# SPEC²: SPECtral SParsE CNN Accelerator on FPGAs

Yue Niu[1], Hanqing Zeng[1], Ajitesh Srivastava[1], Kartik Lakhotia[1], Rajgopal Kannan[2], Yanzhi Wang[3], and Viktor Prasanna[1]

[1]University of Southern California, {yueniu,zengh,ajiteshs,klakhoti,prasanna}@usc.edu
[2]US Army Research Lab-West, rajgopal.kannan.civ@mail.mil
[3]Northeastern University, yanz.wang@northeastern.edu

*Abstract*—To accelerate inference of Convolutional Neural Networks (CNNs), various techniques have been proposed to reduce computation redundancy. Converting convolutional layers into frequency domain significantly reduces the computation complexity of the sliding window operations in space domain. On the other hand, weight pruning techniques address the redundancy in model parameters by converting dense convolutional kernels into sparse ones. To obtain high-throughput FPGA implementation, we propose SPEC² – the first work to *prune and accelerate spectral CNNs*. First, we propose a systematic pruning algorithm based on Alternative Direction Method of Multipliers (ADMM). The offline pruning iteratively sets the majority of spectral weights to zero, without using any handcrafted heuristics. Then, we design an optimized pipeline architecture on FPGA that has efficient random access into the sparse kernels and exploits various dimensions of parallelism in convolutional layers. Overall, SPEC² achieves high inference throughput with extremely low computation complexity and negligible accuracy degradation. We demonstrate SPEC² by pruning and implementing LeNet and VGG16 on the Xilinx Virtex platform. After pruning $75\%$ of the spectral weights, SPEC² achieves $0\%$ accuracy loss for LeNet, and $< 1\%$ accuracy loss for VGG16. The resulting accelerators achieve up to $24\times$ higher throughput, compared with the state-of-the-art FPGA implementations for VGG16.

*Index Terms*—CNN accelerator, Model Compression, Parallel Implementation, Spectral Convolution, Alternative Direction Method of Multipliers, Sparse Computation Engine

## I. INTRODUCTION

Convolutional Neural Networks (CNNs) are powerful deep learning models, widely deployed for computer vision tasks [1], [2]. The high accuracy of CNNs comes with the price of high computation cost. For example, VGG16 [2] requires more than 30 billion multiplications and additions to inference a single image. Although light-weight CNN models [3], [4] have been recently proposed to meet the hardware constraints of embedded systems, large models [2], [5] are still irreplaceable in many advanced computer vision tasks. Motivated by the computation challenge, there have been many attempts to speedup CNN inference. From the algorithm perspective, spectral convolution [6] and spatial weight pruning [7] are two successful methods to reduce the number of operations of a CNN. From the architecture perspective, hardware designs have been tailored on FPGAs to fit the computation and communication characteristics of inference.

Although accelerators co-designed with the algorithmic and architectural innovations have shown impressive throughput

and latency performance, challenges still exist in state-of-the-art approaches. On the one hand, while spectral CNN accelerators [8], [6] perform exact inference and achieve $3\times$ to $4\times$ higher throughput than spatial accelerators, they have significantly higher memory requirement due to enlarged spectral kernels. On the other hand, weight pruning on CNNs has been explored to reduce computation and storage requirements simultaneously. However, no spectral domain pruning has been performed in the literature. Further, many pruning approaches (e.g., [7], [9], [10]), depend on heuristics, making the quality of pruning (in terms of accuracy) sensitive to training data. In addition, the irregular sparse structure of the pruned kernels may result in low hardware efficiency when performing the sliding window operation of spatial convolution. Taking the hardware overhead into account, a heavily pruned spatial CNN may not lead to significantly faster inference. Not only in CNN does sparse operations need particular optimization but also in other applications [11], [12].

To overcome the above challenges, we propose SPEC²: a novel approach to speedup CNN inference by applying systematic weight pruning to spectral convolutional layers. Inference of SPEC² has extremely low computation complexity since (i) the spectral transformation reduces computation redundancy, and (ii) the kernel pruning reduces parameter redundancy. Memory requirement of SPEC² is also significantly reduced, since after pruning, we only store on chip the non-zeros of the sparse kernels. Finally, unlike other implementations based on spatial pruning, SPEC² achieves high hardware efficiency. SPEC² replaces the sliding window operation (of spatial convolution) with element-wise multiplications (of spectral convolution), and enables us to develop a sparse tensor computation engine achieving close to $100\%$ DSP utilization. Overall, the systematic pruning algorithm of SPEC² ensures both high accuracy and high pruning rate of spectral CNNs, and the corresponding FPGA design achieves high inference throughput. The main contributions of this paper are:

- We propose SPEC² to achieve high throughput inference of spectral sparse CNNs, by:
  - *Spectral weight pruning formulation* using Alternative Direction Method of Multipliers (ADMM) algorithm. It minimizes classification error while constraining the number of non-zero weights based on target sparsity.

IEEE computer society

- *Spectral retraining* of the model to iteratively fine-tune non-zero spectral weights and improve accuracy.
- *Sparse convolution accelerator*, which implements the pruned spectral convolution on FPGA. The accelerator exploits various dimensions of parallelism, and uses small number of data replications to significantly reduce BRAM conflicts and pipeline stalls.
- We extensively evaluate $\text{SPEC}^2$ on the below metrics:
  - *Inference accuracy*: We show that pruning 75% weights of the spectral kernels leads to negligible accuracy loss (0% for LeNet on MNIST dataset and $< 1\%$ for VGG16 on CIFAR-10 dataset).
  - *Inference throughput*: We implement the accelerator on Xilinx Virtex-7 XC7VX690T FPGA. The design achieves 99% of the theoretically optimal throughput performance and $24\times$ higher throughput than state-of-the-art.

## II. BACKGROUND AND RELATED WORK

### A. Spectral CNNs

The main building blocks of a CNN are convolutional layers. A convolutional layer convolves the layer inputs with the kernels to generate the layer outputs (i.e., activations). Let $\boldsymbol{X} \in \mathbb{R}^{b \times c_{\text{in}} \times h_{\text{act}} \times h_{\text{act}}}$ and $\boldsymbol{Y} \in \mathbb{R}^{b \times c_{\text{out}} \times h'_{\text{act}} \times h'_{\text{act}}}$ be the input and output tensors of a convolutional layer, where $b$ denotes the batch size, $c_{\text{in}}$ and $c_{\text{out}}$ denote the number of input and output channels respectively, and $h_{\text{act}}$ and $h'_{\text{act}}$ denote the spatial dimensions (width and height, respectively). Let $\boldsymbol{W} \in \mathbb{R}^{c_{\text{out}} \times c_{\text{in}} \times h_{\text{krn}} \times h_{\text{krn}}}$ be the tensor of spatial convolutional kernels, where $h_{\text{krn}}$ denotes the spatial dimension (width and height). We use $\mathcal{F}_n(\cdot)$ and $\mathcal{F}_n^{-1}(\cdot)$ for the $n \times n$ 2D Fast Fourier Transform (FFT) operation and its inverse. Let "~" denote the corresponding spectral domain variable. For example, let $\widetilde{\boldsymbol{W}}_{j,i} = \mathcal{F}_{h_{\text{act}}+h_{\text{krn}}-1}(\boldsymbol{W}_{j,i})$ and $\widetilde{\boldsymbol{X}}_{k,i} = \mathcal{F}_{h_{\text{act}}+h_{\text{krn}}-1}(\boldsymbol{X}_{k,i})$. A convolutional layer operates as below:

$$\boldsymbol{Y}_{k,j} = \underbrace{\sum_{i=1}^{c_{\text{in}}} \boldsymbol{X}_{k,i} * \boldsymbol{W}_{j,i}}_{\text{Spatial convolution}} = \underbrace{\mathcal{F}_{h_{\text{act}}+h_{\text{krn}}-1}^{-1}\left(\sum_{i=1}^{c_{\text{in}}} \widetilde{\boldsymbol{X}}_{k,i} \circ \widetilde{\boldsymbol{W}}_{j,i}\right)}_{\text{Spectral convolution}}$$

(1)

where $1 \leq k \leq b$ and $1 \leq j \leq c_{\text{out}}$; "$*$" denotes the sliding window operation of spatial convolution; "$\circ$" denotes the Hadamard product operation (element-wise multiplication) of spectral convolution. Note that Equation 1 applies to convolution layers of stride 1 and padding $h_{\text{krn}} - 1$. For padding less than $h_{\text{krn}} - 1$, we will further crop the $\boldsymbol{Y}_{k,j}$. For stride larger than 1, we will further slice the $\boldsymbol{Y}_{k,j}$.[1]

We apply the Overlap-and-Add (OaA) technique [8], [6]. To perform spectral convolution with $n \times n$ FFT (where $h_{\text{krn}} \leq n \leq h_{\text{act}} + h_{\text{krn}} - 1$), we follow the below steps for each batch index $k$, and channel indices $i$, $j$:

1) Partition $\boldsymbol{X}_{k,i}$ into tiles of $\boldsymbol{X}_{k,i}^{t,s} \in \mathbb{R}^{m \times m}$, where $m = n - h_{\text{krn}} + 1$ and $1 \leq t, s \leq \lceil \frac{h_{\text{act}}}{m} \rceil$;

2) Transform the tiles $\boldsymbol{X}_{k,i}^{t,s}$ and kernels $\boldsymbol{W}_{j,i}$ into spectral representation (after proper padding), as $\widetilde{\boldsymbol{X}}_{k,i}^{t,s} = \mathcal{F}_n\left(\boldsymbol{X}_{k,i}^{t,s}\right)$ and $\widetilde{\boldsymbol{W}}_{j,i} = \mathcal{F}_n(\boldsymbol{W}_{j,i})$;

3) Obtain output tiles $\boldsymbol{Y}_{k,j}^{t,s} = \mathcal{F}_n^{-1}\left(\sum_{i=1}^{c_{\text{in}}} \widetilde{\boldsymbol{X}}_{k,i}^{t,s} \circ \widetilde{\boldsymbol{W}}_{j,i}\right)$;

4) Combine $\boldsymbol{Y}_{k,j}^{t,s}$ into $\boldsymbol{Y}_{k,j}$ by OaA. For each tile $\boldsymbol{Y}_{k,j}^{t,s}$, place its adjacent tiles (i.e., $\boldsymbol{Y}_{k,j}^{t\pm1,s}$, $\boldsymbol{Y}_{k,j}^{t,s\pm1}$) with $h_{\text{krn}}-1$ horizontally or vertically overlapping pixels. We obtain the final $\boldsymbol{Y}_{k,j}$ by adding the overlapped pixels.

With OaA, we can select the FFT size $n$ to balance the computation and communication capacity of the target FPGA. Therefore, inference throughput can be maximized. It has been shown in [6] that, setting $n = 8$ or $16$ reduces the computation complexity of spatial convolution by $3\times$ to $4\times$, and improves the overall FPGA throughput by up to $5\times$.

While state-of-the-art accelerator [6] already achieves high throughput, such design only applies to unpruned spectral CNNs. In this paper, we significantly improve the design of [6] by reducing the redundancy in model parameters $\widetilde{\boldsymbol{W}}$.

### B. CNN Weight Pruning

Pruning algorithms leverage redundancy in CNN weight parameters. For a CNN with convolutional layers and fully connected layers, in general, convolutional layers contribute to most (99% for VGG16) of the computation workload, but contain only a small portion (10.6% for VGG16) of the model weights. Thus, compared with fully connected layers, pruning convolutional layers is more important, yet much more difficult as well. For spatial CNNs, pruning rate on fully connected layers can be as high as $30\times$ [14], [15], while pruning rate on convolutional layers is often less than $10\times$ [14].

Below we summarize state-of-the-art works in pruning spatial CNNs. The pioneer work [7] uses an iterative heuristic to prune weights of small magnitudes and achieves $2.7\times$ reduction in convolutional weights on AlexNet [1]. This method has been extended in two directions. The first is to improve compression rate by using more sophisticated heuristics. For example, [9] incorporates both weight pruning and growing, and [10] uses $L_1$ regularization. Pruning of [16] is based on a genetic algorithm. The second direction is to enhance the hardware implementation efficiency by deriving an effective tradeoff between accuracy and pruning rate, e.g., the energy-aware pruning [17], and structure-aware pruning [18], [10]. FPGA hardware accelerators [19], [20] have also been investigated to accommodate pruned CNNs, by leveraging the reconfigurability in on-chip resources. Recently, the authors of [14] have developed a systematic weight pruning framework based on the powerful optimization tool ADMM (Alternating Direction Method of Multipliers) [21]. Such framework consistently achieves higher pruning rate than prior arts.

Besides the works on spatial CNN pruning, [22] proposes a frequency-domain compression technique which first converts inputs and kernels of all layers into frequency domain using

---

[1]When stride is larger than 1, spectral convolution may not lead to reduced computation complexity. However, most of the convolutional layers of modern CNNs (e.g., [2], [5], [13]) have stride equal to 1.

discrete cosine transform (DCT), then dynamically prunes unimportant connections. Although this method reduces parameters in convolutional layers by up to 70%, it is difficult to leverage this technique in hardware platform.

With performance brought by spectral convolution algorithm [6], combining it with pruning creates more potential for high throughput hardware implementation. To the best of our knowledge, SPEC$^2$ is the first work to exploit such opportunity.

### C. CNN Accelerators

By leveraging the computation and memory access patterns of CNNs, many domain-specific architectures have been proposed for accelerating inference. Among these works, [23] shows a unified convolutional matrix-multiplication representation for both convolutional and fully-connected layers. This micro-architecture is optimized for both computation speed and resource utilization. In [24], a pipeline and DRAM-free architecture is proposed to simplify data movement between memory and processing elements. This design achieves very high working frequency as well as DSP utilization. On the other hand, to exploit sparsity in pruned CNNs, [25] introduces a sparse computation architecture aware of zeros in both weights and activations. Beside acceleration in spatial domain, [8] and [6] accelerate convolutional layers in spectral domain, where the computation primitives are Fast Fourier Transforms and Hadamard products. Using the additional Overlap-and-Add technique, computation complexity (of unpruned CNNs) can be significantly reduced without hardware reconfiguration. ASICs also demonstrate promising performance in processing CNNs due to their customization ability and low power consumption. Google TPUs [26] employ systolic array based matrix multiplication units to execute convolutional and fully-connected layers with $70\times$ higher power efficiency than GPUs.

## III. PRUNING SPECTRAL CNNs

### A. Overview

The goal of weight pruning is to convert the dense spectral kernels $\widetilde{W}$ into sparse ones, while keeping the overall classification accuracy of CNNs unaffected. ADMM is an optimization framework that we can utilize to iteratively achieve the above goal, without using handcrafted heuristics.

As summarized by Figure 1, the inputs to our pruning algorithm are the $L$-layer spectral CNN with its unpruned weights $\left\{ \widetilde{W}^\ell \mid 1 \leq \ell \leq L \right\}$, and the target pruning factor $\alpha$. The output is the CNN with sparse spectral weights $\left\{ \widetilde{W}_\alpha^\ell \right\}$, where the subscript $\alpha$ ($> 1$) means that at most $\frac{1}{\alpha}$ of the tensor elements are non-zeros. Spectral pruning of SPEC$^2$ consists of three steps. The *ADMM training* step rescales the weights by joint consideration of the classification accuracy and sparsity requirements. The outputs $\left\{ \widetilde{W}_{AD}^\ell \right\}$ are still dense tensors, where $\left(1 - \frac{1}{\alpha}\right)$ portion of the elements are driven to close-to-zero values. The *pruning* step simply removes the close-to-zero values of $\left\{ \widetilde{W}_{AD}^\ell \right\}$ and returns sparse tensors $\left\{ \widehat{W}_{PR}^\ell \right\}$. The *re-training* step fine-tunes the non-zeros values of $\left\{ \widetilde{W}_{PR}^\ell \right\}$ to
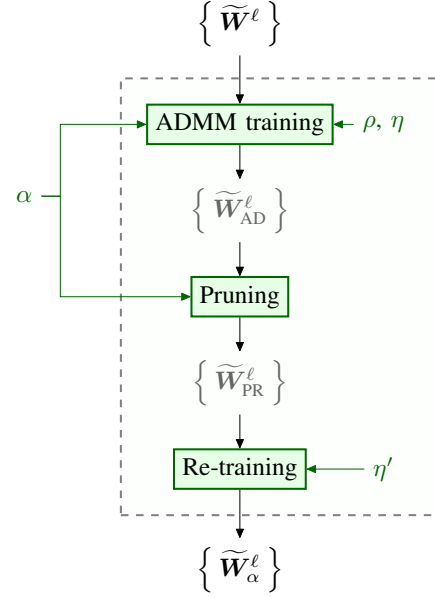


Fig. 1: SPEC$^2$ spectral pruning overview

recover accuracy. The final outputs are the sparse spectral kernels $\left\{ \widetilde{W}_\alpha^\ell \right\}$. It is worth noting that, the ADMM training can be understood as solving a constrained optimization problem, where the optimization objective is loss minimization to ensure high classification accuracy, and the constraint is imposed by the pruning factor $\alpha$. Due to the non-convexity in the objective function and the non-differentiability in the constraint, it is hard to identify analytical solutions. Thus, ADMM breaks the optimization problem into two sub-problems, where one of them has a closed-form solution and the other can be solved iteratively by the stochastic gradient descent algorithm. The ADMM training alternates between the two sub-problems, and eventually they will both converge. The solution identified by ADMM ensures high accuracy, and can be easily pruned.

Different from the existing ADMM based approach [14] which performs pruning in the space domain, SPEC$^2$ performs end-to-end spectral pruning without transformation between $W$ and $\widetilde{W}$. Such spectral pruning enables us to exploit computation redundancy in both the sliding window operation and the spectral kernel weights. Therefore, together with our novel FPGA architecture (Section IV), SPEC$^2$ achieves remarkably high inference throughput without accuracy loss.

### B. ADMM Training

To prune an $L$-layer CNN with pruning factor $\alpha$, we abstract the problem as the below optimization problem:

$$\underset{\left\{ \overline{W}_*^\ell \right\}}{\text{minimize}} \quad \texttt{Loss}\left( L, \overline{L}; \left\{ \widetilde{W}_*^\ell \right\} \right)$$

$$\text{subject to} \quad \texttt{NNZ}\left( \widetilde{W}_*^\ell \right) \leq \frac{1}{\alpha} \cdot \texttt{Size}\left( \widetilde{W}_*^\ell \right), \quad \forall 1 \leq \ell \leq L.$$

197

where $\texttt{Loss}\left(\cdot\right)$ gives the CNN loss calculated by the predicted labels $\boldsymbol{L}$ and the ground-truth $\overline{\boldsymbol{L}}$, under the current model parameters $\left\{\widetilde{\boldsymbol{W}}_*^\ell\right\}$. Function $\texttt{NNZ}\left(\cdot\right)$ measures the **n**umber of **n**on-**z**eros of the tensor, and $\texttt{Size}\left(\cdot\right)$ returns the total number of elements of the tensor (i.e., $\texttt{Size}\left(\widetilde{\boldsymbol{W}}\right) = c_{\text{in}} \cdot c_{\text{out}} \cdot n^2$).

The above problem is hard to solve since the constraint is non-differentiable. Therefore, we perform several steps of transformation to make the problem tractable. Firstly, we rewrite the above optimization problem with an auxiliary variable $\widetilde{\boldsymbol{Z}}_*^\ell$ and an indicator function $\texttt{Ind}\left(\cdot\right)$:

$$\underset{\left\{\widetilde{\boldsymbol{W}}_*^\ell\right\}}{\text{minimize}} \quad \texttt{Loss}\left(\boldsymbol{L}, \overline{\boldsymbol{L}}; \left\{\widetilde{\boldsymbol{W}}_*^\ell\right\}\right) + \sum_{\ell=1}^{L} \texttt{Ind}\left(\widetilde{\boldsymbol{Z}}_*^\ell, \alpha\right) \quad (2)$$

$$\text{subject to} \quad \widetilde{\boldsymbol{Z}}_*^\ell = \widetilde{\boldsymbol{W}}_*^\ell, \quad \forall 1 \leq \ell \leq L.$$

where $\texttt{Ind}\left(\widetilde{\boldsymbol{Z}}_*^\ell, \alpha\right) = \begin{cases} 0, & \text{if } \texttt{NNZ}\left(\widetilde{\boldsymbol{Z}}_*^\ell\right) \leq \frac{1}{\alpha}\texttt{Size}\left(\widetilde{\boldsymbol{W}}_*^\ell\right) \\ +\infty, & \text{otherwise} \end{cases}$

The indicator function, seen as enforcing hard penalty on the spectral kernel sparsity, still makes the optimization difficult. Therefore, as the next step of transformation, we relax the simple constraint of Equation 2. During the optimization process, we do not enforce $\widetilde{\boldsymbol{Z}}_*^\ell = \widetilde{\boldsymbol{W}}_*^\ell$. Instead, we apply to the objective function an additional penalty term $\texttt{Pty}\left(\cdot\right)$ measuring the distance between $\widetilde{\boldsymbol{Z}}_*^\ell$ and $\widetilde{\boldsymbol{W}}_*^\ell$. Then the optimization problem (on $\widetilde{\boldsymbol{W}}_*^\ell$ and $\widetilde{\boldsymbol{Z}}_*^\ell$) becomes unconstrained, and thus, can be solved iteratively. The values of $\widetilde{\boldsymbol{W}}_*^\ell$ and $\widetilde{\boldsymbol{Z}}_*^\ell$ are updated in each iteration, and ultimately, they converge to the same value (i.e., $\widetilde{\boldsymbol{W}}_*^\ell \to \widetilde{\boldsymbol{Z}}_*^\ell$) which is regarded as a good solution to the objective. Theoretically, penalty $\texttt{Pty}\left(\cdot\right)$ should grow with the number of iterations to ensure convergence.

With the above intuitions, we formally describe the algorithm to solve Equation 2 under ADMM. Let $\widetilde{\boldsymbol{U}}_*^\ell$ measure the difference between $\widetilde{\boldsymbol{W}}_*^\ell$ and $\widetilde{\boldsymbol{Z}}_*^\ell$. Define function $f_{\text{AD}} \coloneqq \texttt{Loss}\left(\boldsymbol{L}, \overline{\boldsymbol{L}}; \left\{\widetilde{\boldsymbol{W}}_*^\ell\right\}\right) + \sum_{\ell=1}^{L} \texttt{Ind}\left(\widetilde{\boldsymbol{Z}}_*^\ell, \alpha\right) + \sum_{\ell=1}^{L} \texttt{Pty}\left(\widetilde{\boldsymbol{W}}_*^\ell, \widetilde{\boldsymbol{Z}}_*^\ell, \widetilde{\boldsymbol{U}}_*^\ell\right)$. ADMM decomposes the optimization on $f_{\text{AD}}$ into two sub-problems, one on $\widetilde{\boldsymbol{W}}_*^\ell$ only and the other on $\widetilde{\boldsymbol{Z}}_*^\ell$. ADMM finds a solution by "alternating the optimization direction" along $\widetilde{\boldsymbol{W}}_*^\ell$ and $\widetilde{\boldsymbol{Z}}_*^\ell$. The update rule for $\widetilde{\boldsymbol{W}}_*^\ell$ and $\widetilde{\boldsymbol{Z}}_*^\ell$ at iteration $i+1$ is defined as Equation 3.

Now we use the Frobenius norm as the measure of matrix distance. Thus, $\texttt{Pty}\left(\widetilde{\boldsymbol{W}}, \widetilde{\boldsymbol{Z}}, \widetilde{\boldsymbol{U}}\right) \coloneqq \frac{\rho}{2}\left\|\widetilde{\boldsymbol{W}} - \widetilde{\boldsymbol{Z}} + \widetilde{\boldsymbol{U}}\right\|_F^2 - \frac{\rho}{2}\left\|\widetilde{\boldsymbol{U}}\right\|_F^2$, where $\rho$ is a given constant coefficient. The first sub-problem of Equation 3 often does not have an analytic solution (due to non-convexity of $\texttt{Loss}\left(\cdot\right)$), but can be solved by stochastic gradient descent since both $\texttt{Loss}\left(\cdot\right)$ and $\texttt{Pty}\left(\cdot\right)$ are differentiable.

In fact, the objective of $\texttt{Loss}\left(\left\{\widetilde{\boldsymbol{W}}^\ell\right\}\right) + \sum \texttt{Pty}\left(\widetilde{\boldsymbol{W}}^\ell, \widetilde{\boldsymbol{Z}}^\ell, \widetilde{\boldsymbol{U}}^\ell\right)$ can be understood as the standard CNN loss plus an additional regularization term, and so existing techniques on CNN training are all applicable here.

The second sub-problem admits a simple analytic solution. Clearly, $\widetilde{\boldsymbol{Z}}_{i+1}^\ell$ should satisfy the sparsity constraint by $\alpha$ (so that $\texttt{Ind}\left(\widetilde{\boldsymbol{Z}}_{i+1}^\ell, \alpha\right) = 0$). Then, due to the special form of the penalty (i.e., Frobenius norm), to get optimal $\widetilde{\boldsymbol{Z}}_{i+1}^\ell$, we take $\widetilde{\boldsymbol{W}}_{i+1}^\ell + \widetilde{\boldsymbol{U}}_i^\ell$ and set the elements of smallest magnitude to zero, until $\frac{1}{\alpha} \cdot \texttt{Size}\left(\widetilde{\boldsymbol{Z}}_{i+1}^\ell\right)$ non-zeros remain. As for the variable $\widetilde{\boldsymbol{U}}$, note that it records the cumulative difference between $\widetilde{\boldsymbol{W}}$ and $\widetilde{\boldsymbol{Z}}$, thus enforcing $\widetilde{\boldsymbol{W}}$ to eventually converge to $\widetilde{\boldsymbol{Z}}$.

$$\left\{\widetilde{\boldsymbol{W}}_{i+1}^\ell\right\} = \underset{\left\{\widetilde{\boldsymbol{W}}_*^\ell\right\}}{\arg\min} f_{\text{AD}} = \underset{\left\{\widetilde{\boldsymbol{W}}_*^\ell\right\}}{\arg\min} \left(\texttt{Loss}\left(\boldsymbol{L}, \overline{\boldsymbol{L}}; \left\{\widetilde{\boldsymbol{W}}_*^\ell\right\}\right) \right.$$
$$\left. + \sum_{\ell=1}^{L} \texttt{Pty}\left(\widetilde{\boldsymbol{W}}_*^\ell, \widetilde{\boldsymbol{Z}}_i^\ell, \widetilde{\boldsymbol{U}}_i^\ell\right)\right) \quad (3a)$$

$$\left\{\widetilde{\boldsymbol{Z}}_{i+1}^\ell\right\} = \underset{\left\{\widetilde{\boldsymbol{z}}_*^\ell\right\}}{\arg\min} f_{\text{AD}} = \underset{\left\{\widetilde{\boldsymbol{z}}_*^\ell\right\}}{\arg\min} \left(\sum_{\ell=1}^{L} \texttt{Ind}\left(\widetilde{\boldsymbol{Z}}_*^\ell, \alpha\right)\right.$$
$$\left. + \sum_{\ell=1}^{L} \texttt{Pty}\left(\widetilde{\boldsymbol{W}}_{i+1}^\ell, \widetilde{\boldsymbol{Z}}_*^\ell, \widetilde{\boldsymbol{U}}_i^\ell\right)\right) \quad (3b)$$

$$\left\{\widetilde{\boldsymbol{U}}_{i+1}^\ell\right\} = \left\{\widetilde{\boldsymbol{U}}_i^\ell + \widetilde{\boldsymbol{W}}_{i+1}^\ell - \widetilde{\boldsymbol{Z}}_{i+1}^\ell\right\} \quad (3c)$$

In summary, we iteratively update $\widetilde{\boldsymbol{W}}$ and $\widetilde{\boldsymbol{Z}}$ in an "alternating direction" manner. Throughout all iterations, $\texttt{NNZ}\left(\widetilde{\boldsymbol{Z}}\right) \leq \frac{1}{\alpha} \cdot \texttt{Size}\left(\widetilde{\boldsymbol{Z}}\right)$ is always ensured, while $\texttt{NNZ}\left(\widetilde{\boldsymbol{W}}\right)$ may not satisfy the $\alpha$ constraint. At the end of the ADMM training, $\left(1 - \frac{1}{\alpha}\right)$ fraction of the elements in $\widetilde{\boldsymbol{W}}_{\text{AD}}^\ell$ are close to (but not exactly equal to) zero. We then prune those near-zero elements and fine-tune the remaining $\frac{1}{\alpha}$ fraction of elements, by our spectral re-training algorithm.

### C. Pruning and Re-Training

After the ADMM optimization, we remove the $\left(1 - \frac{1}{\alpha}\right)$ fraction of the near-zero elements of $\widetilde{\boldsymbol{W}}_{\text{AD}}^\ell$, to get sparse kernels $\widetilde{\boldsymbol{W}}_{\text{PR}}^\ell$. Such pruning operation results in slight accuracy degradation ($0.03\%$ for LeNet on MNIST and $0.4\%$ for VGG16 on CIFAR-10). To recover accuracy, we further tune the non-zero values of $\widetilde{\boldsymbol{W}}_{\text{PR}}^\ell$, without changing the kernel sparsity. We term the fine-tuning step as spectral re-training.

Since re-training only aims at improving accuracy, we define loss function simply as $f_{\text{RT}} = \texttt{Loss}\left(\boldsymbol{L}, \overline{\boldsymbol{L}}; \left\{\widetilde{\boldsymbol{W}}_{\text{PR}}^\ell\right\}\right)$. Thus, using stochastic gradient descent, re-training propagates gradients from $f_{\text{RT}}$ backwards and adjusts the values of $\widetilde{\boldsymbol{W}}_{\text{PR}}^\ell$ accordingly. Note that since re-training does not change the sparsity of $\widetilde{\boldsymbol{W}}_{\text{PR}}^\ell$, the gradient $\frac{\partial f_{\text{RT}}}{\partial \widetilde{\boldsymbol{W}}_{\text{PR}}^\ell}$ is masked by the non-zero positions of $\widetilde{\boldsymbol{W}}_{\text{PR}}^\ell$. In summary, at the end of re-training, we have $\texttt{NNZ}\left(\widetilde{\boldsymbol{W}}_\alpha^\ell\right) \leq \frac{1}{\alpha} \cdot \texttt{Size}\left(\widetilde{\boldsymbol{W}}_\alpha^\ell\right)$, and the sparse spectral CNN achieves high classification accuracy ($99.1\%$ for LeNet on MNIST and $90.8\%$ for VGG16 on CIFAR-10 when $\alpha = 4$).

**Algorithm 1** SPEC$^2$ spectral pruning algorithm

**Input:** Unpruned spectral CNN $\left\{\widetilde{\boldsymbol{W}}^\ell\right\}$; Pruning rate $\alpha$; Learning rate $\eta, \eta'$; Penalty coefficient $\rho$;

**Output:** Pruned spectral CNN with sparse kernels $\left\{\widetilde{\boldsymbol{W}}_\alpha^\ell\right\}$

1: $\left\{\widetilde{\boldsymbol{W}}_0^\ell\right\} \leftarrow \left\{\widetilde{\boldsymbol{W}}^\ell\right\}$
2: $\left\{\widetilde{\boldsymbol{Z}}_0^\ell\right\} \leftarrow \left\{\boldsymbol{0}\right\}$
3: $\left\{\widetilde{\boldsymbol{U}}_0^\ell\right\} \leftarrow \left\{\boldsymbol{0}\right\}$
4: **for** iteration $i = 0$ to $i_{\max}$ **do**   ▷ ADMM training
5:   **while** not converged **do**
6:     $\left\{\widetilde{\boldsymbol{W}}_i^\ell\right\} \leftarrow \left\{\widetilde{\boldsymbol{W}}_i^\ell - \eta \cdot \frac{\partial f_{\mathrm{AD}}}{\partial \widetilde{\boldsymbol{W}}_i^\ell}\right\}$
7:   **end while**
8:   $\left\{\widetilde{\boldsymbol{W}}_{i+1}^\ell\right\} \leftarrow \left\{\widetilde{\boldsymbol{W}}_i^\ell\right\}$
9:   $\left\{\widetilde{\boldsymbol{Z}}_{i+1}^\ell\right\} \leftarrow \left\{\mathrm{Pruned}\left(\widetilde{\boldsymbol{W}}_{i+1}^\ell + \widetilde{\boldsymbol{U}}_i^\ell\right)\right\}$
10:   $\left\{\widetilde{\boldsymbol{U}}_{i+1}^\ell\right\} \leftarrow \left\{\widetilde{\boldsymbol{U}}_i^\ell + \widetilde{\boldsymbol{W}}_{i+1}^\ell - \widetilde{\boldsymbol{Z}}_{i+1}^\ell\right\}$
11: **end for**
12: $\left\{\widetilde{\boldsymbol{W}}_{\mathrm{AD}}^\ell\right\} \leftarrow \left\{\widetilde{\boldsymbol{W}}_{i_{\max}}^\ell\right\}$
13: $\left\{\widetilde{\boldsymbol{W}}_{\mathrm{PR}}^\ell\right\} \leftarrow \left\{\mathrm{Pruned}\ \widetilde{\boldsymbol{W}}_{\mathrm{AD}}^\ell\right\}$   ▷ Pruning
14: **for** iteration $i = 0$ to $i'_{\max}$ **do**   ▷ Re-training
15:   $\left\{\widetilde{\boldsymbol{W}}_{\mathrm{PR}}^\ell\right\} \leftarrow \left\{\widetilde{\boldsymbol{W}}_{\mathrm{PR}}^\ell - \eta' \cdot \mathtt{Mask}\left(\frac{\partial f_{\mathrm{RT}}}{\partial \widetilde{\boldsymbol{W}}_{\mathrm{PR}}^\ell}\right)\right\}$
16: **end for**
17: $\left\{\widetilde{\boldsymbol{W}}_\alpha^\ell\right\} \leftarrow \left\{\widetilde{\boldsymbol{W}}_{\mathrm{PR}}^\ell\right\}$   ▷ End of SPEC$^2$ pruning

Algorithm 1 shows the overall SPEC$^2$ pruning algorithm (including ADMM training, pruning and re-training), as described in above. As a final remark, to support the stochastic gradient descent algorithm for any FFT size $n$, we incorporate the OaA technique into ADMM training and re-training. During forward propagation, we partition the input activation of each layer into $n \times n$ tiles and apply $\widetilde{\boldsymbol{W}}_i^\ell$ or $\widetilde{\boldsymbol{W}}_\alpha^\ell$ according to the steps described in Section II-A. During backward propagation, the gradients are derived by the automatic differentiation algorithm of Tensorflow.

## IV. ACCELERATOR DESIGN

### A. Overview

By Equation 1, computation of sparse spectral convolution can be decomposed into three steps: (1) 2D FFT on the input activations; (2) Multiplication-Accumulation (MAC) for Hadamard product computation and reduction along input channel dimension; (3) 2D IFFT on output activations. Figure 2 shows the overall accelerator to compute the above steps.

The external DDR stores the input images, layer activations, as well as the sparse spectral kernel weights of the CNN model. Due to the limited on-chip BRAM, we perform tiling of the activations and kernels. A tile of spectral kernel is pre-loaded into the **kernel buffer**. Tiles of layer activations communicate back and forth between DDR and FPGA to keep the inference pipeline busy. Once a tile of previous-layer activations arrives on-chip, it goes through a **2D FFT** module and

the spectral outputs are stored in the **input buffer**. The **sparse Hadamard** module then computes Hadamard product of the *sparse* spectral kernels and the *dense* spectral activations, by reading into the kernel and input buffers. The partial results of accumulation along the input channel dimension are stored in the **output buffer**. After iterating through all data in the kernel tile, the output activations go through a **2D IFFT** module and return to DDR. We implement 2D FFT and IFFT based on [6]. i.e., we decompose the 2D FFT into two phases of 1D FFTs. We use the 1D FFT pipeline and perform matrix transpose with Streaming Permutation Network proposed in [27].

The following is required to achieve high performance of the inference engine: **(i) Efficient random access** into the data buffers: The sparse Hadamard module fetches data from the dense activation tensor based on the non-zero indices of the pruned kernels. Since kernels can have random non-zero indices, bank conflicts may happen if multiple MAC pipelines of the Hadamard module request data from the same activation BRAM block. Therefore, we replicate activation data to reduce the pipeline stalls due to BRAM conflicts. **(ii) Parallelism** across various tensor dimensions: Spectral convolution with pruning dramatically reduces the CNN computation complexity. We design a parallelization strategy that translates the reduced amount of computation into higher throughput. We parallelize along the batch and channel dimensions of activations, while ensuring fast random access of on-chip data.

### B. Sparse Spectral Convolution Engine

The sparse Hadamard module is the key component of the inference engine. This module processes a kernel tile of shape $c \times c \times n \times n$ and an activation tile of shape $b \times c \times n \times n$, where $c$ is the tiling factor along the input and output channel dimensions (i.e., the $c_{\mathrm{in}}$ and $c_{\mathrm{out}}$ channels are partitioned into tiles of $c$ channels); $n$ is the width and height of each kernel and activation map (i.e., 2D FFT size), and $b$ is the batch size. Note that pruning is along the last two dimensions of the kernel tile, so the pruning algorithm ensures *exactly* $\frac{1}{\alpha} \cdot n^2$ non-zeros in each one of the $c^2$ kernel maps.

Figure 3 illustrates the storage arrangement for sparse kernels. As an example, Figure 3a shows four sparse spectral kernels. The upper part of Figure 3b shows the data structure to represent the data in Figure 3a. For each kernel matrix of input channel $i$ and output channel $j$, we use a $\left(\frac{1}{\alpha} \cdot n^2\right) \times 2$ table to store the non-zero values and the corresponding indices. The lower part of Figure 3b shows the memory layout. We split the value-index table into two tables and populate them with some pre-processed information to facilitate hardware access. The *index table* is $\left(\lambda \cdot \frac{1}{\alpha} \cdot n^2\right)$ by $R$, where $R \geq 1$ is the number of activation replicas and $\lambda \geq 1$ is an overhead coefficient related with $R$. This table stores pre-computed addresses into the input buffer. The *value table* is $\left(\lambda \cdot \frac{1}{\alpha} \cdot n^2\right)$ by 3. This table stores the non-zero values and control signals required by the Hadamard pipeline. More details of the tables and parameters are discussed later in this section.

To compute a single $\widetilde{\boldsymbol{X}}_{k,i} \circ \widetilde{\boldsymbol{W}}_{j,i}$ in Equation 1, the sparse Hadamard module sequentially iterates through the index and
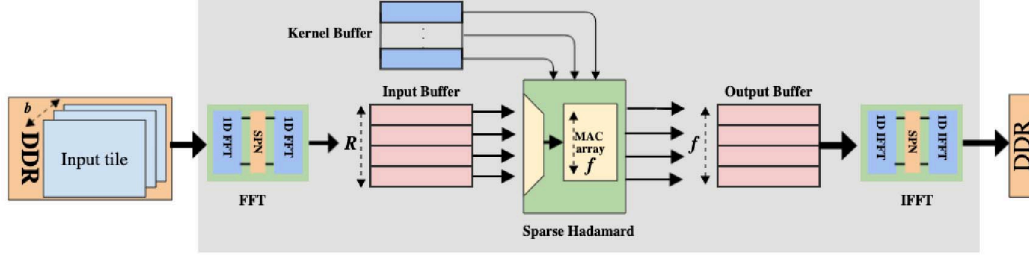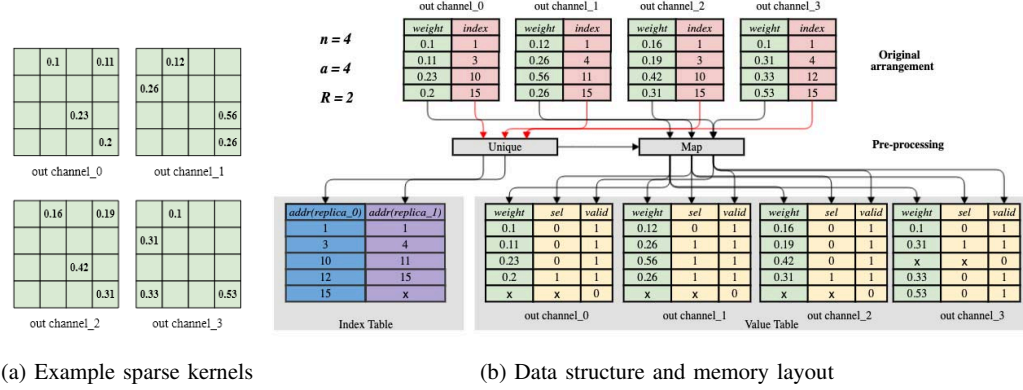
Fig. 2: Overview of SPEC$^2$ inference engine



(a) Example sparse kernels

(b) Data structure and memory layout
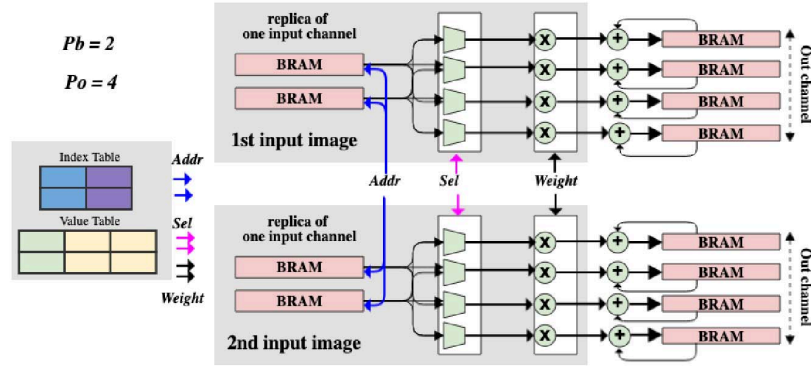
Fig. 3: Storage arrangement for sparse kernels



Fig. 4: Sparse Hadamard pipeline

value tables corresponding to indices $j$, $i$, and initiates random accesses into the input buffer. A single multiplier finishes the computation for a given $(j, i)$ pair in $\lambda \cdot \frac{1}{\alpha} \cdot n^2$ cycles. All multipliers on-chip together exploit parallelism along the dimensions of batch and output channels. Figure 4 shows the design of the sparse Hadamard computation pipeline.

Let $P = P_b \cdot P_o$ be the total number of multipliers in the module, where $P_b$ and $P_o$ denote batch and output channel parallelism, respectively ($P_b = 2$, $P_o = 4$ in Figure 4). The $P_o$ multipliers corresponding to the same $P_b$ index form a *group*, and they all access activations belonging to the same input channel. Thus, each cycle, a group initiates $P_o$ memory requests into the same map $\widetilde{X}_{k,i}$. Ideally, all the $P_o$ requests

have to be served in one cycle to avoid pipeline stalls. In our design, since we store each $\widetilde{X}_{k,i}$ of size $n \times n$ into a single BRAM block, we then need $P_o$ activation replicas in case of the worst case scenario where all the memory requests are distinct. In practice, however, we notice that the spectral sparse kernels demonstrate strong correlation in non-zero locations across channels. Thus, the number of unique addresses in a group of $P_o$ requests is often much less than $P_o$. For this reason, we introduce the number of replicas $R$ (where $1 \leq R \leq P_o$, and $R = 2$ in the example) as an extra design parameter. Higher $R$ reduces the value of $\lambda$ and thus the likelihood of pipeline stalls, but also increases BRAM overhead due to replication.

200

To route from the returned $R$ activation values to the $P_o$ multipliers, we implement a crossbar connecting BRAMs to multipliers for each group. An $R$-to-1 MUX is placed in front of each multiplier for data selection. We further utilize the information in the index and weight tables (Figure 3b) to avoid runtime calculation of addresses and MUX control signals. Since non-zero locations of spectral kernels are fixed during inference, we can determine the unique addresses within each group of requests in offline pre-processing. Specifically, during pre-processing, we sort the group of $P_o$ requests and store the $Q$ number of unique addresses in $\left\lceil \frac{Q}{R} \right\rceil$ rows of the index table. As a result, the $P_o$ runtime requests can be served in $\left\lceil \frac{Q}{R} \right\rceil$ cycles. We also pre-compute the corresponding MUX selection signals (i.e., replica ID) and store them in the "`sel`" column of the value table (Figure 3b). The "`valid`" bit specifies if the value in the corresponding row is valid. Among the $\lambda \cdot \frac{1}{\alpha} \cdot n^2$ rows in the value table, only $\frac{1}{\alpha} \cdot n^2$ of them are valid.

### C. Performance Analysis

We use the notation $S_*$ to denote the total amount of resources available: $S_{BW}$ for external bandwidth (number of complex words per cycle); $S_{DSP}$ for DSPs (number of complex multiplier-adders); $S_{BRAM}$ for BRAMs (number of $36 \times 1K$ BRAM blocks). Also, as a reasonable design choice, we set $P_o = c$, $P_b = b$. Overall throughput (number of multiplication / additions per unit time) is calculated as:

$$T_{sys} = \left( \frac{1}{\lambda_R} \cdot P_o \cdot P_b \cdot 2 \right) \cdot \min \left\{ 1, \frac{1}{2} \cdot \frac{S_{BW}}{S_{BW}^{req}} \right\} \cdot F \quad (4)$$

where $S_{BW}^{req}$ is the required bandwidth to support the DDR to 2D FFT / IFFT communication (Since spectral kernels are reused by large enough number of input tiles, the amortized cost of loading kernels from DDR to FPGA is zero, as analyzed in [6]); $\frac{1}{\lambda_R}$ specifies the average DSP utilization given $R$ replicas; the number "2" indicates multiplication and addition; and $F$ is the working frequency. In the design, the throughput of FFT, sparse Hadamard, and IFFT can be easily matched by adjusting data parallelism of the 2D FFT and IFFT modules. Hence, $S_{BW}^{req}$ is equivalent to required bandwidth by the FFT module, i.e., $S_{BW}^{req} = \frac{2P_b \cdot n^2}{\Omega_H} \cdot F$, where $\Omega_H$ is number of clocks for finishing the corresponding Hadamard product on the tile. Note that FFT and IFFT computations only consist of a small portion of the total convolution layer workload, as shown in [6]. So under a reasonable assumption that there are sufficient logic resources, we implement the 2D FFT / IFFT pipelines with the on-chip logic, and the Multiplication-Accumulation units of the sparse Hadamard module with the DSPs. Thus, the throughput optimized design is obtained by solving the following:

$$\underset{R, P_b, P_o}{\text{maximize}} \quad T_{sys}$$
$$\text{subject to} \quad P_b \cdot P_o \le S_{DSP}, \quad (5)$$
$$P_b \cdot (R + P_o) + 1.5 \cdot P_o \le S_{BRAM}$$

where, the BRAM constraint arises from the number of 1K BRAM blocks for the input/output buffers and the kernel buffer. The number $1.5^2$ in the BRAM constraint means the number of needed $36 \times 1K$ BRAM blocks to store kernel values and indices.

## V. EXPERIMENTS

### A. Experimental Setup

For pruning, we implement the ADMM and re-training algorithm by Tensorflow on NVIDIA P100 GPUs. We use LeNet (on MNIST dataset) and VGG16 (on CIFAR-10 and Flowers-102 [28] datasets) for accuracy and throughput evaluation. As for the hardware implementation, we use Xilinx Virtex-7 XC7VX690T to accelerate convolution layers of the CNN. Other operations, such as ReLU, pooling, partitioning and concatenation, are performed in the host CPU, which communicates with FPGA using PCIe interface. Due to the fact that operations in the host CPU account for a small portion ($< 1\%$) of total computations and most of these operations can be parallelized, CPU execution will not be the bottleneck during inference. We use 16-bit fix-point data format during inference. We use Vivado2018.3 to synthesize and implement the Verilog code. In the following section, to make fair comparison with state-of-the-art spatial CNN accelerators, "image frames per second" (FPS) is adopted to measure inference throughput.

### B. Evaluation Classification Accuracy

In this section, we evaluate the effect of pruning rate $\alpha$ and FFT size $n$ on the classification accuracy. We first obtain the dense spectral kernels $\left\{ \widetilde{W}^{\ell} \right\}$ by converting from the pre-trained spatial CNNs. Then we perform pruning by ADMM and re-training to get the sparse $\left\{ \widetilde{W}_{\alpha}^{\ell} \right\}$. Keeping $\alpha$ the same among different kernels works well in other ADMM based compression works[14]. Besides, the same compression ratio avoids load imbalance when multiple kernels are executed in parallel. We try six pruning configurations, by setting FFT size $n = 8$ or 16 and compression rate $\alpha = 2, 4$ or 8. For each configuration, we perform hyper-parameter tuning independently, by searching over the parameter space defined by: ADMM penalty coefficient $\rho \in \{ 0.001, 0.002, 0.005 \}$, initial learning rate $\eta_0 \in \{ 0.001, 0.0001 \}$, learning rate decaying factor (per 20 epochs[3]) $\gamma \in \{ 0.8 \}$. The ADMM training converges after 200 epochs for LeNet, and 300 epochs for VGG16. The re-training step converges after 10 epochs for LeNet and 30 for VGG16.

**Accuracy under various pruning rates** Figure 5 shows accuracy of the three CNN models when increasing the pruning rate from 2 up to 8. For LeNet on MNIST, accuracy can be fully recovered even when 75% spectral weights are pruned. Small accuracy degradation (0.2%) is resulted from pruning 87.5% of the spectral weights. For VGG16, our

---

[2] 16-bit fix point complex kernels need 1 $36 \times 1K$ BRAM, the corresponding indices need another $18 \times 1k$ BRAM

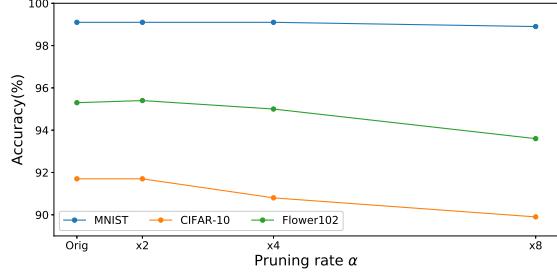[3] An epoch means a full traversal of the training set.

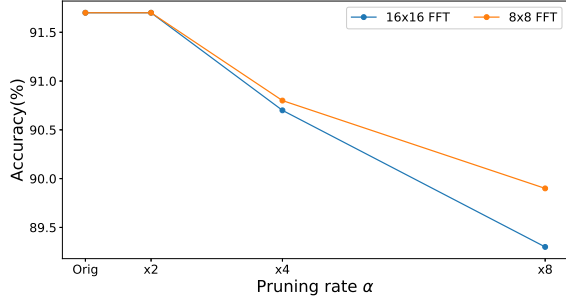Fig. 5: Accuracy under various pruning rate $\alpha$

TABLE I: Accuracy after various pruning steps ($\alpha = 4$)

|  | MNIST | CIFAR-10 | Flowers-102 |
|---|---|---|---|
| Original $\left\{ \widetilde{\boldsymbol{W}}^{\ell} \right\}$ | 99.1 | 91.7 | 95.3 |
| ADMM $\left\{ \widetilde{\boldsymbol{W}}^{\ell}_{\text{AD}} \right\}$ | 99.0 | 90.9 | 95.0 |
| Pruning $\left\{ \widetilde{\boldsymbol{W}}^{\ell}_{\text{PR}} \right\}$ | 99.0 | 90.5 | 94.8 |
| Re-training $\left\{ \widetilde{\boldsymbol{W}}^{\ell}_{\alpha} \right\}$ | 99.1 | 90.8 | 95.0 |

training, small accuracy drop is observed when testing under $\left\{ \widetilde{\boldsymbol{W}}^{\ell}_{\text{AD}} \right\}$. After obtaining $\left\{ \widetilde{\boldsymbol{W}}^{\ell}_{\text{PR}} \right\}$ by removing the near-zero values of $\left\{ \widetilde{\boldsymbol{W}}^{\ell}_{\text{AD}} \right\}$, the accuracy further drops for $0.03\%$ to $0.4\%$. After the re-training step, the accuracy is recovered, and the final accuracy degradation compared with the original unpruned models is $0.0\%$ for MNIST, $0.9\%$ for CIFAR-10 and $0.3\%$ for Flowers-102.



Fig. 6: Accuracy under various FFT sizes $n$ (CIFAR-10)

pruning algorithm achieves less than $0.9\%$ (for CIFAR-10) and $0.3\%$ (for Flowers-102) accuracy loss when pruning rate $\alpha \leq 4$. The accuracy loss is $1.8\%$ (for CIFAR-10) and $1.7\%$ (for Flowers-102) when $\alpha = 8$.

**Accuracy under various FFT sizes** It has been shown in [6] that the FPGA computation and communication workload may depend on the selected FFT size $n$. Hence, it is necessary to evaluate the pruning algorithm performance under various $n$. As shown in Figure 6, $8 \times 8$ spectral kernels leads to higher accuracy under the same pruning rate. As the pruning rate increases, the accuracy gap between the $8 \times 8$ configuration and the $16 \times 16$ one becomes larger.

**ADMM training statistics** As described in Section III-B, the ADMM pruning step drives $\left( 1 - \frac{1}{\alpha} \right)$ portion of the $\widetilde{\boldsymbol{W}}^{\ell}$ elements to close-to-zero values. The re-training step sets these near-zero values to zero and fine-tune the remaining $\frac{1}{\alpha}$ portion. Here we visualize the value distributions of spectral kernels at various stages of the pruning, as shown in Figure 7. The left one represents the distribution of the original kernels $\left\{ \widetilde{\boldsymbol{W}}^{\ell} \right\}$ of VGG16, in which values spread out in the $[0.0, 0.1]$ interval. After ADMM training with $\alpha = 4$, the weight distribution is significantly changed, as most values gather around zero (see the middle figure). By further pruning near-zero weights and re-training, smaller percent ($25\%$) of the weights remain non-zero. We consistently observe such change in distribution for other CNNs and datasets.

Table I shows the change in classification accuracy after various steps of weight pruning ($\alpha = 4$). After ADMM

*C. Evaluation on Inference Throughput*

Overall inference throughput, bottlenecked by either computation or communication speed on FPGA, can be maximized by solving Equation 5. Note that to obtain the optimal design point $(R^*, P_b^*, P_o^*)$, we have to empirically determine the relation between DSP utilization and the parameters $R$, $P_o$.

Figure 8 shows DSP utilization(average ratio of active DSP to total DSP) under various number of replicas $R$ and output channel parallelism $P_o$. In all cases, we do not require a large number of replicas to achieve high DSP utilization. Even when $P_o$ is as high as 128, it suffices to set $R = 10$ for over $80\%$ DSP utilization. In addition, with increasing pruning rate, DSP utilization drops slightly under the same number of replicas. This is due to that the sparser the spectral kernels, the more irregular the non-zero locations.

Based on the statistics of DSP utilization, overall throughput can be obtained by Equation 4, and optimal design point $(R^*, P_b^*, P_o^*)$ can be obtained. Figure 9 shows throughput under various configurations with $\alpha = 4$, $n = 8$ and $P_b = 10$. The solid lines indicate throughput on the target hardware platform, and the dotted lines represent performance bound due to limited number on-chip BRAMs and DSPs. The solid lines terminate at $P_o = 64$ since we set $P_o$ to be power of two. Due to enough DSP resources on the target platform, the design is bound by BRAMs. By Figure 9, the optimal configuration is $P_b^* = 10$, $P_o^* = 64$, $R^* = 16$, corresponding to throughput of $148$ FPS (with average DSP utilization of $99\%$). The total required communication bandwidth between off-chip memory and on-chip processing pipeline is $9$ GB/s, which is lower than the peak system bandwidth of $21$ GB/s.

Given different compression ratios, we can also find optimal configurations based on design exploration algorithm, as shown in Table II. Throughput here is only for computation of convolutional layers. Although kernels are more sparse given higher $\alpha$, like 8, throughput is still almost doubled compared with $\alpha = 4$.
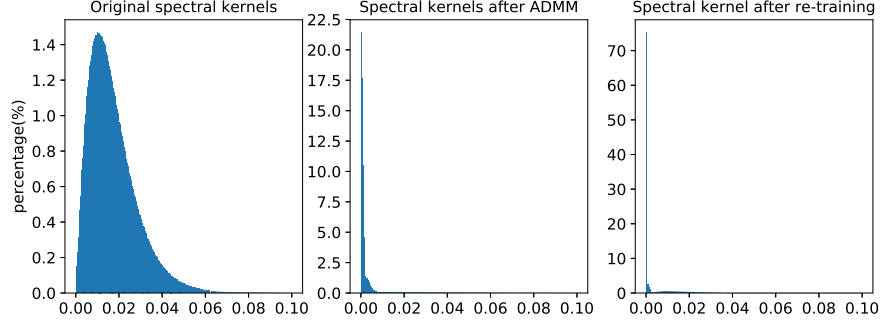
202

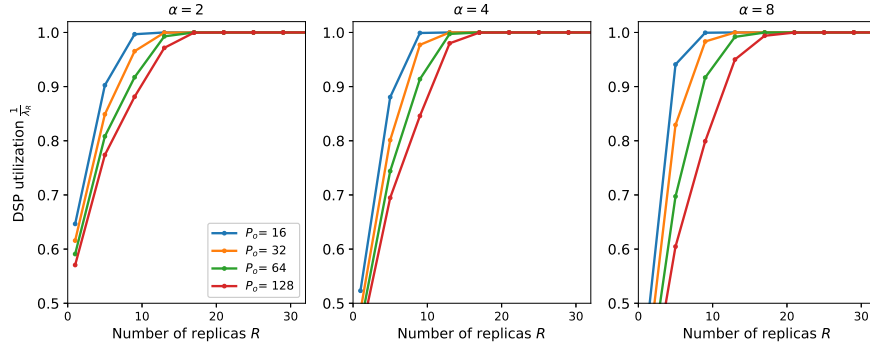Fig. 7: Value distribution of spectral kernels with $\alpha = 4$



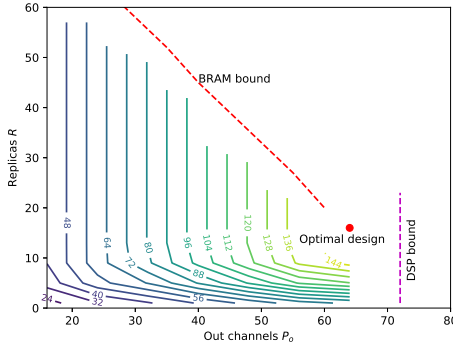Fig. 8: DSP utilization under various number of replica $R$, and output channel parallelism $P_o$



Fig. 9: Throughput under various configurations

TABLE II: Throughput of VGG16 ($224 \times 224$ input images)

| | Pruning rate $\alpha$ | | |
|---|---|---|---|
| | 2 | 4 | 8 |
| Throughput (FPS) | 74 | 148 | 284 |
| Replicas $R$ | 16 | 16 | 16 |
| DSP Utilization $\frac{1}{\lambda_R}$ | 100% | 99% | 96% |

Table III shows the throughput comparison with state-of-the-art on VGG16. [6], [8] accelerates unpruned spectral CNNs. SPEC$^2$ achieves $24\times$ higher throughput, where we estimate that $8\times$ improvement is due to the larger device and $3\times$ improvement comes from our spectral pruning. Compared with [23] which accelerates unpruned spatial CNNs on the same FPGA as us, SPEC$^2$ achieves $9\times$ improvement, where around $3\times$ is due to the spectral convolution algorithm and the other $3\times$ comes from spectral pruning. Compared with pruned spatial CNN implementation [31], SPEC$^2$ achieves $30\times$ higher throughput on a device with 2816 more DSPs. For a fairer comparison, assume throughput of [31] scales with the number of DSPs and [31] can be implemented on the same device as us with the same working frequency. In such case, SPEC$^2$ still achieves $1.8\times$ throughput improvement.

## VI. CONCLUSION

We have proposed SPEC$^2$, an approach to prune and accelerate spectral CNNs. SPEC$^2$ performs systematic weight pruning based on ADMM and deploys an efficient sparse tensor computation pipeline on FPGA to achieve high accuracy and high throughput inference simultaneously.

In the future, we will extend SPEC$^2$ in the following ways. First of all, we will explore model redundancy from various aspects. We will incorporate channel pruning, and weight quantization into the ADMM framework to obtain even

TABLE III: Comparison with state-of-the-art designs on VGG16

| | [6] | [8] | [29] | [23] | [30] | [31] | SPEC[2] |
|---|---|---|---|---|---|---|---|
| FPGA | Intel Stratix V | Intel QPI FPGA | Xilinx Zynq | Xilinx Virtex-7 | Intel Arria 10 | Xilinx Artix-7 | Xilinx Virtex-7 |
| Frequency (MHz) | 200 | 200 | 150 | 150 | 221 | 100 | 200 |
| Datatype | 16-bit FX | 32-bit FT | 16-bit FX | 16-bit FX | 16-bit FX | 16-bit FT | 16-bit FX |
| DSP Usage | 100% (256) | 100% (224) | 89% (780) | 78% (2833) | 49% (1500) | 52% (384) | 89% (3200) |
| LUT Usage | 46% (107K) | - | 84% (183K) | 81% (300K) | 73% (313K) | - | 55% (237K) |
| RAM Usage | 73% (1377) | - | 87% (486) | 42% (624) | 61% (1668) | 53% (388) | 85% (1244) |
| Throughput (fps) | 6 | 4 | 8 | 16 | 19 | 5 | **148** |

\* **FX**: Fixed-point data; **FT**: Floating-point data

more compact spectral CNN models. We will also develop a complete tool chain to automatically prune, quantize and accelerate spectral CNNs.

## VII. ACKNOWLEDGEMENTS

## REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *NIPS'12*, 2012.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, 2014.

[3] A. G., M. Z. Howard, and et al., "Mobilenets: Efficient convolutional neural networks for mobile vision applications," *CoRR*, 2017.

[4] X. Zhang, X. Zhou, M. Lin, and J. Sun, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," *CoRR*, 2017.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[6] H. Zeng, R. Chen, C. Zhang, and V. Prasanna, "A framework for generating high throughput cnn implementations on fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. ACM, 2018.

[7] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems*, 2015.

[8] C. Zhang and V. Prasanna, "Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. ACM, 2017.

[9] G. Yiwen, Y. Anbang, and C. Yurong, "Dynamic network surgery for efficient dnns," ser. NIPS'16, 2016.

[10] W. Wei, W. Chunpeng, W. Yandan, C. Yiran, and L. Hai, "Learning structured sparsity in deep neural networks," ser. NIPS'16, 2016.

[11] S. Zhou, K. Lakhotia, S. G. Singapura, and et al., "Design and implementation of parallel pagerank on multicore platforms," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2017.

[12] D. A. Bader, V. Agarwal, and K. Madduri, "On the design and analysis of irregular algorithms on the cell processor: A case study of list ranking," in *2007 IEEE International Parallel and Distributed Processing Symposium*. IEEE, 2007.

[13] C. Szegedy, W. Liu, Y. Jia, and et al., "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015.

[14] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, "A systematic dnn weight pruning framework using alternating direction method of multipliers," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018.

[15] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding," *arXiv preprint arXiv:1510.00149*, 2015.

[16] D. Xiaoliang, Y. Hongxu, and N. K. Jha, "Nest: A neural network synthesis tool based on a grow-and-prune paradigm," *arXiv*, 2017. [Online]. Available: https://arxiv.org/abs/1711.02017

[17] Y. Tien-Ju, C. Yu-Hsin, and S. Vivienne, "Designing energy-efficient convolutional neural networks using energy-aware pruning," ser. CVPR'17. IEEE, 2017.

[18] H. Yihui, Z. Xiangyu, and S. Jian, "Channel pruning for accelerating very deep neural networks," ser. ICCV'17. IEEE, 2017.

[19] S. Wang, Z. Li, C. Ding, and et al., "C-lstm: Enabling efficient lstm using structured compression techniques on fpgas," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 11–20.

[20] R. Zhao, Y. Hu, J. Dotzel, C. De Sa, and Z. Zhang, "Building efficient deep neural networks with unitary group convolutions," *arXiv preprint arXiv:1811.07755*, 2018.

[21] S. Boyd, N. Parikh, E. Chu, B. Peleato, J. Eckstein *et al.*, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends® in Machine learning*, 2011.

[22] Z. Liu, J. Xu, X. Peng, and R. Xiong, "Frequency-domain dynamic pruning for convolutional neural networks," in *Advances in Neural Information Processing Systems*, 2018.

[23] Z. Chen, S. Guangyu, Zhenman, and et al., "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," *TCAD*, 2018.

[24] E. Wu, X. Zhang, D. Berman, I. Cho, and J. Thendean, "Compute-efficient neural-network acceleration," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. ACM, 2019.

[25] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan *et al.*, "Scnn: An accelerator for compressed-sparse convolutional neural networks," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, June 2017.

[26] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE.

[27] R. Chen, S. Siriyal, and V. Prasanna, "Energy and memory efficient mapping of bitonic sorting on fpga," in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '15. ACM, 2015.

[28] M.-E. Nilsback and A. Zisserman, "Automated flower classification over a large number of classes," in *Proceedings of the Indian Conference on Computer Vision, Graphics and Image Processing*, Dec 2008.

[29] Q. Jiantao, W. Jie, Y. Song, and et al., "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '16. ACM, 2016.

[30] W. Xuechao, Y. Cody Hao, Z. Peng, , and et al., "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," ser. DAC'17. ACM, 2017.

[31] A. Page, A. Jafari, C. Shea, and T. Mohsenin, "Sparcnet: A hardware accelerator for efficient deployment of sparse convolutional networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, 2017.