Kernel-Segregated Transpose Convolution Operation

Vijay Srinivas Tida
University of Louisiana at Lafayette
vijaysrinivas.tida1@louisiana.edu

Sai Venkatesh Chilukoti
University of Louisiana at Lafayette
saivenkatesh.chilukoti1@louisiana.edu

Sonya Hsu
University of Louisiana at Lafayette
hsiu-yueh.hsu@louisiana.edu

Xiali Hei
University of Louisiana at Lafayette
xiali.hei@louisiana.edu

Abstract

Deep learning models having transpose convolution layers requires optimization to deploy in the resource constraint Internet of Things (IoT) devices. The main reason is the presence of zeros at predefined positions in the input feature maps after upsampling layer leads to higher memory and computation load requirements for transpose convolution operations. We propose an algorithmic-level optimization technique based on kernel segregation mechanisms for efficient transpose convolution implementation to address these issues without needing an upsampling layer. Experimental results showed that the proposed approach showed an average of $3.7 \times (3.4\times)$ *faster computation using an Intel Xeon CPU (RTX 2070 GPU) than the conventional method. Further, we analyzed the performance using different transpose convolution layers from the popular Generative Adversarial Network (GAN) models and a simple deep learning model with one transpose convolution layer. There is a significant improvement in computation speed and substantial memory savings from the obtained results.*

keywords:convolution, upsampling layer, transpose convolution, generative adversarial networks, kernel segregation

1. Introduction

GANs consist of two parts, namely, the generator and the discriminator. The transpose convolution layer is mainly used in the generator part, whereas the convolution layer is used in the discriminator part. The general overview of the convolution and transpose convolution is illustrated in Fig. 1. Applying the convolution operation on the input feature map will compress the output feature map. In contrast, the transpose convolution operation will expand the output feature map. The output feature map values will be obtained based on selecting the version of the transpose convolution. The transpose convolution with stride one will not be helpful in deep learning applications because of the checkerboard pattern (Zhou, 4/8/2022 accessed). This problem arises due to more values accumulating at the center pixels. Therefore, the transpose convolution layer with a combination of upsampling and convolution layers is used to avoid the checkerboard problem.

The transpose convolution layer implementation used in this paper is similar to the research of (Yazdanbakhsh et al., 2018b) and is the standard version used in many popular GANs. The upsampling layer transforms the input feature of size $N \times N$ by embedding zeros after each row and column. The transformation results in input feature map size to $(2N-1) \times (2N-1)$ after the upsampling process. Applying the convolution operation with a kernel size of n with stride one on the obtained feature map leads to an output feature map of size $(2N-n) \times (2N-n)$. Fig. 2 explains the basic transpose convolution operation with the input feature map of size 4×4 and a kernel size of 3×3. Unnecessary zeros obtained from the upsampled feature map in transpose convolution operation result in excessive data transfers, memory bottlenecks, and wastage of computing resources.

Prior research primarily focused on optimizing convolution and transpose convolution operations using hardware approaches (Dukhan, 2019; Yan et al., 2018; Yazdanbakhsh et al., 2018b, 2018a; Chang et al., 2018; Van Zee et al., 2015). These implementations require extra hardware, and some need upsampling layers for efficient transpose convolution implementation. To the best of our knowledge, we are the first to introduce the optimization algorithm for transpose convolution without using an upsampling layer.

The significant contributions of this paper are as follows:

a. We propose an optimized transpose convolution algorithm using a kernel segregation mechanism to reduce computation load and memory requirement without requiring specialized hardware.

b. We analyze the speed up in computation time and memory savings of our proposed approach using multiple datasets and the transpose convolution layers

URI: https://hdl.handle.net/10125/103474
978-0-9981331-6-4
(CC BY-NC-ND 4.0)

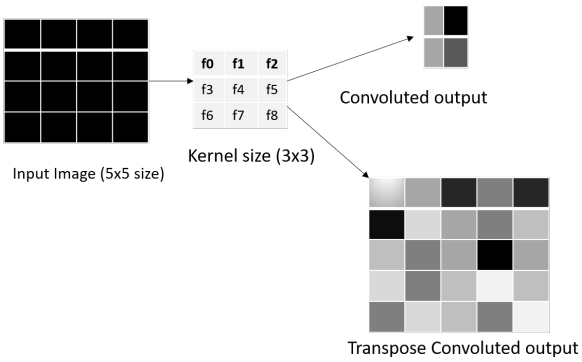

Figure 1. Conventional convolution and transpose convolution operations.

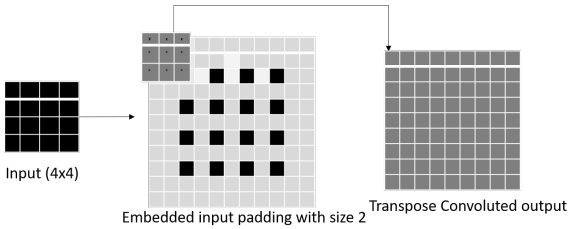

Figure 2. Transpose convolution operation with padding factor of 2.

from popular GANs.

c. We also analyzed the delay, area, and power consumption of the proposed model using Synopsys DC compiler with 45nm and 14nm technologies

d. We investigate the performance of the proposed optimization using a simple deep neural network having one transpose convolution layer. Our experimental results indicate a significant improvement in training time without needing an upsampling layer.

The rest of the paper is organized as follows: Section 2 explains the background, and literature review, whereas Section 3 explains the kernel segregation mechanism along with the optimized transpose convolution process. Section 4 interprets the results and finally, Section 5 concludes the paper with future research directions.

2. Background and Literature Review

The formula for the convolution using the 2D input array can be expressed in Equation 1 (*Speeding up Convolutions*, 2020).

$$out[i,j] = \sum_{u=1}^{n} \sum_{v=1}^{n} in[i+u][j+v] * k[u][v], \quad (1)$$

where the array out represents the output feature map with dimension $(N-n+1)\times(N-n+1)$, the array in represents the input feature map of size $N \times N$ and k represents the kernel of size $n \times n$. The element $out[i,j]$ denotes the output feature map's value at i^{th} row and j^{th} column. The variable $in[i+u][j+v]$ represents the input feature map's value at $(i+u)^{th}$ row and $(j+v)^{th}$ column and $k[u][v]$ represents the value of the kernel at u^{th} row and v^{th} column. The same equation is applicable for transpose convolution, but the input dimension will be $(2N-1)\times(2N-1)$ obtained after the upsampling process.

2.1. Algorithms for convolution operation

The implementation of the general convolution algorithm is clearly explained in (Anderson et al., 2020). GEMM-based algorithms use computations in the convolution operator as a GEneral Matrix Multiplication with highly optimized Basic Linear Algebra Subprograms (BLAS) (Goto et al., 2008; Van Zee et al., 2015). Many deep learning frameworks, including Tensorflow (Abadi et al., 2016), PyTorch (Paszke et al., 2017), and Caffe (Jia et al., 2014), use GEMM-based algorithms introduced in (Chellapilla et al., 2006). However, these algorithms need patch matrices that require more memory storage and bandwidth to perform convolution operations. Several works were proposed to reduce computation costs by reducing the multiplications required for convolution separately for CPU and GPU applications (K.Parhi, accesed 4/7/2022; Chen et al., 2018; Georganas et al., 2018; Heinecke et al., 2016; Zhang et al., 2018).

Fast convolution algorithms using Fourier or Winograd transformations were introduced (Vasilache & Johnson, 2014; Lavin et al., 2016). (Bhattacharya et al., 2016) used a separable convolution operation by converting 2D kernels into the row and column kernels for mobile and embedded platforms. (Wang et al., 2019) proposed a parallel convolution algorithm and showed their performance on multi-core CPUs. The performance evaluation shows a factor ranging from 1.0 to 5.17× than GEMM-based implementation. (Anderson et al., 2020) used smaller patches for computing convolution to reduce the memory overhead. Later, an indirect convolution algorithm helped to eliminate expensive and memory-intensive im2col transformations and replace the im2col buffer with a much smaller indirection buffer (Dukhan, 2019). Therefore, the general convolution method used for transpose convolution implementation is advantageous as it requires less memory than other optimized algorithms. Still, the computation time will be longer and beneficial for resource constraint devices due to lower memory requirements. Furthermore, this approach will make the backpropagation process

also easier for transpose convolution during the training process of neural networks. However, the proposed approach has a significant limitation: It does not support backward propagation for convolutional layers. Moreover, the proposed algorithms might not be efficient for transpose convolution implementation because of nearly 70% zeros embedded in the upsampled input feature map.

2.2. Hardware accelerators for transpose convolution operations

(Yazdanbakhsh et al., 2018b, 2018b) designed hardware accelerators using Application Specific Integrated Circuit (ASIC) and Field Programmable Gate Array (FPGA) for implementing transpose convolution efficiently. However, these hardware accelerators avoid unnecessary computations but demand more memory because of the upsampling layer. On the other hand, efficient implementation of transpose convolution was made using systolic arrays by Huynh *et al.* and filed a patent through Amazon Technologies (V. HUYNH, U.S. Patent WO/2021/061566, April. 2021). However, the authors didn't explain the usage of the proposed hardware for the backpropagation process using systolic arrays. Moreover, the proposed methods above require dedicated hardware that might not be easily available to researchers.

3. Methodology

3.1. Kernel segregation mechanism

This process involves segregating the original kernel into four sub-kernels based on the upsampled input feature map pattern. In the input feature map, zeros are usually embedded along each row and column after every element in a predefined manner, as shown in Fig. 3 after the upsampling process. Four common cases will arise when the original kernel slides through the input feature map. The red dots indicate that the values are zeros in the corresponding input feature map. The kernel elements are inactive at these positions and need to be discarded. An inactive state means that the multiplication operations will give zero at the related positions. The green dots indicate that the values are non-zero in the corresponding input feature map. The kernel elements that are in the active state should be considered for our segregation mechanism. An active state means that the multiplication operation is effective in these locations.

Note that we assume the indexing of elements starts at (0,0) on the input feature map. In the first case, as in Fig. 3a, only a combination of even row and even column elements from the original kernel are in the active state, and others are inactive. In the second case, as in Fig. 3b, only a combination of even row and odd column element operations from the original kernel is in the active state, and all other element positions are useless. In the third case, as in Fig. 3c, only a combination of odd row and even column elements is used for computation, and others remain unused. Similarly, as in Fig. 3d, only a combination of odd row and odd column elements is necessary for the fourth case, and the others remain unnecessary. We can indirectly perform four convolution operations on the same input feature map if four cases are appropriately analyzed. This significant observation will help design the optimization algorithm using kernel segregation. Finally, there will be some offsets based on the particular activation set. Here we ignored the padding effect and assumed the input elements started from the third row and third column. However, the above process still holds for different padding factors, but the order of cases might change. We will explain these offsets and the padding effect in Section 3.3.

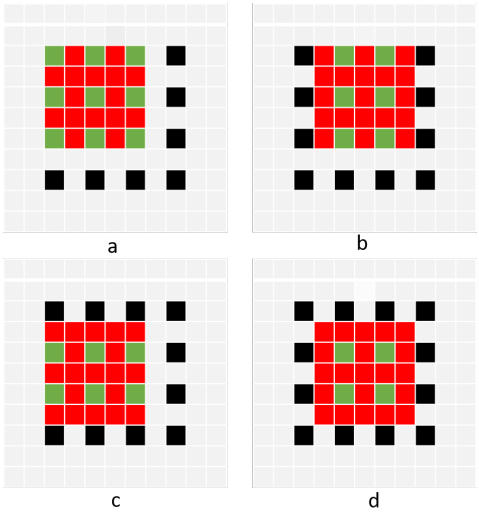

Figure 3. Transpose convolution operation. a), b), c), and d) parts show the computation pattern, throughout the input feature map.

3.2. Generalization of kernel segregation mechanism

We can apply the kernel segregation mechanism to any kernel size of $N \times N$ such that N is odd. The general matrix representation of four sub-kernels can be seen in Equations 3, 4, 5, and 6, respectively from original kernel of size N. The four sub-kernels K_1, K_2, K_3, K_4 are formed by accessing the corresponding locations from the original kernel K. To obtain the first sub-kernel

K_1, the values along with the alternate columns and alternate rows, which start from $(0,0)^{th}$ element, are accessed from the original kernel K. Similarly, the remaining three sub-kernels K_2, K_3, K_4 formed by accessing the elements starting with $(0,1)^{th}$, $(1,0)^{th}$, and $(1,1)^{th}$ elements of the original kernel K, respectively. These four sub-kernels will help perform the four convolution operations on the given input feature map based on the data patch taken each time. The final sizes of four sub-kernels will be $\lceil N/2 \rceil \times \lceil N/2 \rceil$, $\lceil N/2 \rceil \times \lfloor N/2 \rfloor$, $\lfloor N/2 \rfloor \times \lceil N/2 \rceil$, and $\lfloor N/2 \rfloor \times \lfloor N/2 \rfloor$, respectively. We use $N_{11} \times N_{12}$, $N_{21} \times N_{22}$, $N_{31} \times N_{32}$, and $N_{41} \times N_{42}$ as sizes for four segregated kernels. Here, $\lceil . \rceil$ represents the ceiling function and $\lfloor . \rfloor$ represents the floor function. However, the arrangement of elements will vary if an even ordered kernel is used and still follows the same process.

$$K = \begin{bmatrix} k_{00} & k_{01} & k_{02} & \cdots & k_{0(N-1)} \\ k_{10} & k_{11} & k_{12} & \cdots & k_{1(N-1)} \\ k_{20} & k_{21} & k_{22} & \cdots & k_{2(N-1))} \\ k_{30} & k_{31} & k_{32} & \cdots & k_{3(N-1)} \\ \vdots & \vdots & \cdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \cdots & \vdots \\ \vdots & \vdots & \cdots & \cdots & \vdots \\ k_{(N-1)0} & k_{(N-1)1} & k_{(N-1)2} & \cdots & k_{(N-1)(N-1)} \end{bmatrix} \quad (2)$$

$$K_{00} = \begin{bmatrix} k_{00} & k_{02} & \cdots & k_{0(N-1)} \\ k_{20} & k_{22} & \cdots & k_{2(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ k_{(N-1)0} & k_{(N-1)2} & \cdots & k_{(N-1)(N-1)} \end{bmatrix} \quad (3)$$

$$K_{01} = \begin{bmatrix} k_{01} & k_{03} & \cdots & k_{0(N-2)} \\ k_{21} & k_{23} & \cdots & k_{2(N-2)} \\ \vdots & \vdots & \ddots & \vdots \\ k_{(N-1)1} & k_{(N-1)3} & \cdots & k_{(N-1)(N-2)} \end{bmatrix} \quad (4)$$

$$K_{10} = \begin{bmatrix} k_{10} & k_{12} & \cdots & k_{1(N-1)} \\ k_{30} & k_{32} & \cdots & k_{3(N-1)} \\ \vdots & \vdots & \ddots & \vdots \\ k_{(N-2)0} & k_{(N-2)2} & \cdots & k_{(N-2)(N-1)} \end{bmatrix} \quad (5)$$

$$K_{11} = \begin{bmatrix} k_{11} & k_{13} & \cdots & k_{1(N-2)} \\ k_{31} & k_{33} & \cdots & k_{3(N-2)} \\ \vdots & \vdots & \ddots & \vdots \\ k_{(N-2)1} & k_{(N-2)3} & \cdots & k_{(N-2)(N-2)} \end{bmatrix} \quad (6)$$

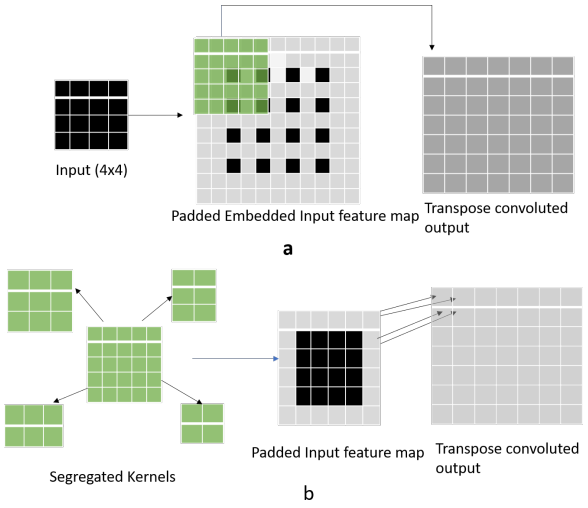

Figure 4. Comparison of the a) conventional transpose convolution with the b) proposed optimized technique.

3.3. Optimization of the transpose convolution operation using segregated kernels

The conventional transpose convolution and the proposed optimized transpose convolution process can be seen in Fig. 4. Fig 4a represents the input feature map of size 4×4 embedded with zeros, and a padding factor of 2 is applied. When a kernel of size 5×5 slides through the upsampled input feature map, its corresponding output values are obtained sequentially for the conventional method can be observed in Fig. 4a. However, using the proposed kernel segregation mechanism, four output values will be acquired using four sub-kernels can be seen in Fig. 4b. Also, the padding factor for the input feature using four segregated kernels will be different from the original padding factor. For example, if the original padding factor is P, the new padding factor will be $\lfloor P/2 \rfloor$.

Fig. 5 illustrates the process of the proposed optimized transpose convolution operation using the kernel segregation mechanism applied on an input feature map of size 4×4. Here the padding factor for the input feature map is reduced to 1 from 2 to obtain the exact output feature map from the transpose convolution operation. Next, the convolution operation is applied on the padded input feature map with four sub-kernels to produce four output values at different locations. The first two output locations and the last two output locations are adjacent. On the other hand, one can get the position for the second pair by adding a specific constant from the place of the first pair.

The optimized transpose convolution operation

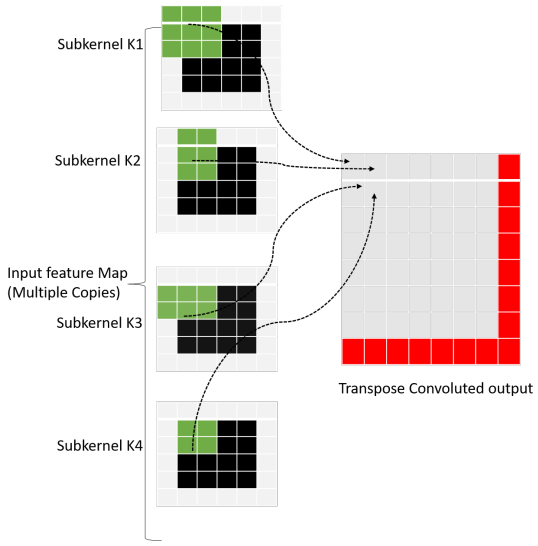

Figure 5. Optimized transpose convolution implementation using four sub-kernels with input size 4×4 and padding with factor 1 (but padding factor is 2 for the original case).

should show four times faster for the ideal case compared to the conventional approach with the same computation load. However, due to the offset problem related to computation in finding specific output locations, there might be some reduction in performance without considering padding and zero embedded time. If the output feature map is of an odd dimension, this continuous process will result in an extra column and row, as indicated in red in Fig. 5. The main reason for the problem is that the optimized algorithm will produce four output feature values in each iteration. The conditional statements can avoid unnecessary computation based on the user requirements. The formulas for calculating the four output feature values by applying the optimization process can be seen in the Equations 7, 8, 9, and 10.

$$out[2*i][2*j] = \sum_{u=1}^{N_{11}} \sum_{v=1}^{N_{12}} in[i+u][j+v] * K_1[u][v], \tag{7}$$

$$out[2*i][2*j+1] = \sum_{u=1}^{N_{21}} \sum_{v=1}^{N_{22}} in[i+u][(j+1)+v] * K_2[u][v], \tag{8}$$

$$out[2*i+1][2*j] = \sum_{u=1}^{N_{31}} \sum_{v=1}^{N_{32}} in[(i+1)+u][j+v] * K_3[u][v], \tag{9}$$

$$out[2*i+1][2*j+1] = \sum_{u=1}^{N_{41}} \sum_{v=1}^{N_{42}} in[(i+1)+u][(j+1)+v] * K_4[u][v], \tag{10}$$

where $out[l][m]$ represents the output feature map located at l^{th} row and m^{th} column; $in[i][j]$ represents the input feature map at the corresponding i^{th} row and j^{th} column; $K_1[u][v]$, $K_2[u][v]$, $K_3[u][v]$ and $K_4[u][v]$ represents the sub-kernels K_1, K_2, K_3 and K_4 obtained after segregation mechanism and their locations at u^{th} row and v^{th} row. The sizes of the corresponding four sub-kernels will be $N_{11} \times N_{12}$, $N_{21} \times N_{22}$, $N_{31} \times N_{32}$, and $N_{41} \times N_{42}$. Here the size of the input feature map will remain the same without upsampled values. The individual output feature map's dimensions depend on the size of the sub-kernels. Finally, the output feature map obtained from the proposed optimization should ensure the same dimensions when conventional transpose convolution is applied. If there are more output values than required, we should discard them.

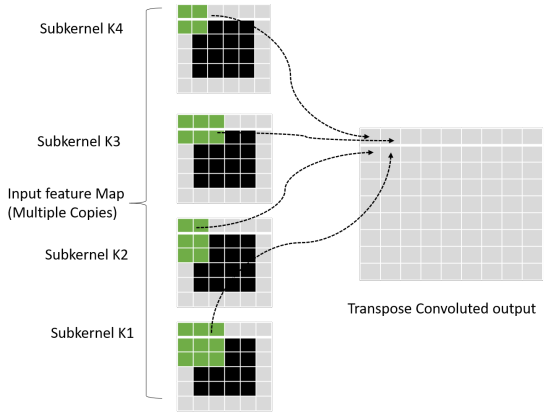

Figure 6. Optimized transpose convolution implementation using four sub-kernels with input size 4×4 and padding with factor 1 (but padding factor is 3 for original case).

Fig. 6 shows the proposed optimization technique when the padding factor is odd, and the kernel size of 5×5 is applied on the input feature map. The new padding factor for the input feature map will be one instead of three in the original case to apply the proposed optimization technique. The above exact process and the equations will still hold here, but the order of sub-kernels will change when the four convolution operations are made on the input feature map. The new set of sub-kernels will be K_4, K_3, K_2, and K_1 instead of K_1, K_2, K_3, and K_4 for this case. In deep learning applications, the proposed optimization technique can also be used to calculate the kernel and the input gradients during the backward propagation process. Since the proposed approach combines the upsampling and convolution layers, there will be a significant advantage in avoiding unnecessary input gradient computations.

Data group	Kernel	Computation time in seconds				Speedup	Speedup	Memory savings
		Conv (GPU)	Conv (CPU)	Prop (GPU)	Prop (CPU)	(GPU)	(CPU)	(Bytes)
Daisy	$5 \times 5 \times 3$	3.6233	61.388	1.018	15.714	3.559	3.906	1,824,320
	$4 \times 4 \times 3$	2.715	38.978	0.741	10.331	3.663	3.772	1,827,900
	$3 \times 3 \times 3$	1.7454	22.491	0.4916	6.098	3.550	3.6882	1,824,304
Dandelion	$5 \times 5 \times 3$	5.073	84.122	1.4929	21.496	3.39	3.913	1,824,320
	$4 \times 4 \times 3$	3.7712	53.573	1.043	14.006	3.615	3.825	1,827,900
	$3 \times 3 \times 3$	2.6008	30.978	0.6962	8.333	3.735	3.717	1,824,304
Rose	$5 \times 5 \times 3$	3.5505	63.06	1.0451	15.963	3.3972	3.9503	1824320
	$4 \times 4 \times 3$	2.736	39.945	0.7838	10.481	3.4906	3.8112	1,827,900
	$3 \times 3 \times 3$	1.9034	23.081	0.553	6.265	3.441	3.684	1,824,304
Sunflower	$5 \times 5 \times 3$	3.3974	58.809	1.0316	15.034	3.2933	3.9117	1824320
	$4 \times 4 \times 3$	2.564	37.438	0.7326	9.829	3.4998	3.8089	1,827,900
	$3 \times 3 \times 3$	1.6225	21.442	0.4756	5.867	3.4114	3.6546	1,824,304
Tulip	$5 \times 5 \times 3$	4.5116	79.113	1.4212	20.23	3.1749	3.9106	1,824,320
	$4 \times 4 \times 3$	3.5148	49.918	1.0202	13.651	3.4452	3.6567	1,827,900
	$3 \times 3 \times 3$	2.2988	28.734	0.6851	7.963	3.3554	3.6084	1,824,304

Table 1. Speedup for GPU and CPU versions and memory savings obtained for Flower dataset for the conventional (Conv) and the Proposed (Prop) approaches

Dataset	Kernel	Computation time in seconds				Speedup	Speedup
		Conv (GPU)	Conv (CPU)	Prop (GPU)	Prop (CPU)	(GPU)	(CPU)
MSCOCO 2017	$5 \times 5 \times 3$	151.49	951.71	39.55	242.252	3.83	3.928
	$4 \times 4 \times 3$	100.22	618.685	30.311	154.301	3.306	4.009
	$3 \times 3 \times 3$	60.543	352.297	17.973	92.268	3.368	3.818
PASCAL VOC 2012 (Classification)	$5 \times 5 \times 3$	210.735	1395.682	58.766	347.362	3.586	4.017
	$4 \times 4 \times 3$	144.471	873.157	43.636	226.842	3.310	3.849
	$3 \times 3 \times 3$	95.922	504.292	29.683	136.671	3.232	3.689
PASCAL VOC 2012 (Detection)	$5 \times 5 \times 3$	35.78	234.1	9.658	57.308	3.704	4.080
	$4 \times 4 \times 3$	25.07	144.72	6.973	37.163	3.595	3.894
	$3 \times 3 \times 3$	15.709	90.248	4.793	22.293	3.277	4.048

Table 2. Speedup for GPU and CPU versions for MSCOCO and PASCAL datasets using conventional (Conv) and proposed (Prop) approaches

4. Results

4.1. Datasets and the evaluation procedure

We considered the flower dataset from the Kaggle website (Mamaev, 4/8/2022 accessed), MSCOCO 2017 (*Common Objects in Context Dataset*, 8/26/2022 accessed), and PASCAL VOC 2012 (*Visual Object Classes Challenge 2012*, 8/26/2022 accessed) datasets to compare the computation times and memory savings for the conventional and proposed optimized approaches for transpose convolution operation. The flower dataset contains five subgroups of classes: sunflower, dandelion, daisy, rose, and tulip. The total number of images in this dataset is 4,323. The sunflower class contains 734; the tulip class includes 984; the daisy class contains 769; the rose class contains 784; and the dandelion class contains 1,052 color images. We considered only 10% of the available images, 11,828, from the MSCOCO 2017 dataset for the experimental analysis. Also, for the PASCAL 2017 dataset, we used both classification and segmentation datasets. The classification dataset contains 17,125 images, whereas the segmentation dataset contains 2,913 images of various sizes. For standard evaluation, all the images from the selected datasets are transformed into a standard format of $224 \times 224 \times 3$. We applied transpose convolution to the images and assessed the computation time using the conventional and the proposed methods. The programming languages used here were C++ and CUDA C for the CPU and GPU, respectively. The computation time and memory requirements are considered for evaluating the benefits of the proposed approach with the conventional implementation.

4.2. Analysis of computation time and memory savings

Compared to the conventional approach, speedup and memory savings from the proposed optimization process with the selected datasets can be seen in Tables 1 and 2. We used the Intel Xeon CPU and Nvidia GeForce RTX 2070 GPU for experimental analysis with GCC 9.4 and CUDA 10.2 versions, respectively. We

				Computation time in seconds				
Model	Layer #	Input Size	Kernel Size	Conv (GPU)	Prop (GPU)	Conv (CPU)	Prop (CPU)	Memory savings (bytes)
DCGAN/ DiscoGAN	2	4×4 × 1024	4×4 × 1024 × 512	0.046753	0.011541	3.023	0.727	495,616
	3	8×8 × 512	4×4 × 512 × 256	0.046085	0.011381	3.101	0.6863	739,328
	4	16×16 × 256	4×4 × 256 × 128	0.043747	0.011296	2.90	0.6598	1,254,400
	5	32×32 × 128	4×4 × 128 × 3	0.003049	0.001551	0.1363	0.0371	2,298,368
			Total	0.139639	0.035769	9.1603	2.1102	
			Total Speedup/ Memory saved		3.9039		4.34	4,787,712
Art-GAN	2	4×4 × 512	4×4 × 512 × 256	0.011886	0.005768	0.7114	0.1782	4,247,808
	3	8×8 × 256	4×4 × 256 × 128	0.011726	0.002971	0.7219	0.1652	369,664
	4	16×16 × 128	4×4 × 128 × 128	0.021727	0.00568	1.3879	0.316	627,200
	6	32×32 × 128	4×4 × 128 × 3	0.001582	0.001532	0.0359	0.0075	67,200
			Total	0.046921	0.015951	2.8571	0.6669	
			Total Speedup/ Memory saved		2.950		4.2841	1,871,872
GP-GAN	2	4×4 × 512	4×4 × 512 × 256	0.011847	0.005787	0.7114	0.1782	247,808
	3	8×8 × 256	4×4 × 256 × 128	0.01171	0.002952	0.7219	0.1652	369,664
	4	16×16 × 128	4×4 × 128 × 64	0.01167	0.002895	0.6995	0.1611	627,200
	5	32×32 × 64	4×4 × 64 × 3	0.001574	0.000852	0.0659	0.016	1,149,184
			Total	0.036801	0.012486	2.1987	0.5205	
			Total Speedup/ Memory saved		2.9474		4.224	2,393,856
EB-GAN	2	4×4 × 2048	4×4 × 2048 × 1024	0.188821	0.046078	16.0994	3.598	991,232
	3	8×8 × 1024	4×4 × 1024 × 512	0.176702	0.045508	14.193	2.919	1,478,656
	4	16×16 × 512	4×4 × 512 × 256	0.172839	0.045071	16.587	2.950	2,508,800
	5	32×32 × 256	4×4 × 256 × 128	0.172694	0.042322	12.197	2.866	4,596,736
	6	64×64 × 128	4×4 × 128 × 64	0.175486	0.041105	11.745	2.774	8,786,432
	7	128×128 × 64	4×4 × 64 × 64	0.349605	0.082192	22.398	5.233	17,172,736
			Total	1.236147	0.302276	93.2194	20.34	
			Total Speedup/ Memory Saved		4.089464		4.583	35,534,592

Table 3. Speedup for GPU and CPU versions and memory savings obtained from transpose convolution layers for popular GAN models

varied the kernel size of 5×5, 4×4, and 3×3 to apply the transpose convolution operation for the input dimension of $224 \times 224 \times 3$. We reported the flower dataset's computation time and memory savings obtained from both approaches. The results showed that the sub-classes of the flower dataset reached $3.4\times$ ($3.7\times$) speedup on average for GPU (CPU), with memory savings above 11,824,304 bytes based on the kernel size. Similarly, the average speedup of $3.4\times$ ($3.8\times$) for GPU(CPU) was achieved for the MSCOCO 2017 and PASCAL VOC 2012 datasets. Since all the input images for these datasets are preprocessed into the exact size of $224 \times 224 \times 3$, the memory savings still holds the same for these datasets from Table 1. The speedup is significantly improved with the increase in the kernel sizes for all three datasets, with the corresponding memory savings. However, the even order kernel showed more memory savings because it didn't produce offset elements during computation.

4.3. Ablation study

The computation time, memory savings, and computation load for the transpose convolution layers commonly used in the popular GAN architectures (Yazdanbakhsh et al., 2018b) are reported in Tables 3 and 4. The forward propagation phase for the layers is only considered by taking only one input sample during experimental analysis. In the DC-GAN/DiscoGAN, the average speedup of $3.9\times$ ($4.34\times$) was achieved for GPU (CPU) from the proposed approach with the overall memory savings of 4,787,712 bytes from the transpose convolution layers. Similarly, Art-GAN and GP-GAN got an average speedup of $2.95\times$ ($4.2\times$) for GPU (CPU). EB-GAN model showed the highest speedup of $4.08\times$ ($4.583\times$) because of more computation load needed for the transpose convolution layers in the model. We obtained limited GPU speedup for Art-GAN and GP-GAN models since the number of floating point operations like multiplications and additions is relatively less than in other models, which results in lower memory transfers. Among all the analyzed models, EB-GAN showed the highest memory savings in bytes of 35,534,592 from all transpose convolution layers. Additionally, there will be considerable improvement in the speedup from the transpose convolution layers during the training process, especially from backward

Model	Layer #	# of multiplications (original)	# of multiplications (proposed)	# of additions (original)	# of additions (proposed)
DCGAN /DISCOGAN	2	536,870,912	134,217,728	536,838,144	134,184,960
	3	536,870,912	134,217,728	536,805,376	134,152,192
	4	536,870,912	134,217,728	536,739,840	134,086,656
	5	25,165,824	6,291,456	25,153,536	6,279,168
Total		1,635,778,560	408,944,640	1,635,536,896	408,702,976
Total # of reductions in operations			1,226,833,920		1,226,833,920
Art-GAN	2	134,217,728	33,554,432	134,201,344	33,538,048
	3	134,217,728	33,554,432	134,184,960	33,521,664
	4	268,435,456	67,108,864	268,304,384	66,977,792
	6	25,165,824	6,2914,56	25,153,536	6,279,168
Total		562,036,736	140,509,184	561,844,224	140,316,672
Total # of reductions in operations			421,527,552		421,527,552
GP-GAN	2	134,217,728	33,554,432	134,201,344	33,538,048
	3	134,217,728	33,554,432	134,184,960	33,521,664
	4	134,217,728	33,554,432	134,152,192	33,488,896
	5	12,582,912	3,145,728	12,570,624	3,133,440
Total		415,236,096	103,809,024	415,109,120	103,412,048
Total # of reductions in operations			311,427,072		311,697,072
EB-GAN	2	2,147,483,648	536,870,912	2,147,418,112	536,805,376
	3	2,147,483,648	536,870,912	2,147,352,576	536,739,840
	4	2,147,483,648	536,870,912	2,147,221,504	536,608,768
	5	2,147,483,648	536,870,912	2,146,959,360	536,346,624
	6	2,147,483,648	536,8709,12	2,146,435,072	535,822,336
	7	4,294,967,296	1,073,741,824	4,290,772,992	1,069,547,520
Total		15,032,385,536	3,758,096,384	15,026,159,616	3,751,870,464
Total # of reductions in operations			11,274,289,152		11,274,289,152

Table 4. Number of floating point operations like multiplications and additions required for the conventional and proposed methods

	45nm technology			14nm technology		
Design	Delay (ns)	Area (cell units)	Power (mW)	Delay (ns)	Area (cell units)	Power (mW)
$3\times3 kernel$						
Conventional / 1 output	1.53	29413.37	19.23	0.49	3105.55	2.93
Proposed / 4 outputs	1.31	29019.63	19.91	0.44	3070.52	2.96
$4\times4 kernel$						
Conventional /1 output	1.66	54174.12	31.90	0.52	5835.89	5.19
Proposed /4 outputs	1.35	51217.06	37.57	0.44	5645.68	5.70
$5\times5 kernel$						
Conventional /1 output	1.77	78509.66	46.48	0.54	8966.66	7.77
Proposed /4 outputs	1.52	71270.24	56.38	0.49	8549.79	8.31

Table 5. Synthesis results of the conventional and proposed methods using 45nm and 14nm technologies with three different integer kernels

Original model	Shape	Modified model	Shape	Proposed model	Shape
Input layer	28×28×1	Input layer	28×28×1	Input layer	28×28×1
————-	———	Upsampling layer	55×55×1	Proposed optimized layer	51 × 51 × 8 (5×5 kernel)
CONV layer	24×24×8 (5×5 kernel)	CONV layer	51×51×8 (5×5 kernel)		
ReLU	24×24×8	ReLU	51×51×8	ReLU	51×51×8
Max pooling	12×12×8	Max pooling	26×26×8	Max pooling	26×26×8
FC layer	10	FC layer	10	FC layer	10

Table 6. Simple Deep Learning model configuration along with the total number of neurons for the MNIST dataset

propagation.

4.4. Hardware implementation

The functional unit for the transpose convolution operation is implemented using the Verilog language to understand the hardware characteristics for the conventional and proposed optimization methods, as depicted in Table 5. Here, Synopsys DC Compiler with 45nm and 14nm technology is used to analyze the original and proposed methods' performance using integer kernels of 32 bits with an input size of 8 bits. Results indicate that the proposed model requires more power but less delay and area than the conventional implementation. However, the power consumption for the proposed method is higher because it writes four output values instead of one, compared to the conventional implementation.

4.5. Evaluation using a simple neural network model

We evaluated the training time using a simple convolutional neural network model for practical application in deep learning to illustrate the advantage of the proposed optimization. The model design having one convolutional layer trained on the MNIST dataset (LeCun, 4/8/2022 accessed) is considered for the analysis, and the model's structure can be seen in Table. 6. It consists of an input layer with a shape of $28 \times 28 \times 1$ followed by a convolutional layer (CONV layer) with a Rectified Linear Unit (ReLU) as an activation function and a max-pooling layer. Finally, a fully connected layer (FC layer) is added with ten neurons, as there are ten classes of MNIST images. Later, the convolutional layer is replaced with conventional and proposed transpose convolution layers to compare the training time for both models. The model was trained using Intel dual-core processor with all the layers implemented using C++. The training time is taken for the model when 100,000 epochs with a minibatch size of 1 for comparing the two models. The training time obtained for the original model was 1,100 seconds, whereas the proposed model took only 501 seconds. Results showed that our proposed optimized algorithm performed $2.2\times$ faster than the conventional approach.

5. Conclusion and future work

This manuscript proposed a novel optimization technique for transpose convolution operations using the kernel segregation mechanism. And it obtained an average speedup of $3.7\times$ $(3.4\times)$ on computation time for the CPU (GPU) compared to the naive transpose convolution implementation with notable memory savings. Furthermore, the optimized technique is applied to the simple deep learning model, which consists of a single transpose convolutional layer. The results indicated that the proposed method achieved $2.2\times$ faster than the conventional method. However, the proposed optimization method needs more power consumption than the traditional method as it writes four output values in the memory simultaneously instead of one. There is also a need to reduce power consumption for the proposed approach, which can be viewed as a future research direction.

Acknowledgement

We sincerely thank Drs. Md Imran Hossen and Liqun Shan for their valuable time for the suggestions and for his dedicated help in organizing the manuscript effectively. We are also grateful to Dr. Tzeng for his support. This work is supported in part by the US NSF under grants OIA-1946231 and CNS-2117785.

References

Abadi, M., et al. (2016). {TensorFlow}: A system for {Large-Scale} machine learning. In *12th usenix symposium on operating systems design and implementation (osdi 16)* (pp. 265–283).

Anderson, A., et al. (2020). High-performance low-memory lowering: Gemm-based algorithms for dnn convolution. In *2020 ieee 32nd international symposium on computer*

architecture and high performance computing (sbac-pad) (pp. 99–106).

Bhattacharya, S., et al. (2016). Sparsification and separation of deep learning layers for constrained resource inference on wearables. In *Proceedings of the 14th acm conference on embedded network sensor systems cd-rom* (pp. 176–189).

Chang, J.-W., et al. (2018). An energy-efficient fpga-based deconvolutional neural networks accelerator for single image super-resolution. *IEEE Transactions on Circuits and Systems for Video Technology*, *30*(1), 281–295.

Chellapilla, K., et al. (2006). High performance convolutional neural networks for document processing. In *Tenth international workshop on frontiers in handwriting recognition.*

Chen, T., et al. (2018). {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th usenix symposium on operating systems design and implementation (osdi 18)* (pp. 578–594).

Common objects in context dataset. (8/26/2022 accessed). https://cocodataset.org/#download.

Dukhan, M. (2019). *The indirect convolution algorithm.* arXiv. Retrieved from https://arxiv.org/abs/1907.02129 doi: 10.48550/ARXIV.1907.02129

Georganas, E., et al. (2018). Anatomy of high-performance deep learning convolutions on simd architectures. In *Sc18: International conference for high performance computing, networking, storage and analysis* (pp. 830–841).

Goto, K., et al. (2008). Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, *34*(3), 1–25.

Heinecke, A., et al. (2016). Libxsmm: accelerating small matrix multiplications by runtime code generation. In *Sc'16: Proceedings of the international conference for high performance computing, networking, storage and analysis* (pp. 981–991).

Jia, Y., et al. (2014). Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd acm international conference on multimedia* (pp. 675–678).

K.Parhi, K. (accesed 4/7/2022). *Fast convolution.* http://www.ece.umn.edu/users/parhi/SLIDES/chap8.pdf.

Lavin, A., et al. (2016). Fast algorithms for convolutional neural networks. In *Proceedings of the ieee conference on computer vision and pattern recognition* (pp. 4013–4021).

LeCun, Y. (4/8/2022 accessed). *The mnist database of handwritten digits.* http://yann.lecun.com/exdb/mnist/.

Mamaev, A. (4/8/2022 accessed). *Flowers recognition.* https://www.kaggle.com/datasets/alxmamaev/flowers-recognition.

Paszke, A., et al. (2017). Automatic differentiation in pytorch.

Speeding up convolutions. (2020). https://scocoyash.github.io/speeding-up-convolutions/#naive-convolution.

Van Zee, F. G., et al. (2015). Blis: A framework for rapidly instantiating blas functionality. *ACM Transactions on Mathematical Software (TOMS)*, *41*(3), 1–33.

Vasilache, N., & Johnson, o. (2014). Fast convolutional nets with fbfft: A gpu performance evaluation. *arXiv preprint arXiv:1412.7580*.

V. HUYNH, J. T. V. (U.S. Patent WO/2021/061566, April. 2021). *Transposed convolution using systolic array.*

Visual object classes challenge 2012. (8/26/2022 accessed). http://host.robots.ox.ac.uk/pascal/VOC/voc2012/.

Wang, Q., et al. (2019). Parallel convolution algorithm using implicit matrix multiplication on multi-core cpus. In *2019 international joint conference on neural networks (ijcnn)* (pp. 1–7).

Yan, J., et al. (2018). Gna: Reconfigurable and efficient architecture for generative network acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *37*(11), 2519–2529.

Yazdanbakhsh, A., et al. (2018a). Flexigan: An end-to-end solution for fpga acceleration of generative adversarial networks. In *2018 ieee 26th annual international symposium on field-programmable custom computing machines (fccm)* (pp. 65–72).

Yazdanbakhsh, A., et al. (2018b). Ganax: A unified mimd-simd acceleration for generative adversarial networks. In *2018 acm/ieee 45th annual international symposium on computer architecture (isca)* (pp. 650–661).

Zhang, J., et al. (2018). High performance zero-memory overhead direct convolutions. In *International conference on machine learning* (pp. 5776–5785).

Zhou, S. (4/8/2022 accessed). *Transposed convolutions.* https://www.coursera.org/lecture/build-basic-generative-adversarial-networks-gans/transposed-convolutions-H02dK.