SDCNN: An Efficient Sparse Deconvolutional Neural Network Accelerator on FPGA

Jung-Woo Chang, Keon-Woo Kang and Suk-Ju Kang
Dept. of Electronic Engineering, Sogang University, Seoul, South Korea
{zwzang91, kkw0526, sjkang}@sogang.ac.kr

***Abstract*— Generative adversarial networks (GANs) have shown excellent performance in image generation applications. GAN typically uses a new type of neural network called deconvolutional neural network (DCNN). To implement DCNN in hardware, the state-of-the-art DCNN accelerator optimizes the dataflow using DCNN-to-CNN conversion method. However, this method still requires high computational complexity because the number of feature maps is increased when converted from DCNN to CNN. Recently, pruning has been recognized as an effective solution to reduce the high computational complexity and huge network model size. In this paper, we propose a novel sparse DCNN accelerator (SDCNN) combining these approaches on FPGA. First, we propose a novel dataflow suitable for the sparse DCNN acceleration by loop transformation. Then, we introduce a four stage pipeline for generating the SDCNN model. Finally, we propose an efficient architecture based on SDCNN dataflow. Experimental results on DCGAN show that SDCNN achieves up to 2.63 times speedup over the state-of-the-art DCNN accelerator.**

I. Introduction

Convolutional neural networks (CNNs) have been widely used in various computer vision applications with high level of accuracy. However, it depends on the fairly large amount of labeled training data. Generative adversarial networks (GANs), which produce novel data samples from high-dimensional data distributions, emerge as a solution [1]. GANs usually consist of a generator and discriminator, which compete with each other for learning data distributions. The generator produces similar data so that real and fake samples cannot be distinguished. On the other hand, the discriminator distinguishes real samples from fake samples produced by the generator.

Currently, research on FPGA-based CNN acceleration has attracted much attention in various fields [2], [3]. However, there is a lack of research on FPGA-based acceleration for GAN. Especially, the generator used for the GAN inference consists of deconvolutional neural networks (DCNNs), which are different from existing CNN structures. Therefore, to accelerate GAN inference, we should consider a fundamentally different type of mathematical operator, deconvolution.

Chang *et al*. [3] proposed a method of transforming the deconvolutional layer to the convolutional layer (TDC), which is a new computational methodology for the deconvolutional layer processor. Although the TDC method increases the computational efficiency over conventional deconvolution operations, it has the disadvantage of increasing the number of feature maps. Pruning [4] has been recognized as an effective solution for reducing the computational complexity of CNNs and huge network model size. However, the efficiency of the accelerator depends on the sparse structure of the 4D weight tensor [5]. There are different sparse structures depending on the two pruning methods. First, weight pruning [4] creates an irregular sparse pattern by removing some weights in the filters. However, it can cause load imbalanced workload between processing elements (PEs) because the ratio of non-zeros is different for each filter [6]. In contrast, filter pruning [7] removes weights while maintaining the regularity of the weight tensor. It has the advantage of bringing regular sparsity patterns that are more effective for hardware acceleration, but it cannot reduce the number of spatial filtering operations. Therefore, we must find the best sparse structure of weight tensor for DCNN acceleration.

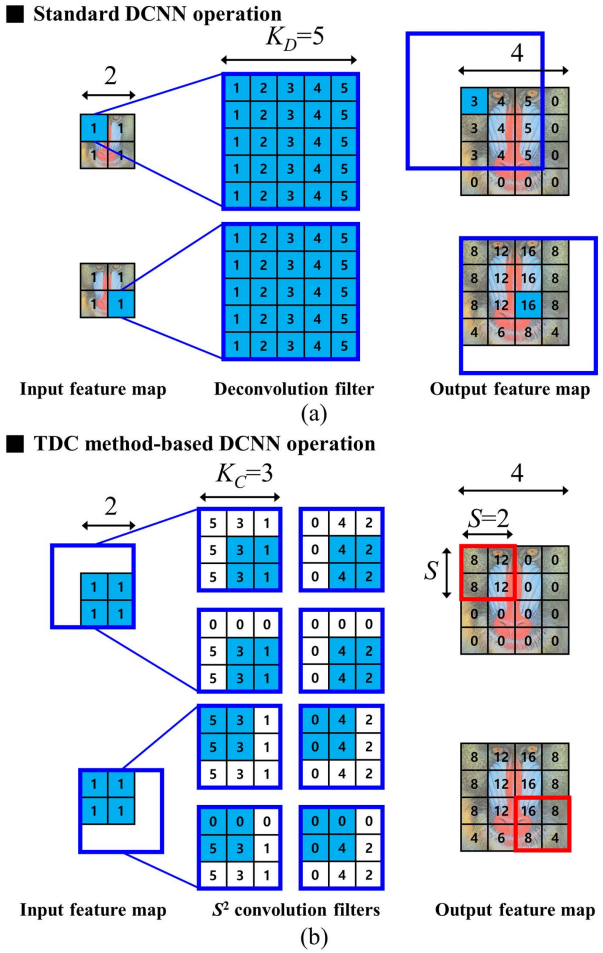

Fig. 1. Examples of DCNN implementations for each existing computational methodology: (a) Standard DCNN operation and (b) TDC method-based DCNN operation [3]. The red bounding box indicates the $S\times S$ output block. All output pixels within $S\times S$ output block can be created simultaneously.

In this paper, we propose a sparse DCNN accelerator (SDCNN) for DCNN inference exploiting TDC method and filter sparsity. The main contributions of this paper are as follows.

- We propose a specific dataflow suitable for the sparse DCNN acceleration by loop transformation. In this new dataflow, we group filters with similar distribution of non-zero weights to minimize total idle cycles of the PEs.

- We introduce a four stage pipeline that converts the DCNN model into an acceleration-efficient model by transforming, compressing, and encoding.
- We propose a novel architecture based on the SDCNN dataflow for GAN inference.

II. Background

A. Deconvolutional neural network (DCNN)

The deconvolution operation is used to extend a small-sized input image to a larger-sized output image. Using this characteristic, DCNN reconstruct the 3D feature maps by repeatedly performing 2D deconvolution through the learned filters. Specifically, kernel-sized deconvolution results are added to the output feature maps at stride intervals.

Fig. 1(a) shows that a 4×4 output feature map is formed when a standard DCNN operation is performed on a 2×2 input feature map. K_D is a width of the deconvolution filter and S is a stride. Since S is set to 2, the resolution of the output feature map is two times greater than that of the input feature map.

B. TDC Method

The standard deconvolution operation adds nested portions of previously existing output feature maps. Therefore, when implemented as an accelerator, dataflow between PEs and memory is inefficient [3]. To solve this problem, the state-of-the-art accelerator effectively optimizes the dataflow through the TDC method using the new source of parallelism in the standard deconvolution operation.

Fig. 1(b) shows how the output feature map is generated through the TDC method-based DCNN operation. K_C is the width of the convolution filter created by the TDC method. As shown in Fig. 1(b), all pixels in the $S\times S$ output block can be created in parallel because there is no dependency between the inputs when generating each output. Fig. 1 demonstrates that standard deconvolution and TDC method-based deconvolution form the same output feature map.

Template-code 1 The mapping of DCNN to the tiling engine.

```
1:  Deconvolution ():
2:  for (row=0; row<R; row+=Tr){
3:    for (col=0; col<C; col+=Tc){
4:      for (to=0; to<S^2×M; to+=Tm){
5:        for (ti=0; ti<N; ti+=Tn){
6:          // Load input feature maps & weights
7:          compute()
8:          // Store output feature maps
9:  }}}}
```

Template-code 2 The tiled accelerating engine for DCNN.

```
1:  compute ():
2:  for (i=0; i<K_C; i++){
3:  for (j=0; j<K_C; j++){
4:    for (r=0; r<Tr; r++){
5:    for (c=0; c<Tc; c++){
6:  #pragma HLS PIPELINE
7:      for (m=0; m<Tm; m++)
8:  #pragma HLS UNROLL
9:      for (n=0; n<Tn; n++)
10: #pragma HLS UNROLL
11:         f = m mod S^2
12:         fi =floor( f / S)
13:         fo = f mod S
14:         out[m][r×S+fi][c×S+fj]+=weight[S^2×m+S×fi+fj][n][i][j]
            ×in[n][r+i][c+j]
15: }}}}
```

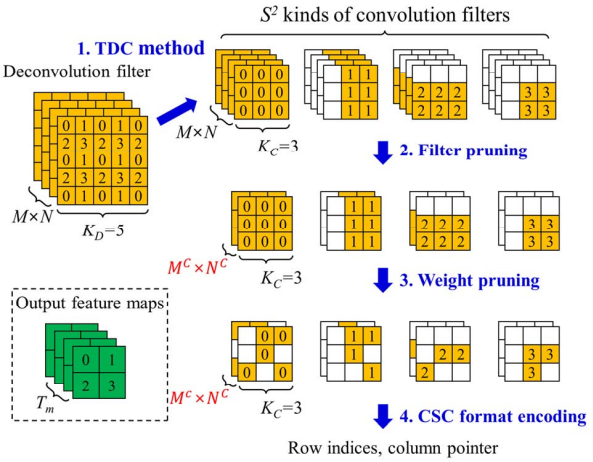

Fig. 2. The four stage pipeline for SDCNN: TDC method, filter pruning, weight pruning, and CSC format encoding.

Recently, many FPGA-based accelerators have been developed using high-level synthesis (HLS) tools to cope with the rapidly evolving deep neural networks. Template-code 1 shows HLS-based C/C++ code for mapping of TDC method-based DCNN operation to the tiling engine. The loops for R, C, $S^2\times M$, and N are tiled with parameters Tr, Tc, Tm, and Tn. The number of output feature maps is increased by the square of S by the TDC method. Before executing the tiled-operation, inputs and weights are loaded from external memory. Then *compute*() performs a 2D convolution on tiled inputs and filters. Finally, the outputs are stored in external memory.

Template-code 2 shows the process accelerated by loop optimization techniques [2] for TDC method-based operations. f, fi and fj are indices indicating the position of an output pixel in the $S\times S$ output block. Thus, the S^2 kinds of filters generate an $S\times S$ output block by repeatedly using the same accelerating engine. However, when the weight pruning is applied to the S^2 kinds of filters, we observe that the pruned S^2 kinds of filters have different distributions with respect to the number of non-zero weights. In addition, conventional engine cannot skip operations including zero-weights. To solve these problems, we optimize the dataflow of the conventional tiling engine through loop transformation so that the operation of each filter with different granularity can be efficiently processed.

III. DCNN Compression

Fig. 2 shows the process of transforming, compressing, and encoding a DCNN model using a four stage pipeline. In Fig. 2, M^C and N^C are the number of compressed output feature maps and input feature maps. The weights in each filter are labeled with indices indicating the position of the output to be created.

As shown in Fig. 2, we first apply filter pruning to S^2 kinds of convolution filters created by the TDC method. However, since the generator learns by competing with the discriminator, the pre-trained discriminator can easily distinguish the fake image created by the pruned generator from the real image. This interferes with the retraining of the generator. Thus, we also reduce the accuracy of the pre-trained discriminator using pruning. Next, we retrain the entire network after filter pruning and repeat this process until the target compression ratio is

TABLE I
THE AVERAGE NUMBER OF NON-ZERO WEIGHTS IN S^2 KINDS OF FILTERS AND THE NUMBER OF COMPRESSED INPUT/OUTPUT FEATURE MAPS. (FP: FILTER PRUNING, WP: WEIGHT PRUNING)

Case		dense	P1	P2	P3	P4	P5	P6
FP (%)		0	0	15	30	45	60	75
WP (%)		0	75	60	45	30	15	0
Output index	0	9	2.30	2.63	3.13	3.86	5.42	9
	1	6	1.91	2.12	2.49	2.97	4.11	6
	2	6	1.88	2.09	2.43	2.94	4.06	6
	3	4	1.51	1.63	1.86	2.09	2.78	4
M^C		512	512	511	490	420	340	280
N^C		1024	1024	803	620	560	485	400

reached. After the filter pruning is completed, we perform the weight pruning with the same process as above. Finally, we encode the sparse DCNN model in Compressed Sparse Column (CSC) format. We set the ratios of filter pruning and weight pruning differently for each case to find the best sparsity pattern for DCGAN [8]. Since all layers of DCGAN have K_D and S of 5 and 2 respectively, K_C is always set to 3. Table I shows the average number of non-zero weights for each filter when compressing the first deconvolutional layer at different ratios. Since S is 2 and there are four kinds of filters, the number of output indices is 4.

As shown in Table I, the number of non-zero weights in each filter is much smaller than the square of K_C. If we use a dense accelerating engine like Template-code 2, there is a serious performance degradation. In addition, the latency of a PE is bounded by the maximum number of non-zero weights. Furthermore, the average number of non-zero weights varies by filter type. Thus, we should reduce idle cycles of the tiled accelerating engine considering the types of each filter. Table I also shows the number of compressed input and output feature maps. Compared to weight pruning, filter pruning makes it easier to design accelerator by creating regular sparse patterns.

Template-code 3 Proposed method for mapping to the tiling engine.

```
1:  Sparse-deconvolution ():
2:  …
3:      for (to=0; to<M^C; to+=Tm){
4:          for (ti=0; ti<N^C; ti+=Tn){
5:              // Load input feature maps & weights
6:              for (fi=0; fi<S; fi++){
7:                  for (fj=0; fj<S; fj++){
8:                      sparse-compute()
9:              }}
10:             // Store output feature maps
11: …
```

Template-code 4 Proposed tiled accelerating engine for DCNN.

```
1:  sparse-compute ():
2:  for (i=0; i<K_C; i++){
3:      for (r=0; r<Tr; r++){
4:      for (c=0; c<Tc; c++){
5:  #pragma HLS PIPELINE
6:          for (m=0; m<Tm; m++)
7:  #pragma HLS UNROLL
8:          for (n=0; n<Tn; n++)
9:  #pragma HLS UNROLL
10:             for (j=colptr[m][n][i]; j<colptr[m][n][i+1]; j++)
11:                 out[m][r×S+fi][c×S+fj]+=in[n][rowind[m][n][j]][i]
                    ×weight[S^2×m+S×fi+fj][n][rowind[m][n][j]][i]
12: }}}}
```

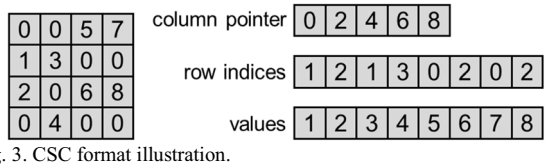

Fig. 3. CSC format illustration.

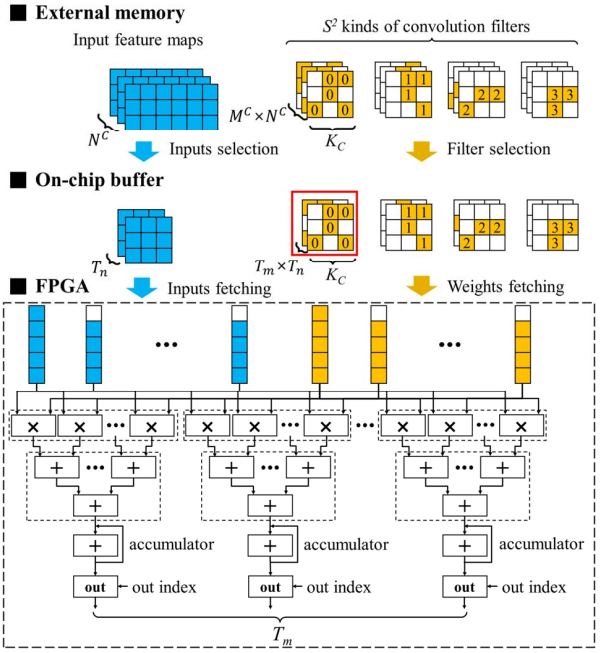

Fig. 4. Overview of our SDCNN architecture.

IV. SDCNN DESIGN

A. SDCNN Dataflow

Template-code 3 shows the C/C++ code mapping through the proposed method to the tiling engine. We split the loop for the $S^2 \times M$ in Template-code 1 into several pieces. In this manner, we process the convolution operations for the S^2 kinds of filters separately. The TDC method-based deconvolution generates S^2 outputs by operating convolutions with S^2 filters in the same input block. If we insert a split loop before loading the input feature maps, we must reload the previously fetched data. To improve the data reuse, we insert a split loop on lines 6-9. Template-code 3 also shows that the input and output feature maps are compressed by filter pruning.

The proposed tiled accelerating engine supports TDC method-based sparse deconvolution operation, as shown in Template-code 4. The *sparse-compute*() process a column-major convolution with column pointers and row indices expressed in CSC format. Fig. 3 illustrates a sparse matrix and its representation in the CSC format as an example.

The latency of the PE is limited by the maximum number of non-zero weights in each column. Especially, the sparsity structure between the filters is different and the patterns are irregular. To alleviate this problem, we process the S^2 kinds of filters separately on the accelerating engine. By balancing the load of each operation, the total idle cycles can be reduced.

B. SDCNN Architecture

Fig. 4 shows an overview of our SDCNN architecture. We first store CSC-formatted weights and input feature maps in

external memory. External memory transfers tile-sized data to the on-chip buffer for calculating the output vector. Through the proposed loop transformation, we implement each type of filter individually on the accelerating engine. The non-zero weights are stored in the weight buffer in column order, as shown in Fig. 4. Similarly, inputs that are in the same position as the non-zero weights are stored in the input buffer. Finally, the PEs receive the $Tm \times Tn$ tile from the on-chip buffer and parallelizes the input and output feature maps. We repeat this process for the remaining types of filters.

V. Experimental Results

We validated SDCNN on the Xilinx Virtex-7 485T FPGA. We converted C/C++ code to RTL implementation using Vivado HLS 2016.4 and simulated the SDCNN. We used floating-point precision in our design. To fairly compare with prior work, we implemented an existing DCNN accelerator [3] for DCGAN on the Virtex-7 485T FPGA. We set the optimal tile size among the possible design spaces through the roofline model [9]. Thus, Tm and Tn were set to 128 and 4. We used 2560 DSPs for the tiled accelerating engine and 294 BRAMs for the input/output buffers and encoded weights, respectively.

Fig. 5 shows the performance comparison between the conventional dataflow and the SDCNN dataflow in dense DCGAN. SDCNN achieved 1.4 times faster throughput on average than conventional accelerator. As shown in Fig. 5, in the 4th layer, the speed of SDCNN was 0.35 times lower than conventional accelerator. Because the number of output feature maps in the 4th layer is 1, all of the S^2 kinds of filters in the conventional dataflow are processed simultaneously, but in the proposed dataflow, the sparse filters are executed individually. Nevertheless, the proposed dataflow allows more efficient acceleration in sparse DCNN structures, as shown in Fig. 6.

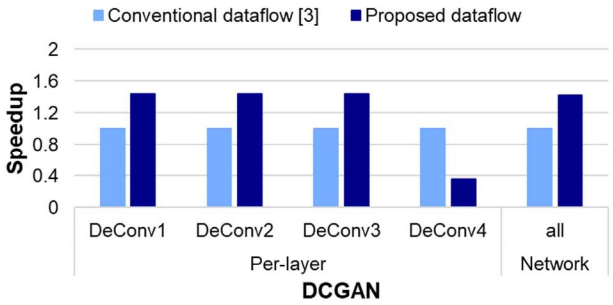

Fig. 5. Performance comparison between conventional dataflow and our proposed dataflow in dense DCNN on MNIST dataset.

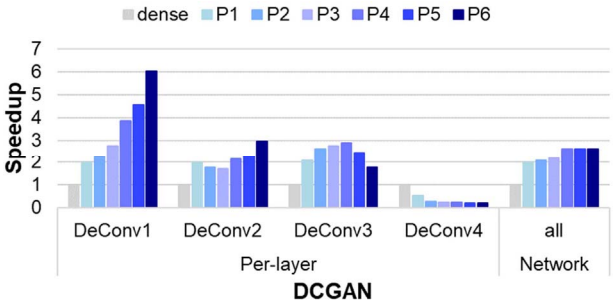

Fig. 6. Performance comparison between dense DCNN and SDCNNs with different sparse structures on MNIST dataset.

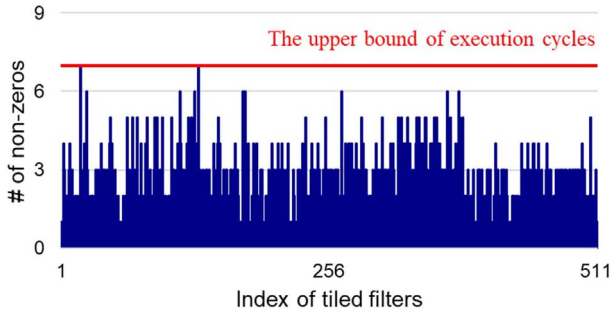

Fig. 7. Examples of cases where the number of non-zero weights is different for each of the tiled filters in sparse DCNN.

Fig. 6 shows the performance of SDCNNs according to the ratios of filter pruning and weight pruning shown in Table I. We observed that the performance of the SDCNN is relatively faster when the ratio of filter pruning is high. To be specific, Fig. 7 shows the unbalanced distribution of non-zeros within tiled filters at P1 case. Each column in the PE was processed in parallel with the same number of multipliers. In this manner, the upper bound of the latency for the PE was determined by the maximum number of non-zeros. Thus, there was a restriction on speedup in the case of P1 with a highest ratio of weight pruning. In contrast, filter pruning uniformly improved the throughput of the accelerating engine by reducing the number of feature maps. As a result, in the case of P5, SDCNN achieved 2.63 times faster than the state-of-the-art accelerator.

VI. Conclusion and Future Work

In this paper, we propose a sparse DCNN accelerator (SDCNN) exploiting TDC method and sparsity. We first propose a new dataflow suitable for sparse DCNN acceleration through loop transformation. Then we introduce a four stage pipeline that converts the dense model to the SDCNN model. Finally, we propose an architecture based on new dataflow. We implement our design on the Xilinx Virtex-7 485T and the results show 2.63 times speedup over the state-of-the-art work. In the future, we will evaluate our SDCNN for more GANs.

Acknowledgment

This work was supported by Samsung Electronics and the Korea Institute of Energy Technology Evaluation and Planning (KETEP) and the Ministry of Trade, Industry & Energy (MOTIE) of the Republic of Korea (No. 20161210200560).

References

[1] I. Goodfellow *et al.*, "Generative adversarial nets," in *NIPS*, 2014.
[2] C. Zhang *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," in *FPGA*, 2015.
[3] J.-W. Chang *et al.*, "Optimizing fpga-based convolutional neural networks accelerator for image super-resolution," in *ASP-DAC*, 2018.
[4] S. Han *et al.*, "Learning both weights and connections for efficient neural network," in *NIPS*, 2015.
[5] H. Mao *et al.*, "Exploring the granularity of sparsity in convolutional neural networks," in *CVPRW*, 2017.
[6] A. Parashar *et al.*, "SCNN: an accelerator for compressed-sparse convolutional neural networks," in *ISCA*, 2017.
[7] Y. He *et al.*, "Channel pruning for accelerating very deep neural networks," in *ICCV*, 2017.
[8] A. Radford *et al.*, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv*, 2015.
[9] S. Williams *et al.*, "Roofline: an insightful visual performance model for multicore architectures," *Commun, ACM*, 52(4):65-76, Apr. 2009.