Fast 2D Convolution Algorithms for Convolutional Neural Networks

Chao Cheng and Keshab K. Parhi, *Fellow, IEEE*

Abstract— **Convolutional Neural Networks (CNN) are widely used in different artificial intelligence (AI) applications. Major part of the computation of a CNN involves 2D convolution. In this paper, we propose novel fast convolution algorithms for both 1D and 2D to remove the redundant multiplication operations in convolution computations at the cost of controlled increase of addition operations. For example, when the 2D processing block size is 3×3, our algorithm has multiplication saving factor as high as 3.24, compared to direct 2D convolution computation scheme. The proposed algorithm can also process input feature maps and generate output feature maps with the same flexible block sizes that are independent of convolution weight kernel size. The memory access efficiency is also largely improved by the proposed method. These structures can be applied to different CNN layers, such as convolution with $stride > 1$, pooling and deconvolution by exploring flexible feature map processing tile sizes. The proposed algorithm is suitable for both software and hardware implementation.**

Index Terms — **Convolutional neural network, Fast convolution, Kronecker product, Deconvolution Parallel FIR filter, Winograd algorithm**

I. Introduction

Convolution Neural Network (CNN) is the foundation of state-of-the-art AI applications such as image recognition [1][2][3][4], text classification[5][6][7], and generative adversarial network (GAN) [8]. More than 90% of the computation in CNN is occupied by convolution layers [9]. It is thus beneficial if we can shorten computation time in these convolution layers in order to achieve good CNN acceleration.

Computation of convolution layers has a lot of redundancy due to the nature of convolution especially when the convolution kernel size is large. Even though large kernel sizes are less popular as a recent trend, the CNN acceleration solution proposed in this paper covers both large and small kernel sizes.

Fast convolution of short lengths have been explored to reduce the computational complexity of convolution in previous works [10][11]. However, they can be improved. For example, the Winograd fast convolution [10] handles 2D short convolution with minimal multiplications but it has three issues: 1) when convolution length or kernel size gets larger (than 3), the number of addition or multiplication operations in both preprocessing and post-processing matrices increases dramatically; 2) when either input tile size or convolution kernel size changes, the preprocessing and post-processing matrices will change too; 3) preprocessing and post-processing involve left and right matrix multiplication, which makes it hard to expand convolution into large matrix multiplication. Other CNN fast convolution algorithms [11] based on short convolution algorithms [12], although achieve balanced and regular structures, exploit the redundancy only at one dimension of the feature map (row or column dimension) and thus the computation efficiency are not pushed to the limit from both the row and column dimensions. Furthermore, the sub-filter structures in the existing 1D convolution is not suitable for software implementation and can be improved.

Another limitation of the previous works is the convolution weight kernel size is usually tied to the tile size. This makes the convolution layer with $stride > 1$ and other subsequent CNN operations (such as pooling layers) hard to handle. We would have to apply multiple memory banks or perform extra memory access for data rearrangement. Today's CNN implementation requires higher and higher memory bandwidth. It's very important that we keep memory access rate low.

This paper focuses on improving the convolution efficiency by proposing a novel convolution processing algorithm in 2D of the input feature map. This 2D convolution core can be easily scaled to handle different convolution kernel sizes. The fact that feature map tile size is independent of the weight kernel size enables smooth data flow for different stride sizes in convolution, deconvolution and pooling and thus keeps memory access rate low.

Parallel filtering algorithms in [12][13][14] can be used for 1D CNN. We derive efficient 2D convolution algorithms and their general formula for 2D CNN in this paper. We show that, if the computation complexity saving factor of 1D convolution is F, then its corresponding 2D convolution can have a saving factor of F^2. To the best of the authors' knowledge, this is the first paper to demonstrate this saving factor for 2D convolution with parallel filter structures.

There are also many other published papers that cover different aspects of CNN implementation and acceleration [15][16]17]; for example, reducing data bandwidth by applying alternative data-reuse strategies [16] or addressing computation latency [17]. The proposed fast 2D algorithms in this paper are complementary to these existing architectures because it can save more computational complexity by exploiting fast convolution in 2D domain similar to the popular Winograd algorithms [10]. However we make it more convenient to

This paper was first submitted on Nov. 5th, 2019.

Chao Cheng is with AI Computation Technologies Lab, Alibaba Damo Academy, 525 Almanor Avenue, Sunnyvale, CA 94085 USA (e-mail: chen0867@umn.edu).

Keshab K. Parhi is with Electrical and Computer Engineering, University of Minnesota at Twin Cities, MN 55455 USA (e-mail: parhi@umn.edu). K.P.'s research was supported by the National Science Foundation under grant number CCF-1814759.

.

exploit fast 2D convolution for both hardware and software implementations. For example, transforming kernel and input feature map tiles to frequency domain would be done only by left matrix multiplication and no right matrix multiplication is needed. The pre-processing and post-processing matrices could have more flexible choices with only {0,1,-1} as their elements.

This paper is organized as follows. Section II reviews 1D convolution structures. The proposed 2D convolution algorithm is then derived in Section III. Application of the proposed 2D convolution structure to CNN convolution layers is discussed with implementation examples and is generalized in Section IV. Computational complexity analysis is performed in Section V.

II. One-Dimensional Convolutional Algorithms

We assume the input feature map has dimension $M \times N$, and weight kernel size of $K \times K$. The one-dimensional (1D) L-parallel FIR filter structure is described by (2.1) in [13]. We first apply it to the feature map in the row dimension.

$$y_i^{L(t+1)-1:Lt} = P^T H_L Q^T A^T R_i^{L(t+1)-1:Lt-L+1} \tag{2.1}$$

where, t is the sample index, $y_i^{L(t+1)-1:Lt} = \begin{pmatrix} y_i^{L(t+1)-1} & y_i^{L(t+1)-2} & \cdots & y_i^{Lt}\end{pmatrix}^T$ are the output feature map (OFM) elements at ith row and columns $L(t+1)-1:Lt$, $R_i^{L(t+1)-1:Lt-L+1} =$ $\begin{pmatrix} R_i^{L(t+1)-1} & R_i^{L(t+1)-2} & \cdots & R_i^{Lt} & | & R_i^{Lt-1} & \cdots & R_i^{Lt-L+1}\end{pmatrix}^T$ are the input feature map (IFM) elements at ith row and columns $L(t+1)-1:Lt-L+1$, and we have $R_i^j = 0$ for $j < 0$ or $j \geq N$; $H_L = diag(P \times (H_0 \quad H_1 \quad \cdots \quad H_{L-1})^T)$

and H_i, (i =0, 1, …, L -1), are the subfilters containing the coefficients w_i^{Lt+j}, which are the weight values at ith row and $Lt+j$ columns of a weight matrix of size $K \times K$.

A^T is used only when we need to factorize L into $L = L_1 L_2 \cdots L_r$ and is listed here for the completeness of the algorithm. Since state-of-art CNN kernel sizes are usually small, a large L value with factorization is not usually used and thus A^T can be ignored. More detail about A^T can be found in [13]. P, Q and H_L matrices are from short linear convolution decomposition.

Assume consuming as input and producing as output $L = 3$ columns of pixels of a feature map without loss of generality. Here we can apply a 3-parallel FIR filter structure as follows:

$$\begin{pmatrix} y_i^{3t+2} \\ y_i^{3t+1} \\ y_i^{3t} \end{pmatrix} = P_3^T diag \begin{pmatrix} w_i^2 \\ w_i^1 \\ w_i^0 \\ w_i^2 + w_i^1 \\ w_i^2 + w_i^0 \\ w_i^1 + w_i^0 \end{pmatrix} Q_3^T \begin{pmatrix} R_i^{3t+2} \\ R_i^{3t+1} \\ R_i^{3t} \\ R_i^{3t-1} \\ R_i^{3t-2} \end{pmatrix} \tag{2.2}$$

where, R_i^j is the IFM element on i-th row and j-th column, the 1D kernel coefficients are $\{w_i^0, w_i^1, w_i^2\}$ (kernel size here is K=3). P_3 and Q_3 are defined as follows,

$$P_3 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix}, Q_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & 0 & 1 & 0 & 0 \\ -1 & 1 & -1 & 0 & 1 & 0 \\ 0 & -1 & -1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix},$$

$H_L = diag(P_3 \times (H_0 \quad H_1 \quad H_2)^T)$ with $H_0 = w_i^2$, $H_1 = w_i^1$ and $H_2 = w_i^0$.

There is another 3-parallel FIR filter structure:

$$\begin{pmatrix} y_i^{3t+2} \\ y_i^{3t+1} \\ y_i^{3t} \end{pmatrix} = P_3^T diag \begin{pmatrix} \frac{1}{2} w_i^2 \\ \frac{1}{2}(w_i^2 + w_i^1 + w_i^0) \\ \frac{1}{6}(w_i^2 + 2w_i^1 + w_i^0) \\ \frac{1}{6}(w_i^2 - w_i^1 + w_i^0) \\ w_i^0 \end{pmatrix} Q_3^T \begin{pmatrix} R_i^{3t+2} \\ R_i^{3t+1} \\ R_i^{3t} \\ R_i^{3t-1} \\ R_i^{3t-2} \end{pmatrix} \tag{2.3}$$

Where, $P_3 = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 2 & 4 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$,

$Q_3 = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ -1 & 2 & -1 & -2 & 2 \\ -2 & 1 & 0 & 3 & -1 \\ 1 & -1 & 1 & -1 & -2 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$ and

$H_L = diag(diag(1/2 \quad 1/2 \quad 1/6 \quad 1/6 \quad 1) \times P_3 \times (H_0 \quad H_1 \quad H_2)^T)$ with $H_0 = w_i^2$, $H_1 = w_i^1$ and $H_2 = w_i^0$.

Since the row size of the feature map is N, 1D convolution would conventionally need to perform KN multiplications and $(K-1)N$ additions. 3-parallel structures in (2.2) and (2.3) require $\frac{N}{3} \cdot 6 \cdot \lceil K/3 \rceil) \approx \frac{2}{3}KN$ and $\frac{N}{3} \cdot 5 \cdot \lceil K/3 \rceil) \approx \frac{5}{9}KN$ multipliers, respectively. Therefore, compared to conventional 1D convolution computation, (2.2) and (2.3) can save 1.5 and 1.8 times of the multiplication operations, respectively.

We continue our discussion based on fast convolution in (2.2) without loss of generality.

For kernel size K>3, for example, K=9, a 1D 3-parallel FIR filter implementation can be represented as:

$$\begin{pmatrix} y_i^{3t+2} \\ y_i^{3t+1} \\ y_i^{3t} \end{pmatrix} = P_3^T * diag \begin{pmatrix} \{w_i^8, w_i^5, w_i^2\} \\ \{w_i^7, w_i^4, w_i^1\} \\ \{w_i^6, w_i^3, w_i^0\} \\ \{w_i^8 + w_i^7, w_i^5 + w_i^4, w_i^2 + w_i^1\} \\ \{w_i^8 + w_i^6, w_i^5 + w_i^3, w_i^2 + w_i^0\} \\ \{w_i^7 + w_i^6, w_i^4 + w_i^3, w_i^1 + w_i^0\} \end{pmatrix} * Q_3^T \begin{pmatrix} R_i^{3t+2} \\ R_i^{3t+1} \\ R_i^{3t} \\ R_i^{3t-1} \\ R_i^{3t-2} \end{pmatrix} \tag{2.4}$$

Where, $\{w_i^8, w_i^5, w_i^2\}$, $\{w_i^7, w_i^4, w_i^1\}$, $\{w_i^6, w_i^3, w_i^0\}$, $\{w_i^8 + w_i^7, w_i^5 + w_i^4, w_i^2 + w_i^1\}$,

$\{w_i^8 + w_i^7, w_i^5 + w_i^4, w_i^2 + w_i^1\}$ and $\{w_i^7 + w_i^6, w_i^4 + w_i^3, w_i^1 + w_i^0\}$ are subfilters, and w_i^j is row i and column j of the 2D filter kernel. "*" is the 1D convolution operator.

Note that the length of subfilters in (2.4) is $\lceil K/L \rceil = 3$. These subfilter taps are embedded inside subfilters multiplied by delayed versions of their input sources. This is equivalent to convolving these subfilter taps to their preprocessed IFM tile by Q_3^T. To remove these subfitler convolution operations and achieve fully-controlled pipeline flow in the convolution engine, we expand these subfilter taps and move their input to the IFM side by concatenating IFM elements of size $(2L-1) \times 1$ with their delayed versions. This 1D subfilter expansion by removing the 1D convolution operator * is shown in (2.5).

$$\begin{pmatrix} y_i^{3t+2} \\ y_i^{3t+1} \\ y_i^{3t} \end{pmatrix} = P_3^T [I_6 \quad I_6 \quad I_6] diag\left((I_3 \otimes P_3) \begin{pmatrix} w_i^8 \\ w_i^7 \\ \vdots \\ w_i^1 \\ w_i^0 \end{pmatrix} \right) (I_3 \otimes Q_3^T) \begin{pmatrix} R_i^{3t+2} \\ R_i^{3t+1} \\ \vdots \\ R_i^{3t-2} \\ ---- \\ R_i^{3(t-1)+2} \\ R_i^{3(t-1)+1} \\ \vdots \\ R_i^{3(t-1)-2} \\ ---- \\ R_i^{3(t-2)+2} \\ R_i^{3(t-2)+1} \\ \vdots \\ R_i^{3(t-2)-2} \end{pmatrix} \qquad (2.5)$$

where $\otimes$ is the Kronecker product. I_i is the identity matrix of size *i*.

The proposed 1D fast convolution algorithm can be generalized and summarized as (2.6).

$$\begin{pmatrix} y_i^{Lt+L-1} \\ y_i^{Lt+L-2} \\ \vdots \\ y_i^{Lt} \end{pmatrix} = P^T diag\left(D \begin{pmatrix} w_i^{K-1} \\ w_i^{K-2} \\ \vdots \\ w_i^1 \\ w_i^0 \end{pmatrix} \right) Q^T \begin{pmatrix} R_i^{Lt+L-1} \\ R_i^{Lt+L-2} \\ \vdots \\ R_i^{Lt-L+1} \\ ---- \\ R_i^{L(t-1)+L-1} \\ R_i^{L(t-1)+L-2} \\ \vdots \\ R_i^{L(t-1)-L+1} \\ ---- \\ R_i^{L(t-2)+L-1} \\ R_i^{L(t-2)+L-2} \\ \vdots \\ R_i^{L(t-2)-L+1} \end{pmatrix} \qquad (2.6)$$

where, $D = I\left(\left\lceil \frac{K}{L} \right\rceil\right) \otimes (h_L P_L)$ and h_L is part of $L \times L$ convolution decomposition; for example, h_L is $diag([\frac{1}{2}, \frac{1}{2}, \frac{1}{6}, \frac{1}{6}, 1])$ and I_6 in (2.3) and (2.2), respectively;

$P^T = P_L^T \left(ones\left(1, \left\lceil \frac{K}{L} \right\rceil\right) \otimes I(P_L^{rows\#}) \right)$, $Q^T = I\left(\left\lceil \frac{K}{L} \right\rceil\right) \otimes Q_L^T$ and $I(x)$ is the identity matrix of size x. $P_L^{rows\#}$ is the number of rows of matrix P_L.

(2.6) can be used as a computation engine for feature map tiles of size $1 \times L$ in 1D convolution applications, such as text classification. For example, when $L = 3$, the number of required multiplication operation can be reduced by a factor of 1.5 and 1.8, respectively, using (2.2) and (2.3).

III. 2D Convolution algorithms

Input could be a feature map for convolutional neural network. A 2D filter could be a CNN kernel of size $K \times K$. We target processing an input feature map and producing an output feature map of a tile size of $L \times L$ in one operation time unit, where L can be read as parallelism level of the 2D parallel FIR filter engine. It can be chosen from a number set, such as {2,3}, and the numbers that can be factorized into the same number set. More discussion about parallelism level can be found in [13]. To make the following presentation easier, we proceed with L=3 in this section.

A. Weight kernel size $\boldsymbol{K = 3}$ and feature map tile size $\boldsymbol{L = 3}$

Define the 3×3 kernel as,

$$\begin{pmatrix} w_0^0 & w_0^1 & w_0^2 \\ w_1^0 & w_1^1 & w_1^2 \\ w_2^0 & w_2^1 & w_2^2 \end{pmatrix} \text{ and } w_i = \{w_i^0, w_i^1, w_i^2\}$$

We can present the 2-D convolution which processes *L*=3 rows of feature map as (3.1).

$$\begin{pmatrix} y_{3s+2} \\ y_{3s+1} \\ y_{3s} \end{pmatrix} = P_3^T * diag \begin{pmatrix} w_2 \\ w_1 \\ w_0 \\ w_2 + w_1 \\ w_2 + w_0 \\ w_1 + w_0 \end{pmatrix} * Q_3^T * \begin{pmatrix} R_{3s+2} \\ R_{3s+1} \\ R_{3s} \\ R_{3s-1} \\ R_{3s-2} \end{pmatrix} \qquad (3.1)$$

Where, as defined in section II, y_i, w_k and R_j are row vectors from OFM, weigh kernel and IFM, respectively. y_i, w_k and R_j are zero vectors when $i < 0$ or $i \geq N$, $k < 0$ or $k \geq K$, and $j < 0$ or $j \geq N$, respectively. We also define the following 1D convolution operation over * operator.

$$(x \quad y) * \begin{pmatrix} R_i \\ R_j \end{pmatrix} = x * R_i + y * R_j$$

The computational complexity saving of parallel filter structures are from decomposition of the original 1D/2D convolution into many 1D sub-filter structures with smaller sizes and aggregation of these sub-filter outputs. 1D convolution operators are used to simplify the representation of these decomposition and aggregation procedures in the sub-filter structures.

We next discuss how to remove the 1D convolution operator * from (3.1).

We define,

$$\begin{pmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \\ Z_4 \\ Z_5 \end{pmatrix} = Q_3^T * \begin{pmatrix} R_{3t+2} \\ R_{3t+1} \\ R_{3t} \\ R_{3t-1} \\ R_{3t-2} \end{pmatrix}$$

where Z_i is a vector of intermediate results, after applying 1D convolution between IFM rows and Q_3^T. We then carry out the deduction shown in (3.2) to help understand how to remove 1D convolution at subfilter level.

$$\begin{pmatrix} y_{3t+2} \\ y_{3t+1} \\ y_{3t} \end{pmatrix} = P_3^T * diag\begin{pmatrix} w_2 \\ w_1 \\ w_0 \\ w_2 + w_1 \\ w_2 + w_0 \\ w_1 + w_0 \end{pmatrix} * \begin{pmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \\ Z_4 \\ Z_5 \end{pmatrix}$$

$$= P_3^T * diag\begin{pmatrix} w_2 * Z_0 \\ w_1 * Z_1 \\ w_0 * Z_2 \\ (w_2 + w_1) * Z_3 \\ (w_2 + w_0) * Z_4 \\ (w_1 + w_0) * Z_5 \end{pmatrix} = P_3^T * (I_6 \otimes P_3^T) *$$

$$diag\begin{pmatrix} diag(P_3\ w_2) \\ diag(P_3\ w_1) \\ diag(P_3\ w_0) \\ diag(P_3\ (w_2 + w_1)) \\ diag(P_3\ (w_2 + w_0)) \\ diag(P_3\ (w_1 + w_0)) \end{pmatrix} (I_6 \otimes Q_3^T) \begin{pmatrix} Z_0 \\ Z_1 \\ Z_2 \\ Z_3 \\ Z_4 \\ Z_5 \end{pmatrix} \tag{3.2}$$

We can thus get the 2D subfilter expansion by removing the 1D convolution operator * in (3.1). The obtained OFM of block size 3x3 is given in (3.3).

$$\begin{pmatrix} y_{3s+2}^{3t+2:3t} \\ y_{3s+1}^{3t+2:3t} \\ y_{3s}^{3t+2:3t} \end{pmatrix} = (P_3^T \otimes I_3)(I_6 \otimes P_3^T) diag\left((I_6 \otimes P_3)(P_3 \otimes I_3) \begin{pmatrix} w_2^2 \\ w_2^1 \\ w_2^0 \\ \vdots \\ w_0^1 \\ w_0^0 \end{pmatrix} \right)$$

$$(I_6 \otimes Q_3^T)(Q_3^T \otimes I_5) \begin{pmatrix} R_{3s+2}^{3t+2:3t-2} \\ R_{3s+1}^{3t+2:3t-2} \\ R_{3s}^{3t+2:3t-2} \\ R_{3s-1}^{3t+2:3t-2} \\ R_{3s-2}^{3t+2:3t-2} \end{pmatrix} \tag{3.3}$$

Where, $3s + i$ and $3t + j$ are the row and column indices of feature maps, respectively.

B. Weight kernel size $\boldsymbol{K = 9}$ for $\boldsymbol{K > 3}$ and feature map tile size $\boldsymbol{L = 3}$.

Let us define 9x9 kernel as follows,

$$\begin{pmatrix} w_0^0 & w_0^1 & w_0^2 & \dots & w_0^7 & w_0^8 \\ w_1^0 & w_1^1 & w_1^2 & \dots & w_1^7 & w_1^8 \\ w_2^0 & w_2^1 & w_2^2 & \dots & w_2^7 & w_2^8 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ w_7^0 & w_7^1 & w_7^2 & \cdots & w_7^7 & w_7^8 \\ w_8^0 & w_8^1 & w_8^2 & \cdots & w_8^7 & w_8^8 \end{pmatrix}$$

Where, $w_i = \{w_i^0, w_i^1, w_i^2, \cdots, w_i^8\}$

We can present the 2-D convolution which processes L=3 rows of feature map as (3.4).

$$\begin{pmatrix} y_{3s+2} \\ y_{3s+1} \\ y_{3s} \end{pmatrix} = P_3^T *$$

$$diag\begin{pmatrix} \{w_8, w_5, w_2\} \\ \{w_7, w_4, w_1\} \\ \{w_6, w_3, w_0\} \\ \{w_8 + w_7, w_5 + w_4, w_2 + w_1\} \\ \{w_8 + w_6, w_5 + w_3, w_2 + w_0\} \\ \{w_7 + w_6, w_4 + w_3, w_1 + w_0\} \end{pmatrix} * Q_3^T * \begin{pmatrix} R_{3s+2} \\ R_{3s+1} \\ R_{3s} \\ R_{3s-1} \\ R_{3s-2} \end{pmatrix} \tag{3.4}$$

After applying subfilter expansion to (3.4), we get (3.5).

$$\begin{pmatrix} y_{3s+2} \\ y_{3s+1} \\ y_{3s} \end{pmatrix} = P_3^T * [I_6 \quad I_6 \quad I_6] * diag\left((I_3 \otimes P_3) \begin{pmatrix} w_8 \\ w_7 \\ \vdots \\ w_1 \\ w_0 \end{pmatrix} \right) *$$

$$(I_3 \otimes Q_3^T) * \begin{pmatrix} R_{3s+2} \\ R_{3s+1} \\ \vdots \\ R_{3s-2} \\ ---- \\ R_{3(s-1)+2} \\ R_{3(s-1)+1} \\ \vdots \\ R_{3(s-1)-2} \\ ---- \\ R_{3(s-2)+2} \\ R_{3(s-2)+1} \\ \vdots \\ R_{3(s-2)-2} \end{pmatrix} \tag{3.5}$$

After removing 1D convolution operator *, we get (3.6) to process IFM of tile size 3x3 and generate OFM of the same tile size.

$$\begin{pmatrix} y_{3s+2}^{3t+2:3t} \\ y_{3s+1}^{3t+2:3t} \\ y_{3s}^{3t+2:3t} \end{pmatrix} = P^T diag\left(D \begin{pmatrix} w_8^8 \\ w_8^7 \\ \vdots \\ w_8^0 \\ w_7^8 \\ w_7^7 \\ \vdots \\ w_7^0 \\ w_6^8 \\ \vdots \\ w_0^1 \\ w_0^0 \end{pmatrix} \right) Q^T \begin{pmatrix} R_{3s+2}^{3t+2:3t-2} \\ R_{3s+1}^{3t+2:3t-2} \\ \vdots \\ R_{3s-2}^{3t+2:3t-2} \\ ---- \\ R_{3(s-1)+2}^{3t+2:3t-2} \\ R_{3(s-1)+1}^{3t+2:3t-2} \\ \vdots \\ R_{3(s-1)-2}^{3t+2:3t-2} \\ ---- \\ R_{3(s-2)+2}^{3t+2:3t-2} \\ R_{3(s-2)+1}^{3t+2:3t-2} \\ \vdots \\ R_{3(s-2)-2}^{3t+2:3t-2} \end{pmatrix} \quad (3.6)$$

Where, $D = (I_{18} \otimes I_3 \otimes P_3)(I_3 \otimes P_3 \otimes I_9)$,
$P^T = (P_3^T \otimes I_3)([I_6 \quad I_6 \quad I_6] \otimes I_3)(I_{18} \otimes (P_3^T [I_6 \quad I_6 \quad I_6]))$,
and $Q^T = (I_{18} \otimes I_3 \otimes Q_3^T)(I_3 \otimes Q_3^T \otimes I_{15})$.

The Proposed 2D fast convolution algorithm can be generalized and summarized as (3.7).

$$\begin{pmatrix} y_{Ls+L-1}^{Lt+L-1:Lt} \\ y_{Ls+1}^{Lt+L-1:Lt} \\ \vdots \\ y_{Ls}^{Lt+L-1:Lt} \end{pmatrix} = P^T diag\left(D \begin{pmatrix} w_{K-1}^{K-1} \\ w_{K-1}^{K-2} \\ \vdots \\ w_{K-1}^0 \\ --- \\ w_{K-2}^{K-1} \\ w_{K-2}^{K-2} \\ \vdots \\ w_{K-2}^0 \\ --- \\ \vdots \\ --- \\ w_0^{K-1} \\ \vdots \\ w_0^1 \\ w_0^0 \end{pmatrix} \right) Q^T \begin{pmatrix} R_{Ls+L-1}^{Lt+L-1:Lt-L+1} \\ R_{Ls+L-2}^{Lt+L-1:Lt-L+1} \\ \vdots \\ R_{3s-L+1}^{Lt+L-1:Lt-L+1} \\ ------ \\ R_{L(s-1)+L-1}^{Lt+L-1:Lt-L+1} \\ R_{L(s-1)+L-2}^{Lt+L-1:Lt-L+1} \\ \vdots \\ R_{L(s-1)-L+1}^{Lt+L-1:Lt-L+1} \\ ------ \\ \vdots \\ ------ \\ R_{L\left(s-\left\lceil\frac{K}{L}\right\rceil\right)+L-1}^{Lt+L-1:Lt-L+1} \\ R_{3\left(s-\left\lceil\frac{K}{L}\right\rceil\right)+L-2}^{Lt+L-1:Lt-L+1} \\ \vdots \\ R_{L\left(s-\left\lceil\frac{K}{L}\right\rceil\right)-L+1}^{Lt+L-1:Lt-L+1} \end{pmatrix} \quad (3.7)$$

Where,

$$D = \left(I\left(P_L^{rows\#}\left\lceil\frac{K}{L}\right\rceil\right) \otimes I\left(\left\lceil\frac{K}{L}\right\rceil\right) \otimes (h_L P_L) \right) \left(I\left(\left\lceil\frac{K}{L}\right\rceil\right) \otimes (h_L P_L) \otimes I\left(L\left\lceil\frac{K}{L}\right\rceil\right) \right)$$

h_L is part of $L \times L$ convolution decomposition, for example, h_L is $diag([\frac{1}{2}, \frac{1}{2}, \frac{1}{6}, \frac{1}{6}, 1])$ and I_6 in (2.3) and (2.2), respectively;

$$P^T = (P_L^T \otimes I(L)) \left(ones\left(1, \left\lceil\frac{K}{L}\right\rceil\right) \otimes I(P_L^{rows\#} L) \right) \left(I\left(P_L^{rows\#}\left\lceil\frac{K}{L}\right\rceil\right) \otimes \left(P_L^T \left(ones\left(1, \left\lceil\frac{K}{L}\right\rceil\right) \otimes I(P_L^{rows\#}) \right) \right) \right)$$

and

$$Q^T = \left(I\left(P_L^{rows\#}\left\lceil\frac{K}{L}\right\rceil\right) \otimes I\left(\left\lceil\frac{K}{L}\right\rceil\right) \otimes Q_L^T \right) \left(I\left(\left\lceil\frac{K}{L}\right\rceil\right) \otimes Q_L^T \otimes I\left(Q_L^{rows\#}\left\lceil\frac{K}{L}\right\rceil\right) \right),$$

$I(x)$ is the identity matrix of size x. Q_L and P_L are the Q and P matrix for $L \times L$ 1D fast convolution, respectively. $P_L^{rows\#}$ and $Q_L^{rows\#}$ are the number of rows of matrices P_L and Q_L, respectively.

We can see that the decomposition and computation of the proposed (3.7) does not depend on dimension size of IFM.

If we compare the derived 1D (2.6) and the 2D convolution algorithm (3.7), we can see that they have unified structure. This allows the proposed convolution to enable easy switch among 1D/2D applications, for example, text classification (1D) and image classification (2D).

(3.7) is used as a computation engine for feature map tiles of size $L \times L$. We will discuss how to use it in the whole data flow via implementation examples in the next section.

IV. Implementation and Examples

To compute (3.7), we first slice input feature map into $L \times L$ tiles; each tile is then transformed into a vector with the order from right to left and bottom to top. We consume these tiles from IFM in the order of left to right and then top to bottom. We use Example 4.1 and Example 4.2 to illustrate how this is done, where IFM is processed and consumed in (3.3) and (3.6) for $K = L$ and $K > L$. Both examples share an original IFM of size 7x7. It's represented as $IFM_{org}^{c\#}$ in (4.1).

$$IFM_{org}^{c\#} = \begin{pmatrix} 1 & 2 & 3 & \ldots & 6 & 7 \\ 8 & 9 & 10 & \ldots & 13 & 14 \\ 15 & 16 & 17 & \ldots & 20 & 21 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 36 & 37 & 38 & \cdots & 41 & 42 \\ 43 & 44 & 45 & \cdots & 48 & 49 \end{pmatrix} \quad (4.1)$$

where, the superscript c# is the IFM channel index

A. *Example 4.1 $L = 3$, kernel size $K = 3$ and 7×7 IFM*

After slicing $IFM_{org}^{c\#}$ into tiles of size $L \times L$, we get (4.2). Note that we pad zeros to the right and bottom of IFM to make sure all tiles have the same size.

$IFM_{sliced}^{c\#} =$

1	2	3	4	5	6	7	0	0
8	9	10	11	12	13	14	0	0
15	16	17	18	19	20	21	0	0
22	23	24	25	26	27	28	0	0
29	30	31	32	33	34	35	0	0
36	37	38	39	40	41	42	0	0
43	44	45	46	47	48	49	0	0
0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0

(4.2)

We convert $IFM_{sliced}^{c\#}$ into $IFM_{mem}^{c\#}$ in (4.2) for efficient memory access. We can see that each $L \times L$ slice is converted to a column vector of the order from the lower right corner to upper left corner of its corresponding original slice in $IFM_{sliced}^{c\#}$. For example, the upper left tile of (4.2) is converted to the left most column of (4.3).

$$IFM_{mem}^{c\#} = \begin{pmatrix} 17 & 20 & 0 & 38 & 41 & 0 & 0 & 0 & 0 \\ 16 & 19 & 0 & 37 & 40 & 0 & 0 & 0 & 0 \\ 15 & 18 & 21 & 36 & 39 & 42 & 0 & 0 & 0 \\ 10 & 13 & 0 & 31 & 34 & 0 & 0 & 0 & 0 \\ 9 & 12 & 0 & 30 & 33 & 0 & 0 & 0 & 0 \\ 8 & 11 & 14 & 29 & 32 & 35 & 0 & 0 & 0 \\ 3 & 6 & 0 & 24 & 27 & 0 & 45 & 48 & 0 \\ 2 & 5 & 0 & 23 & 26 & 0 & 44 & 47 & 0 \\ 1 & 4 & 7 & 22 & 25 & 28 & 43 & 46 & 49 \end{pmatrix} \tag{4.3}$$

$IFM_{mem}^{c\#}$ read from memory needs to be transformed locally into $IFM_{4conv}^{c\#}$ shown in (4.4)

$IFM_{4conv}^{c\#}=$

$$\begin{array}{ccc|ccc|ccc}
17 & 20 & 0 & 38 & 41 & 0 & & & \\
16 & 19 & 0 & 37 & 40 & 0 & & & \\
15 & 18 & 21 & 36 & 39 & 42 & \multicolumn{3}{c}{0_{5\times3}} \\
0 & 17 & 20 & 0 & 38 & 41 & & & \\
0 & 16 & 19 & 0 & 37 & 40 & & & \\
\hline
10 & 13 & 0 & 31 & 34 & 0 & & & \\
9 & 12 & 0 & 30 & 33 & 0 & & & \\
8 & 11 & 14 & 29 & 32 & 35 & \multicolumn{3}{c}{0_{5\times3}} \\
0 & 10 & 13 & 0 & 31 & 34 & & & \\
0 & 9 & 12 & 0 & 30 & 33 & & & \\
\hline
3 & 6 & 0 & 24 & 27 & 0 & 45 & 48 & 0 \\
2 & 5 & 0 & 23 & 26 & 0 & 44 & 47 & 0 \\
1 & 4 & 7 & 22 & 25 & 28 & 43 & 46 & 49 \\
0 & 3 & 6 & 0 & 24 & 27 & 0 & 45 & 48 \\
0 & 2 & 5 & 0 & 23 & 26 & 0 & 44 & 47 \\
\hline
& & & 17 & 20 & 0 & 38 & 41 & 0 \\
& & & 16 & 19 & 0 & 37 & 40 & 0 \\
\multicolumn{3}{c|}{0_{5\times3}} & 15 & 18 & 21 & 36 & 39 & 42 \\
& & & 0 & 17 & 20 & 0 & 38 & 41 \\
& & & 0 & 16 & 19 & 0 & 37 & 40 \\
\hline
& & & 10 & 13 & 0 & 31 & 34 & 0 \\
& & & 9 & 12 & 0 & 30 & 33 & 0 \\
\multicolumn{3}{c|}{0_{5\times3}} & 8 & 11 & 14 & 29 & 32 & 35 \\
& & & 0 & 10 & 13 & 0 & 31 & 34 \\
& & & 0 & 9 & 12 & 0 & 30 & 33 \\
\hline
C_1^{c\#} & C_2^{c\#} & C_3^{c\#} & C_4^{c\#} & C_5^{c\#} & C_6^{c\#} & C_7^{c\#} & C_8^{c\#} & C_9^{c\#}
\end{array} \tag{4.4}$$

Each column $C_i^{c\#}$ in (4.4) corresponds to the right-most column of input vector in (3.3).

Note that, OFM would have the same format as IFM, which means each column of OFM represents a $L \times L$ tile of OFM and the size of OFM is the same as IFM. This means we can directly use OFM for next layer processing without extra memory access for feature map re-arrangement.

If we represent each sliced block in (4.4) as $R_1^{c\#}$ with 'i' corresponding to the i'th row of $IFM_{org}^{c\#}$, for example (4.5),

$$R_1^{c\#} = \begin{pmatrix} 3 & 6 & 0 \\ 2 & 5 & 0 \\ 1 & 4 & 7 \\ 0 & 3 & 6 \\ 0 & 2 & 5 \end{pmatrix} \tag{4.5}$$

(4.4) can then be simplified as (4.6).

$$IFM_{4conv}^{c\#} = \begin{pmatrix} R_3^{c\#} & R_6^{c\#} & 0 \\ R_2^{c\#} & R_5^{c\#} & 0 \\ R_1^{c\#} & R_4^{c\#} & R_7^{c\#} \\ 0 & R_3^{c\#} & R_6^{c\#} \\ 0 & R_2^{c\#} & R_5^{c\#} \end{pmatrix} \tag{4.6}$$

Interestingly, we can find that (4.5) and (4.6) have the same pattern.

B. *Example 4.2. $L = 3$, $K = 9$ kernel and 7×7 IFM.*

For $K = 9$, input feature map can be arranged as (4.7), where each stacked layer corresponds to one subfitler tap and thus the pattern of one layer is the delayed (right shifted) version of the section right above it. Both row-wise (4.7) and pixel-wise (4.8) representation have this same pattern among their stacked layers.

$IFM_{4conv}^{c\#}=$

$$\begin{array}{ccccc}
R_3^{c\#} & R_6^{c\#} & 0 & 0 & 0 \\
R_2^{c\#} & R_5^{c\#} & 0 & 0 & 0 \\
R_1^{c\#} & R_4^{c\#} & R_7^{c\#} & 0 & 0 \\
0 & R_3^{c\#} & R_6^{c\#} & 0 & 0 \\
0 & R_2^{c\#} & R_5^{c\#} & 0 & 0 \\
\hline
0 & R_3^{c\#} & R_6^{c\#} & 0 & 0 \\
0 & R_2^{c\#} & R_5^{c\#} & 0 & 0 \\
0 & R_1^{c\#} & R_4^{c\#} & R_7^{c\#} & 0 \\
0 & 0 & R_3^{c\#} & R_6^{c\#} & 0 \\
0 & 0 & R_2^{c\#} & R_5^{c\#} & 0 \\
\hline
0 & 0 & R_3^{c\#} & R_6^{c\#} & 0 \\
0 & 0 & R_2^{c\#} & R_5^{c\#} & 0 \\
0 & 0 & R_1^{c\#} & R_4^{c\#} & R_7^{c\#} \\
0 & 0 & 0 & R_3^{c\#} & R_6^{c\#} \\
0 & 0 & 0 & R_2^{c\#} & R_5^{c\#}
\end{array} \tag{4.7}$$

Where, for example, we have

$R_1^{c\#} =$

$$\begin{array}{ccccc}
3 & 6 & 0 & 0 & 0 \\
2 & 5 & 0 & 0 & 0 \\
1 & 4 & 7 & 0 & 0 \\
0 & 3 & 6 & 0 & 0 \\
0 & 2 & 5 & 0 & 0 \\
\hline
0 & 3 & 6 & 0 & 0 \\
0 & 2 & 5 & 0 & 0 \\
0 & 1 & 4 & 7 & 0 \\
0 & 0 & 3 & 6 & 0 \\
0 & 0 & 2 & 5 & 0
\end{array}$$

$$\begin{array}{ccccc}
0 & 0 & 3 & 6 & 0 \\
0 & 0 & 2 & 5 & 0 \\
0 & 0 & 1 & 4 & 7 \\
0 & 0 & 0 & 3 & 6 \\
0 & 0 & 0 & 2 & 5 \\
\hline
C_1^{c\#} & C_2^{c\#} & C_3^{c\#} & C_4^{c\#} & C_5^{c\#}
\end{array} \tag{4.8}$$

We next generalize the implementation procedure.

Computation data flow of a convolution layer could be of two types: pointwise and depth wise.

C. Pointwise operation

This needs a large on-chip buffer to save the intermediate computational results as large as the whole feature map, while the weight kernel of a channel does not need to be swapped back and forth until next feature map input. (3.7) can be implemented with this scheme as shown in Fig. 4.2. Each shaded block is a $L \times L$ feature map processing tile and the number in it shows the order index it's processed. We can see that the tiles (Fig. 4.2(a)) in the same IFM are consumed from upper left corner to right bottom corner and then moved on to the next feature map channel. Fig. 4.2(b) shows the map between IFM tiles and their corresponding weight kernel.

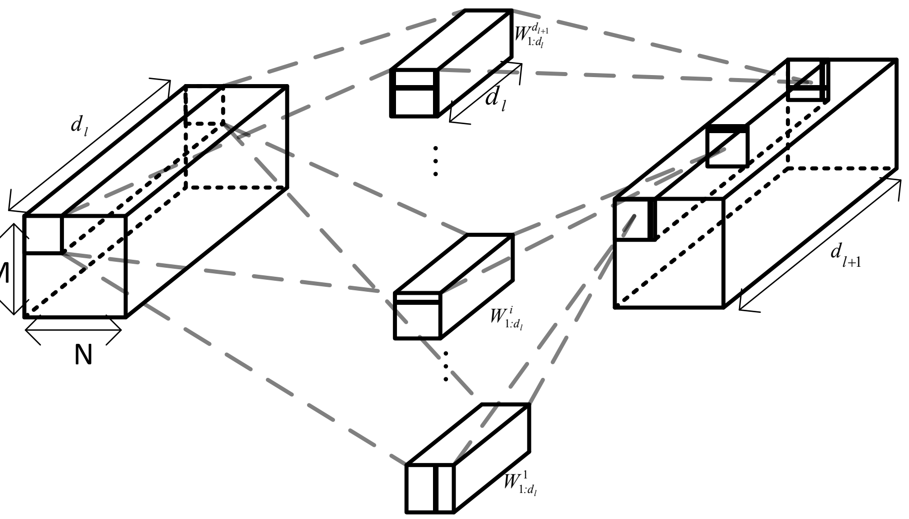
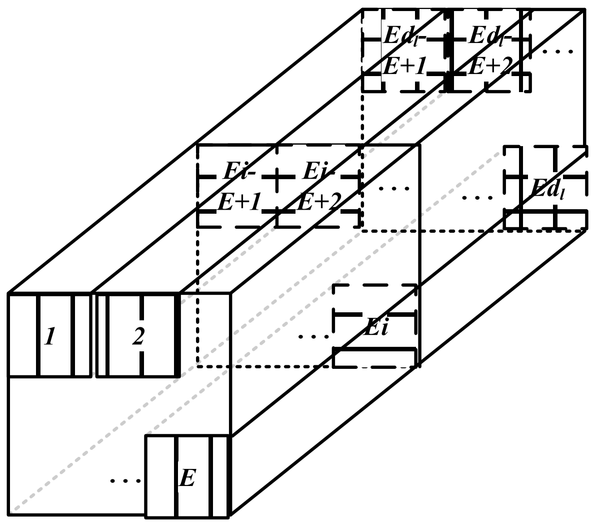


Fig. 4.1. CNN convolutional layer computation from layer l to $l+1$. d_l is the feature map depth at layer l. W_i^j is the weight kernel for IFM channel i and OFM channel j.



(a)

$C_1^1\ \ C_2^1\ \ \dots\ \ C_E^1$	$C_1^2\ \ C_2^2\ \ \dots\ \ C_E^2$	...	$C_1^{d_l}\ \ C_2^{d_l}\ \ \dots\ \ C_E^{d_l}$
W_1^1	W_2^1	...	$W_{d_l}^1$
$diag(DW_1^1)Q^T C_{1:E}^1$	$diag(DW_2^1)Q^T C_{1:E}^2$	...	$diag(DW_{d_l}^1)Q^T C_{1:E}^{d_l}$
$OFM_{C_{1:E}}^1 = P^T \sum_{i=1}^{d_l} diag(DW_i^1)Q^T C_{1:E}^i$			

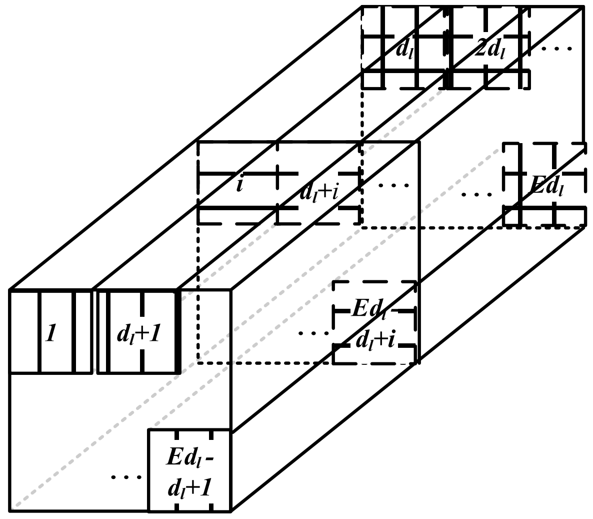
$C_1^1\ \ C_2^1\ \ \dots\ \ C_E^1$	$C_1^2\ \ C_2^2\ \ \dots\ \ C_E^2$	...	$C_1^{d_l}\ \ C_2^{d_l}\ \ \dots\ \ C_E^{d_l}$

W_1^2	W_2^2	…	$W_{d_l}^2$
$diag(DW_1^2)Q^T C_{1:E}^1$	$diag(DW_2^2)Q^T C_{1:E}^2$	…	$diag(DW_{d_l}^2)Q^T C_{1:E}^{d_l}$
$OFM_{C_{1:E}}^2 = P^T \sum_{i=1}^{d_l} diag(DW_i^2)Q^T C_{1:E}^i$			
⋮			
$C_1^1 \quad C_2^1 \quad … \quad C_E^1$	$C_1^2 \quad C_2^2 \quad … \quad C_E^2$	…	$C_1^{d_l} \quad C_2^{d_l} \quad … \quad C_E^{d_l}$
$W_1^{d_{l+1}}$	$W_2^{d_{l+1}}$	…	$W_{d_l}^{d_{l+1}}$
$diag(DW_1^{d_{l+1}})Q^T C_{1:E}^1$	$diag(DW_2^{d_{l+1}})Q^T C_{1:E}^2$	…	$diag(DW_{d_l}^{d_{l+1}})Q^T C_{1:E}^{d_l}$
$OFM_{C_{1:E}}^{d_{l+1}} = P^T \sum_{i=1}^{d_l} diag(DW_i^{d_{l+1}})Q^T C_{1:E}^i$			

$$*E = \left\lceil \frac{M+L\left\lfloor\frac{K}{L}\right\rfloor-1}{L} \right\rceil \left\lceil \frac{N+L\left\lfloor\frac{K}{L}\right\rfloor-1}{L} \right\rceil$$

(b)

Fig. 4.2. IFM tile access sequence for pointwise operation to generate one channel of OFM (repeated d_{l+1} times to get all OFM channels). $\boldsymbol{C_j^i}$ means the j'th tile in the i'th input feature map channel.



(a)

$C_1^1 \quad C_1^2 \quad … \quad C_1^{d_l}$	…	$C_E^1 \quad C_E^2 \quad … \quad C_E^{d_l}$	…	$C_1^1 \quad C_1^2 \quad … \quad C_1^{d_l}$	…	$C_E^1 \quad C_E^2 \quad … \quad C_E^{d_l}$
$W_1^1 \quad W_2^1 \quad … \quad W_{d_l}^1$	…	$W_1^1 \quad W_2^1 \quad … \quad W_{d_l}^1$	…	$W_1^{d_{l+1}} \quad W_2^{d_{l+1}} \quad … \quad W_{d_l}^{d_{l+1}}$	…	$W_1^{d_{l+1}} \quad W_2^{d_{l+1}} \quad … \quad W_{d_l}^{d_{l+1}}$
$OFM_{C_1}^1 = P^T \sum_{i=1}^{d_l} diag(DW_i^1)Q^T C_1^i$	…	$OFM_{C_E}^1 = P^T \sum_{i=1}^{d_l} diag(DW_i^1)Q^T C_E^i$		$OFM_{C_1}^{d_{l+1}} = P^T \sum_{i=1}^{d_l} diag(DW_i^{d_{l+1}})Q^T C_1^i$	…	$OFM_{C_E}^{d_{l+1}} = P^T \sum_{i=1}^{d_l} diag(DW_i^{d_{l+1}})Q^T C_E^i$

(b)

Fig. 4.3. IFM tile access sequence for Depthwise operation to generate one channel of OFM (repeated d_{l+1} times to get all OFM channels). $\boldsymbol{C_j^i}$ means the j'th tile in the i'th input feature map channel.

D. *Depthwise operation*

This requires swapping weight kernel among IFM channels. (3.7) can be implemented with this scheme as shown in Fig. 4.3. Each shaded block is a $L \times L$ feature map processing tile and the number in it shows the order index it's processed. We can see that the tiles (Fig. 4.3(a)) in the same feature map position among the IFM channels are consumed first and then move on to next feature map position from upper left corner to right bottom corner. Fig. 4.3(b) shows the map between IFM tiles and their corresponding weight kernel.

V. Computational Complexity Analysis

The proposed 2D parallel FIR filter structures are flexible for both data flow types.

For an input feature map of size $M \times N$ and channel depth d_l, without the loss of generality, let's assume the output feature of size $M \times N$ and channel depth d_{l+1}. Direct computation of convolution from layer l to layer $l + i$ requires

$K \cdot K \cdot M \cdot N \cdot d_{l+1} \cdot d_l$ multiplication and $(K^2-1) \cdot M \cdot N \cdot d_{l+1} \cdot d_l$ addition operations. However, with the proposed 2D fast convolution algorithm, the required number of multiplications can be reduced to (5.1).

$$R.M. = \left(M_L \left\lceil \frac{K}{L} \right\rceil\right)^2 \left(\frac{M}{L}\right)\left(\frac{N}{L}\right) d_l d_{l+1} \qquad (5.1)$$

where M_L is the number of multipliers required by $K \times K$ fast 1D convolution algorithm of parallelism level L. The multiplication saving ratio ($M.S.R.$) is given by (5.2).

$$M.S.R. = \left(\frac{KL}{M_L\left\lceil\frac{K}{L}\right\rceil}\right)^2 \sim = \left(\frac{L^2}{M_L}\right)^2 \qquad (5.2)$$

For example, in equations (2.2) and (2.3), we have M_L equal to 6 and 5 for $L=3$, and thus $M.S.R.$ is 2.25 and 3.24, respectively.

The $M.S.R.$ could be estimated independent of the kernel size especially when K is divisible by L.

Although we assume the same parallelism levels at row and column processing dimensions as L, they do not have to be the same.

The number of required addition operations can be estimated too. From above discussion, we can see that the addition operations for the convolution layers are all embedded in matrices P^T, Q^T and D in (3.7) and the required number of addition operations involved in these matrices are given in Table 5.1. We also summarize dimensions of these matrices in Table 5.1.

The total number of addition operations (R.A.) is given by (5.3).

$$R.A. = \left(A(P^T) + d_l\left(A(Q^T) + A(D)\right)\right)\left(\frac{M}{L}\right)\left(\frac{N}{L}\right) d_{l+1} \qquad (5.3)$$

The required number of addition operations is larger than the conventional convolution computation scheme. We define addition increase ratio (A.I.R.) to estimate the magnitude of this increase.

$$A.I.R. = \frac{\left(A(P^T)+d_l\left(A(Q^T)+A(D)\right)\right)\left(\frac{M}{L}\right)\left(\frac{N}{L}\right)d_{l+1}}{(K^2-1)\cdot M \cdot N \cdot d_{l+1} \cdot d_l} \qquad (5.4)$$

$M.S.R.$ for different weight kernel size K and feature map processing block size L are shown in Fig. 5.1.

$A.I.R.$ for different weight kernel size K and feature map processing block size L are shown in Fig. 5.2.

D matrix in (3.7) is computed from P_L and thus the density of matrix D relies on that of P_L. The required multiplication operations are located in matrix D. When K is not divisible by L or K<L, diagonal matrix generation in (3.7) needs zero padding. The $M.S.R.$ would be more sensitive to a D matrix (and thus P_L) with higher density. For example, P_3 in (2.3) is denser than (2.2), $M.S.R.$ values computed based on (2.2) shows higher $M.S.R.$ variation if we compare Fig. 5.1(b) with (2.3) and Fig. 5.1(a) with (2.2). Similar higher $A.I.R.$ variation trend is also observed in Fig. 5.2(b) with (2.3) than that shown in Fig. 5.2(a) with (2.2).

As we can see from above discussion, the feature map processing block size L is independent of weight kernel size K. Although so far we used $L=3$ as example to illustrate how the proposed algorithm works, L could take other values too; for example, (5.5) shows one 1D convolution decomposition for $L=2$.

$$\begin{pmatrix} y_i^{2t+1} \\ y_i^{2t} \end{pmatrix} = P_2^T H_L \, Q_2^T \begin{pmatrix} R_i^{2t+1} \\ R_i^{2t} \\ R_i^{2t-1} \end{pmatrix} \qquad (5.5)$$

where, $P_2 = \begin{pmatrix} 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{pmatrix}, Q_2 = \begin{pmatrix} 1 & 0 & 0 \\ 1 & -1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$,
$H_L = diag(P_2 \times (H_0 \quad H_1)^T)$.

Since K=3 is the most popular weight kernel size, we summarize $M.S.R.$ and $A.I.R.$ values with L=2, 3 and 4 in Table 5.2 for computation complexity comparison.

We can see $L=3$ is most efficient. This is because $L=3$ naturally aligns with $K=3$ and thus no zero padding is need to for tile size ($L=4$) or kernel size ($L=2$). However, $L=4$ and $L=2$ can still flexibly support $K=3$ with fairly good computation efficiency. We will show later that this flexibility is important for other CNN computation scenarios, such as deconvolution and convolution with $stride > 1$.

A. *Other Implementation complexity Analysis*

As discussed above, the proposed algorithm can support different tile sizes of L to process the same weight kernel size with good efficiency. This flexibility to support different processing block size is quite important to the overall processing efficiency of the proposed CNN computation structure. This section covers the application of the proposed fast 2D convolution structure to other aspects of CNN implementation.

B. *Different Stride Size*

When we have convolution stride size $stride > 1$, it's equivalent to first computing convolution of $stride = 1$ and then down-sampling the output by $stride \times stride$ in 2D. If we set $L = stride$ and only retain the computation path that leads to the one (for example, the last one) out of $stride \times stride$ OFM elements, we can achieve both structure consistence and computation efficiency. Since the $stride \times stride$ OFM is represented as the $stride^2 \times 1$ vector, this simplification starts from ignoring the processing of first $stride^2 - 1$ rows of P^T matrix. Since we only need to handle the last row of P^T matrix, the diagonal elements in matrix D that correspond to the zero elements of the last P^T row can be ignored, which means the multiplication operation at these positions can be removed; for the same reason, the addition operations in the corresponding rows in Q^T matrix can be ignored as well. This is illustrated in Fig. 5.3, where "×" elements are ignored, $last\_w = \left(P_L^{rows\#}\left\lceil\frac{K}{L}\right\rceil\right)^2$ and $last\_ifm = \left((2L-1)\left\lceil\frac{K}{L}\right\rceil\right)^2$.

As shown in (3.6), matrix P^T and Q^T are pre-determined when L and K values are chosen in advance. Therefore, the locations of the saved multiplication in matrix D and addition operations for Q^T are known in advance as well.

L can be factorized into smaller values if large size of L can be applied for large K, for example, K=11 in the first convolutional layer of AlexNet. More details on this factorization procedure can be found in [13].

C. Memory requirement

The memory size to save $IFM_{mem}^{c\#}$ in (4.3) has a size of $L^2 \times \left\lceil \frac{M+L\left\lceil \frac{K}{L}\right\rceil-1}{L} \right\rceil \left\lceil \frac{N+L\left\lceil \frac{K}{L}\right\rceil-1}{L} \right\rceil$.

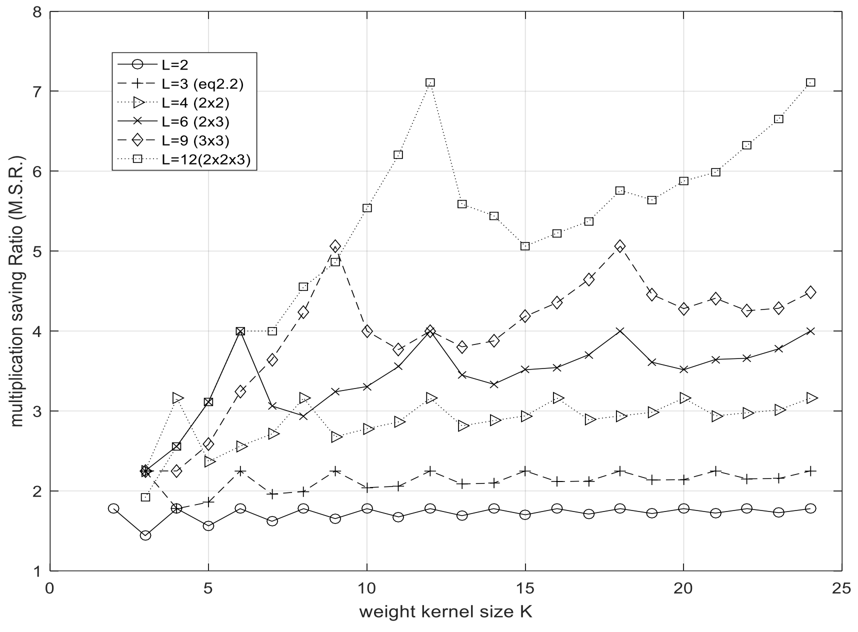
We also need local memory of different sizes for delayed values for $IFM_{4conv}^{c\#}$ in (4.5) and (4.7) if we explore pointwise and depthwise opertations. For pointwise operation in Fig. 4.1, we need a memory size of $M \times N$ for intermediate results of OFM and $\left(L(L-1)+(L-1)(2L-1)\left\lceil \frac{N+L\left\lceil \frac{K}{L}\right\rceil-1}{L} \right\rceil\right) \times 1$ for delayed values of IFM. For depthwise operation in Fig. 4.2, we need a memory size of $\left(L(L-1)+(L-1)(2L-1)\left\lceil \frac{N+L\left\lceil \frac{K}{L}\right\rceil-1}{L} \right\rceil\right) d_l \times 1$ for delayed values of IFM. (3.7) needs input of size $\left((2L-1)\cdot\left\lceil \frac{K}{L}\right\rceil\right)^2 \times 1$, but only $L^2 \times 1$ of these values are from a IFM tile saved in $IFM_{mem}^{c\#}$ of size $L \times L$ and the rest values are from delayed IFM values as shown in (4.5)-(4.8).

We can see the depth-wise implementation requires d_l (where d_l is the IFM channels) times more local memory for delayed values of IFM than point-wise implementation. However, point-wise implementation requires $\frac{\boldsymbol{M \cdot N}}{\boldsymbol{L \cdot L}}$.times more local memory for intermediate results of OFM than depth-wise implementation. Therefore, local memory requirement ratio between depth-wise and point-wise implementation is estimated as $\frac{\boldsymbol{d_l \cdot L \cdot L}}{\boldsymbol{M \cdot N}}$.

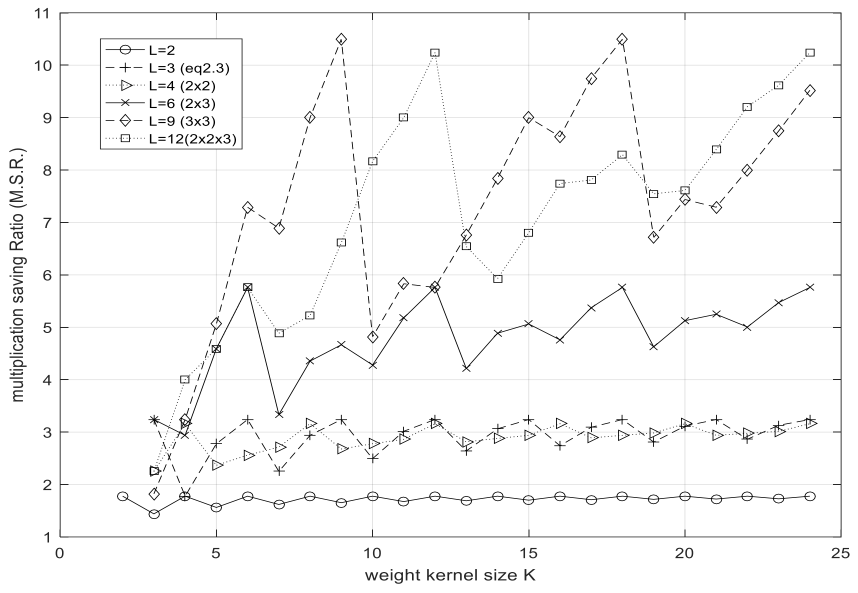
TABLE 5.1 ADDITION OPERATION ANALYSIS

Matrices	dimension	Addition Operation
P^T	$L^2 \times \left(P_L^{rows\#}\left\lceil \frac{K}{L}\right\rceil\right)^2$	$A(P^T) = A(P_L^T)L + P_L^{rows\#}L\left(\left\lceil \frac{K}{L}\right\rceil - 1\right) + P_L^{rows\#}\left\lceil \frac{K}{L}\right\rceil\left(A(P_L^T) + P_L^{rows\#}\left(\left\lceil \frac{K}{L}\right\rceil - 1\right)\right) = \left(A(P_L^T) + P_L^{rows\#}\left(\left\lceil \frac{K}{L}\right\rceil - 1\right)\right)\left(P_L^{rows\#}\left\lceil \frac{K}{L}\right\rceil + L\right)$
Q^T	$\left(P_L^{rows\#}\left\lceil \frac{K}{L}\right\rceil\right)^2 \times \left((2L-1)\left\lceil \frac{K}{L}\right\rceil\right)^2$	$A(Q^T) = A(Q_L^T)(P_L^{rows\#} + Q_L^{rows\#})\left\lceil \frac{K}{L}\right\rceil^2$
D	diagonal $\left(P_L^{rows\#}\left\lceil \frac{K}{L}\right\rceil\right)^2$	$A(D) = A(h_L P_L)(P_L^{rows\#} + L)\left\lceil \frac{K}{L}\right\rceil^2$

*For example, $A(P_3^T) = 6$, $A(P_3\ ) = 3$ and $A(Q_3^T) = 6$ in (2.2).

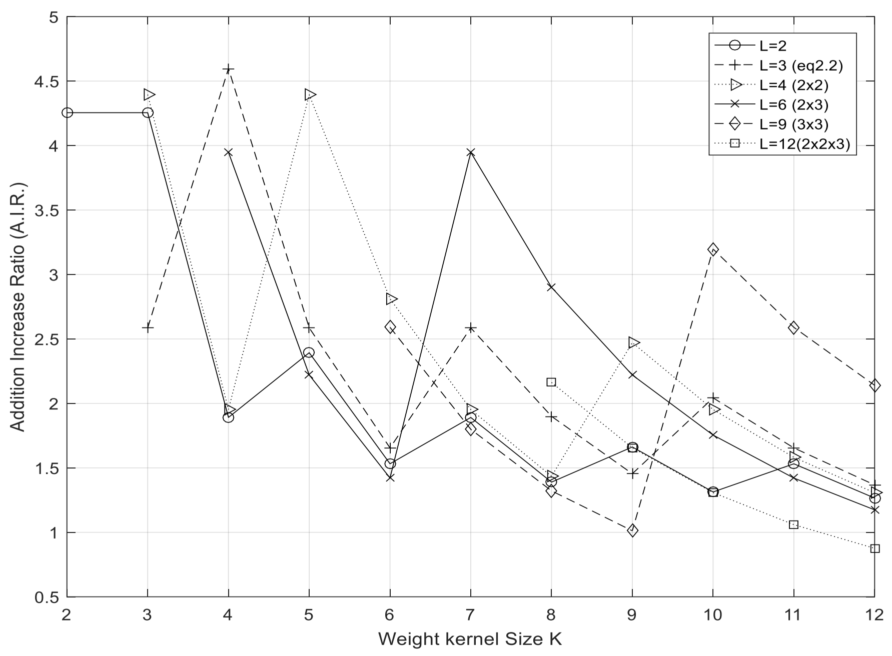


(a)

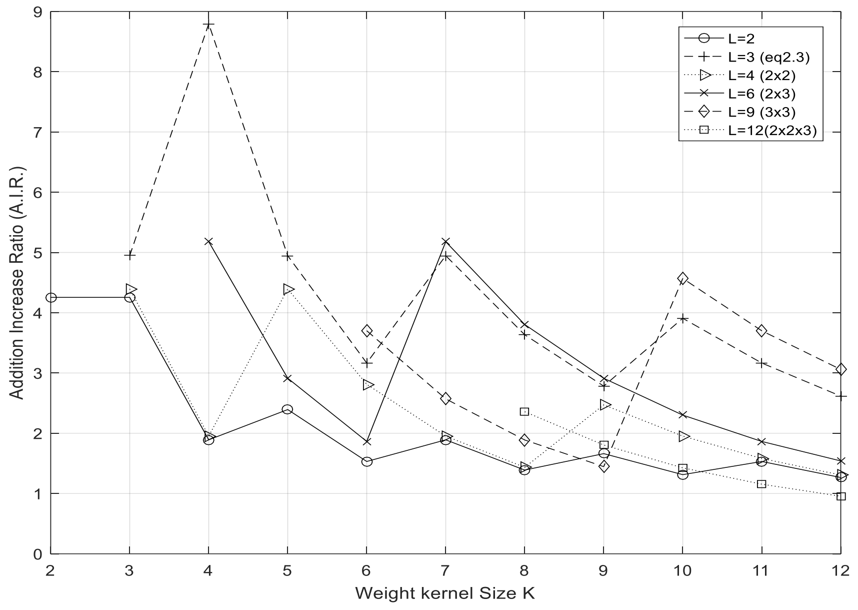


(b)

Fig. 5.1. $M.S.R.$ for different $\{K,L\}$ combination, (a) with equation (2.2); (b) with equation (2.3)



(a)



(b)

Fig. 5.2. $A.I.R.$ for different $\{K,L\}$ combination, (a) with equation (2.2); (b) with equation (2.3)

TABLE 5.2 $M.S.R.$ AND $A.I.R.$ FOR $K=3$ WITH DIFFERENT FEATURE MAP TILE SIZE L

L	1D convolution	$M.S.R.$	$A.I.R.$
2	Eq(5.5)	1.5	4.3
3	Eq(2.2)	2.25	2.6
3	Eq(2.3)*	3.24	5
4	Eq(5.5) 2x2	2.25	4.4

*Assuming 1/6=1/8+1/32+1/128+1/512, 1/6 can be represented with 3 addition operation with quantization error of -64dB, which is good for 10bit fixed point computation.

D. Deconvolution

Deconvolution is widely used in applications such as generative adversarial network (GAN) [8] and feature extraction [18]. We need to perform 2D up-sampling by $stride \times stride$ before we apply convolution computation with weight kernel size $K \times K$. If we set $L = stride$ and only leave the computation path that starts from the only one non-zero value of $L \times L$ IFM elements, we can achieve both structure consistence and computation efficiency as well.

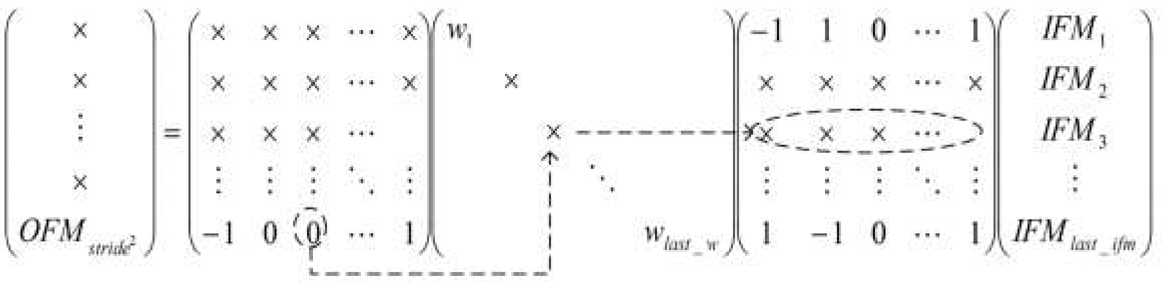


Fig. 5.3. Multiplication and addition operation can be saved with $L = stride$.

Since we represent a IFM block of size of $L \times L$ as column vector $L^2 \times 1$, we would only have one non-zero value in the IFM column of Fig. 5.3, which means only one column of Q^T matrix needs to be used. The multiplication operations in matrix D that correspond to the zero elements in this column of Q^T can be ignored. The saving of addition operations in P^T can be obtained by following the same reasoning as shown in Fig. 5.3.

The locations of the saved multiplication in matrix D and addition operations for P^T and Q^T are known for the chosen L and K values.

E. Pooling

Pooling computation would have a window size of $K \times K$ with a stride of usually $stride > 1$. Since pooling operation, either max-pooling or average-pooling, is carried out within a window of $K \times K$ and we represent IFM and OFM tiles in the format of column vector $L^2 \times 1$, if we can dramatically improve memory access efficiency if choose tile size L as the same or a factor of K at the convolution layer before pooling layer. If this not possible, we would choose an L value that could re-organize the output of the convolution layer output into this desired format. For example, The first convolution layer of AlexNet has $stride = 4$, which means we would let this layer have tile size $L = 4$ for easy downsampling, but its following pooling layer has $K = 3$. However, this should not be an issue, because we can choose $L = 12$ and $stride = 4$ down-sampling would naturally generate 3×3 tiles size as input to max-pooling of window with the same size.

F. Fully-connected Layer

In fully-connected layer, we would not have convolution related redundancy to explore and thus no saving on multiplication operation. It would be pure matrix multiplication. In this case we would bypass Q^T matrix and configure the last row of P^T matrix to be all 1's for inner product output.

G. Sparsity

If we apply weight sparsity [19][20] in the training, the computation saving factor could be less unless we apply sparsity in a way that is friendly to the subfilter generation procedure in this paper. Interested readers can explore sparsity in this direction.

H. Round-off noise

Since Winograd needs both left and right multiplications for both preprocessing and post-processing, the proposed algorithms in this paper only need left multiplication and thus its round-off/quantization noise performance should be better.

VI. Conclusion

Unified 1D and 2D convolution algorithms are proposed in this paper for CNN computation optimization. Multiplication computation efficiency is largely improved with controlled addition operation increase. Processing tile size and output format of OFM is the same as those of IFM. Tile size is independent of convolution kernel size. Structure regularity and consistency are achieved among convolution layers and other CNN processing layers. The matrix multiplication is limited to left multiplication and is applicable for CNN implementation based on large matrix multiplication. The proposed algorithm with potential tweak is suitable for both software and hardware CNN acceleration.

Since the 2D parallel convolution algorithms in this paper can be used in a plug-and-play fashion, it is independent of the specific convolution neural networks it is used in and the other implementation considerations.

The sections IV and V provide only an example of possible CNN implementation based on the proposed algorithms. Usually, for example, to process an IFM of size LxL for K=L, we would need to access the memory and read in a tile of size (2L-1)x(2L-1). We reduce memory BW requirement by reducing the memory access to LxL and saving/refreshing the other needed pixels in local memory based on sliding widows. This presented example does not mean the proposed algorithms can only be implemented this way.

References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks.”, *NIPS*, pp. 1106–1114, 2012.

[2] K. Simonyan and A. Zisserman, "Deep convolutional networks for large-scale image recognition." arXiv preprint arXiv:1409.1556, (2014).

[3] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions." In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1-9. 2015.

[4] K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition.", *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770-778, 2016.

[5] Y. Kim, "Convolutional neural networks for sentence classification.", arXiv preprint arXiv:1408.5882, 2014.

[6] X. Zhang, J. Zhao, and Y. LeCun. "Character-level convolutional networks for text classification.", *Advances in neural information processing systems*, pp. 649-657, 2015.

[7] A. Conneau, H. Schwenk, L. Barrault, and Y. Lecun, "Very deep convolutional networks for text classification.", arXiv preprint arXiv:1606.01781 (2016).

[8] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. "Generative adversarial nets.*", Advances in neural information processing systems*, pp. 2672-2680, 2014.

[9] H. Nakahara and T. Sasao. "A deep convolutional neural network based on nested residue number system." *In 2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1-6. IEEE, 2015.

[10] A. Lavin, S. Gray, "Fast algorithms for convolutional neural networks*", Proc. IEEE Conf. Comput. Vis. Pattern Recognit*., pp. 4013-4021, Aug. 2016.

[11] J. Wang, J. Lin, and Z. Wang. "Efficient hardware architectures for deep convolutional neural network." *IEEE Transactions on Circuits and Systems I: Regular Paper*s, vol. 65, no. 6 (2018): 1941-1953.

[12] K. K. Parhi, *VLSI Digital Signal Processing Systems: Design and Implementation,* John Wiley and Sons, 1999.

[13] C. Cheng, K. K. Parhi, "Hardware efficient fast parallel FIR filter structures based on iterated short convolution*", IEEE Trans. Circuits Syst. I Reg. Pape*rs, vol. 51, no. 8, pp. 1492-1500, Aug. 2004.

[14] D.A. Parker and K.K. Parhi, "Low Area/Power Parallel FIR Digital Filter Implementations," *Journal of VLSI Signal Processing*, 17(1), pp. 75-92, Sept. 1997

[15] W. Xu, Z. Wang, X. You and C. Zhang, “Efficient fast convolution architectures for convolutional neural network,” in *proceedings of 2017 IEEE 12th International Conference on ASIC (ASICON)*, Guiyang, 2017, pp. 904-907.

[16] A. Ardakani, C. Condo, M. Ahmadi, and W. J. Gross, “An architecture to accelerate convolution in deep neural networks,” *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 4, pp. 1349–1362, Apr. 2017.

[17] Y.-J. Lin and T. S. Chang, “Data and hardware efficient design for convolutional neural network,” *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 65, no. 5, pp. 1642–1651, May 2018.

[18] A. Radford, L. Metz, and S. Chintala. "Unsupervised representation learning with deep convolutional generative adversarial networks." arXiv preprint arXiv:1511.06434 (2015).

[19] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, “EIE: Efficient Inference Engine on Compressed Deep Neural Network,” in *IEEE Symposium on Computer Architecture (ISCA),* pp. 243-254, 2016

[20] C. Deng, S. Liao, Y. Xie, K.K. Parhi, X. Qian, and B. Yuan, "PermDNN: Efficient Compressed Deep Neural Network Architecture with Permuted Diagonal Matrices," Proc. 51st *Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 189-202, Fukuoka, Japan, Oct. 20-24, 2018

Chao Cheng (M’2004) received his Ph.D in Electrical and Computer Engineering from University of Minnesota at Twin Cities, Minneapolis, USA in 2007.

From 2007 to 2018, he gained his industrial experience in Storage and Wireless communication system from Intel, Marvell and Qualcomm in Silicon

Valley. He has authored 20 papers, including 9 IEEE Transaction papers. He has invented or coinvented 7 US patents.

He is currently a research scientist at Damo Academy, Alibaba. His research interest includes Novel CNN algorithms and architectures for better accuracy and lower cost, Graph based Neural Network for recommendation applications and Multi-task object detection and understanding platform of multi-modalities.

Keshab K. Parhi (S'85–M'88–SM'91–F'96) received the B.Tech. degree from Indian Institute of Technolgy, Kharagpur, India, in 1982; the M.S.E.E. degree from the University of Pennsylvania, Philadelphia, PA, USA; in 1984, and the Ph.D. degree from the University of California at Berkeley, Berkeley, CA, USA, in 1988.

He has been with the University of Minnesota, Minneapolis, MN, USA, since 1988, where he is currently a Distinguished McKnight University Professor and Edgar F. Johnson Professor in the Department of Electrical and Computer Engineering. He has published over 650 papers, is the inventor or co-inventor of 31 patents, has authored the textbook *VLSI Digital Signal Processing Systems* (New York, NY, USA: Wiley, 1999), and coedited the reference book *Digital Signal Processing for Multimedia Systems* (Boca Raton, FL, USA: CRC Press, 1999). His current research interests include the VLSI architecture design and implementation of signal processing, and machine learning systems, neuromorphic computing, data-driven neuroscience and biomarkers for neuro-psychiatric disorders, hardware security and molecular computing.

Dr. Parhi has served on the Editorial Boards of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS PART I AND PART II, the IEEE TRANSACTIONS ON VLSI SYSTEMS, IEEE TRANSACTIONS ON SIGNAL PROCESSING, the IEEE SIGNAL PROCESSING LETTERS, and the *IEEE Signal Processing Magazine*, and served as the Editor-in-Chief of the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS PART I from 2004 to 2005. He currently serves on the Editorial Board of the *Journal of Signal Processing Systems* (Springer). He has served as the Technical Program Co-Chair of the 1995 IEEE VLSI Signal Processing Workshop and the 1996 Application Specific Systems, Architectures, and Processors conference, and as the General Chair of the 2002 IEEE Workshop on Signal Processing Systems. He was the Distinguished Lecturer of the IEEE Circuits and Systems Society during 1996-1998 and 2019-2020. He served as a Board of Governors Elected Member of the IEEE Circuits and Systems Society from 2005 to 2007. He is the recipient of numerous awards including the 2017 Mac Van Valkenburg award, the 2012 Charles A. Desoer Technical Achievement award and the 1999 Golden Jubilee medal, from the IEEE Circuits and Systems society, the 2013 Distinguished Alumnus Award from IIT Kharagpur, the 2013 Graduate/Professional Teaching Award from the University of Minnesota, the 2004 F. E. Terman award from the American Society of Engineering Education, the 2003 IEEE Kiyo Tomiyasu Technical Field Award, and the 2001 IEEE W. R. G. Baker Prize Paper Award.