In Proc of the 33rd ACM Int Conf
on Supercomputing (ICS), June 2019

Full-stack Optimization for Accelerating CNNs Using Powers-of-Two Weights with FPGA Validation

Bradley McDanel*
Harvard University
mcdanel@fas.harvard.edu

Sai Qian Zhang*
Harvard University
zhangs@g.harvard.edu

H. T. Kung
Harvard University
kung@harvard.edu

Xin Dong
Harvard University
xindong@g.harvard.edu

ABSTRACT

We present a full-stack optimization framework for accelerating inference of CNNs (Convolutional Neural Networks) and validate the approach with a field-programmable gate array (FPGA) implementation. By jointly optimizing CNN models, computing architectures, and hardware implementations, our full-stack approach achieves unprecedented performance in the trade-off space characterized by inference latency, energy efficiency, hardware utilization, and inference accuracy. An FPGA implementation is used as the validation vehicle for our design, achieving a 2.28ms inference latency for the ImageNet benchmark. Our implementation shines in that it has 9x higher energy efficiency compared to other implementations while achieving comparable latency. A highlight of our approach which contributes to the achieved high energy efficiency is an efficient Selector-Accumulator (SAC) architecture for implementing CNNs with powers-of-two weights. Compared to an FPGA implementation for a traditional 8-bit MAC, SAC substantially reduces required hardware resources (4.85x fewer lookup tables) and power consumption (2.48x).

CCS CONCEPTS

• **Computing methodologies** → **Neural networks**; • **Computer systems organization** → **Systolic arrays**; **Neural networks**; • **Hardware** → **Hardware-software codesign**.

KEYWORDS

systolic arrays; neural networks; sparsity; joint optimization; co-design; powers-of-two weights

ACM Reference Format:
Bradley McDanel, Sai Qian Zhang, H. T. Kung, and Xin Dong. 2019. Full-stack Optimization for Accelerating CNNs Using Powers-of-Two Weights with FPGA Validation. In *2019 International Conference on Supercomputing (ICS '19), June 26–28, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3330345.3330385

*Equal Contribution.

ICS '19, June 26–28, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6079-1/19/06…$15.00
https://doi.org/10.1145/3330345.3330385

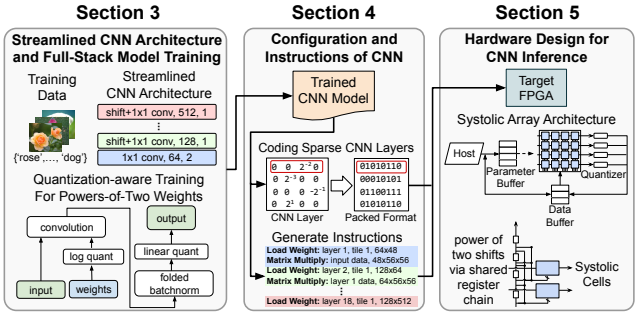

Figure 1: An overview of the proposed full-stack optimization framework for accelerating inference of sparse and quantized CNNs. Section 3 details CNN training which includes sparsity and quantization constraints to match the proposed computing architecture. Section 4 describes the process of converting a trained CNN into a packed representation for systolic array implementation and how FPGA instructions are generated for each convolution layer. Section 5 outlines the proposed architecture with selector-accumulator (SAC) based systolic cells which are used to perform inference for all convolution layers on the FPGA.

1 INTRODUCTION

Due to the widespread success of Convolutional Neural Networks (CNNs) across a variety of domains, there have been extraordinary research and development efforts placed on improving the inference latency, energy efficiency, and accuracy of these networks. Generally, these research efforts can be viewed from two distinct perspectives: (1) machine learning practitioners who focus on reducing the complexity of CNNs through more efficient convolution operations [45], weight and activation quantization [22], and weight pruning [13] and (2) hardware architecture experts who design and build CNN accelerators aiming to minimize power consumption, memory footprint, and I/O cost [8, 23, 43, 48].

However, approaching the problem from only one of these two viewpoints can lead to suboptimal solutions. For instance, as discussed in Section 2.4, many low-precision weight quantization methods may omit significant cost factors in an end-to-end implementation such as using full-precision weights and data for the first layer [1, 50] or using full-precision batch normalization [21]. On the hardware side, most CNN accelerators are designed to support some target CNNs (*e.g.,* AlexNet [26] and VGG-16 [39]) at 8-bit or 16-bit precision for weights and data [7, 12]. Therefore, these accelerators generally are not directly applicable to many of the recent advances

in CNN design including efficient CNN structures (using, *e.g.,* separable filters [16]), weight pruning (using, *e.g.,* Lasso [40]), and low-precision quantization.

To address this disparity, in this paper we propose a *full-stack optimization* framework, where the design of the CNN model is *jointly* optimized (co-designed) with the computing architectures and circuit implementations on which it will run. Figure 1 provides an overview of the proposed method in three stages, covered in three sections. Section 3 describes the training stage, which uses a hardware-aware quantization graph to facilitate training a CNN. The trained CNN can allow direct implementation of quantized computations on an FPGA without any additional overhead. Section 4 describes the process of generating instructions to perform inference given both the trained CNN and the systolic array of some fixed size implemented on the target FPGA. It also covers how the trained sparse and quantized CNN is coded for efficient use of FPGA memory. Section 5 depicts the bit-serial systolic array architecture which includes the use of multiplication-free sparse systolic cells, based on the Selector-Accumulator (SAC) architecture for the multiplier-accumulation (MAC) operation.

The novel techniques of the paper are as follows:

- A **full-stack optimization framework** where the hardware architecture informs the CNN structure in training.
- **Selector-accumulator** (SAC) which provides efficient multiplication-free inference for CNNs trained with powers-of-two weights. We replace traditional 8-bit MAC hardware with inexpensive SAC facilitated by simple shared register chains (Section 5.2).
- A **systolic array building block** for low-precision CNNs (Section 5.1) which uses shared register chains for two purposes: propagating data into adjacent systolic cells and performing multiplication with powers-of-two weights. Systolic arrays are used as an example of processor arrays (our register chain design may extend to other parallel processing array architectures).
- A **streamlined CNN structure** which achieves competitive performance on ImageNet in the mobile setting using only 1×1 convolution without residual connections (Section 3.1).
- **Input reshaping** which decreases the spatial resolution of the input image while increasing the number of channels (Section 3.3). This significantly increases the systolic array utilization for the first convolution layer which is a significant portion of the total runtime.

Leveraging all these advances into a single system is challenging and one of the main accomplishments of this work. We have built an efficient CNN inference engine on an FPGA (Xilinx VC707 evaluation board) and have validated its correctness by checking the output against our software output. All the timing and power consumption results reported in this paper are based on the actual measurements obtained from this FPGA implementation. Our FPGA design is composed almost entirely of lookup tables (LUTs). We use DSPs only to implement the final fully connected layer.

We believe our design provides a useful base for future ASIC implementation. Our CNN training code (using PyTorch [35]), python code which converts a trained sparse and quantized CNN into a packed representation for the FPGA, and Verilog code for FPGA implementation are available at https://github.com/BradMcDanel/multiplication-free-dnn.

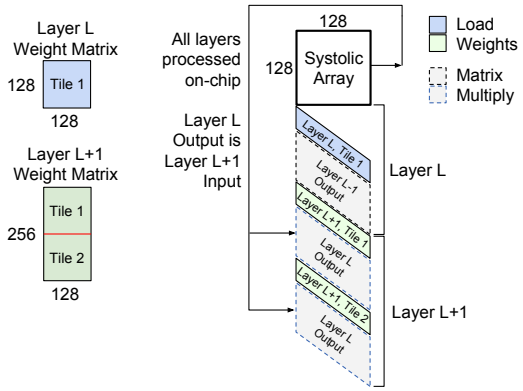

Figure 2: CNN inference for two consecutive layers (layer L and L + 1) using a single 128×128 systolic array similar to the implementation in this paper. The systolic array alternatively executes load weight and matrix multiply instructions for all tiles in a layer (six instructions in total for this example; see Section 4.2).

2 BACKGROUND AND RELATED WORK

In this section, we first describe CNN inference using systolic arrays and summarize recent FPGA-based CNN accelerators which we compare against in Section 6. Then, we review advances in efficient CNNs which are used as a starting point for our approach.

2.1 Systolic Arrays

A systolic array [27] is a collection of interconnected systolic cells. Generally, each systolic cell in a systolic array is hard-coded to perform the same simple arithmetic operation (*e.g.,* a MAC operation). Additionally, communication (*i.e.,* dataflow) in the systolic array occurs only between neighboring cells. These properties make systolic arrays appealing for computational problems which rely heavily on a single type of arithmetic operation such as MAC in CNN inference.

Figure 2 depicts how a systolic array implemented on an FPGA performs CNN inference by reusing the array across the CNN layers. Note that for systolic array synchronization, items input to and output from the array are properly skewed, as shown in the figure. When a layer has more filters than the systolic array can handle, we partition the layer into *vertical tiles* across filters, as shown on the left of the figure, and reuse the systolic array across these tiles. When a layer has more channels than the systolic array can handle, we partition the layer into *horizontal tiles* across channels (these horizontal tiles are not shown in the figure).

2.2 FPGA Accelerators for CNNs

In recent years, numerous FPGA designs for CNN inference have been proposed (generally targeting prominent networks such as LeNet-5 [29], AlexNet [26], and VGG-16 [39]) with the key objectives of low latency and high energy efficiency. A common strategy

deployed by these designs is to minimize the degree of weight and data movement, especially from off-chip memory, as they add significant overhead in terms of both latency and power consumption.

One approach for minimizing data movement is layer fusion, where multiple CNN layers are processed at the same time in a pipelined manner to allow for instant use of intermediate data without external memory accesses [20, 30, 46]. Another approach, used for 3×3 or larger convolutional filters, is determining the order of inference computation which minimizes the number of partial sums that must be stored [33, 49]. Since our CNN architecture (Section 3) uses only 1×1 filters, convolution is reduced to matrix multiplication, which can be efficiently implemented using systolic arrays. Additionally, different computation strategies are often taken for the first layer [47], as it has only three input channels in the case of RGB images and final fully connected layer [36], where there are significantly more weights than data. In this work, we propose to use the same systolic array building block for efficient implementations of all convolution layers in a CNN.

2.3 Efficient CNN Structures

Since VGG-16 [39] was introduced in 2014, there has been a general trend towards designing deeper CNNs through the use of residual connections (ResNets[14]) and concatenative connections (DenseNet [19]) as deeper networks tend to achieve higher classification accuracy for benchmark datasets such as ImageNet [6]. However, as pointed out in Table 2 of the original ResNet paper [14], residual connections appear to add little improvement in classification accuracy of a shallower (18 layer) CNN. Based on these observations, we have chosen to use a shallower CNN (19 layers) without any residual or concatenative connections, which we outline in Section 3.1. In our evaluation (Section 6.5.3) we show that for this shallower CNN, the exclusion of additional connections has minimal impact on classification accuracy while significantly simplifying our hardware implementation and improving its efficiency.

Additionally, several alternatives to standard convolution have been recently proposed to reduce the computation cost. Depthwise separable convolution [3] dramatically reduces the number weights and operations by separating a standard convolution layer into two smaller layers: a depthwise layer that only utilize neighboring pixels within each input channel and a pointwise layer which operates across all channels but does not use neighboring pixels within a channel (*i.e.,* it only uses 1×1 filters). Wu *et al.* [45] showed that a channel shift operation can be used to replace the depthwise layer without a 4significant impact on classification accuracy. As described in Section 3.1, our proposed CNN use this channel shift operation immediately preceding a 1×1 convolution layer. Note that the training paradigm outlined in Section 3 is not specific to convolution with shift and can also be applied to more conventional networks (*e.g.,* AlexNet, VGG, and ResNet).

2.4 Weight and Data Quantization

Several methods have been proposed to quantize the CNN weights after training, using 16-bit [12] and 8-bit [7] fixed-point representations, without dramatically impacting classification accuracy. More recently, low-precision quantization methods (*i.e.,* 1-4 bits) such as binary [4, 17] and ternary quantization [42, 50, 52] methods have also been studied, which may incur some loss in classification accuracy compared to higher precision approaches. Generally, for these low-precision methods, training is still performed using full-precision weights, but the training graph is modified to include quantization operations which match the fixed-point arithmetic used at inference. In this paper, log quantization [50] is adopted for weights, with each quantization point being a power of two. This allows for significantly more efficient inference, as fixed-point multiplication is replaced with bit shift operations corresponding the powers-of-two weight, as discussed in Section 5.

In addition to weight quantization, we may also quantize activated data output from each CNN layer [1, 2, 37, 50, 51]. Data quantization reduces not only the cost of MAC operations but also the cost of memory access for these intermediate output between layers in a CNN during inference. However, it has been shown that low-precision quantization of activation (*i.e.,* 1-4 bits) appears to lead to a significant degradation in classification accuracy compared to weight quantization [5, 31]. Due to these considerations, we use 8-bit linear quantization for data in this paper and focus on an efficient implementation of powers-of-two weight multiplications with 8-bit data.

Additionally, we note that many of the proposed methods for low precision weights and data omit details which are important for efficient end-to-end system performance. First, work in this area often treats the first layer in a special manner by keeping the weights and data full-precision to mitigate a potential drop in classification accuracy [1, 5, 31]. Second, they often explicitly omit quantization considerations of batch normalization and use standard full-precision computation as performed during training [1, 51]. Since batch normalization is essential to the convergence of low-precision CNNs, this omission makes it difficult to efficiently implement many low-precision approaches as floating-point units are required for batch normalization. In this work, as discussed in Section 3.2, we handle both of these issues by (1) quantizing the weights and data in all layers (including the first layer) under a single quantization scheme and by (2) including batch normalization quantization in the training graph (depicted in Figure 6) so that it adds zero overhead during inference.

2.5 Weight Pruning

It is well known in the literature that the majority of weights in a CNN (up to 90% for large models such as VGG-16) can be set to zero (pruned) without having a significant impact on the classification accuracy [13]. The resulting pruned network may have sparsely distributed weights with an irregular sparsity structure, which is generally difficult to implement efficiently using conventional hardware such as GPUs. Subsequent methods have proposed structured pruning techniques which result in models with structured sparsity [11, 15, 18, 32, 34, 44]. While these methods allow more efficient CPU and GPU implementations, they appear unable to achieve the same level of reduction in model size that unstructured pruning can achieve.

Column combining is a pruning method which allows for sparse CNN layers, but enforces that the remaining sparse weights can be packed into a denser format when deployed in a systolic array [28].

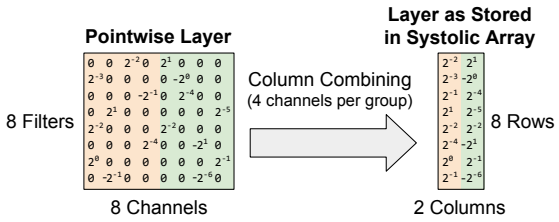

Figure 3: A pointwise convolution layer (left) with four channels per group resulting from pruning for column combining [28]. After combining columns in the filter matrix (left), each group of four channels (shown in cream and green) are reduced into a single column (right). Note that during column combing, for each group, all entries in each row are removed (pruned) but one with the largest magnitude.

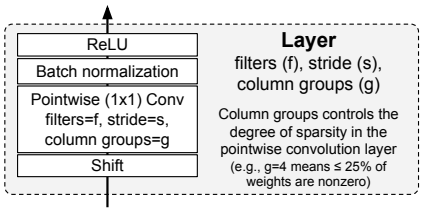

Figure 4: Each layer of the evaluation CNN models in this paper consists of a shift operation [45], pointwise (1x1) convolution, batch normalization and ReLU activation. A layer is parameterized with a number of filters (f), a stride (s), and column groups (g) for column combining in packing a sparse convolutional layer [28].

In our proposed training pipeline, we use column combining in addition to weight and data quantization as discussed in the previous section, in order to achieve efficient sparse CNN inference. Figure 3 shows how a sparse pointwise convolution layer with powers-of-two weights is converted into a denser format by removing all but the largest nonzero entry in each row across the combined channels when stored in a systolic array. In this example, column combining reduces the width of the small layer by factor of 4× from 8 to 2. In Section 5, we describe bit-serial design for efficient hardware implementation of this packed format shown on the right side of Figure 3.

3 FULL-STACK MODEL TRAINING

In this section, we first provide an overview of our streamlined CNN structure in Section 3.1, targeted for the FPGA implementation reported in this paper. Then, we outline the various design choices to improve the utilization and efficiency of the FPGA system. Specifically, we employ a quantization-aware training graph, including quantized batch normalization (Section 3.2) and an input reshaping operation to improve the utilization of our systolic array for the first convolution layer (Section 3.3). The methods proposed in Section 3.2 and Section 3.3 are not specifically designed for the evaluation network used in the paper (described in Section 3.1) and can be generally applied to any CNN structure.

3.1 Proposed Streamlined CNN Architecture

Our objective in designing a CNN architecture is to achieve high classification accuracy using a simplified structure across all convolution layers which can be mapped efficiently onto a systolic array. Figure 4 shows the structure of each convolutional layer in our network. To achieve similar performance to standard 3×3 convolution while using only pointwise (1×1) convolution, every layer begins with a channel shift operation as described in [45]. The output of the *shift operation* is then applied to a sparse pointwise convolution layer, followed by batch normalization and rectified linear unit (ReLU). During training, the weights in the pointwise convolution layer are pruned with column combining using the column groups parameter (g) as in [28]. For the earlier convolution layers in a network which have fewer weights, a column group size of 2 is used, which reduces the number of nonzero weights by roughly 50%. For the latter CNN layers, which are larger and have higher redundancy, a group size of 8 is used (an 87.5% reduction). Each layer is progressively pruned over the course of training, such that after training they will reach their target sparsity set by the column groups for the layer.

Figure 5 shows the evaluation models for the proposed streamlined CNN structure for the CIFAR-10 [25] and ImageNet [6] datasets. As discussed in Section 2.3, we have chosen to keep the network relatively shallow (19 layers) and without any residual or concatenative connections. In Section 6, we show that this streamlined structure can achieve competitive Top-1 ImageNet classification accuracy with low latency and high energy efficiency. We evaluate ImageNet under three network settings: ImageNet-Small/224, ImageNet-Small/56, and ImageNet-Large/56, where 224 and 56 refer to the width and height of the input image after the prepossessing stage. The small models have 1.5M weights and the large model has 8.5M weights after training. These evaluation model were chosen to evaluate the importance of model size and the spatial input size on classification accuracy, latency, and throughput. Additionally, as described in Section 3.3, for the settings with (56×56) input size, we use an *reshaping operation* to increase the number of input channels from 3 (for RGB images) to 48 for increased systolic array utilization.

3.2 Quantization-aware Training

In order to achieve high classification accuracy using powers-of-two weights, we add quantization operations to the CNN training graph to match the fixed-point weights and data used at inference. Figure 6 shows the training and inference graphs for a single layer in the CNN shown in Figure 4. As discussed in Section 2.4, this approach of injecting quantization into the training graph is known in the literature and has mainly been used to train binary and ternary networks [4, 51]. In our training graph, we use log quantization for the weights, which quantizes an underlying full-precision weight (shown in blue) to the nearest power of two. During training, backpropagation uses full-precision gradients to update the full-precision weights in each layer as in [4].

Additionally, we perform quantization on the batch normalization operations which follow each convolutional layer. Generally,

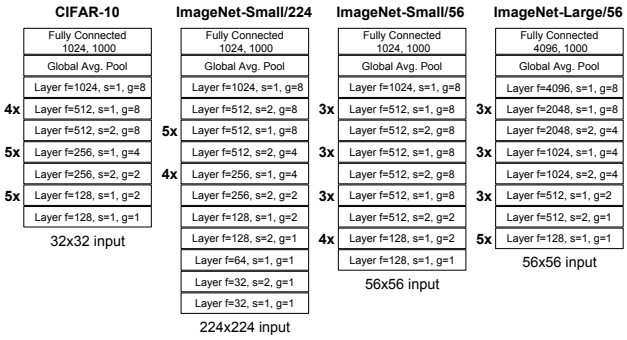

Figure 5: The evaluation models for the CIFAR-10 and ImageNet datasets. Each network consists of 19 layers of trainable weights, where each layer has a structure shown in Figure 4, with the first layer not including a shift component. Downsampling is performed using strided convolution denoted by layers with a stride of 2 (s=2).

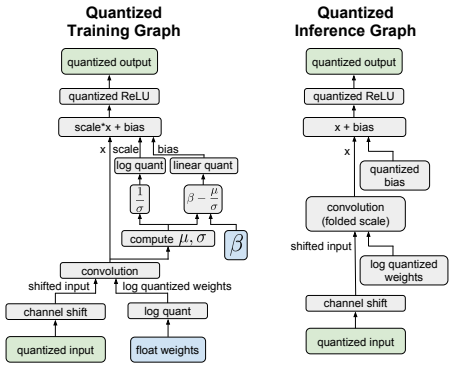

Figure 6: The quantized training graph (left) performs both linear quantization for input data and powers-of-two quantization (log quantization) for weight and batch normalization parameters during training. The inference graph (right) uses the quantized version of the full-precision weights learned during training and therefore does not require any floating-point operations.

for higher precision weights (*e.g.,* 8-bit weights), these batch normalization parameters can be folded directly into the weights and bias terms of the preceding convolution layer after training, so that they have no additional overhead [24]. However, for lower precision weights (such as binary or powers-of-two weights), this folding process introduces significant quantization error, leading to a notable drop in classification accuracy. For this reason, prior works using low-precision weights employ full-precision batch normalization, incurring the corresponding full-precision computation cost. For our proposed bit-serial architecture, these full-precision batch normalization operations would introduce significant overhead and break our objective of multiplication-free inference. Therefore, as shown in Figure 6, we include quantization of the batch normalization parameters in our training graph. Applying log quantization on the batch normalization scale parameters allows them to be folded into the log quantized weights without introducing any quantization error.

Batch normalization is defined as

$$x_{bn} = \gamma\left(\frac{x-\mu}{\sigma}\right) + \beta \tag{1}$$

where μ and σ are the mean and standard deviation of each mini batch during training and average running statistics during inference. γ and β are learnable parameters which are introduced to improve the representation power of the network. When followed by ReLU, as is the case in our CNN, the effects of the learnable scale parameter γ can be captured in the following convolution layer and can therefore be omitted by setting γ as 1 [10]. We then factor μ, σ, and β into a scale and bias term as follows

$$x_{bn} = \underbrace{\frac{1}{\sigma}}_{\text{scale}} x + \underbrace{\beta - \frac{\mu}{\sigma}}_{\text{bias}} \tag{2}$$

After applying quantized batch normalization to the output from the preceding convolution layer, a non-linear activation function (ReLU) is performed, which sets all negative values to 0. Additionally, it applies 8-bit linear quantization on the data so that it matches the fixed-point computation at inference. The inference graph of Figure 6 shows how computation is performed on the FPGA during inference. The log quantized batch normalization scale factor is folded into the log quantized weights in the preceding convolution layer. Only channel shift and fixed-point addition operations are performed during inference.

3.3 Input Reshaping to Improve Utilization

For CNNs trained on ImageNet, the first convolution layer represents 10-15% of the total inference computation performed due to the large spatial size of the input image (3×224×224). However, as recently discussed by Xilinx [47], the computation in this layer does not map well onto a systolic array, because the input image has only three input channels, meaning that the majority of the array's input bandwidth may not be utilized. To address this imbalance, Xilinx proposes to use two systolic arrays, one systolic array specifically designated to the first layer and the other systolic array used for the remaining convolution layers in the CNN.

In this paper, rather than using a different systolic array for the input layer, we reshape the input image to the CNN to increase the utilization of our single systolic array for the first convolutional layer. Figure 7 shows the input reshaping operation for an RGB image with 3 channels and 224×224 pixels per channel. Each 2×2 block of pixels is divided into four groups (denoted 1, 2, 3, and 4) with 1 pixel per group. The pixels in the same group across all 2×2 blocks are then placed into a new set of RGB channels (4 groups, each with RGB channels, leading to 12 channels total). Each of these channels has 112×112 pixels, which is one quarter of the original input image. In Section 6, we evaluate the ImageNet-Small/56 and ImageNet-Large/56 networks with an even more aggressive reshaping operation, where we use 16 groups to convert the 3×224×224 input image into a 48×56×56 input for the CNN.

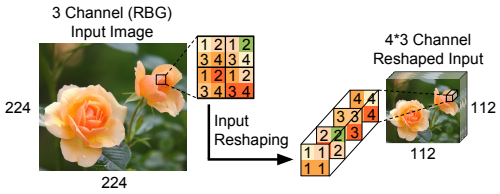

Figure 7: Reshaping the input data by decreasing the spatial size and increasing the number of channels in order to improve utilization of the systolic array in processing the first layer.

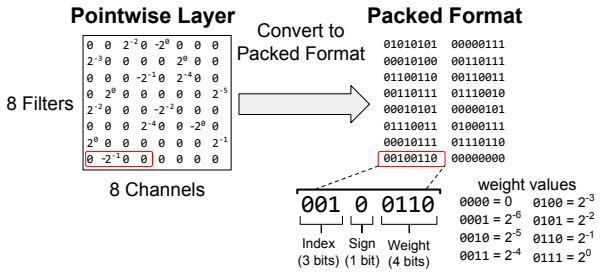

Figure 8: A sparse pointwise layer with powers-of-two weights (left) is converted into a packed representation for efficient storage on an FPGA (right), where each group of combined channels (4 in this example) produces 1 8-bit encoding per filter (row).

4 CONFIGURATION AND INSTRUCTIONS

In this section, we show how our trained CNN described in Section 3 is coded for efficient use on an FPGA (Section 4.1). We then explain how the weights in each layer are divided into tiles which fit in a given systolic array and the corresponding instructions for each tile which run on the FPGA (Section 4.2).

4.1 Coding Sparse CNN Layers for FPGA

After training is complete, each convolution layer will have reached a target sparsity set by the column group parameter for the layer as described in Section 3.1. The left side of Figure 8 illustrates the weights of a pointwise convolution layer after training with 8 filters, 8 channels, and column groups of size 4. For this layer, each group of 4 channels will be combined into a single column in the systolic array on the FPGA, as there is at most one nonzero entry per filter (row) in each group. The remaining nonzero weights are powers of two due to the weight quantization scheme discussed in Section 3.2.

To illustrate the coding procedure, we have outlined 4 weights in red in the pointwise layer shown in Figure 8 which will be converted into an 8-bit representation in the packed format. The second element in this group is the nonzero weight -2^{-1}. The first 3 bits in the encoding store the index of the nonzero weight which is 1 (`001`) corresponding to the second element in the group. Note that for larger layers, we combine up to 8 channels, requiring a 3-bit index. The remaining 5 bits are composed of a sign bit and 4 bits to indicate the power-of-two weight, which is `00110` for -2^{-1}. As depicted in Figure 8, the representable powers-of-two weights are ordered from smallest to largest (*e.g.*, 2^{-6} is `0001` and 2^{0} is `0111`), with `0000` being used to represent 0. In summary, to configure each systolic cell, only 8 bits are required.

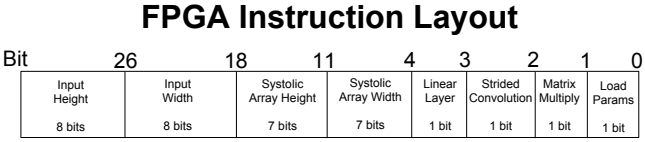

Figure 9: The FPGA instruction layout for a 128×64 systolic array on an FPGA.

4.2 Instructions for FPGA

CNN inference is composed of a series of matrix-matrix multiplications, one per convolution layer, between the data which is input to a layer and the learned weights of a layer. When using a single systolic array to perform the matrix multiplications for all layers, generated instructions will carry out a relatively straightforward process of alternatively loading weights into the systolic array and performing matrix multiplication between the data and loaded weights, in sequential order of the CNN layers. However, when a weight matrix is larger than the fixed size of the systolic array, it must be partitioned into smaller *tiles*, where each tile can fit into the systolic array. Then, a pair of weight loading and matrix multiplication instructions are scheduled for each tile. In this paper, we use column combining to dramatically reduce the number of columns for inference (*e.g.*, from 512 channels to 64 columns in the systolic array via 8-way combining) decreasing the total number of tiles.

Figure 2 shows how inference is performed across two layers (layer L and layer L + 1) using a single systolic array. First, a load weights instruction is used to load the 128 filters by 128 channels weight matrix for layer L. Then, matrix multiplication is performed between the loaded weights and the previous layer output (layer L - 1) by passing the data into the systolic array. This matrix multiplication generates the layer L output which will be used for layer L + 1. Since the layer L + 1 weight matrix of 256×128 is larger than the systolic array of 128×128, it is partitioned into two tiles. The first tile in layer L + 1 is then loaded into the systolic array and is multiplied with the layer L output, which generates half the output for layer L + 1. The second tile in layer L + 1 is then processed in the same manner as the first tile. A total of six instructions are used in total, one pair of weight load and matrix multiply instructions for layer L and two pairs of instructions for layer L + 1.

Figure 9 shows the FPGA instruction layout for the systolic array architecture described in Section 5. A load weight instruction is indicated when the first bit is set, with the systolic array width and height fields controlling the size of the tile being loaded into the array. A matrix multiply instruction is indicated when the second bit is set. The height of the data matrix to be multiplied with the loaded weights is set by the input width and height fields (*e.g.*, 56×56 for the first layer).

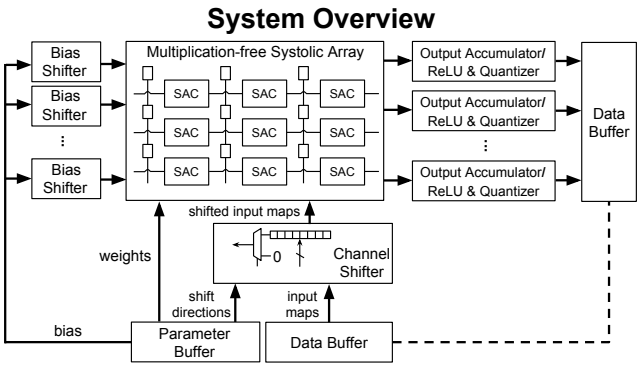

Figure 10: System design as implemented on an FPGA.

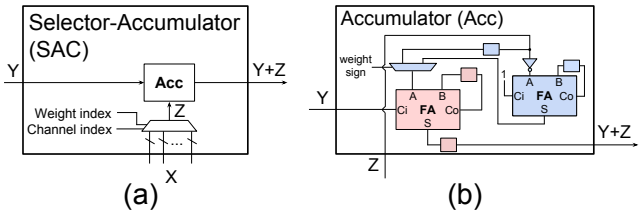

Figure 11: Bit-serial Selector-Accumulator (SAC).

Table 1: Comparison of FPGA resources and power for a 64×64 systolic array implemented with MAC and SAC.

	64×64 MAC	64×64 SAC	MAC / SAC
LUT	212388	43776	4.85×
FF	192293	54330	3.54×
Power	4.21W	1.7W	2.48×

5 FPGA DESIGN

In this section, we provide a detailed description of our FPGA design for sparse CNN inference with powers-of-two weights.

5.1 System Description

Figure 10 shows an overview of the CNN inference system as implemented on an FPGA. The parameter buffer stores the filter weights, biases, and shift directions for the channel shift operation [45]. During a load weight instruction, filter weights are loaded into the systolic array (Section 5.2) and filter bias and channel shift directions are loaded into the bias and the channel shifters, respectively. During a matrix multiplication instruction, input data is loaded from the data buffer into the channel shifters, which perform the shift operations before sending the data to the systolic array in a bit-serial fashion [28]. Each column in the systolic array takes input data from multiple input channels to support column combining shown in Figure 3. Each Selector-Accumulator (SAC) cell (Figure 11) within a column of the systolic array takes in the multiple input channels at different power of two shift offsets to select both the channel index and powers-of-two weight index for the cell corresponding to the packed format in Figure 8.

The output from each row of the systolic array is passed to the ReLU & Quantization block (Section 5.4) before storing the results back to the data buffer. Output data stored in the data buffer for the previous layer is the input data to the next layer. The output accumulator (Section 5.5) is used only in the final (fully connected) layer to reduce the feature map for each class to a single number used for prediction. The parameters for the next weight tile are loaded the off-chip DRAM (not shown in Figure 10) to the parameter buffer as matrix multiplication is performed on the systolic array for the current tile. During inference, all intermediate results are stored in on-chip RAM in the data buffer.

5.2 Selector-Accumulator (SAC) for Multiplication-free Systolic Array Design

In this section, we describe our Selector-Accumulator (SAC) design for a multiplication-free systolic array for sparse matrix multiplication. We choose a bit-serial design for efficient multiplexing of multiple data streams into a single column of the array to support column combining [28]. In the layout of the multiplication-free systolic array (shown in Figure 10), each column in the array takes up to eight input channels (to support column combining) into the register chain for the column in a bit-serial fashion. Each cycle, input data is shifted up to the next register in the chain. This register chain serves the standard purpose in a systolic array of propagating data through the cells in the column. However, when using powers-of-two weights, it can serve an additional purpose of powers-of-two weight multiplication, as each increasing position in the register chain corresponds to the input data being multiplied by an increasing power of two. In our design, we utilize this observation of the dual purpose nature of the register chain when using powers-of-two weights to design a more efficient systolic cell.

Figure 11a shows the *selector-accumulator* (SAC) cells which takes the input data from multiple points on the register chain and selects the point corresponding to the powers-of-two weight stored in cell using a multiplexer. Additionally, it uses a channel index, also stored in the cell, to determine the position of the weight in the original sparse filter matrix (see Figure 8 for details on the indexing scheme). The selected element is then passed into the bit-serial accumulator shown in Figure 11b. The blue logic elements in the accumulator negate the product Y based on the sign of the powers-of-two weight and add the result to the bit-serial accumulator (pink full-adder) before passing the result to the SAC to the right.

Compared with a 8-bit multiplier-accumulator (MAC) which requires 8 1-bit full adders for multiplication, the SAC requires only a multiplexer to select an offset corresponding to a powers-of-two weight. Using Xilinx Vivado design suite, we observe that compared to a traditional 8-bit MAC, SAC substantially reduces required LUTs (4.85×), FFs (3.54×), and power consumption (2.48×), as shown in Table 1. As we discuss in Section 6.2, this dramatically reduces the hardware cost of each systolic cell and allows for substantially larger systolic array to be implemented on the FPGA than with standard 8-bit MAC cells.

Figure 12 shows how a register chain is used in generating a shifted version of the input data 10010 (red) with the shift amount corresponding to the powers-of-two weight associated with the cell over time steps ($T = 0, 1, 2, \ldots$). As depicted in Figure 12 (a), (b) and (c), suppose that the SAC requires a shifted version of the original

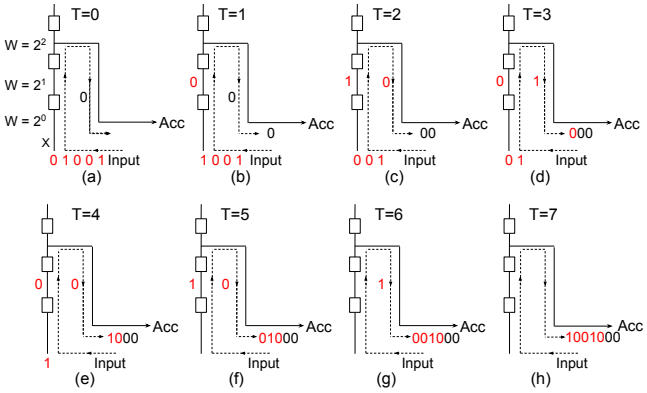

Figure 12: An example of sending a shifted version of input stream (corresponding to a multiplication with a powers-of-two weight) to a SAC cell in a bit-serial fashion. In this example, the weight is 2^2.

input with two pending zeros in the beginning (filter weight is four). Then, the Accumulator (Acc) will grab the input data stream at the second register in the register chain, so the first two bits sent to the Acc are zeros (black). After 4 additional cycles, the Acc receives an input of 1001000, which is four times of the original input 10010.

Figure 13 shows how the register chain can be shared across two consecutive SAC cells in one column of systolic array. Suppose each of two SAC cells may require any one of the three shifted versions of the original input (corresponding to three possible powers-of-two weights). Then, this leads to use of two windows with span of three on the register chain (shown in green and blue in Figure 13). The red lines in the figures show the positions where the SAC cells grab the shifted versions of the original input from the register chain. Thus, the register chain is used for two purposes: (1) shifts the input data upwards to all the SAC cells in the same column and (2) generates the shifted versions of the input data for the powers-of-two multiplication.

5.3 Energy-efficient SAC with Zero-Skipping

Each SAC can be turned off to save power when either the weight or the input to the SAC is zero, which we call zero-skipping. The structure of a SAC with and without the zero-skipping mechanism are shown in Figure 14. For the SAC with zero-skipping, the zero signal is set when either the input or weight is 0 and is used as the enable signal for the gated clock. When the zero signal is set due to the current input being 0, the accumulation Y_i bypasses the SAC and is forwarded directly to the next SAC on the row in the systolic array. Note that due to ReLU, approximately half of the data elements are zero, meaning that the SAC will be disabled roughly half of the time. When the weight for the SAC is 0, then the SAC will be disabled for the entire matrix multiplication. In Section 6.3, we show that this zero-skipping mechanism reduces power by roughly 30%.

5.4 Design of ReLU and Quantization Block

As mentioned in Section 2.4, we use an 8-bit fixed-point representation for the input data to each layer. Therefore, the quantization

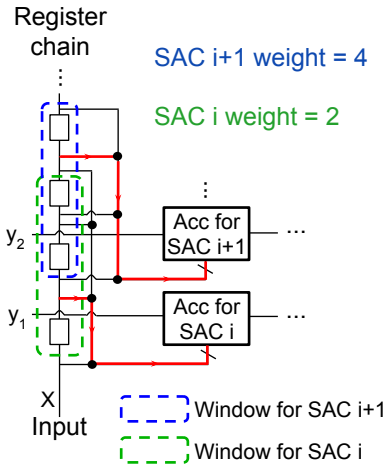

Figure 13: Register chain with per-cell window for powers-of-two weights for two adjacent cells on a column of the systolic array. The red lines show the position where the shifted versions of the original input are grabbed from the register chain.

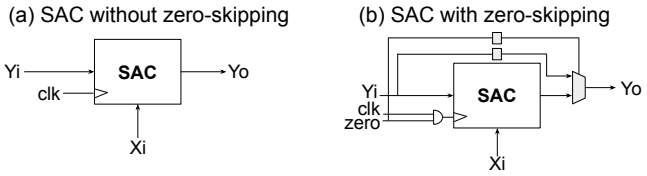

Figure 14: SAC without zero-skipping (a) and with zero-skipping (b).

process must convert the higher precision (32-bit) accumulator output from the systolic array back into an 8-bit range to be used as input to the next layer. In this paper, since the fixed-point scale factor is shared across all layers, this quantization step simplifies to extracting an 8-bit range from the 32-bit accumulator. This quantization step can be fused with the ReLU activation function, by setting negative accumulator outputs to 0.

Figure 15 shows the architecture of the ReLU & Quantization block. A register array is used to collect the 32-bit output from the systolic array. The 32-bit result is shifted by the smallest representable powers-of-two weight (*e.g.*, 2^{-6} as shown in Figure 8) and passed to the comparator. The comparator generates the indicator bit for the multiplexer, which clips the result between (0, 255) before storing it back to the buffer.

5.5 Design of Output Accumulator

Given the output channels produced by the final convolutional layer, average pooling is used to reduce the spatial components of each channel to a single averaged value. For our single systolic array implementation, we fold this pooling operation into the weights of fully connected layer. Let x_i^k and $\bar{x}^k = \frac{\sum_{i=1}^{R} x_i^k}{R}$ denote the i-th element of the input map k and the average of the input channel k,

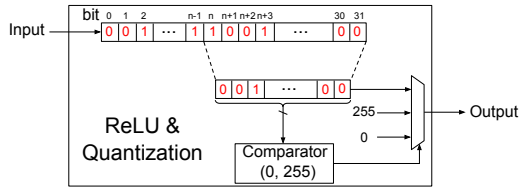

Figure 15: Design of ReLU and Quantization blocks.

where R is total number of elements in each channel. Denote $\vec{x} = \{\bar{x}^1, \bar{x}^2, ..., \bar{x}^M\}$ as the vector of channel averages, where M is the total number of input channels. We have the following derivation for the output $\vec{y}$ of the fully connected layer:

$$\vec{y} = W\vec{x} = W\frac{\sum_{i=1}^{R}\vec{x_i}}{R} = \sum_{i=1}^{R}\frac{W}{R}\vec{x_i} \tag{3}$$

where $\vec{x_i} = \{x_i^1, x_i^2, ..., x_i^M\}$ and W is the weight matrix of the fully connected layer. From equation 3, we notice that $\frac{W}{R}\vec{x_i}$ can be computed by carrying out the matrix multiplication between $\frac{W}{R}$ and $X = \{x_i^k\}$ with the systolic array, and $\vec{y}$ can be computed by summing up all the $\frac{W}{R}\vec{x_i}$.

The output accumulator is used to calculate the sum of $\frac{W}{R}\vec{x_i}$. The 32-bit output stream from a row in the systolic array enters the output accumulator in a bit-serial fashion. This stream is stalled in a register array until the final bit arrives. The resulting 32-bit output is added to the next 32-bit output stream. We use DSPs on the FPGA to carry out these 32-bit additions.

6 EVALUATION

In this section, we first briefly reiterate the key contributions of our proposed approach from the perspectives of both the CNN training and hardware and tie each contribution to the corresponding evaluation section (Section 6.1). Then, we evaluate the performance of our FPGA implementation against state-of-the-art FPGA accelerators on the ImageNet dataset (Section 6.2). Next, we measure the impact of zero skipping on energy efficiency (Section 6.3) for different sized systolic arrays and the impact of folding batch normalization (Section 6.4). Finally, in Section 6.5, we analyze the impact of our streamlined CNN structure and training procedure presented in Section 3 on classification accuracy including input reshaping, using powers-of-two weights, and the omission of residual connections from the CNN structure mentioned in Section 3.1.

We focus on two primary performance metrics: (1) *latency* from image input to classification output, and (2) *energy efficiency*, which is the number of images the inference engine can process per joule. Note that the latter is also the number of images/sec (i.e., throughput) per watt. For high-throughput inference applications we may use multiple inference engines in parallel. If these individual engines each offer low latency and high-energy efficient inference, then the aggregate system will delivery high-throughput inferences per watt while meeting low inference latency requirements.

6.1 Recap of Full-stack Optimization

Full-stack optimization via training has enabled the following design advances which lead to our efficient FPGA implementation presented in Section 5.

- Using powers of two for weights and the batch normalization scale parameters, outlined in Section 2.4, for all convolution layers in the CNN. This allows for a simplified design, where a single sparse multiplication-free systolic array is used for all CNN layers. In Section 6.5.2, we discuss the impact of the proposed quantization scheme on classification accuracy.
- Zero-skipping of the quantized data (Section 5.3). In Section 6.3, we show that zero-skipping reduces the power consumption during matrix multiplication by roughly 30%.
- Packing sparse CNNs using column combining [28] for efficient storage and use on FPGAs, which we describe in Section 4.1. Our ImageNet-Small/56 evaluation model has only 1.5M powers-of-two weights, which is 40× smaller than AlexNet and 92× smaller than VGG-16 (the two CNNs used by other FPGA designs).
- Using channel shifts [45] to replace 3×3 convolutions with 1×1 convolutions. As with column combining, this reduces the number of model parameters. Additionally, it streamlines the design of the systolic array system, as 1×1 reduces to a smaller matrix multiplication, as opposed to 3×3 convolutional filters.
- Input reshaping (Section 3.3) to increase the bit-serial systolic array utilization and dramatically reduce the latency for the first convolution layer. In Section 6.5.1, we show that input reshaping alleviates some of the accuracy loss when using a smaller spatial input size of 48×56×56 instead of the conventional 3×224×224.

6.2 Comparing to Prior FPGA Accelerators

We compare our 170 MHz FPGA design to several state-of-the-art FPGA accelerators on the ImageNet dataset in terms of top-1 classification accuracy, latency for a single input image, and energy efficiency when no batch processing is performed (*i.e.,* batch size of 1). By choosing these metrics, we focus on real-time scenarios where input samples must be processed immediately to meet a hard time constraint. Our evaluation model is the ImageNet-Small/56 network shown in Figure 5 with input reshaped to 48×56×56. Our FPGA can fit a systolic array with 128 rows by 64 columns. Each of the columns can span up to 8 channels in convolution weight matrix, *i.e.,* when the column group parameter is set to 8, for a total of 512 channels.

Table 2 provides a comparison of our FPGA implementation with the other FPGA-based CNN accelerators. Our design achieves a per-image latency of 2.28 ms, which is among the lowest across all the designs. In addition, compared with some of the most recent works [41, 49], our design outperforms them by 5.64× and 3.26×, respectively, in term of energy efficiency. Additionally, compared to an implementation which achieves comparable low latency [30], our implementation has 9.29x higher energy efficiency.

Our design achieves the highest energy efficiency among all these designs for several reasons. First, we use a highly efficient CNN structure (Section 3.1) with only 1.5M weights (compared

Table 2: Comparison with other FPGA-based CNN accelerators.

	[49]	[36]	[46]	[33]	[30]	[38]	[41]	Ours
Xilinx FPGA Chip	VC706	ZC706	ZC706	Arria-10	VC709	Virtex-7	ZC706	VC707
FF	51K(12%)	127k(29%)	96k(22%)	-	262k(30%)	348k(40%)	51k(12%)	201K(33%)
LUT	86k(39%)	182k(83%)	148k(68%)	161k(38%)	273k(63%)	236k(55%)	86k(39%)	239K(78%)
DSP	808(90%)	780(89%)	725(80%)	1518(100%)	2144(59%)	3177(88%)	808(90%)	112(4%)
BRAM	303(56%)	486(86%)	901(82%)	1900(70%)	1913(65%)	1436(49%)	303(56%)	834(81%)
Accuracy (Top-1)	53.30%	64.64%	N/A	N/A	N/A	55.70%	52.60%	50.84%
Frequency (MHz)	200	150	100	150	150	100	200	170
Latency (ms)	5.88	224	17.3	47.97	2.56	11.7	5.84	2.28
Efficiency (img./S/W)	23.6	0.46	6.13	0.98	12.93	8.39	40.7	120.7

Table 3: Power consumption comparison of zero-skipping.

	Without Skipping	With Skipping
32×64	1.0W	0.7W
64×64	1.7W	1.3W
128×64	3.0W	2.2W

to 60M for AlexNet and 136M weights for VGG-16 [33, 36]). Our model in Table 2 is significantly smaller and all weights (including weights in batch normalization layers) are quantized. Our accuracy is 50.84% (about 2% worse than nearest competitive designs [41] in terms of energy efficiency). However, our implementation has at least 3x higher energy efficiency. Second, our proposed powers-of-two quantization (Section 2.4) enables the use of a multiplication-free systolic array (Section 5.1), where each cell contains only a selector and two full adders (see Figure 11) which are more efficient compared with [33] and have simpler structure compared with [38]. This allows for a large systolic array (128×64) to fit on the FPGA, thereby reducing the number of tiles required to perform inference for each sample. Moreover, by using column combining we can pack sparse CNN layers for efficient systolic array implementation with high hardware utilization [28]. Additionally, DSPs are used in the Output Accumulator (Section 5.5) only for a single fully connected layer and are turned off for the rest of the layers. Finally, the zero-skipping mechanism, which we evaluate in more detail in Section 6.3, further saves power by dynamically turning off systolic cells when the data entering a cell is zero.

6.3 Power Reduction by Zero Skipping

In order to evaluate the power reduction due to zero skipping, we measure the power consumption of the FPGA during matrix multiplication under two settings. The "Without Skipping" setting uses inputs which are all nonzero, meaning that every cell will be active during matrix multiplication. The "With Skipping" setting uses inputs which are half zero, in order to approximate the output of ReLU, which sets roughly half of the elements to zero [9].

Table 3 shows the amount of power consumption for inference for the "Without Skipping" and "With Skipping" settings for three systolic arrays of increasing sizes. For all three systolic array sizes, we observe that "With Skipping" reduces the power consumption of matrix multiplication by roughly 30%.

6.4 Power Reduction by Folding Batch Normalization

We further evaluate the savings on hardware and power by folding batch normalization parameters into filter weights. For comparison, we implement a batch normalization block on the FPGA, which performs 8-bits fixed-point MAC operations on the outputs of 128 × 64 systolic array. We then measure the additional hardware and power consumed by the batch normalization block. Introducing the additional block for batch normalization not only consumes additional LUTs and FFs (12653 and 9377 respectively), but also increases the power consumption by 0.6W.

6.5 Impact of Full-stack Training on Accuracy

We now evaluate the impact of the modifications to both the CNN structure and training procedure as proposed in Section 3 on classification accuracy.

6.5.1 Impact of Input Reshaping. In order to determine the effectiveness of the input reshaping operation described in Section 3.3, we compare models using the same spatial input size with and without reshaping (*e.g.,* 3×56×56 versus 48×56×56) and models with different spatial input size (*e.g.,* 3×224×224 versus 48×56×56). Additionally, we train a larger ImageNet model (ImageNet-Large/56) using input reshaping to see the best accuracy that our proposed approach can achieve when used with a small spatial input size.

Table 4 shows the classification accuracy for the four evaluated network settings. First, we observe that the ImageNet-Small/56 with reshaping is able to achieve similar classification accuracy to the ImageNet-Small/224 without reshaping, even with a 16× fewer pixels in each channel. This shows that input reshaping allows for input images with additional channels to negate some of the loss in accuracy due to the smaller spatial input size. Additionally, for the two ImageNet-Small/56 models (with and without reshaping), we see that input reshaping provides a substantial improvement of around 4% accuracy. This is especially interesting considering these two networks have identical structures except for the initial layer (48 channels with input reshaping versus 3 channels without reshaping). Finally, the ImageNet-Large/56 model achieves an impressive 67.57% which is only 2% behind full-precision MobileNet [16] using 224×224 input. This shows that the proposed CNN structure and powers-of-two quantization method can achieve high classification accuracy with reshaped input when using a larger CNN.

Table 4: Evaluating impact of input reshaping.

Model	Input Reshaping	Accuracy (%)
ImageNet-Small/224	No	52.32
ImageNet-Small/56	No	46.92
ImageNet-Small/56	Yes	50.84
ImageNet-Large/56	Yes	67.57

Table 5: Comparing full-precision and powers-of-two weights for the CIFAR-10 and ImageNet-Small/56 models.

	CIFAR-10	ImageNet-Small/56
Full-Precision	95.28	57.16
Powers-of-Two	92.80	50.84

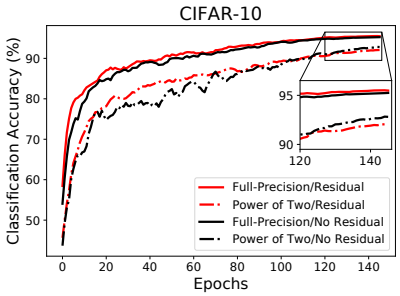

Figure 16: Classification accuracy over 150 epochs CIFAR-10 models trained with/without residual connections and with/without powers of two quantization.

6.5.2 Impact of Powers-of-Two Weight Quantization. While powers-of-two weight quantization allow for an exceedingly efficient implementation, they introduce some loss in classification accuracy when compared against a full-precision version of the same network. Additionally, if these schemes are only evaluated on easier datasets (such as CIFAR-10), the reduction in accuracy can be understated when transition to harder datasets (such as ImageNet). Table 5 shows the classification accuracy for the CIFAR-10 and ImageNet-Small/56 models using full-precision and powers-of-two weights. We see that while the gap between the CIFAR-10 models is only around 2.5%, the gap for ImageNet is closer to 6%. However, as we demonstrate in Section 6.5.1, this reduction in classification accuracy can often be alleviated by increasing the model size.

6.5.3 Impact of Removing Residual Connections. Figure 16 shows the impact of residual connections by evaluating the CIFAR-10 network structure with and without residual connections. In order to ensure that there is not an unseen interaction between the powers-of-two quantization and residual connections, we compare the impact of residual connections on networks with and without quantization. We see that, regardless of quantization, networks trained without residual connections achieve similar performance to the networks trained with residual connections. This shows that residual connections have minor impact on classification accuracy for the 19 layer networks as shown by He *et al.* in the original ResNet paper [14].

7 CONCLUSION

In this paper, we propose using full-stack optimizations for accurate, low-latency and high energy-efficiency CNN inference. We demonstrate that designs ranging from CNN model training at a high level, to those of computing structures and FPGA implementation at a low level can all be optimized simultaneously to ensure they fit one another, thereby achieving high system performance. While cross-layer optimization (co-design) is a known concept in the literature, the system reported in this paper is one of the most comprehensive realizations based on full-stack optimization for the design of deep learning implementations on a chip.

We describe implementation details of various optimization techniques, including (1) channel shifts instead of computationally more expensive 3×3 convolutions, (2) packing sparse CNNs of irregular sparsity structure for efficient implementations on regular processor arrays, (3) quantizing data activations for power-saving with zero-skipping and efficient storage of intermediate data between layers, and (4) use of powers-of-two weights and batch normalization for efficient computation.

Our Selector-Accumulator (SAC) design resulting from full-stack optimization with powers-of-two weights represents an extremely efficient way of implementing MAC by selecting from a shift register rather than performing arithmetic operations. (It seems difficult to have a more efficient MAC design, short of analog implementations!) Given that MAC is the basic operation in the dot-product computation for matching data against filters, we believe our SAC design is significant.

ACKNOWLEDGMENTS

This work is supported in part by the Air Force Research Laboratory under agreement number FA8750-18-1-0112, a gift from MediaTek USA and a Joint Development Project with TSMC.

REFERENCES

[1] Zhaowei Cai, Xiaodong He, Jian Sun, and Nuno Vasconcelos. 2017. Deep learning with low precision by half-wave gaussian quantization. (2017). arXiv:1702.00953
[2] Jungwook Choi, Zhuo Wang, Swagath Venkataramani, Pierce I-Jen Chuang, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. 2018. PACT: Parameterized Clipping Activation for Quantized Neural Networks. (2018). arXiv:1805.06085
[3] François Chollet. 2016. Xception: Deep Learning with Depthwise Separable Convolutions. (2016). arXiv:1610.02357
[4] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*. 3123–3131.
[5] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830* (2016).
[6] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. Imagenet: A large-scale hierarchical image database. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. IEEE, 248–255.
[7] Tim Dettmers. 2015. 8-bit approximations for parallelism in deep learning. *arXiv preprint arXiv:1511.04561* (2015).
[8] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 92–104.
[9] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. 2011. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*. 315–323.
[10] Google. [n. d.]. Functional interface for the batch normalization layer. https://www.tensorflow.org/api_docs/python/tf/layers/batch_normalization. ([n. d.]).

Accessed: 2018-12-04.
[11] Scott Gray, Alec Radford, and Diederik Kingma. 2017. GPU Kernels for Block-Sparse Weights. https://s3-us-west-2.amazonaws.com/openai-assets/blocksparse/blocksparsepaper.pdf. (2017). [Online; accessed 12-January-2018].
[12] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *International Conference on Machine Learning*. 1737–1746.
[13] 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).
[14] 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.
[15] Yihui He, Xiangyu Zhang, and Jian Sun. 2017. Channel pruning for accelerating very deep neural networks. In *International Conference on Computer Vision (ICCV)*, Vol. 2.
[16] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).
[17] Qinghao Hu, Peisong Wang, and Jian Cheng. 2018. From hashing to CNNs: Training BinaryWeight networks via hashing. *arXiv preprint arXiv:1802.02733* (2018).
[18] Gao Huang, Shichen Liu, Laurens van der Maaten, and Kilian Q Weinberger. 2017. CondenseNet: An Efficient DenseNet using Learned Group Convolutions. *arXiv preprint arXiv:1711.09224* (2017).
[19] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks.. In *CVPR*, Vol. 1. 3.
[20] IEEE Press 2016. *Fused-layer CNN accelerators*. IEEE Press.
[21] Sergey Ioffe and Christian Szegedy. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167* (2015).
[22] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. [n. d.]. Quantization and training of neural networks for efficient integer-arithmetic-only inference. ([n. d.]).
[23] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246
[24] Raghuraman Krishnamoorthi. 2018. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342* (2018).
[25] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. 2014. The CIFAR-10 dataset. (2014).
[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*. 1097–1105.
[27] H. T. Kung. 1982. Why systolic architectures? *IEEE Computer* 15 (1982), 37–46. Issue 1.
[28] H. T. Kung, Bradley McDanel, and Sai Qian Zhang. 2019. Packing Sparse Convolutional Neural Networks for Efficient Systolic Array Implementations: Column Combining Under Joint Optimization. *24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2019).
[29] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (1998), 2278–2324.
[30] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*. IEEE, 1–9.
[31] Zechun Liu, Baoyuan Wu, Wenhan Luo, Xin Yang, Wei Liu, and Kwang-Ting Cheng. 2018. Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm. *arXiv preprint arXiv:1808.00278* (2018).
[32] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. 2017. Thinet: A filter level pruning method for deep neural network compression. *arXiv preprint arXiv:1707.06342* (2017).
[33] 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.
[34] Sharan Narang, Eric Undersander, and Gregory F. Diamos. 2017. Block-Sparse Recurrent Neural Networks. *CoRR* abs/1711.02782 (2017). arXiv:1711.02782 http://arxiv.org/abs/1711.02782
[35] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).
[36] 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.
[37] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*. Springer, 525–542.
[38] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*. IEEE, 535–547.
[39] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[40] Robert Tibshirani. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), 267–288.
[41] Junsong Wang, Qiuwen Lou, Xiaofan Zhang, Chao Zhu, Yonghua Lin, and Deming Chen. 2018. Design Flow of Accelerating Hybrid Extremely Low Bit-width Neural Network in Embedded FPGA. *arXiv preprint arXiv:1808.04311* (2018).
[42] Peisong Wang and Jian Cheng. 2017. Fixed-point factorized networks. In *Computer Vision and Pattern Recognition (CVPR), 2017 IEEE Conference on*. IEEE, 3966–3974.
[43] Shihao Wang, Dajiang Zhou, Xushen Han, and Takeshi Yoshimura. 2017. Chain-NN: An energy-efficient 1D chain architecture for accelerating deep convolutional neural networks. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1032–1037.
[44] 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.
[45] Bichen Wu, Alvin Wan, Xiangyu Yue, Peter Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. 2017. Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions. *arXiv preprint arXiv:1711.08141* (2017).
[46] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. 2017. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs. In *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM, 62.
[47] Xilinx. 2018. *Accelerating DNNs with Xilinx Alveo Accelerator Cards*. Technical Report. Xilinx.
[48] 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.
[49] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 56.
[50] Aojun Zhou, Anbang Yao, Yiwen Guo, Lin Xu, and Yurong Chen. 2017. Incremental network quantization: Towards lossless cnns with low-precision weights. *arXiv preprint arXiv:1702.03044* (2017).
[51] 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).
[52] Chenzhuo Zhu, Song Han, Huizi Mao, and William J Dally. 2016. Trained ternary quantization. *arXiv preprint arXiv:1612.01064* (2016).