SpWA: An Efficient Sparse Winograd Convolutional Neural Networks Accelerator on FPGAs

Liqiang Lu, Yun Liang†
Center for Energy-efficient Computing and Applications, School of EECS, Peking University
{liqianglu,ericlyun}@pku.edu.cn

ABSTRACT

FPGAs have been an efficient accelerator for CNN inference due to its high performance, flexibility, and energy-efficiency. To improve the performance of CNNs on FPGAs, fast algorithms and sparse methods emerge as the most attractive alternatives, which can effectively reduce the complexity of CNNs. Using fast algorithms, the feature maps are transformed to special domain to reduce the arithmetic complexity. On the other hand, compressing CNN models by pruning the unimportant connections reduces both storage and arithmetic complexity.

In this paper, we introduce sparse Winograd convolution accelerator (SpWA) combining these two orthogonal approaches on FPGAs. First, we employ a novel dataflow by rearranging the filter layout in Winograd convolution. Then we design an efficient architecture to implement SpWA using line buffer design and Compress-Sparse-Column (CSC) format-based processing element. Finally, we propose an efficient algorithm based on dynamic programming to balance the computation among different processing elements. Experimental results on VGG16 and YOLO network show a 2.9x~3.1x speedup compared with state-of-the-art technique.

1 INTRODUCTION

Convolutional neural networks (CNNs) have been widely used in various computer vision tasks including image classification, object detection, and semantic segmentation [18, 20]. Various hardware accelerators have been proposed to accelerate the performance of CNN models. Among these accelerators, Field Programmable Gate Arrays (FPGAs) turns out to be a promising solution due to its high performance, flexibility, and energy-efficiency. In addition, High Level Synthesis (HLS) helps to greatly lower the barrier of hardware programming [5, 7, 13, 19]. For example, FPGA accelerators for CNNs have been successfully designed using C or OpenCL programming model with high performance. [4, 15, 21, 22, 24, 25, 27].

However, the resources on FPGAs are limited, such as the Block RAMs (BRAMs) for data storage and Digital Signal Processors (DSPs) for computation. Therefore, it is critical to reduce the arithmetic complexity of CNN models so that the FPGAs can yield the expected performance. To address this limitation, an effective approach is to compress CNNs by pruning. [8, 9, 12, 14] have shown that there is significant redundancy in the neural networks, which can be pruned significantly during the training process. For example, Han et al. [8, 9] have shown that the weights in typical CNNs can be pruned to more than 95% sparsity without accuracy loss. Another important trend in acceleration is to transform the feature maps into special domains using fast algorithms which can significantly reduce the number of multiplications in convolution. Winograd and Fast Fourier Transformation (FFT), as fast algorithm representatives, have been widely adopted in many highly-optimized HPC libraries such as CuDNN[3] and MKL[2]. In Winograd algorithm, the input tile and filter are transformed to Winograd domain then perform element-wise matrix multiplication (EWMM). Last, an inverse transformation of the EWMM results is required. For example, using Winograd algorithm with the tile size 4×4 can reduce the number of multiplications from 3.7 GOPs to 1.64 GOPs for the second layer in VGG16 network [20]. Recent work demonstrated that the filters can be trained directly in Winograd domain, which can achieve 90% sparsity without accuracy loss [12].

Prior efforts on the implementations of fast algorithms do not consider the case of sparsity [4, 11, 15, 17, 26]. On the other hand, the hardware design mainly for ASIC platform do not apply fast algorithms to the sparse CNN models [6, 10, 16, 28]. Although fast algorithms and sparse methods are appealing in acceleration, FPGA implementations combining these two techniques have not appeared yet due to several challenges. First, it is difficult to maintain high resource efficiency while leveraging the multiplication reduction from sparsity. Second, sparsity will cause imbalance workload in the Winograd hardware dataflow. Fast algorithm convolution is a regular computation with a tile as a basic operating unit. For example, in Winograd convolution, the transformed input tile and filter perform EWMM operation tile by tile, then the results across different channels need to be accumulated. However, when introducing sparsity to Winograd filters, the workload among different processing elements will be imbalanced.

To overcome these challenges, we present the sparse Winograd CNN accelerator (SpWA) on FPGAs, which accelerates CNN inference that exploits both Winograd fast algorithms and filter sparsity. We first propose a novel dataflow for SpWA. Then, we design an efficient architecture to conduct Winograd transformation and sparse computation. Finally, we develop an efficient algorithm based on dynamic programming to balance the computation among different processing elements.

This work makes the following contributions:

- We present a dataflow that applies sparse Winograd algorithm for accelerating CNNs on FPGAs. In this dataflow, we batch the transformed input tiles and rearrange the filter layout.
- We propose an architecture based on SpWA dataflow. The SpWA architecture employs efficient line-buffer and PE designs.

DAC '18, June 24–29, 2018, San Francisco, CA, USA
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5700-5/18/06…$15.00
https://doi.org/10.1145/3195970.3196120

†Corresponding Author

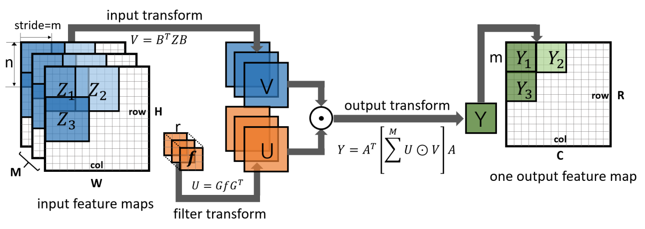

Figure 1: Apply Winograd to convolutional layers

- We design an efficient algorithm to determine the partition of sparse matrices into groups so as to minimize the total idle cycles.

We evaluate our design by implementing VGG16[20] and YOLO[18] on Xilinx ZC706 platform. Experimental results show that our design achieves 2.9x~3.1x speedup compared with state-of-the-art works.

2 BACKGROUND

2.1 Spatial Convolution

Consider the feedforward procedure in a typical convolutional layer which receives M channels of $H \times W$ input feature maps $Z_{M \times H \times W}$ and outputs N channels of $R \times C$ feature maps $Y_{N \times R \times C}$, To generate N channels of output feature map, M channels of input feature maps are convolved with $N \times M$ filters with the size of $r \times r$, where S is the stride of filter.

$$Y(k, i, j) = \sum_{t=1}^{M} \sum_{p=1}^{r} \sum_{q=1}^{r} f(k, t, p, q) \times Z(t, i*S+p, j*S+q) \qquad (1)$$

2.2 Winograd Algorithm

Let us denote the result of computing m outputs with the filter size of r as $F(m, r)$. Using spatial convolution for $F(2, 3)$ requires $2 \times 3 = 6$ multiplications. Winograd algorithm computes $F(2, 3)$ in the following way, which only needs 4 multiplications:

$$Z = \begin{bmatrix} z_0 & z_1 & z_2 & z_3 \end{bmatrix}^{\mathrm{T}} f = \begin{bmatrix} x_0 & x_1 & x_2 \end{bmatrix}^{\mathrm{T}} Y = \begin{bmatrix} y_0 & y_1 \end{bmatrix}^{\mathrm{T}}$$

$$\begin{bmatrix} z_0 & z_1 & z_2 \\ z_1 & z_2 & z_3 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} m_1 + m_2 + m_3 \\ m_2 - m_3 + m_4 \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \end{bmatrix} \qquad (2)$$

m_1,m_2,m_3,m_4 are:

$$\begin{aligned} m_1 &= (z_0 - z_2)x_0 \quad m_2 = (z_1 + z_2)\frac{x_0 + x_1 + x_2}{2} \\ m_4 &= (z_1 - z_3)x_2 \quad m_3 = (z_2 - z_1)\frac{x_0 - x_1 + x_2}{2} \end{aligned} \qquad (3)$$

The 1-D convolution using Winograd algorithm can be formulated using the transformation matrices A, B and G as follows,

$$Y = A^{\mathrm{T}}[(Gf) \odot (B^{\mathrm{T}}Z)] \qquad (4)$$

$$B^T = \begin{bmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \quad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{bmatrix}$$

where $\odot$ is element-wise multiplication (EWMM). In this paper, we use 2D-Winograd algorithm $F(m \times m, r \times r)$, where the output tile size is $m \times m$, the filter size is $r \times r$ and the input tile size is $n \times n$ ($n = m + r - 1$). By nesting 1D-Winograd algorithm with itself, 2D-Winograd algorithm can be formulated as follows,

$$Y = A^{\mathrm{T}}[(GfG^{\mathrm{T}}) \odot (B^{\mathrm{T}}ZB)]A \qquad (5)$$

The number of multiplications is determined by $\odot$. To compute the $m \times m$ tile in the output feature map, Winograd algorithm requires

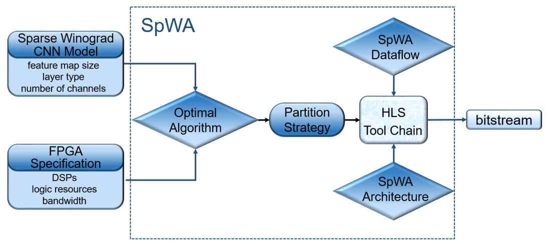

Figure 2: Framework overview

n^2 multiplications while the conventional algorithm requires $m^2 r^2$ multiplications. In this paper, we use a uniform Winograd size $F(2 \times 2, 3 \times 3)$.

Assuming $S = 1$ in Equation 1 that $H = R, W = C$, to apply Winograd algorithm in a convolutional layer, feature maps are divided into tiles as shown in Figure 1. Specifically, the input feature maps are divided into multiple $n \times n$ tiles denoted by Z_{tile} ,where $Z_{tile}(t, i, j)$ means an $n \times n$ tile in the t^{th} channel at the coordinates (i, j), and the output feature maps are divided into multiple $m \times m$ tiles denoted by Y_{tile}, where $Y_{tile}(k, i, j)$ means an $m \times m$ tile in the k^{th} channel at the coordinates (i, j). Note that the stride of the input tile is m, which results in $(r-1)$ rows/columns overlapping. Each time Winograd algorithm is called, it generates an $m \times m$ tile in the output feature maps. The 2-D convolution using the Winograd can be formulated as follows, where $Q_w(k)$, $U_w(k, t)$ and $V_w(t)$ are $n \times n$ tiles in Winograd domain,

$$\begin{aligned} &Y_{tile}(k, i, j) = A^{\mathrm{T}} Q_w(k) A \quad Q_w(k) = \sum_{t=1}^{M} U_w(k, t) \odot V_w(t) \\ &U_w(k, t) = Gf(k, t)G^{\mathrm{T}} \quad V_w(t) = B^{\mathrm{T}} Z_{tile}(t, i, j) B \end{aligned} \qquad (6)$$

Here, $U_w(k, t)$ is the transformed Winograd filter in the k^{th} output channel and t^{th} input channel. $V_w(t)$ is the transformed input tile in the t^{th} input channel. $Q_w(k)$ represents the results of EWMM-accumulation in the k^{th} output channel. In sparse Winograd convolution, $U_w(k, t)$ is pruned with sparsity.

3 SPWA FRAMEWORK

We propose SpWA to map sparse Winograd convolution onto FPGAs. As shown in Figure 2, it takes the sparse Winograd CNN model and specification of target FPGA as inputs and generates the bitstream of FPGA configuration. The CNN model involves the topological structure of the network and sparse Winograd filters of each layer. The FPGA specification includes DSPs, logic resources and bandwidth. Our framework mainly involves three components: SpWA dataflow, SpWA architecture, efficient algorithm:

- **SpWA Dataflow.** We propose a new dataflow for SpWA by rearranging the data layout in sparse Winograd convolution. By rearrangement, we transform the EWMM-accumulation to sparse vector-matrix multiplications.
- **SpWA Architecture.** Based on SpWA dataflow, we present an efficient architecture to perform sparse Winograd convolution using line buffer and sparse Winograd PE designs.

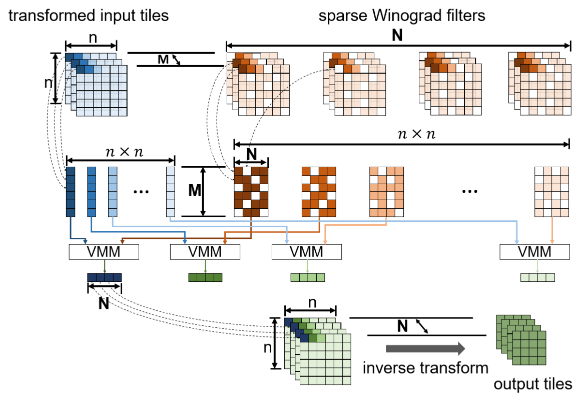

Figure 3: Dataflow of SpWA

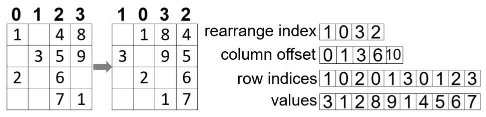

Figure 4: re-CSC format illustration

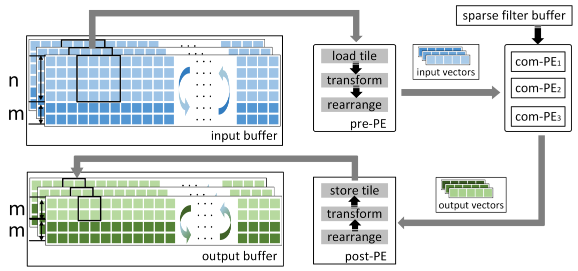

Figure 5: Architecture overview

- **efficient Algorithm.** We propose an efficient algorithm based on dynamic programming to minimize the total idle cycles of multiplication of different PEs.

4 SPWA DATAFLOW

After the input tile $Z_{tile}(t, i, j)$ and filter $f(k, t)$ are transformed to $V_w(t)$ and $U_w(k, t)$ in Winograd domain, they perform multiple EWMM operations as shown in Equation 6. Then, an $n \times n$ tile $Q_w(k)$ is generated by accumulating the EWMM results from all the input channels. Previous architectures [4, 15, 17] that target dense Winograd convolution are not efficient for sparse models. This is because these architectures performed the multiplications usin the EWMM operation in parallel, which is inefficient when the Winograd filters have unbalanced non-zeros distribution.

To address this problem, we present SpWA dataflow where we transform EWMM-accumulation operations to vector-matrix multiplication (VMM) operations. Figure 4 shows the dataflow of SpWA. We first batch M channels of transformed input tiles up before EWMM-accumulation operation. Then, we rearrange M channels of the transformed tiles to $n \times n$ vectors with the length of M, where $V_w(t, i, j)$ means the pixel in the tile $V_w(t)$ at the coordinates (i, j) and $V'_w(i * n + j, t)$ means the pixel in the vector $V'_w(i * n + j)$ at the coordinates t.

$$V'_w(i * n + j, t) = V_w(t, i, j) \tag{7}$$

Similarly, we rearrange the sparse filters from $N \times M$ matrices to $n \times n$ matrices as follows, where $U_w(k, t, i, j)$ means the pixel in the tile $U_w(k, t)$ at the coordinates (i, j) and $U'_w(i * n + j, k, t)$ means the pixel in the matrix $U'_w(i * n + j)$ at the coordinates (k, t).

$$U'_w(i * n + j, k, t) = U_w(k, t, i, j) \tag{8}$$

In this manner, EWMM-accumulation operation in Equation 6 is transformed to $n \times n$ vector-matrix multiplication (VMM) operations as follows, where $p \in [0, n^2 - 1]$,

$$Q'_w(p) = U'_w(p)V'_w(p) \tag{9}$$

Then we inversely rearrange the $n \times n$ vectors resulting from VMM to $Q(k)$ of Winograd domain as follows,

$$Q'_w(i * n + j, k) = Q_w(k, i, j) \tag{10}$$

Finally, N channels of the output tiles are generated by transforming $Q_w(k)$ to the spatial domain. In sparse Winograd convolution, the matrix $U'_w(p)$ in Equation 9 is sparse which can be obtained after training. In SpWA, we propose rearranged Compressed Sparse Column (re-CSC) format to store the sparse matrix $U'(p)$ as shown in Figure 4. In CSC format [1], values are read column by column with a row index stored for each value, which can reduce the memory requirement from $O(M \times N)$ to $O(2 \times nonzeros + 2N)$. In re-CSC format, the sparse matrix is first rearranged according the number of nonzeros in the columns. Then the rearranged matrix is compressed in CSC format together with the index of rearrangement.

5 SPWA ARCHITECTURE DESIGN

5.1 Architecture Overview

Figure 5 shows an overview of SpWA architecture, which mainly consists of pre-processing element (pre-PE), computing processing element (com-PE) and post-processing element (post-PE). The input and output feature maps are transferred to on-chip buffer via a FIFO. The pre-PE first loads an $n \times n$ tile from the input buffer channel by channel, then transforms and rearranges it to several vectors. com-PEs receive the input vector from pre-PE and load the weights from the sparse filters buffer, then calculates the output vector. The post-PE receives vectors from the com-PEs and rearranges them to several tiles. Last, post-PE transforms these tiles to the output feature maps.

5.2 Line Buffer Design

On-chip memory of FPGA is not large enough to store the entire feature maps, so we split the feature maps in the channel dimension with the factor 32. In Winograd convolution, the input tile slides with a stride of m, which results in $(n - m) \times n$ data reuse between two neighboring tiles. To increase data reuse opportunities and overlap transfer time with convolution operations, we store $(n + m)$ lines of the input feature maps and $2m$ lines of the output feature maps as shown in Figure 5. Specifically, when calculating the first n lines in the input buffer, the next m lines of the input feature maps are being loaded to input buffer. In the next iteration, the first m lines in the input buffer will load the input feature maps.

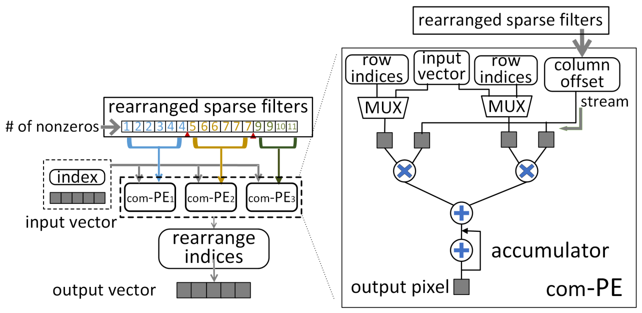

Figure 6: com-PE Architecture

5.3 PE Architecture

Figure 6 shows the architecture of com-PEs. In re-CSC format, the row indices are discontinuous, which leads to the irregular access pattern of the input vector. Therefore, multiplexers are inserted to select certain pixel in the input vectors. The values in the column are continuous in re-CSC format, so the weights are fed into com-PEs in stream. Then the selected input pixel and the weights from sparse filters perform multiplications and the multiplication results are added via an adder tree. At the end of the com-PE, there is an accumulator to update the final output pixel. In re-CSC format, the sparse matrix is rearranged by sorting the columns according to the number of nonzeros, to balance the workload of each com-PE, the matrix is partitioned into several groups with each group corresponding to a PE. All com-PEs work in parallel and each column in the same group is computed in parallel. The details of how to partition the columns into groups will be discussed in Section 6. After all com-PEs complete computation, the output pixels are concatenated and rearranged to the final output vector according to the rearrange index as shown in Figure 4.

Based on our PE architectures, we employ two level pipeline design: intra-PE pipeline and inter-PE pipeline. Intra-PE pipeline. For pre/post-PEs, the process for the tiles is pipelined via different channels. Specifically, the execution of transformation for the i^{th} channel is overlapped with the operation that loading the tile from the $(i+1)^{th}$ channel. Inter-PE pipeline. We use double buffers among pre-PE, com-PE and post-PE to overlap transformation operations and computations.

6 EFFICIENT ALGORITHM

In the implementation of SpWA, double buffer design is used between PEs and the latency of pre/post PEs is relatively short compared to com-PEs. So to improve the performance of SpWA, we focus on minimizing the latency of com-PEs. In Figure 6, the sparse matrix is partitioned into $\mathcal{T}$ groups and each group corresponds to a com-PE. All com-PEs work in parallel and the allocation of multipliers for each com-PE is proportionate to the maximal nonzeros of the column in the corresponding group. Each column in the PE is computed in parallel with the same number of multipliers, so the latency of a com-PE is always bounded by the column with the maximal nonzeros. For example, in Figure 7, the black bar means the number of nonzeros of the column in a rearranged sparse matrix. In Figure 7(d), the matrix is partitioned into 4 groups by 5 points k_0, k_1, k_2, k_3, k_4. The columns from the first to the sixth are in the first group with one multiplier for each column. Assuming it takes

Table 1: Parameter Description

M, N	height of the matrix, width of the matrix
Num	the number of rearranged sparse matrices
$nr[Num][N]$	the number of nonzeros in each column
$\mathcal{T}$	the number of groups
k_i	partition point where $k_0 = 1$, $k_T = N$
s_{ij}	the shaded area summation between point i and j

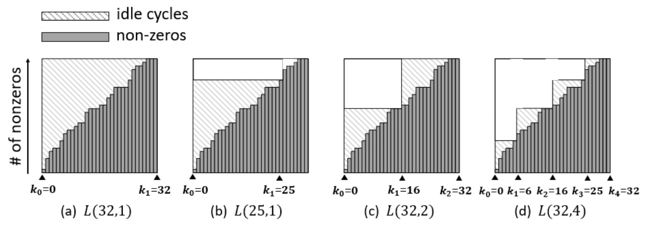

Figure 7: Different cases in dynamic programming: (a) the end point is 32 **and there is one group; (b) the end point is** 25 **and there is one group; (c) the end point is** 32 **and there are two groups; (d) the end point is** 32 **and there are four groups.**

one cycle for a multiplier to perform one multiplication, it takes 1 cycle for the first column to finish computation, and the sixth column needs 9 cycles, which leads to 8 idle cycles in the multipliers for the first column totally. As a consequence, the irregular distribution of nonzeros in the columns leads to imbalanced workload for each com-PE. To solve the problem, we propose an algorithm to minimize the total idle cycles. The parameter descriptions are shown in Table 1.

DEFINITION 1. *Given* ***Num*** *rearranged sparse matrices with the size of* ***M***×***N*** *and* ***nr[Num][N]*** *denoted as the number of nonzeros in each column for* ***Num*** *matrices, we define the* s_{ij} *as the idle cycles between between point i and point j.*

For example, in Figure 7(d) the total idle cycles between point k_0 and k_1 is 20 (= 8 + 5 + 3 + 2 + 2). For all Num matrices, s_{ij} can be formulated as follows,

$$s_{ij} = \sum_{k=1}^{k \le Num} (nr[k][j] * (j - i) - \sum_{t=i+1}^{t \le j} nr[k][t]) \tag{11}$$

Our goal is to minimize the total idle cycles, so the partition problem can be described as follows,

PROBLEM 1. *Given* ***Num*** *rearranged sparse matrices with the size of* ***M*** × ***N***, ***nr[Num][N]****, the goal is to find an partition strategy* $\boldsymbol{k}_0, \boldsymbol{k}_1, \boldsymbol{k}_2 \ldots \boldsymbol{k}_T$ *to minimize the total idles.*

We develop a dynamic programming algorithm to solve Problem 1. Ideally, the total idle cycles will drop as we increase $\mathcal{T}$. However, the hardware complexity will also increase with $\mathcal{T}$. To strike a balance, we set $\mathcal{T}$ to 4 in this paper. Let $L(e, l)$ represent the partition strategy with minimal idle cycles, where e is the endpoint in the matrix meaning that the problem interval is $[0, e]$ and l is the number of groups between $[0, e]$. Figure 7 shows some examples of $L(e, l)$. In Figure 7, the gray area means the idle cycles of multipliers caused by the discrepancy between different numbers of nonzeros in a group.

When finding the solution $L(e, l)$, as long as the last previous point is determined, the problem shrinks to find the solution $L(e.pre, l - 1)$.

Observing that, we derive the following recursion formula,

$$L(e, l) = \begin{cases} s_{0e} & if\, l = 1 \\ \min\limits_{0<i<e} \{L(i, l-1) + s_{ie}\} & if\, l > 1 \end{cases} \quad (12)$$

Using the recursion formula, we can solve the Problem 1 using dynamic programming as shown in algorithm 1.

Algorithm 1: Algorithm for Problem 1

```
Input: $T$, s[N][N]
Output: $k[T]$
1  for $int\ i = 1; i \le N; ++i$ do
2      $L[i][1].sum = s[0][i]$
3      $L[i][1].pre = -1$
4  for $int\ i = 2; i \le T; ++i$ do
5      for $int\ j = i; j \le N; ++j$ do
6          $L[j][i].sum = \infty$
7          $L[j][i].pre = -1$
8          for $int\ k = i - 1; k < j; ++k$ do
9              if $L[j][i].sum < L[k][i-1].sum + s[k][j]$ then
10                 $L[j][i].sum = L[k][i-1].sum + s[k][j]$
11                 $L[j][i].pre = k$
12 $int\ pre\_e = N$
13 for $int\ i = T - 1; i \ge 1; --i$ do
14     $pre\_e = L[pre\_e][i+1].pre$
15     $k[i] = L[pre\_e][i]$
```

In algorithm 1, $L(e, l).sum$ means the summation of the shaded area and $L(e, l).pre$ means the previous partition point of e. we first set the initial cases that the group size is 1 with endpoint ranging from 1 to N (Line 1-3). When the endpoint and group numbers are determined, we search the previous partition point that minimizes the summation of the shaded area then mark the point (Line 9-11).

7 EXPERIMENTAL EVALUATION

7.1 Experiments Setup

In this work, we first use Xilinx Vivado HLS (v2017.1) tool chain to transform C code into RTL implementation. Then, we employ Xilinx SDSoC (v2017.1) to compile the source code into bitstream. We evaluate our techniques on Xilinx ZC706 platform. ZC706 platform consists of a Kintex-7 FPGA and dual ARM Cortex-A9 processors. The external memory is 1GB DDR3. The bandwidth between on-chip memory and external memory is 4 GB/s. We use 16-bit fixed data type in our design. The operating frequency of our implementation is 166MHz. In the experiments, we compare our design with [15], the state-of-art Winograd implementation on FPGAs. However, [15] ignores sparsity, which may cause unbalanced execution in the pipeline. We profile the Winograd kernel execution time using Xilinx SDSoC (v2017.1).

7.2 Experiments of Synthetic Sparse Matrices

In this subsection, we test the performance by using synthetic sparse matrices. We use a typical input feature map size: 224×224 with $M = N = 32$ and the stride is 1. We randomly generate the sparse matrices with different sparsity and deviation. Here, the deviation refers to the standard deviation of the number of nonzeros in each column. Since the maximal number of nonzeros in a column is 32, we set three deviation values $\{3, 7, 10\}$.

Table 2: Resource utilization

	BRAM18K	DSP48E	FF	LUT
[15]	540(50%)	532(59%)	91874(21%)	89628(41%)
Ours	732(67%)	768(85%)	153020(35%)	155206(71%)

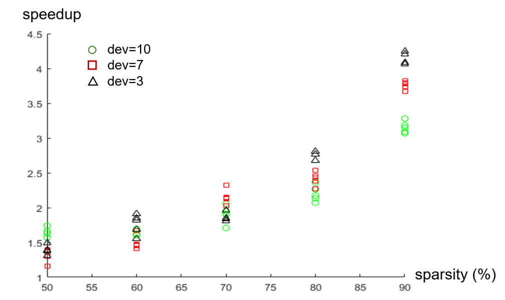

Figure 8: Speedup of sparse Winograd implementation

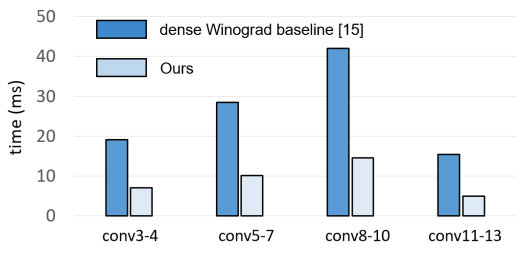

Figure 9: Comparison between prior work and ours on VGG16

Figure 8 shows the comparison between SpWA and previous Winograd implementation on FPGAs[15]. When sparsity is low, the speedup among different deviations shows almost the same. Because the patterns of nonzeros in the columns are always continuous, different cases can show the similar speedup after partition. When sparsity becomes higher, there are many zeros. A lower deviation case can achieve higher performance after partition than a higher deviation case. Because a high deviation shows less continuous.

7.3 Case study of VGG16

We first compare our design to state-of-the-art work proposed by [15] using VGG16 [20]. VGG16 consists of 5 convolution groups with different input size and all convolutional layers use 3×3 filters. We implement [15] and our design using $F(2 \times 2, 3 \times 3)$ Winograd. The sparsity is pruned to 80% based on [12]. As shown in Figure 9, we achieve 2.9x performance speedup on average.

Table 2 shows the resource utilization of our design and [15]. In [15], Look Up Tables (LUTs) are mainly used to implement constant multipliers and adders for transformation, since multiple tiles are transformed together. In our design, the transformation in pre/post-PE is conducted tile by tile, LUTs are mainly used to implement multiplexers to fetch the address in com-PEs. The difference of DSP utilization comes from different parallel computing strategies. Our design requires more BRAMs, because the filters are stored in transform format and additional memories are required to store indices in re-CSC format.

7.4 Case study of YOLO Network

You only look once (YOLO) is a state-of-the-art network for real-time object detection system [18]. We use Tiny-YOLO version to evaluate our design. Tiny-YOLO consists of 9 convolutional layers and 6 max pooling layers. All convolutional layers use 3×3 filters.

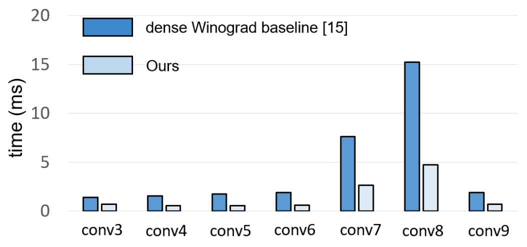

Figure 10: Comparison between prior work and ours on YOLO network

Figure 10 shows the comparison results. The speedup in the first few layers is not significant. Because the number of channels is small $(16, 32, 64)$, which means a small sparse matrix in Equation 9. In SpWA architecture, we apply pipeline technique in com-PEs. When the workload is small, the latency of com-PE can be bounded by the length of pipeline. For the rest layers, they have higher performance speedup than other layers. This is because these layers are with more channels, leading to more effective design when pipeline is enabled. On average, our design shows a 3.1x speedup.

8 RELATED WORK

The pursuit of faster deep learning FPGA accelerators never stops, there have been many efforts to explore different architectures for CNNs.

Architecture for dense CNNs using spatial convolution. Prior efforts to accelerate CNNs using spatial convolution have shown substantial successes on FPGAs. Zhang [25] et al. proposed a design space exploration technique to optimize the throughput from computation resources and bandwidth aspects. Wei et al. [23] implemented CNN on an FPGA using a systolic array architecture, which can achieve high clock frequency under high resource utilization. Zhang et al. [27] proposed a performance model that optimized the OpenCL kernels to efficiently utilize the hardware resources.

Architecture for dense CNNs using fast algorithm. Recently, the implementations of fast algorithms have been explored on FPGAs. Zhang et al. [26] implemented FFT with the size of 8 on FPGA platform for CNN. Ko et al. [11] proposed an FFT-based architecture for CNN model, which can be used for training and inference process. In [11] design, the training process is completed in frequency domain. Podili et al. [17] proposed a novel data layout to reduce the required memory bandwidth in the implementation of Winograd. Aydonat et al. [4] applied Intel OpenCL tool chain to map 1-D Winograd algorithm on Arria 10 FPGA platform. Lu et al. [15] proposed a performance model for Winograd convolution to optimize implementations and predict resource utilization.

Architecture for sparse CNNs using spatial convolution. Recently, there are a few accelerators that exploit sparsity of CNNs on ASICs. Eyeriss [6] gated computation cycles for zeros in the input feature maps to save energy and the data is stored in compress format in DRAM. Han et al. [10] proposed EIE CNN accelerator which operated directly on compressed networks and enables the large neural network models to fit in on-chip SRAM. EIE exploits sparsity both in input feature maps and filters but only focused on the fully-connected layer. Parashar et al. [16] proposed SCNN accelerator based on Cartesian-product operation in which all nonzeros have to be multiplied with one another. Zhang et al. [28] presented Cambricon-X accelerator which applied step indexing techniques. In Cambricon-X design, the nonzeros in the same row are divided into multiple segments with the same size in subsequent addresses.

9 CONCLUSIONS

In this work, we propose sparse Winograd CNNs accelerator (SpWA) on FPGAs which exploits Winograd fast algorithm and sparsity. We first present the dataflow of SpWA in which we rearrange data order. Based on the dataflow, we design an efficient architecture, which employs line buffer design and sparse Winograd PEs. Then we develop an efficient algorithm based on dynamic programming to guide our implementation strategy. Finally, we evaluate our design on Xilinx ZC706 platform and the results show a 2.9x∼3.1x speedup compared with state-of-the-art work.

ACKNOWLEDGMENTS

This work is supported by Beijing Natural Science Foundation (No. L172004) and National Science Foundation China (No. 61672048).

REFERENCES

[1] https://en.wikipedia.org/wiki/Sparse_matrix. 2018.
[2] Intel Math Kernel Library. https://github.com/01org/mkl-dnn. 2018.
[3] NVIDIA CuDNN. https://developer.nvidia.com/cudnn. 2018.
[4] U. Aydonat et al. An OpenCL Deep Learning Accelerator on Arria 10. In *FPGA*, 2017.
[5] A. Canis et al. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *FPGA*, 2011.
[6] Y.-H. Chen, J. Emer, and V. Sze. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *ISCA*, 2016.
[7] J. Cong et al. High-Level Synthesis for FPGAs: from Prototyping to Deployment. In *TCAD*, 2011.
[8] S. Han et al. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. In *ICLR*, 2015.
[9] S. Han et al. Learning both Weights and Connections for Efficient Neural Network. In *NIPS*, 2015.
[10] S. Han et al. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *ISCA*, 2016.
[11] J. H. Ko et al. Design of an Energy-Efficient Accelerator for Training of Convolutional Neural Networks using Frequency-Domain Computation. In *DAC*, 2017.
[12] S. Li et al. Enabling Sparse Winograd Convolution by Native Pruning. In *arXiv preprint arXiv:1702.08597*, 2017.
[13] Y. Liang et al. High-Level Synthesis: Productivity, Performance, and Software Constraints. In *Electrical and Computer Engineering*, 2012.
[14] B. Liu et al. Sparse Convolutional Neural Networks. In *CVPR*, 2015.
[15] L. Lu et al. Evaluating Fast algorithms for Convolutional Ceural Networks on FPGAs. In *FCCM*, 2017.
[16] A. Parashar et al. SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks. In *ISCA*, 2017.
[17] A. Podili, C. Zhang, and V. Prasanna. Fast and Efficient implementation of Convolutional Neural Networks on FPGA. In *ASAP*, 2017.
[18] J. Redmon et al. You Only Look Once: Unified, Real-Time Object Detection. In *CVPR*, 2016.
[19] B. C. Schafer et al. Machine Learning Predictive Modelling High-Level Synthesis Design Space Exploration. In *IET computers & digital techniques*, 2012.
[20] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *arXiv preprint arXiv:1409.1556*, 2014.
[21] N. Suda et al. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In *FPGA*, 2016.
[22] S. Wang et al. FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs. In *DAC*, 2017.
[23] X. Wei et al. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *DAC*, 2017.
[24] Q. Xiao et al. Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs. In *DAC*, 2017.
[25] C. Zhang et al. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In *FPGA*, 2015.
[26] C. Zhang and V. Prasanna. Frequency Domain Acceleration of Convolutional Neural Networks on CPU-FPGA Shared Memory System. In *FPGA*, 2017.
[27] J. Zhang et al. Improving the Performance of OpenCL-based FPGA Accelerator for Convolutional Neural Network. In *FPGA*, 2017.
[28] S. Zhang et al. Cambricon-X: An Accelerator for Sparse Neural Networks. In *MICRO*, 2016.