An Efficient Hardware Architecture for Sparse Convolution using Linear Feedback Shift Registers

Murad Qasaimeh, Joseph Zambreno and Phillip H. Jones
Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, USA 50010
Email: {qasaimeh, zambreno, phjones} @iastate.edu

***Abstract*—Deep convolutional neural networks (CNNs) have shown remarkable success in many computer vision tasks. However, their intensive storage, bandwidth and computational requirements limit their deployment to embedded platforms. Although several research efforts have shown that pruning redundant weights could significantly reduce storage and computations, working with sparse weights remains challenging. The irregular computation of sparse weights and the overhead of managing their representation limit the efficiency of the underlaying hardware. To address these issues, we propose a hardware-friendly pruning algorithm that generates structured sparse weights. In this algorithm, locations of non-zero weights are derived on-chip in real-time using Linear Feedback Shift Registers (LFSRs) to eliminate the overhead of managing sparse weight representations. In this paper, we also propose a hardware inference engine for sparse convolution on FPGAs. It uses LFSRs to localize non-zero weights within weights tensors and avoids copying sparse weights indices by generating them on-chip. Experimental results show that the proposed pruning method can reduce the size of VGG16, ResNet50, and InceptionV3 models by 80%, 76% and 65% with less than 2% accuracy loss. Experiments also demonstrate that our accelerator can achieve 456-534 effective GOP/s for the modern CNNs on Xilinx ZCU102, which provides a 1.2-2.7× speedup over previous sparse CNN accelerators on FPGAs.**

***Index Terms*—DNNs, Sparsity, FPGA, Pruning, Convolution.**

I. Introduction

Deep neural networks (DNNs) have achieved remarkable success in many challenging tasks including image classification, object detection, and image segmentation [1] [2], [3]. Although DNNs provide state-of-the-art accuracy, they require considerable storage, memory bandwidth and computational resources, which limit their deployment to embedded environments. A major challenge in deploying DNNs to resource limited platforms is the large amounts of energy consumed when accessing model parameters from external DRAMs. For example, in 45nm CMOS technology, accessing a 32bit DRAM memory requires 640pJ, which is 3 order of magnitudes higher than a 32-bit floating point add operation (0.9 pJ) [4]. Therefore, deploying DNNs with large memory bandwidth requirements on battery constrained embedded platforms remains a challenging task.

One promising approach is model compression by pruning redundant and less important weights. Recent researche works show that significant redundancy exists in DNN models that can be pruned without sacrificing accuracy [4], [5]. However, despite the significant reduction in a models' parameters (up to 90%), these methods have hardly sped up inference time. The irregular distribution of weights in pruned models poorly fit current computing platforms such as CPUs and GPUs. The speedup can be negative compared to dense convolution when the sparsity ratio is low [6]. Another challenge with sparse convolution is the overhead of managing sparse representations. The amount of data used to record non-zero weight locations can be high with low sparsity ratios. In summary, mapping irregular sparse convolution to hardware is challenging and could require twice the memory of the dense model.

The irregularity of sparse weights could be reduced by applying constraints on the locations of weights during the pruning process. In unstructured pruning methods, only the magnitude of weights are used to decide whether to prune or retain a weight. All weights below a specific threshold are pruned from the network (converting a dense model into a sparse model). The remaining weights are re-trained to recover accuracy losses. Since in unstructured pruning no location constraints are placed on the non-zero weights, the generated sparse weights have an irregular distribution within the weight tensor. In structured pruning methods, constraints on locations of pruned weights are applied (e.g. channel-wise, filter-wise, shape-wise, etc.). For example, in channel-wise pruning methods, all weights in a channel will be pruned or retained. A general advantage of structured pruning is the retainment of hardware friendly regularity that can be leveraged to simplify sparse convolution operations. However, strict pruning constraints can negatively impact accuracy.

In this paper, we propose a hardware-friendly pruning algorithm that generates structured sparse weight patterns. We also propose an FPGA-based inference engine for sparse convolution, which uses pruned models generated by our prunning approach to speed up convolution computation. We avoid copying sparse weight indices from off-chip memory by computing these indices on-chip in real-time. The contribution of this work can be summarized as follows:

- We propose a structured pruning methods that uses pseudo random sequences generated by LFSRs with known seeds, to regularize and prune CNN models.
- We explore several pruning methods and evaluate their accuracy across multiple sparsity ratios on ImageNet.
- We design an FPGA-based accelerator by leveraging our proposed pruning method to create an architecture that efficiently performs sparse convolution operations.

The remainder of this paper is organized as follows. Section II provides background on CNNs, pruning, sparse representations, and LFSRs. Section III reviews related work. In Section IV, the proposed structured pruning algorithm is presented. Section V explains the implementation details of our hardware architecture. In Section VI, we discuss the experimental setup and results. Finally, Section VII concludes the paper with outlooks for future work.

II. Background and Motivations

In this section, we give an overviw of convolution operations, weight pruning algorithms, and common sparse represenitions. We then review the concept of linear feedback shift registers (LFSRs), and how they can be used to generate pesdo random sequences.

A. Convolution Operations in CNNs

The convolution operation takes two inputs: an image tensor (I), and a list of filters (F). It outputs a map of extracted features also called a feature map (O). The operation can be represented by 6-nested loops as shown in Algorithm (1), where H is the height of output feature map, W is the width of output feature map, C is the number of input channels and equals number of filters channels, N is the number of output channels and equals number of filters. R, S are the filter dimensions. Figure (1) shows a convolution layer with H=4, W=4, C=3, R=3, S=3 and N=2.

B. DNN Weight Pruning

Weight pruning is the process of removing unimportant and redundant weights from DNN models without scarifying accuracy. It is an efficient way to compress dense models by reducing the number of parameters, and operations. During the pruning process, the importance of weights is defined based on a given metric, and less important weights are pruned first. The most commonly used metric is the absolute value of weights, first presented by [5]. If a weight's absolute value is less than a certain threshold, it is zeroed out. Other criteria focus on

Algorithm 1: Convolution Operation

```
1 for $h = 0;\ h < H;\ h+ = 1$ do
2     for $w = 0;\ w < W;\ w+ = 1$ do
3         for $n = 0;\ n < N;\ n+ = 1$ do
4             for $c = 0;\ c < C;\ c+ = 1$ do
5                 for $r = 0;\ r < R;\ r+ = 1$ do
6                     for $s = 0;\ s < S;\ s+ = 1$ do
7                         O(n, h, w) += F(n, c, r, s) × I(c, h+r, w+s)
```

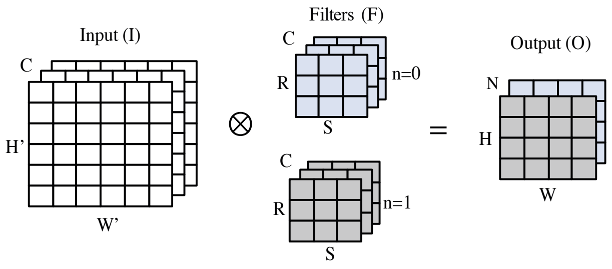

Fig. 1: Convolution Layer with Two Filters

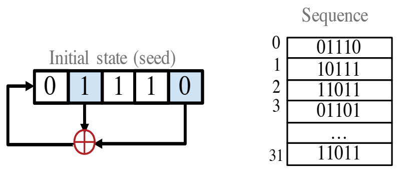

Fig. 2: Linear Feedback Shift Register (LFSR)

the energy consumption of a CNN layer to guide the pruning process [7]. The geometrical location of weights is also used to reduce the irregularity of sparse weights [8] [9], such pruning schemes are referred to as structured pruning algorithms.

C. Sparse Matrix Representation

One solution to work with sparse weights efficiently is to use an alternative data structure to represent sparse data. The three most common sparse representations are : (1) Coordinate List (COO) where non-zero weights are stored as a list of tuples with each tuple containing: filter's row and colum indices, channel number, filter's number and non zero value. (2) Compressed Sparse Row (CSR) or Yale, format represents a matrix by three arrays containing nonzero values, column indices, and the extents of rows. It is similar to COO, but compresses the row indices. (3) Compressed Sparse Column (CSC) representation is similar to CSR except that a row index for each non-zero weight, and column pointers are stored. Table (I) shows the encoding overhead and the minimum sparsity (SP) required for these representations to be smaller than the dense representition.

TABLE I: Sparse Representition Overhead

Representition	Encoding Overhead	Required Sparsity
COO	3×(R×S×C×N)×(1-SP)	SP≥0.667
CSR	(R×S×C×N)×(1-SP) + N×C	SP≥0.5(1/(R×S) +1)
CSC	(R×S×C×N)×(1-SP) + R×S	SP≥0.5(1/(C×N) +1)

D. Linear Feedback Shift Register (LFSR)

Figure 2 shows an example of a 5 bits, LFSR and its sequence. An n-bit LFSR consists of n cells, each of which holds a state variable $s_i \in \{0, 1\}$, and a coefficient (tap) $c_i \in \{0, 1\}$, for i = 0, 1, ..., n-1. The feedback function (XOR function) computes the new state s_n using the coefficients and the state values as shown in Equation 1. The period of a LSFR is a function of its coefficients and initial state values. The maximal sequence period generated by an n bit LFSR is $2^n - 1$ unique states, since the all-zero state is excluded. The maximal period is generated when the coefficients form a primitive polynomial (irreducible). Table (II) lists primitive polynomials for LFSRs of order n= (2-9).

$$s_n = c_0 \cdot s_0 \oplus c_1 \cdot s_1 ... \oplus c_{n-1} \cdot s_{n-1} \qquad (1)$$

TABLE II: Primitive Polynomials

	n=2	n=3	n=4	n=5	n=6	n=7	n=8	n=9
Taps	(0,1)	(0,1)	(0,1)	(0,2)	(0,1)	(0,1)	(0,2,3,4)	(0,4)

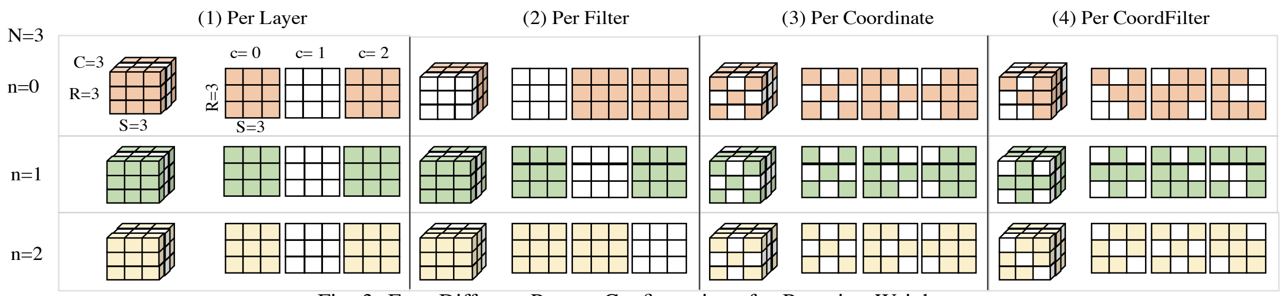

Fig. 3: Four Different Pattern Configurations for Prunning Weights.

III. Related Work

Designing custom hardware architectures has been explored to efficiently support sparse convolution on both ASICs and FPGAs: (1) ASIC-based accelerators. In [10], a CNN accelerator (EIE) is proposed, which exploits the sparsity of both input feature maps and filters. EIE focuses on fully connected (FC) layers and performs optimized sparse matrix vector multiplications to improve computation efficiency. The work in [11] proposed an SCNN accelerator that supports processing convolutional layers in a compressed format. It uses pixel-oriented dataflow, where the innermost computation is a Cartesian product. Zhang et al. [12] present the Cambricon-X accelerator, which applies step indexing techniques, and uses wide (256$\times$16-bit width) memory and (256-to-1) multiplexers (MUXs) in convolution layers to gather sparse weights into a vector, which needs to dynamically select the input vector. (2) FPGA-based accelerators. In [13], a tile look-up table and a channel multiplexer is used to match the index between unstructured sparse weights, and input pixels. A large tile look-up table is used to locate sub-input tiles and avoid indexing. The work in [14] uses a vector generator module to match the index between sparse weights and input activations, and uses shape-wise pruning to allow sharing the same indexes of weights between processing units. A more detailed comparison between our work and FPGA-based accelerators is presented in Section (VI).

IV. Proposed Pruning Method

Our pruning method takes a pre-trained model and a desired sparsity ratio as an input, and generates sparse weights and a list of LFSRs seeds. The method consists of four steps. First, we choose a suitable LFSR pattern configuration for our model. Second, for that configuration, we find the best seeds for each LFSR registers. Third, we train our models with a customized regularization. Finally, we prune and re-train our model. The following sub-sections explain each step in more detail:

A. LFSR Pattern Configurations

In this step, we explore four different pattern configurations for prunning weights in convolution layers. Some of these configurations add strict constraints on the location of non-zero weights within filters, while others use looser constraints at the expense of adding complexity to the pruning process, and inference engine. Figure (3) shows these configurations for a convolution layer composed of 3 filters of size (R=3, S=3, C=3). The configurations are described as follows:

1) PerLayer: one pattern is used to prune weights in all filters. For example, pattern (0-2) is used in all filters to represent non-zero weights indices. This configuration uses only one LFSR to generate one sequence, which generates the most strictly constrainted pattern.
2) PerFilter: one pattern is used to prune weights for each filter. For example, sequences (1-2), (0-2) and (0-1) are used in filter (n=0), (n=1) and (n=2), respectively to represent non-zero weights indices. A total of N (number of filters) LFSRs are required for this configuration.
3) PerCoordinate: A total of R$\times$S patterns are required for this configuration. The same patterns is shared between different filters at the same coordinate (r, s).
4) PerCoordFilter: A total of R$\times$S$\times$N patterns (i.e. LFSR) are used in this configuration. In this configuration, the number of non-zero weights in the sub-tensor (r, s, :) is the same for (r= 0, 1, ..., R) and (s= 0, 1, ..., S).

B. Finding Best Seeds

In this step, we use a ranking algorithm to find the best initial seeds for each LFSR. We use the magnitude of weights in pre-trained models to indicate how important each weight is. We evaluate if the sequence generated by a seed keeps most of the important weights, and prunes unimportant weights at multiple sparsity levels. In this way, we can guarantee using the best starting sequence for our model. The proposed ranking algorithm explores all possible sequences, and returns the best LFSR seeds. Pseudo code of the method is shown in Algorithm (2). The input is a dense weight (F) of size (N$\times$C$\times$R$\times$S) and a desired sparsity level (S_d). The output is a list of seeds; one seed for each LFSR. For tge PerCoordinate pattern, for each coordinate (r, s) we evaluate the sequence generated by the (2^C) different seed combinations. We compute a score for each seed by multipling the filters weight by a linear significant values $sig(idx)$ as in line 10 . The method returns the seeds with the highest scores.

Figure (4) shows an example for a sequence of length 10 and three filters in $perCoordinate$ configuration (i.e. the same pattern is shared across filters). Based on the sparsity level (S_d), we use the first $C \times S_d$ indices from the pattern as shown in the white cells in Figure (4), where C is number of channels. We also add a significance for each weights value. Weights

Algorithm 2: Choosing Best Seeds (PerCoordinate)

```
Input : Filters ($F$) of size ($N \times C \times R \times S$),
        Sparsity ($S_d$).
Output: List of LFSR seeds ($ListSeeds$).
$Size \leftarrow \lceil \log_2(C) \rceil$
$ListSeeds \leftarrow 0$
for $r = 0;\ r < R;\ r+ = 1$ do
    for $s = 0;\ s < S;\ s+ = 1$ do
        $bestSeed \leftarrow 0$
        foreach $(seed_i) \in set(2^{Size})$ do
            $LFSR_{seq} \leftarrow$ generateLFSRseq($seed_i$)
            for $i = 0;\ i < (C \times S_d);\ i+ = 1$ do
                $F(idx) \leftarrow abs(F(r, s, LFSR_{seq}[i]))$
                $sig(idx) \leftarrow (1 - i/C)$
                $score(seed_i)+ = F(i) * sig(idx)$
            if $score(seed_i) > bestSeed$ then
                $bestSeed \leftarrow score(seed_i)$
        $ListSeeds \leftarrow append(bestSeed)$
Return $ListSeeds$
```

that will be pruned first have lower significance compared to the weights pruned at a higher sparsity level. For example: W_8 has the lowest significance value of 0.1 and W_7 has the highest significance value of 1. The input to this step are filter weights and the desired sparsity level (S_d). The output is a list of best seeds $bestSeed$. For each possible seed value $seed_i$, we generate its sequence of length C (# of channels), and use it to compute seed rank $score(seed_i)$.

C. Training with Regularization

In the third step, we change the distribution of weights to follow the selected LFSR sequence generated by the best seeds. We use customized L1 regularization during the training process to force weights in specific locations to have low values or to be zeroed-out without degrading accuracy. We add a regularizer term to the cost function (J), as shown in Equation (2), where n is the number of training samples. λ is the regularization parameter; note, increasing λ adds more penalty on weight values making them closer to zero. F_l is the weight tensor of layer l. S_l is the signficance tensor of layer l with the same shape as F_l. Each value in S_l falls between [0-1] based on the importance of each corresponding weight.

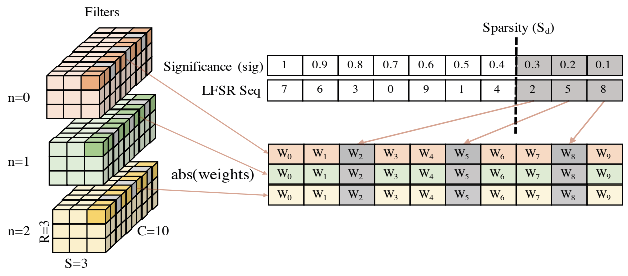

Fig. 4: Example of Pruning Filters in perCoord Configuration.

Finally, the regularization term is computed by a dot product of the weight tensor and the signficance tensor. This way, we force the optimization algorithm to re-distribute weights within each tensor to follow our LFSR patterns.

$$J = \frac{1}{n} \sum_{i=1}^{n} L(y_i - \hat{y}_i) + \lambda \sum_{l=1}^{L} \| F_l \cdot S_l \| \qquad (2)$$

D. Pruning and Re-training

In this step, we use an iterative pruning and re-training process to prune weights in dense models to follow LFSR patterns. We start with a dense model (current sparsity level (S_c)= 0), and then increase (S_c) from [0 - desired sparsity level (S_d)] gradually over number of training iterations. Every weight below the S_c threshold is zeroed out, and the rest remain the same. After each pruning steps, we re-train our model to compesnate for accuracy loss. After a number of training iterations, our model will have S_d×R×S×C×N×L non-zero weights, where L is number of layers.

V. Hardware Accelerator

In this section, we describe the hardware optimizations and implementation details of our sparse CNN accelerator, and its sparse dataflow. We also provide a quantitative analysis of the computing throughput, and required memory bandwidth of our accelerator.

A. Hardware Architecture and Dataflow

An FPGA-based CNN accelerator design consists of four major components: processing elements (PEs), on-chip buffers, external memory, and on-chip interconnect [15]. Before computing, all data including the input image and model weights are stored in external memory. Due to limitations in

Algorithm 3: Pseudo Code of the Dataflow

```
for $h = 0;\ h < H;\ h+ = T_h$ do
    for $w = 0;\ w < W;\ w+ = T_w$ do
        for $n = 0;\ n < N;\ n+ = T_n$ do
            for $c = 0;\ c < NNZ;\ c+ = T_c$ do
                /* Load Input Tile */
                $get(I[T_{h'}][T_{w'}][T_c])$
                /* Load Weight Tile */
                $get(F[T_r][T_s][T_c][T_n])$
                /* Clear Output Tile */
                $O[T_h][T_w][T_n] = 0;$
                for $r = 0;\ r < R;\ r++$ do
                    for $s = 0;\ s < S;\ s++$ do
                        for $n_i = 0;\ n_i < T_n;\ n_i++$ do
                            $seed = readLFSRSeed();$
                            for $c_i = 0;\ c_i < T_c;\ c_i++$ do
                                $c_x = generateNext(seed);$
                                for $h_i = 0;\ h_i < T_h;\ h_i++$ do
                                    for $w_i = 0;\ w_i < T_w;\ w_i++$ do
                                        $O[h_i][w_i][n_i]+ = F[r][s][c_i][n_i] \times$
                                        $I[h_i + r][w_i + s][c_x];$
                /* Store Output Tile */
                $store(O[T_h][T_w][T_n]);$
```

the size of on-chip memory, we cannot copy all data from the external memory to on-chip buffers at once. First, we divide the input data into small portions (Tiles) and cache them into on-chip buffers a tile at a time for feeding the PEs. On-chip buffers are used to store tiles of the input image, model filters, and output (partial sums). Data communication channels between PEs, and on-chip buffer banks are provided through the on-chip interconnect. Finally, the PEs are responsible for all computations.

To achieve high performance, we start our optimization from Algorithm 1 (see Section II). We transform the convolution computation from window-based to element-matrix multiplication. Using this approach, we can process each weight individually, which is more efficient when computing sparse convolution. Then, we apply loop tiling to keep a small portion of data stored on-chip, increasing data reuse and reducing external memory access. External memory access happens only when we finish all computations on the current tile, and a new tile is needed. Loop tiling and ordering decide the dataflow from/to our accelerator.

In our dataflow (shown in Algorithm 3), lines 1-4 show the order in which we process tiles. In lines 5 and 6, we copy a tile of input pixels and weights from external memory into on-chip buffers. The size of an input tile is $[T_{h'}, T_{w'}, T_c]$, where $T_{h'} = T_h + R - 1$, $T_{w'} = T_w + C - 1$. The size of a weight tile is $[T_r, T_s, T_c, T_n]$. We also clear an output tile of size $[T_h, T_w, T_n]$ in line 7. In this section, we explain the dataflow for the perCoordFiler configuration. As discussed in Section IV.B, for the perCoordFiler configuration, we generate a unique pseudo random sequence at each filter's coordinates (r, s, n_i). We achieve this by loading a unique LFSR seed in line 11 and generating a sequence of indices of length T_c as shown in line 13. The sparse weight at index c_i matches the input pixel at index c_x. In order to increase parallelism in our architecture, the nested loops in lines 10, 14, and 15 are unrolled and mapped to a parallel hardware. We choose tile and unrolling sizes such that we fully utilize of all computation resources provided by the FPGA hardware platform.

Figure (5) shows our hardware architecture when input tile size equals $[T'_h = 5, T'_w = 5, T_n = 1, T_c = 1]$. In this architecture, we use 5×5 BRAMs to stores input pixels. Each BRAM stores pixels at the same coordinates over all channels i.e. $I(h, w, :)$. This way, we can read multiple pixels at the same time. To match sparse weights with their corresponding input pixels, we connect T_n LFSRs to the address port of BRAMs (broadcast). LFSRs are used to generate a sequence of sparse weight indices. Thus, every time a new index is generated by a LFSR, a new data will be read from all BRAMs at the same channel. Then, the output ports of BRAMs are multiplied with the same weight value to perform element-matrix multiplication as shown in Figure (6). The partial sums from the multiplication operation are accumulated in $[T_h = 3, T_w = 3]$ registers. Multiplxers of size $(R \times S)$ are used to select the correct input for each accumulator. After $(R \times S \times C \times SP)$ clock cycles, the results in the accumulator registers become valid and can be streamed out to external memory, where SP refers to sparsity ratio.

The attainable system throughput is constrained by either computation (computation-bounded) or communication (memory-bounded). In [16], a roofline performance model is proposed to relate system performance to the peak performance provided by the hardware platform, and off-chip memory traffic. The actual performance of an algorithm on a hardware platform is the minimum of two terms. The first term is the peak throughput (GOP/s) provided by all computation resources in the platform assuming all data is available on-chip. The second term is the maximum performance that the memory system can provide. It depends on the algorithm's computation to communication (CTC) ratio and platform bandwidth. In the next two sections, we provide a quantitative analysis of the computation throughput, and required memory bandwidth of our accelerator.

B. Computation Optimization

The peak computational performance (also called computational roof) is the maximum number of operations per second provided by hardware when all required data is available on-chip. Given a specific output tile size $[T_h, T_w, T_n, T_c]$, the peak computational performance can be computed by Equations (3 and 4). It is a function of loop unrolling factors in the T_h, T_w, T_n, T_c dimensions. Loop unrolling increases peak performance but also increases the resource utilization in FPGA devices. Our architecture needs $(C \times SP \times R \times S)$ clock cycles to finish the execution of each output tile.

$$P(GOP/s) = \frac{total\ number\ of\ operations}{number\ of\ execution\ cycles} \times f \tag{3}$$

$$P = \frac{2 \times H \times W \times N \times (C \times R \times S \times SP)}{\left\lceil \frac{H}{T_h} \right\rceil \times \left\lceil \frac{W}{T_w} \right\rceil \times \left\lceil \frac{N}{T_n} \right\rceil \times \left\lceil \frac{C \times SP}{T_c} \right\rceil \times (C \times SP \times R \times S)} \times f \tag{4}$$

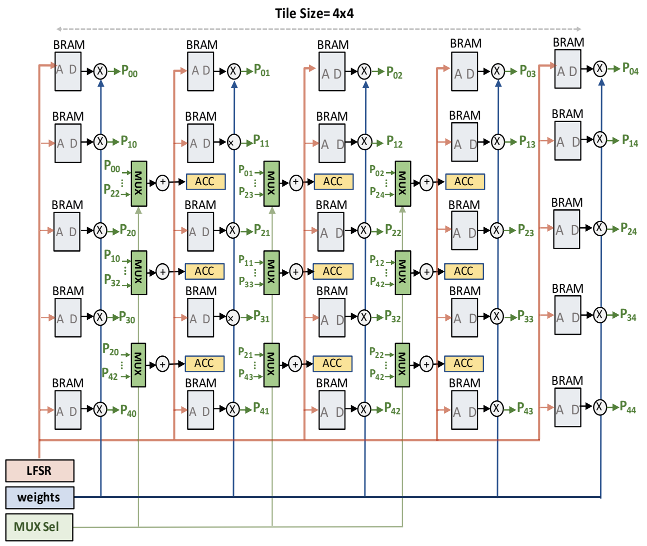

Fig. 5: Hardware architecture of sparse convolution engine for a input Tile of size= $[T'_h = 5, T'_w = 5, T_n = 1, T_c = 1]$

C. Memory Access Optimization

In this subsection, we show how reducing communication volume can increase attainable performance. We reduce memory traffic volume by generating sparse weight indices on-chip using LFSRs, and applying efficient data reuse. Thus, increasing the computation to communication (CTC) ratio of our architecture. This ratio describes the total number of computations per memory access. Equations (5 and 6) show the CTC ratio calculation for our accelerator, where α_{in}, α_{weight}, α_{out} and B_{in}, B_{weight}, B_{out} refer to memory access counts and on-chip buffer sizes for input, weights and output feature maps, respectively. SP is the sparsity ratio, and SP_{En} is the buffer size used for storing the sparse weight representation overhead (in our implementation SP_{En} =0).

$$CTC = \frac{total\ number\ of\ operations}{total\ amount\ of\ external\ data\ access} \tag{5}$$

$$CTC = \frac{2 \times H \times W \times N \times (C \times R \times S \times SP)}{\alpha_{in} \times B_{in} + \alpha_{weight} \times B_{weight} + \alpha_{out} \times B_{out}} \tag{6}$$

where:

$$B_{in} = T_c \times (T_h + R - 1) \times (T_w + S - 1) \tag{7}$$

$$B_{weight} = (R \times S \times C \times N) \times SP + SP_{En} \tag{8}$$

$$B_{out} = T_h \times T_w \times T_n \tag{9}$$

$$\alpha_{in} = \left\lceil \frac{H}{T_h} \right\rceil \times \left\lceil \frac{W}{T_w} \right\rceil \times \left\lceil \frac{N}{T_n} \right\rceil \times \left\lceil \frac{C}{T_c} \right\rceil \tag{10}$$

$$\alpha_{weight} = \left\lceil \frac{H}{T_h} \right\rceil \times \left\lceil \frac{W}{T_w} \right\rceil \times \left\lceil \frac{N}{T_n} \right\rceil \times \left\lceil \frac{C \times SP}{T_c} \right\rceil \tag{11}$$

$$\alpha_{out} = \left\lceil \frac{H}{T_h} \right\rceil \times \left\lceil \frac{W}{T_n} \right\rceil \times \left\lceil \frac{N}{T_n} \right\rceil \tag{12}$$

VI. Experiemnt

In this section, we first evaluate the performance of our pruning algorithm. We measure the accuracy of its four different configurations at multiple sparsity levels. We compare our algorithm's accuracy with dense, and unstructured pruning models. Then, we evaluate the performance of our sparse convolution engine on three modern DNNs, and compare its performance with previous dense, and sparse accelerators.

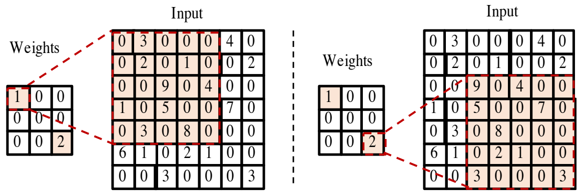

Fig. 6: Element-Matrix Convolution Operation

TABLE III: DNN Models

Model	Top1	Top5	#OP (GOP)	Size (MBs)	# Parameters
VGG16	0.713	0.901	30.92	528 MB	138,357,544
ResNet50	0.749	0.921	6.65	98 MB	25,636,712
InceptionV3	0.779	0.937	11.25	92 MB	23,851,784

A. Pruning Algorithm Evaluation

To evaluate the performance of our pruning algorithm, we measure the classification accuracy of three modern DNNs models: VGG16, ResNet50, and InceptionV3 on the ImageNet dataset (ILSVRC2012) [17]. A summary of the characteristics, and Top1 (Top5) accuracies of the used models are shown in Table (III). The number of operations is computed at 224×224, 224×224 and 229×229 image resolution for VGG16, ResNet50 and InceptionV3, respectively. In our implementation, we extend the TensorFlow framework [18] by associating masks with weight tensors. We update these masks based on the generated LFSRs patterns. During the pruning process, we start with weights from pre-trained models [19]. Then, we apply our pruning algorithm gradually increasing sparsity from an initial sparsity ratio s_i (usually 0) to a final sparsity ratio s_f over a span of n pruning steps. We re-train our models between each pruning for 10 epochs. We carried out the experiments using TensorFlow 1.14 on an Nvidia GeForce GTX TITAN X.

The Top1 (Top-5) classification accuracy of VGG16, ResNet50, and InceptionV3 models at [0,0.1,...,0.9] sparsity levels is shown in Figure (7). We use dense models and unstructured pruning models as our references. Figure (7) also shows the accuracy of our algorithm for four different configurations: perLayer, perFilter, perCoordinate, and perCoordFilter. It can be seen that the perLayer and perFilter configurations have the lowest accuracies. Their accuracy also drops quickly as sparsity increases. This is due to the strict constraints inherent in these configurations, and the complexity of the ImageNet classification problem. The perLayer and perFilter configurations could be useful when applied to simpler problems. While the perCoordinate configuration has an acceptable accuracy, the perCoordFilter configuration achieves the best results. The perCoordFilter configuration was able to increase the sparsity of the VGG16, ResNet50 and InceptionV3 models to 80%, 76%, and 65% respectively with an accuracy (Top1) loss of less than 2%. This configuration was also able to achieve around a 54%, 52%, and 46% sparsity ratio with no accuracy degradation.

B. Hardware Architecture Evaluation

1) Experiments Setup: We evaluate our design on the Xilinx ZCU102 platform. It consists of an UltraScale FPGA, quad ARM CortexA53 processors, 4GB PS DDR4 and 512MB PL DDR4 (14.9GB/s). In our experiments, we use the Xilinx Vivado HLS (v2019.1) tool chain to transform optimized C code into an RTL implementation. Our synthesized design runs at 200MHz on this platform. In this work, we evaluate the performance of our accelerator on the VGG-16, Resnet50 and InceptionV3 benchmarks at a model sparsity of 80%, 76%, 65% respectively

C. Resource Utilization

We evaluated the resource utilization of our accelerator for a number of parallelism options $[T_h, T_w, T_n]$ (see Figure (8)). The utilization of BRAMs is determined by the input tile

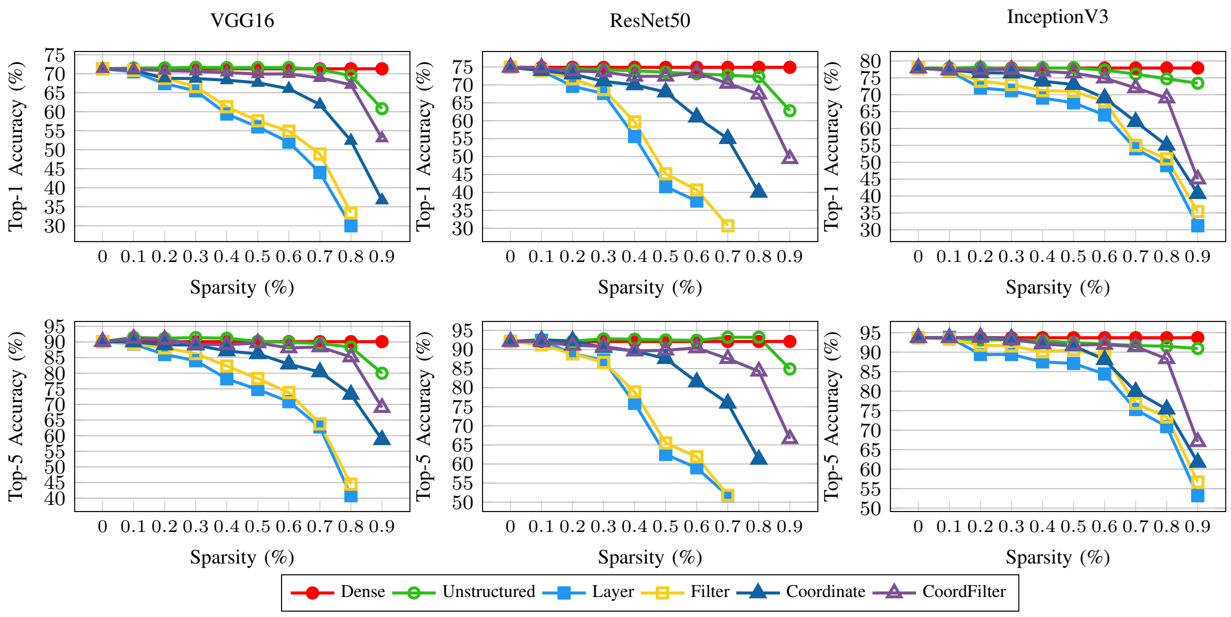

Fig. 7: An accuracy comparison between our pruning algorithm vs. dense and unstructed pruning algorithms for multiple sparsity ratio for VGG16, Resnet50 and InceptionV3 models.

size ($T_{h'}, T_{w'}$), and the number of filters in the weight tile T_n. We use BRAMs to store the input and weight tiles on-chip. The parameters $T_{h'}, T_{w'}$ determine the size of input buffers and T_n determines the number of weight buffers. LUT and FF utilization increases as the parallelism factors T_h, T_w, T_n increase because the number and size of MUXs in our design increase, as shown in Section V.B. We use FF registers to implement $T_h \times T_w \times T_n$ accumulators, because we need to access all registers at the same clock cycle. We also observe that the LUT and FF utilization is almost linear to tile size. The number of DSPs used in our architecture can be calculated as $T_h \times T_w \times T_n$. Each DSP can perform a 16-bit $\times$ 16-bit multiplication operation. Our accelerator configured with $[T_h = 8, T_w = 8, T_n = 24]$ almost fully utilizes the FPGA's hardware resources, and achieves the hightest peak computational performance. Table (V) reports the available resources in the Xilinx Zynq ZCU102 platform.

1) Performance Analysis: We evaluate the performance of our accelerator using three modern CNNs: VGG16, ResNet50 and InceptionV3 at 54%, 52% and 46% sparsity levels. We configure our accelerator to $[T_h, T_w, T_n]$ = [8, 8, 24]. In this configuration, we utilize most of the FPGA resources, and obtain a peak performance of $2 \times 0.2GHz \times 24 \times 8 \times 8 = 614.4\ GOP/s$ when the width of operand is 16-bits.

Our accelerator achieves 534.2 GOP/s effective performance on sparse VGG16 which shows 1.7× and 1.4× speedup compared with [13] and [14], respectively. For Resnet50 and InceptionV3, our architecture achieves 456.0 GOP/s and and 458.0 GOP/s which is 1.56× and 1.8× higher than the effective performance of [13]. In terms of image per second, our accelerator has 1.2× and 2.7× speedup compared with [13]. The work in [14] was optimized to target VGG-16

TABLE IV: Performance Comparison with Related Work. In our work, $[T_h = 8, T_w = 8, T_n = 24]$ is used.

	(2019)[13]	(2019)[13]	(2019)[13]	(2020)[14]	Ours	Ours	Ours
CNN type	VGG16	ResNet50	GoogLeNet	VGG16	VGG16	ResNet50	InceptionV3
Device	ZCU102	ZCU102	ZCU102	ZCU102	ZCU102	ZCU102	ZCU102
Frequency (MHz)	200	200	200	200	200	200	200
Precision	16 bit fixed	16 bit fixed	16 bit fixed	16 bit fixed	16 bit fixed	16 bit fixed	16 bit fixed
DSPs	1144 (45%)	1144 (45%)	1144 (45%)	1350 (53%)	1564 (62%)	1564 (62%)	1564 (62%)
BRAMs	912 (48%)	912 (48%)	912 (48%)	1460 (80%)	840 (46%)	840 (46%)	840 (46%)
LUTs	132K (48%)	132K (48%)	132K (48%)	390K (65%)	203K (74%)	203K (74%)	203K (74%)
FFs	68K (12%)	68K (12%)	68K (12%)	278K (51%)	433K (79%)	433K (79%)	433K (79%)
Sparsity(%)	67.5%	58.7%	65.8%	65.4%	54% / 80%	52% / 76%	46% / 65%
Accuracy loss(%)	0%	0%	0%	0 %	0% / 2%	0% / 2%	0% / 2%
Performance (GOP/s)	309.0	291.4	257.4	495.4	534.2	456.0	458.0
Images/sec	31	104	65	46	38 / 86	154 / 285	75 / 114

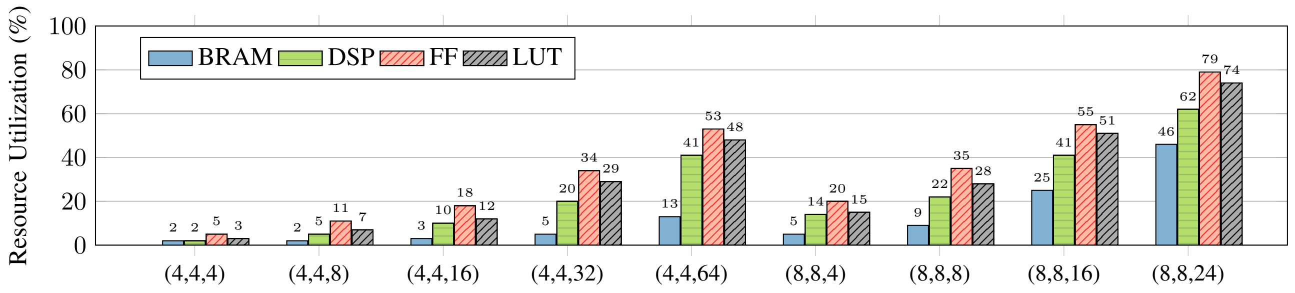

Fig. 8: Resource utilization for configuration (T_h, T_w, T_n), where T_h, T_w, T_n are the output tile dimensions.

TABLE V: Available Resource in the ZCU102 Platform

	BRAMs	DSPs	FFs	LUTs
ZCU102	1,824	2,520	548,160	274,780

giving a throughput of 46 image/sec compared to our 38 image/sec. However, our more generalized architecture supports efficient compuation over a wide range of models. Compared to previous works, we do not need to transfer sparsity index information from off-chip to on-chip. This allows imporved I/O usage and less BRAM requirements. A given configuration of cour architecture performs most optimally when the output feature map dimensions (H, W, N) divide evenly by our output tile dimensions (T_h, T_w, T_n).

VII. Conclusion

In this paper, we proposed a pruning algorithm that generates hardware-friendly structured sparse weights. The locations of non-zero weights are generated in real-time using Linear Feedback Shift Registers (LFSRs). The advantage of using this approach is two-folds: First, eliminating the overhead of managing sparse representations; second, avoiding copying extra data from external memory to on-chip buffers. We also proposed a hardware inference engine that leverages the structure of our pruning algorithm imposes to efficiency perform sparse convolution on FPGAs. It uses LFSRs to compute the positions of non-zero weights within weight tensors on-chip. Experimental results demonstrate that our accelerator can achieve 456-534 GOP/s for the modern CNNs on the Xilinx ZCU102, which provides a 1.5-1.8× speedup over previous sparse CNN FPGA accelerators.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[2] N. McLaughlin, J. M. del Rincon, and P. C. Miller, "Person reidentification using deep convnets with multitask learning," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 27, no. 3, pp. 525–539, 2016.

[3] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3431–3440, 2015.

[4] S. Han, H. Mao, and W. J. Dally, "Deep compression: compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR*, vol. abs/1510.00149, 2016.

[5] 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 28* (C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, eds.), pp. 1135–1143, Curran Associates, Inc., 2015.

[6] Z. Yao, S. Cao, W. Xiao, C. Zhang, and L. Nie, "Balanced sparsity for efficient dnn inference on gpu," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 5676–5683, 2019.

[7] T.-J. Yang, Y.-H. Chen, and V. Sze, "Designing energy-efficient convolutional neural networks using energy-aware pruning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 5687–5695, 2017.

[8] H.-J. Kang, "Accelerator-aware pruning for convolutional neural networks," *IEEE Transactions on Circuits and Systems for Video Technology*, 2019.

[9] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 548–560, 2017.

[10] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: efficient inference engine on compressed deep neural network," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.

[11] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally, "Scnn: An accelerator for compressed-sparse convolutional neural networks," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 2, pp. 27–40, 2017.

[12] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen, "Cambricon-x: An accelerator for sparse neural networks," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, IEEE, 2016.

[13] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang, "An efficient hardware accelerator for sparse convolutional neural networks on fpgas," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 17–25, IEEE, 2019.

[14] C. Zhu, K. Huang, S. Yang, Z. Zhu, H. Zhang, and H. Shen, "An efficient hardware accelerator for structured sparse convolutional neural networks on fpgas," *arXiv preprint arXiv:2001.01955*, 2020.

[15] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays*, pp. 161–170, 2015.

[16] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[17] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255, Ieee, 2009.

[18] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: large-scale machine learning on heterogeneous systems. software available from tensorflow. org. 2015," *URL https://www. tensorflow. org*, 2015.

[19] Keras, "Keras: Using pre-trained models." https://keras.rstudio.com/articles/applications.html#applications-1, 2020.