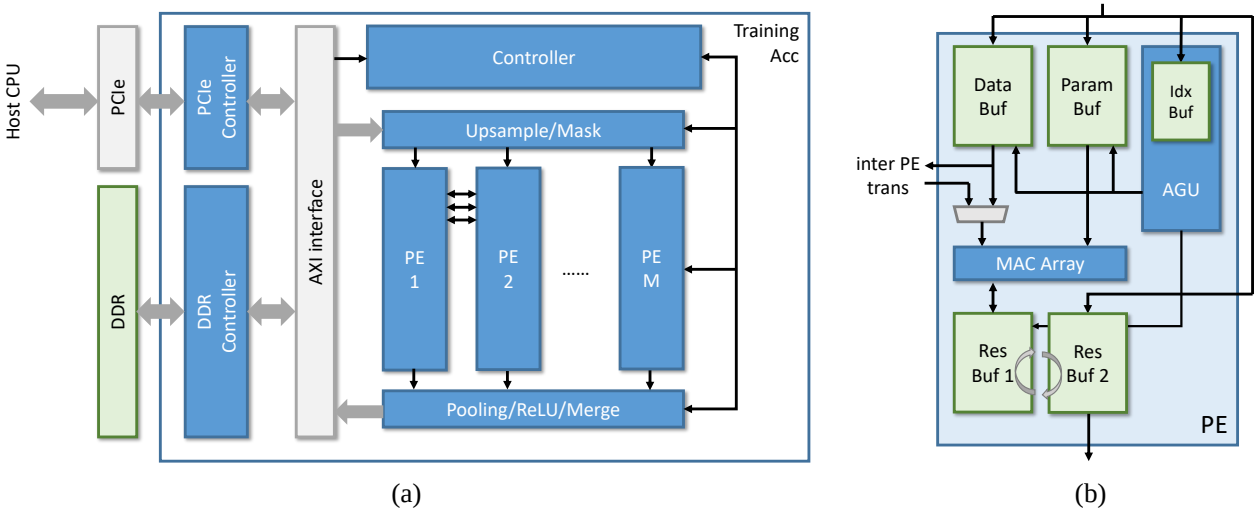

Figure 4: The CNN training accelerator architecture. (a) The overall system architecture. (b) The structure of a single PE.

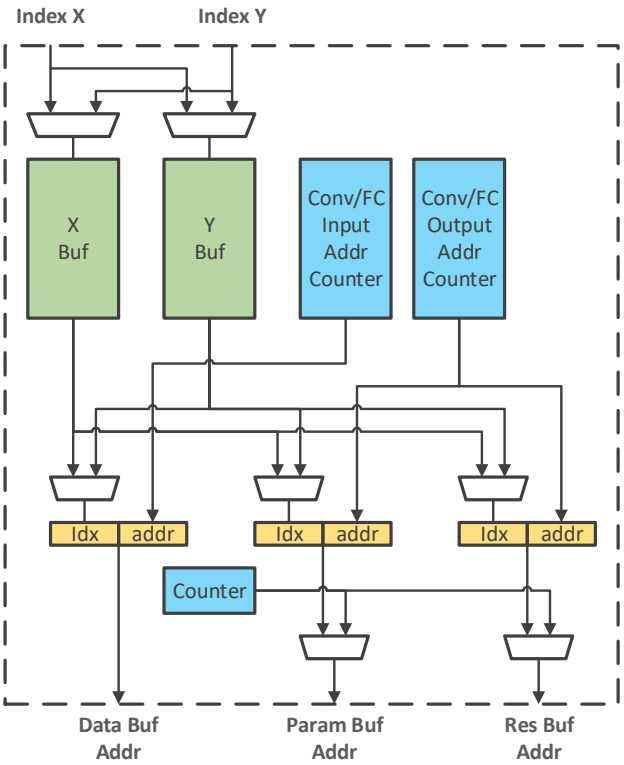

Figure 5: AGU architecture.

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. This dimension is not preferred.

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 unit to process in parallel. Because the network is structured sparse, the number of 2-d convolution kernels for each hardware unit to process may different. Some experiments are carried out to show the parallelism affects the hardware utilization ratio as shown in Figure 7. In Figure 7 (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 7 (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 is usually 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 FF and NG 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

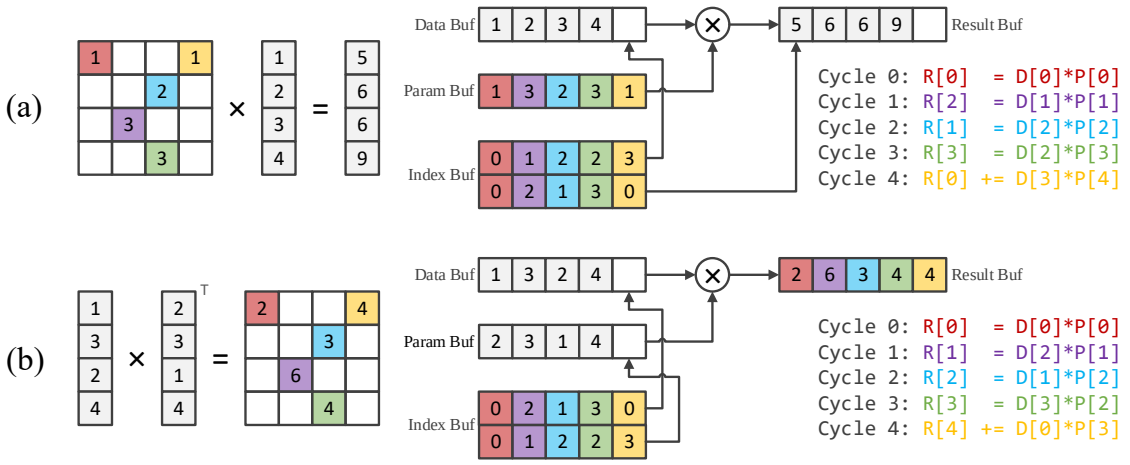

Figure 6: The hardware behavior for sparse network process. (a) Sparse matrix vector multiplication for inference and calculation of gradient of activations. (b) Vector vector multiplication to calculate the gradient of sparse parameters.

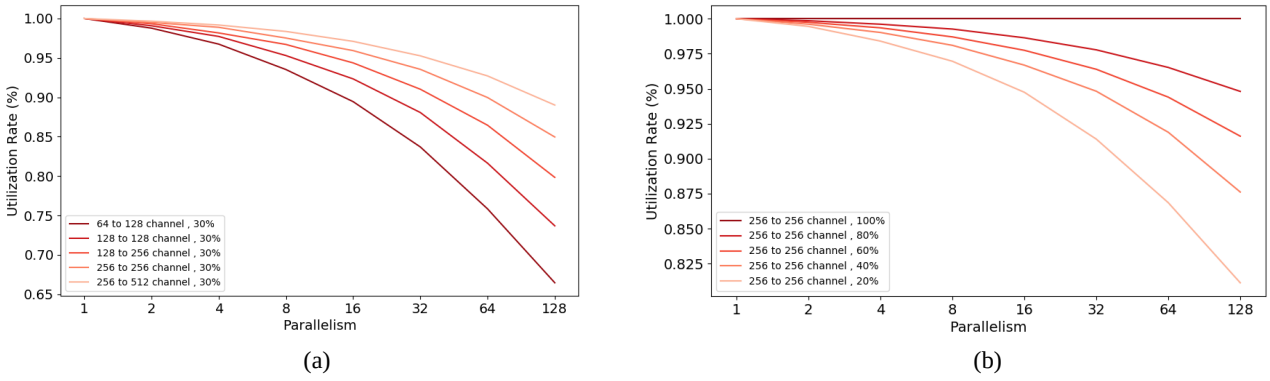

Figure 7: Hardware utilization ratio under different sizes of sparse parameters and sparsity. (a) Estimation with a fixed sparsity and different parameter scales. (b) Estimation with a fixed parameter size and different sparsities.

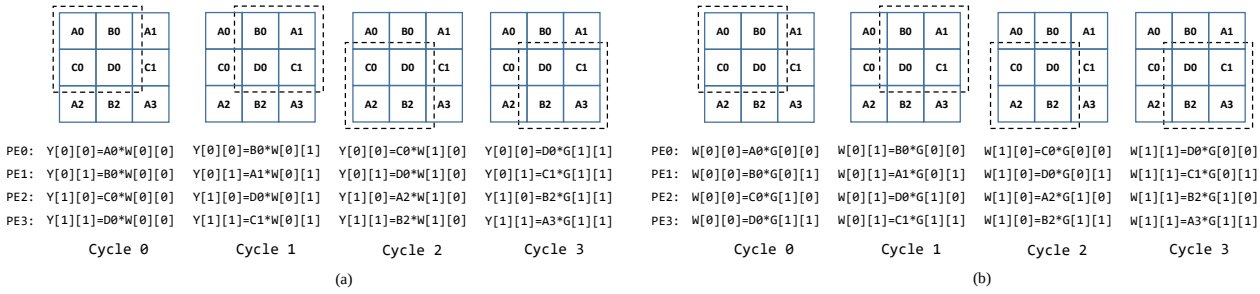

Figure 8: Example of 2×2 convolution on 3×3 feature map with 4 PEs. The letter in each pixel denotes the PE it belongs to. The number denotes the address it stores in the buffer. The feature map needed for each cycle is marked with the dashed line box. (a) convolution in FF step. (b) convolution in WG step.

reduce the channel level parallelism, we explore the possibility of unrolling feature map and convolution kernel dimension in CONV layers. We propose a configurable unroll method between these two dimension.

For FF and NG steps, 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 8(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.

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 8(b). The result are added together when data is written back to external memory.

5.4 Data Organization

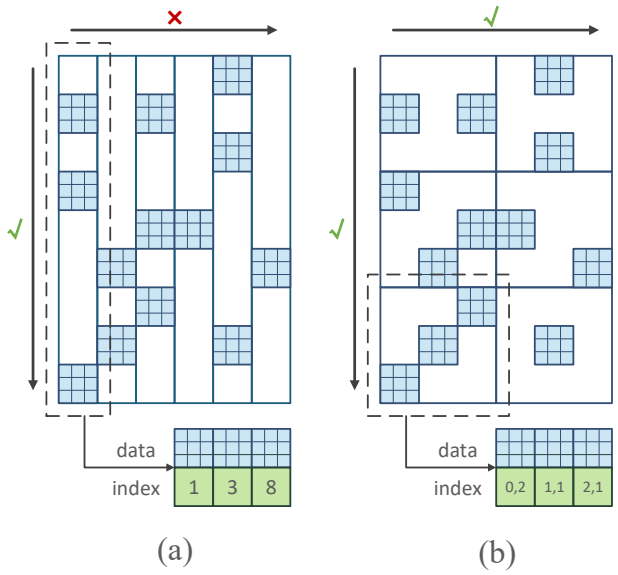

Figure 9: Example of different sparse matrix store formats for structured sparse convolution kernels. (a) Compressed Sparse Column(CSC) (b) Compressed Sparse Block(CSB)

First, we use the compressed sparsed block (CSB) data format 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 it as a 2-D matrix of 2-D convolution kernels. The commonly adopted CSC format, as shown in Figure 9(a) stores data column by column, thus leads to the difficulty in row major access. So we adopt the CSB format [**?**] where the elements are encoded 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 untill 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 FF and WG steps, this is the common feature maps. In NG step, this is the common feature map gradient.
- **Local Load**: Load local data to a PE or a group of PE. In FF and NG steps, this is the network weights and index. In WG step, 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 10(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 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 section 4, we have shown the model accuracy and epoch consumption of the proposed hardware friendly training process. In this section, we show the hardware experimental results. A prototype design is implemented on a Xilinx XCKU115 chip with 2 on-board DDR4 SDRAM. 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 MSB for computation and 24-bit LSB 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 200MHz. Each 2×2 PEs are grouped together for CONV layers.

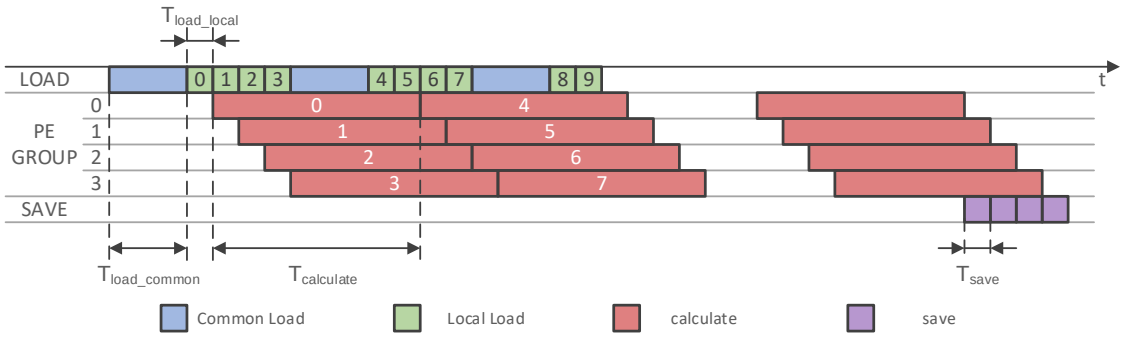

Figure 10: Example scheduling time line. The number for load local and calculate denote the data dependency

Table 4: Prototype design resource utilization

Resource	LUT	Reg	Block RAM	DSP
Available	663360	1326720	2160	5520
Utilization	193821	144594	1216	1024
Ratio	29%	11%	56%	19%

Table 5: 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			1	1.050	1.078

6.1 Effect of Workload Imbalance

Before introducing the hardware performance, we first analyze the theoretical performance loss brought by workload imbalance in Table 2. As introduced in section 5.3, unroll parameters are limited by loop dimension variety and sparsity. As we cannot group the PEs for fully connected layers, we only compare on the convolutional layers. The normalized speed on the convolutional layers of each configuration is shown in Table 5.

Here, we see that using 32 PEs for this network will suffer from 7% performance loss compared with the ideal case where no workload imbalance occurs. Group each 4 PEs together brings about 5% performance increase.

6.2 Hardware Performance

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 6.

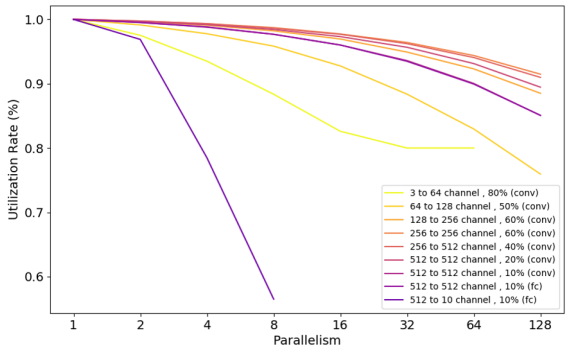

Figure 11: Theoretical utilization ratio for each layer over the number of PEs.

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 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.

A performance and energy efficiency comparison is shown in Table 7. The GPU used for comparison is GTX Titan X GM200. Alough GPU achieves about 2x speed compared with FPGA, the FPGA part only consumes 1/5 power compared with GPU and achieves 3x energy efficiency.

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 6.1, 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.

Table 6: 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		Feed Forward (FF)				Neuron Gradient(NG)				Neuron Gradient(NG)			
	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	-	-

Table 7: Performance and energy efficiency comparison with GPU.

	Data Format data+gradiant	Forward time (us)	Backward time (us)	Total time (us)	Performance (GOP/s)	Power (W)	Power efficiency (GOP/s/W)
GPU	float + float	7795	15700	23495	1252.7	150	8.4
FPGA	8bit + 24bit	10931	34981	45912	641.1	29	22.1

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. Increase the the

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.

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

[3] 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.

[4] 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.

[5] 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).

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

[13] 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.

[14] 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).

[15] 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.

[16] 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.

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

[18] Wei Tang, Gang Hua, and Liang Wang. 2017. How to train a compact binary neural network with high accuracy?. In *AAAI*. 2625–2631.

[19] 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.

[20] 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.

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

[22] 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).