Stride 2 1-D, 2-D, and 3-D Winograd for Convolutional Neural Networks

Juan Yepez and Seok-Bum Ko, *Senior Member, IEEE*

***Abstract*— Convolutional neural networks (CNNs) have been widely adopted for computer vision applications. CNNs require many multiplications, making their use expensive in terms of both computational complexity and hardware. An effective method to mitigate the number of required multiplications is via the Winograd algorithm. Previous implementations of CNNs based on Winograd use the 2-D algorithm $F(2 \times 2, 3 \times 3)$, which reduces computational complexity by a factor of 2.25 over regular convolution. However, current Winograd implementations only apply when using a stride (shift displacement of a kernel over an input) of 1. In this article, we presented a novel method to apply the Winograd algorithm to a stride of 2. This method is valid for one, two, or three dimensions. We also introduced new Winograd versions compatible with a kernel of size 3, 5, and 7. The algorithms were successfully implemented on an NVIDIA K20c GPU. Compared to regular convolutions, the implementations for stride 2 are 1.44 times faster for a 3 × 3 kernel, 2.04× faster for a 5 × 5 kernel, 2.42× faster for a 7 × 7 kernel, and 1.73× faster for a 3 × 3 × 3 kernel. Additionally, a CNN accelerator using a novel processing element (PE) performs two 2-D Winograd stride 1, or one 2-D Winograd stride 2, and operations per clock cycle was implemented on an Intel Arria-10 field-programmable gate array (FPGA). We accelerated the original and our proposed modified VGG-16 architectures and achieved digital signal processor (DSP) efficiencies of 1.22 giga operations per second (GOPS)/DSPs and 1.33 GOPS/DSPs, respectively.**

***Index Terms*— Convolutional neural network (CNN), deep neural network, field-programmable gate array (FPGA), stride, Winograd.**

I. Introduction

CONVOLUTIONAL neural networks (CNNs) are widely used in many deep learning systems. CNNs have shown state-of-the-art accuracy in a variety of interdisciplinary research, including image classification [1]–[3], object detection [4]–[6], and speech recognition [7], [8], leading to their widespread adoption.

To reduce memory bandwidth requirements, recent research into CNN hardware acceleration has focused on increasing parallelism, reducing bits via quantization, or using fixed points rather than floating points [9], [10].

Manuscript received June 6, 2019; revised September 22, 2019 and November 22, 2019; accepted December 12, 2019. Date of publication January 13, 2020; date of current version March 20, 2020. This work was supported by the Research and Development Program of MOTIE/KEIT (Developing Processor-Memory-Storage Integrated Architecture for Low Power, High Performance Big Data Servers) under Grant 10077609. *(Corresponding author: Seok-Bum Ko.)*

The authors are with the Electrical and Computer Engineering Department, University of Saskatchewan, Saskatoon, SK S7N 5A9, Canada (e-mail: juan.yepez@usask.ca; seokbum.ko@usask.ca).

Color versions of one or more of the figures in this article are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2019.2961602

The Winograd minimal filtering algorithms (WMFAs), capable of being used for any stride [11], take advantage of overlapping computations between adjacent windows [12] to reduce the number of multiplications required for convolution, trading multiplication for addition. Given that the hardware required for multiplication is complex and large compared to that of a simple adder, the multiplication–addition tradeoff proposed by Winograd is desirable.

Modern CNN architectures replace pooling layers with strided convolutions for downsampling [13], where stride is defined as the element-wise shift displacement of a kernel over an input along a particular axis [14]. Convolutional layers learn feature properties during training; conversely, pooling is a fixed downsampling operation, and pooling layers have no trainable weights. A convolutional layer with stride >1 is advantageous in that it has trainable parameters and downsamples.

Recent architectures (e.g., the MobileNet family) use increasingly more layers with stride >1. Therefore, it is important to develop methods to process these layers efficiently. In this article, we introduce a novel way to use Winograd stride 1 algorithms to produce the effect of stride 2. We propose algorithms for 3 × 3 and 5 × 5 kernels for 1-D and 2-D cases. These algorithms require special cases of the Winograd algorithm that are described in this article. Therefore, the main contributions of this article are as follows.

1) Novel Winograd algorithms with stride 2 for 1-D, 2-D, and 3-D convolutions. These algorithms reduce the multiplication complexity of convolution. We provided a quantitative analysis of the required number of multiplications and additions.
2) We determined the matrices for 1-D Winograd $F(2,2)$ and $F(2,4)$ and for 2-D Winograd $F(2 \times 2, 2 \times 3)$, $F(2 \times 2, 3 \times 2)$, $F(2 \times 2, 3 \times 4)$, and $F(2 \times 2, 4 \times 3)$. These versions are required to solve the proposed Winograd with stride 2. To obtain the values of these respective matrices, the popular Winograd $F(2,3)$ is referenced and the Chinese Remainder Theorem is used. We do a quantitative analysis of these matrices.
3) We implemented the proposed algorithm on an NVIDIA K20c GPU. It shows 1.44× improvement for a 3 × 3 kernel, 2.04× improvement for a 5 × 5 kernel, 2.42× improvement for a 7 × 7 kernel, and 1.73× improvement for a 3 × 3 × 3 kernel.
4) We implemented a CNN accelerator on an Intel Arria-10 field-programmable gate array (FPGA) for both

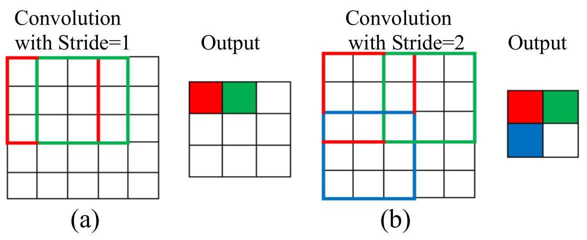

Fig. 1. Convolution with strides (a) 1 and (b) 2.

an original and modified VGG-16 architecture, which achieved digital signal processor (DSP) efficiencies of 1.22 giga operations per second (GOPS)/DSPs and 1.33 GOPS/DSPs, respectively. For the accelerator, we proposed a Winograd processing element (PE) for a 3×3 kernel, compatible for both stride 1 and 2 convolutions. Our combined one- and two-stride Winograd PE uses only 146 and 100 more lookup tables (LUTs) and registers, respectively, and the same number of DSPs (32), as a PE for two Winograd stride 1 calculations. It also uses 25 less DSPs than would be required by independent PEs for calculating two Winograd stride 1 and one Winograd stride 2 operations.

The rest of this article is organized as follows. Section II describes the background of this work. Related research is discussed in Section III. The proposed Winograd method with stride 2 is introduced in Section IV. Section V presents the CNN architectures with layers stride >1. The GPU implementation is shown in Section VI. Section VII shows the design, implementation, and results of the CNN accelerator and compares our proposed method to prior studies. Finally, Section VIII concludes this article.

II. Background

A. Convolutional Neural Networks

CNNs comprise a series of multiple layers, and each layer is convolved with a predefined kernel [1].

Input to the first layer in a CNN is the data the network is to analyze. Input to the next layer, and for all other layers, is the output feature map generated by the previous layer. The number of layers, the quantity of which is referred to as the depth of the CNN, can potentially reach hundreds—creating a need for massive and efficient computation.

With the conventional convolutional algorithm, each element in the output is computed individually by multiplying and accumulating the corresponding kernel and the input data.

B. Stride in CNNs

The stride controls how the kernel convolves around the input. When the stride is 1, the kernel is shifted over the input one element at a time. The stride is normally set in such a way that the output volume is an integer.

Fig. 1(a) shows a 5×5 input with a 3×3 kernel. With stride 1, the output matrix generated is of size 3×3. A stride of 1 is normally used to extract the maximum number of features, as it provides the maximum overlapping between the kernel and input—but at maximum computational complexity. Fig. 1(b) shows the kernel shifting by two units over the input, generating an output 2×2 matrix. Generally, when the stride is bigger than 1, the receptive fields overlap less. A smaller output is produced. If the stride were 3, there would be issues with spacing, as the receptive field would not fit around the input as an integer.

C. Winograd Algorithm

The conventional convolution algorithm is simple to implement, but it is not efficient. An efficient alternative convolution method can be realized via the WMFA [12].

In the case of a size 4 input data vector and size 3 kernel vector, conventional convolution requires six multiplications to generate the final result, whereas the Winograd algorithm requires only four multiplications.

The 1-D convolution using the Winograd algorithm can be formulated using the transformation matrices A, B, and G, input data d, and kernel g as follows:

$$Y = A^{\mathrm{T}}[[(Gg) \cdot (B^{\mathrm{T}}d)]]. \tag{1}$$

Lavin and Gray [12] introduced the matrices for $F(2,3)$. The matrices are also compatible for two dimensions. In the 2-D Winograd algorithm F($m \times m$, $r \times r$), the output size is $m \times m$, the kernel size is $r \times r$, and the input size is $n \times n$, where $n = m + r - 1$. The Winograd algorithm for two dimensions can be written in matrix form as

$$Y = A^{\mathrm{T}}[(GgG^{\mathrm{T}}) \cdot (B^{\mathrm{T}}dB)]A. \tag{2}$$

Shen *et al.* [15] proposed a method for the 3-D Winograd algorithm $F(m \times m \times m, r \times r \times r)$, which can be represented with the following equation:

$$Y = [A^{\mathrm{T}}[(GgG^{\mathrm{T}})^{R}G^{\mathrm{T}} \cdot (B^{\mathrm{T}}dB)^{R}B]A]^{R}A. \tag{3}$$

To obtain the convolution for $F(2 \times 2 \times 2, 3 \times 3 \times 3)$ using the 3-D Winograd algorithm, first, the 3-D matrix should be split into channels. For each channel, a 2-D Winograd transformation is applied to the kernel and the feature map. The new 3-D matrices containing the results should be rotated 90° clockwise. After this, an additional transformation using the transpose is required. At this point, the dot multiplications are performed. The results can be separated into channels. Utilizing these separate channels, the 2-D transformation is applied using A and rotated 90° clockwise. Then, the matrix is multiplied using A^{T}. Following these steps, the convolution result is achieved in a $2 \times 2 \times 2$ matrix.

III. Related Research

Researchers have developed many approaches to reducing the computational cost for CNNs. In [16], an algorithm improvement is proposed through the analysis of the algebraic properties of CNNs. The algorithm achieves 47% reduction in computation without affecting the accuracy. Lavin and Gray [12] used WMFA to develop new algorithms for stride 1 convolution. This algorithm is implemented for 2-D on a GPU and lessens multiplications by a factor of 2.25, thus achieving better performance than the CuDNN library. Lu *et al.* [17] and Huang *et al.* [18] implemented 2-D WMFA on an FPGA. WMFA uses fewer multiplications and little extra memory. In [19], an FPGA implementation using OpenCL is presented.

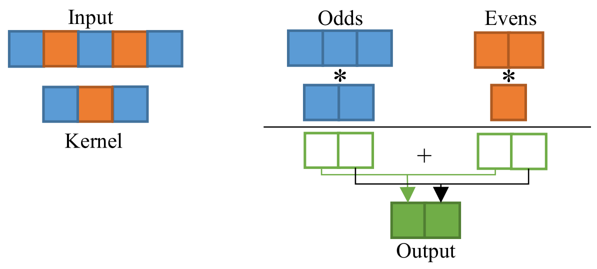

Fig. 2. Proposed convolution with stride = 2 for kernel = 3.

This implementation uses DSPs in parallel to process 1-D Winograd $F(4, 3)$, while using the entire FPGA's processing capacity for a more efficient implementation. In [15], an architecture for accelerating 3-D CNNs is presented, which is 13× faster than regular convolution. In [20], an instruction-driven CNN accelerator is proposed which supports the Winograd algorithm and cross-layer scheduling; their accelerator achieves 7× speedup compared to another cross-layer accelerator [21] on the same platform. In [17], a Winograd algorithm implementation is presented; their design uses a line buffer structure to reuse feature map data and achieves 2940 GOPS for VGG16 on a Xilinx ZCU102 platform. Qiu *et al.* [22] proposed an instruction-driven accelerator. However, the performance is limited for the large data transfer between on-chip and off-chip memory. In [23], a WMFA FPGA implementation is proposed that uses an optimization algorithm to determine a "fusion" and "strategy" for each layer. The implementation achieves 0.91 GOPS/DSPs for VGG-16.

IV. Winograd With Stride 2

A. One Dimensions

In 1-D convolution with stride 2, odd positions of the input are multiplied with odd positions of the kernel, and even-position input elements are multiplied with even-position kernel elements; no multiplication between odd-position and even-position elements occurs. Thus, we can separate the input and kernel elements into two groups: odd and even. Using these groupings, it is possible to convert a convolution of stride 2 into two convolutions of stride 1. Fig. 2 shows the procedure for a size 5 input vector and a size 3 kernel vector.

The even group contains two elements for the input and one element for the kernel. This cannot be further simplified. However, the odd group contains three elements for the input and two elements for the kernel, which can be calculated using the $F(2,2)$ Winograd.

There is no available $F(2,2)$ Winograd based on matrices. However, $F(2,2)$ can be determined using the popular $F(2,3)$ Winograd. In these matrices, the values of the variables g_2 and d_3 are zero when using $F(2,2)$ because the fourth row for both G and B^T are zero and can be deleted. Similarly, the fourth column of B^T and A^T and the third column of G can also be eliminated. Therefore, the matrices for $F(2,2)$ are as follows:

$$g = \begin{bmatrix} g_0 & g_1 \end{bmatrix}^T d = \begin{bmatrix} d_0 & d_1 & d_2 \end{bmatrix}^T$$
$$B^T = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 1 \\ 0 & -1 & 1 \end{bmatrix} G = \begin{bmatrix} 1 & 0 \\ 1/2 & 1/2 \\ 1/2 & -1/2 \end{bmatrix} A^T = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & -1 \end{bmatrix}. \quad (4)$$

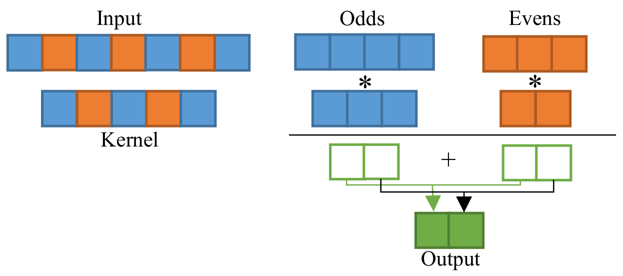

Fig. 3. Proposed convolution with stride = 2 for kernel = 5.

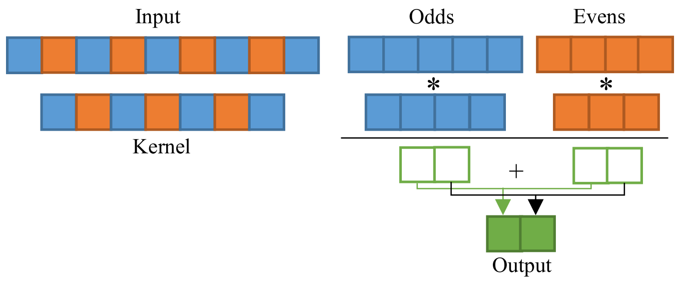

Fig. 4. Proposed convolution with stride = 2 for kernel = 7.

The minimal filtering algorithm for computing m outputs with an r-tap kernel, which is called $F(m, r)$, requires a number of multiplications defined by

$$\mu(F(m,r)) = m + r - 1. \quad (5)$$

The $F(2,2)$ algorithm uses just three multiplications and is therefore minimal by the formula $\mu(F(2,2)) = 2+2-1 = 3$. The algorithm also uses three additions involving the data, two additions and two multiplications by a constant involving the kernel, and three additions to reduce the products in the final result, whereas six is required for regular convolution.

To use a size 5 kernel with stride 2, the procedure is similar to using a size 3 kernel: there is no multiplication between the odd-position and even-position elements, and elements are separated into respective odd- and even-position groups. For a size 5 kernel, the input vector contains seven elements. The odd group contains four input elements and three kernel elements, allowing the Winograd $F(2,3)$ to be applied. The even group contains three input and two kernel elements, and Winograd $F(2,2)$ is used. Fig. 3 shows the procedure for a size 7 input vector and a size 5 kernel vector.

The odd-position group uses four multiplications $(\mu(F(2,3)) = 4)$, and the even-position group uses three multiplications $(\mu(F(2,2)) = 3)$. Therefore, the total number of multiplications is seven, whereas ten would be required for regular convolution.

For a size 7 kernel, the even group contains four input and three kernel elements, and the popular Winograd $F(2,3)$ will be used. The odd group contains five input elements and four kernel elements, allowing the Winograd $F(2,4)$ to be applied. Fig. 4 shows the procedure for a size 9 input vector and a size 7 kernel vector.

Because there is no available $F(2,4)$ Winograd based on matrices, we derive $F(2,4)$ using the Chinese Remainder Theorem, which is the same technique provide by Winograd.

The four-element kernel g and five-element data d are represented as polynomials in the following:

$$g(x) = g_3x^3 + g_2x^2 + g_1x + g_0$$
$$d(x) = d_4x^4 + d_3x^3 + d_2x^2 + d_1x + d_0.$$

The lineal convolution $g * d$ is

$$y(x) = g(x)d(x).$$

For the polynomial $m(x)$, we can write

$$y(x) = g(x)d(x) \bmod m(x).$$

Applying the Chinese Remainder Theorem, we solved the equations that can be represented as matrices, as shown in the following equation:

$$g = \begin{bmatrix} g_0 & g_1 & g_2 & g_3 \end{bmatrix}^T d = \begin{bmatrix} d_0 & d_1 & d_2 & d_3 & d_4 \end{bmatrix}^T$$
$$A^T = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} B^T = \begin{bmatrix} 2 & -1 & -2 & 1 & 0 \\ 0 & -2 & -1 & 1 & 0 \\ 0 & 2 & -3 & 1 & 0 \\ 0 & -1 & 0 & 1 & 0 \\ 0 & 2 & -1 & -2 & 1 \end{bmatrix}$$
$$G = \begin{bmatrix} 1/2 & 0 & 0 & 0 \\ -1/2 & -1/2 & -1/2 & -1/2 \\ -1/6 & 1/6 & -1/6 & 1/6 \\ 1/6 & 1/3 & 2/3 & 4/3 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \quad (6)$$

The odd-position group uses five multiplications ($\mu(F(2,4)) = 5$), and the even-position group uses four multiplications ($\mu(F(2,3)) = 4$). Therefore, the total number of multiplications is 9, whereas 14 would be required for regular convolution.

B. Two Dimensions

1) Using Kernel 3×3*:* To apply the Winograd with stride 2 in two dimensions, the elements of the input and the kernel should be separated into four groups. The first group comprises the elements at row-odd, column-odd indices of the input and kernel; the second one comprises the elements at row-odd, column-even indices of the input and kernel; the third one comprises the elements at row-even, column-odd indices of the input and kernel; the last one comprises the remaining four elements of the input and one element of the kernel. Fig. 5 shows the groups with a 3×3 kernel.

The first group contains 3×3 elements for input and 2×2 elements for the kernel. In this case, the Winograd $F(2 \times 2, 2 \times 2)$ can be used; the matrices for 2-D are the same as those for 1-D $F(2,2)$.

Two consecutive Winograd $F(2,2)$ operations can be used for the second and the third groups.

The fourth group cannot be further simplified, so regular multiplications are used. Each group's convolution produces a 2×2 matrix, and the final value of this process is simply the sum of these intermediate matrices.

The number of multiplications per group is as follows: nine multiplications for the first group; six multiplications for the second; six multiplications for the third; and four multiplications for the last. The total number of multiplications used is 25, whereas 36 is used with regular convolution.

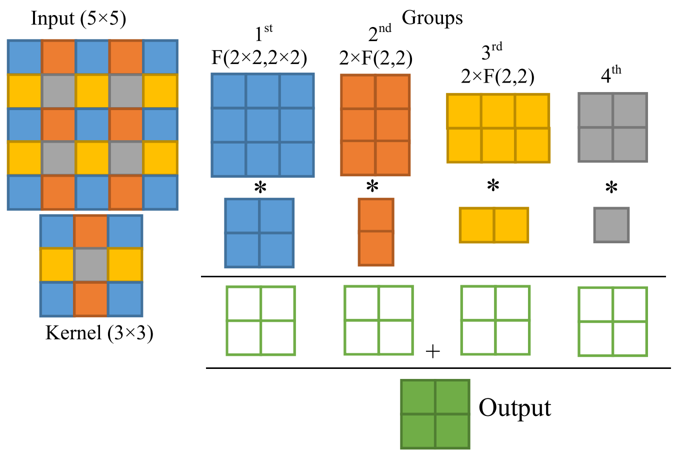

Fig. 5. Proposed convolution with stride = 2 for kernel = 3 × 3.

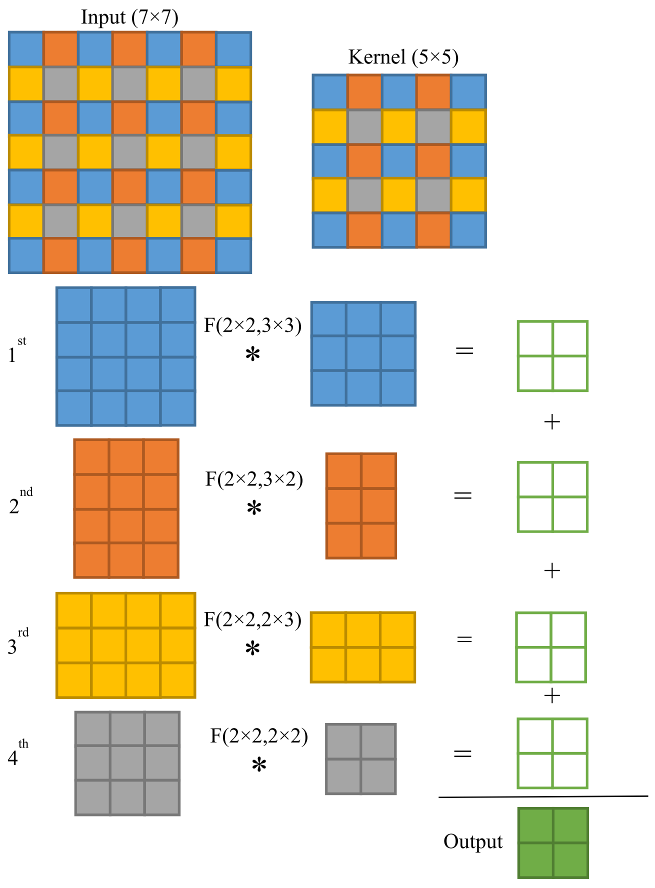

Fig. 6. Proposed convolution with stride = 2 for kernel = 5 × 5.

2) Using Kernel 5×5*:* For two dimensions with a 5×5 kernel, like the 3×3 kernel case, the elements of the (presumably 7×7) input and the kernel can be separated into four groups: the elements of the row-odd, column-odd indices of the input and kernel; the elements of the row-odd, column-even indices; the elements of the row-even, column-odd indices; and the remainder. Fig. 6 shows the groups with a 5×5 kernel.

The first group contains 4×4 elements for input and 3×3 elements for the kernel, which can be implemented using the popular Winograd $F(2 \times 2, 3 \times 3)$.

The second group has 4 × 3 input elements and 3 × 2 kernel elements. This special case can be implemented using a combination of the matrices of the different methods of Winograd. The matrices of $F(2,3)$ are used for A^{T}, G, and B^{T}, and the matrices of $F(2,2)$ are used for G^{T}, B, and A.

The third group has 3 × 4 input elements and 2 × 3 kernel elements. This is a swapped version of the second group's case, and matrices of $F(2,2)$ are used for A^{T}, G, and B^{T}, and the matrices of $F(2,3)$ are used for G^{T}, B, and A.

The fourth group contains 3 × 3 input elements with a 2 × 2 kernel, and uses Winograd $F(2 \times 2,2 \times 2)$.

The result of the convolution of the input 7 × 7 with a kernel 5 × 5 will be the sum of the intermediate 2 × 2 matrices produced via the respective convolutions of each group.

The numbers of multiplications per group using this algorithm are as follows: 16 for the first group; 12 in each of the second and third; and nine multiplications for the last. The total is 49 multiplications, whereas 100 would be used by regular convolution.

3) Using Kernel 7 × 7: There are popular architectures where the size 7 × 7 kernel with stride 2 is used at the first layer. Commonly, a kernel size 7 could be operated with fast Fourier transform (FFT). However, the stride 2 make the invalid FFT operation. Due to the procedure of separating the elements into odd- and even-position, it also reduces the size of the kernel for the operation. It makes possible the use of Winograd in an efficient manner.

For two dimensions with a 7 × 7 kernel, like the case of previous kernels, the input and the kernel can be separated into four groups: the elements of the row-odd, column-odd indices of the input and kernel; the elements of the row-odd, column-even indices; the elements of the row-even, column-odd indices; and the remainder. Fig. 7 shows the groups with a 7 × 7 kernel.

The result of the convolution of the input 9 × 9 with kernel 7 × 7 is the sum of the intermediate 2 × 2 matrices produced via the respective convolutions of each group. The number of multiplications per group using this algorithm is as follows: 25 for the first group; 20 in each of the second and third; and 16 multiplications for the last. The total is 81 multiplications, whereas 196 would be used by regular convolution.

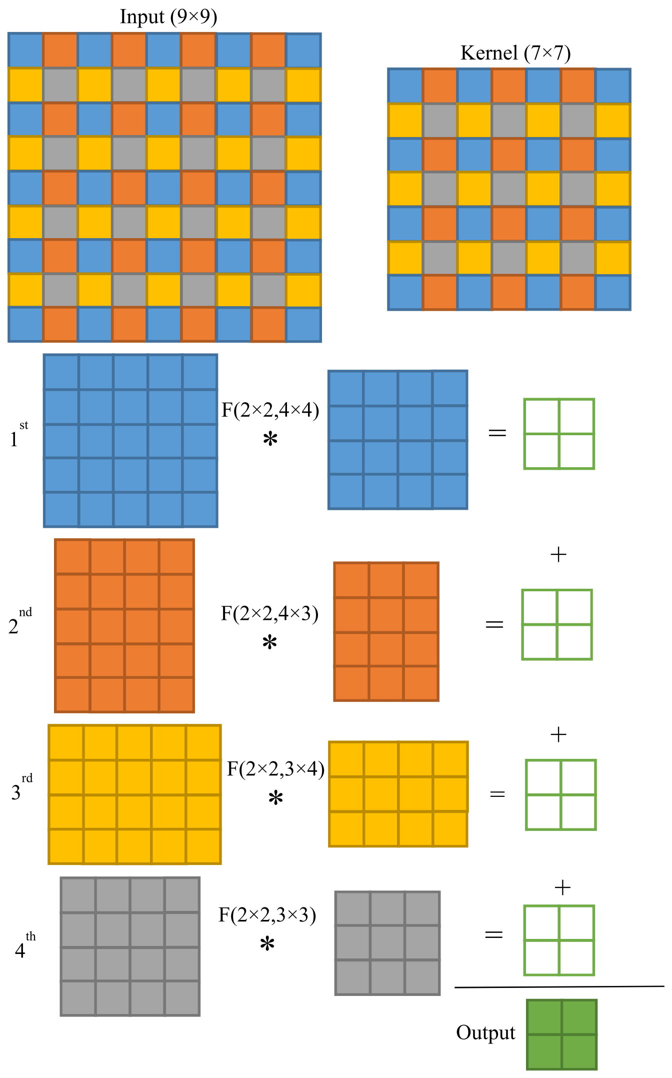

Fig. 7. Proposed convolution with stride = 2 for kernel = 7 × 7.

C. Three Dimensions

To apply Winograd with stride 2 in three dimensions using a 3 × 3 × 3 kernel, the elements of the input and the kernel should be in eight groups. The first group comprises the elements at row-odd, column-odd, channel-odd of the input and the kernel; the second group comprises the elements at row-odd, column-even, channel-odd indices of the input and the kernel; the third group comprises the elements at row-even, column-odd, channel-odd indices of the input and the kernel; the fourth group comprises the elements at row-odd, column-odd, channel-even indices of the input and the kernel; the fifth group comprises the elements at row-even, column-even, channel-odd indices of the input and the kernel; the sixth group comprises the elements at row-odd, column-even, channel-even indices of the input and the kernel; the seventh group comprises the elements at row-even, column-odd, channel-even indices of the input and the kernel form the seventh group; the last group comprises the remaining four elements of the input and one element of the kernel. Fig. 8 shows the groups with a 3 × 3 × 3 kernel.

The first group contains 3 × 3 × 3 elements for the input and 2 × 2 × 2 elements for the kernel. In this case, the Winograd $F(2 \times 2 \times 2,2 \times 2 \times 2)$ can be used; the matrices for three dimensions will be the same as those for 1-D $F(2,2)$.

Two consecutive 2-D Winograd $F(2 \times 2,2 \times 2)$ operations can be used for the second, third, and fourth groups.

Four consecutive 1-D Winograd $F(2,2)$ operations can be used for the fifth, the sixth, and the seventh groups.

The eighth group cannot be further simplified, so regular multiplications are used. Each group's convolution produces a 2 × 2 × 2 matrix, and the final value of this process is simply the sum of these intermediate matrices.

Two consecutive 2-D Winograd $F(2 \times 2,2 \times 2)$ operations can be used for the second, third, and fourth groups.

Four consecutive 1-D Winograd $F(2,2)$ operations can be used for the fifth, the sixth, and the seventh groups.

The eighth group cannot be further simplified, so regular multiplications are used. Each group's convolution produces a

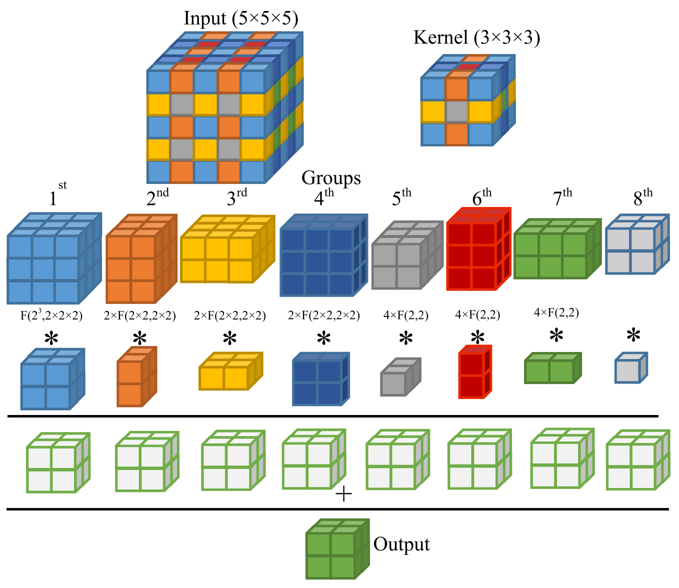

Fig. 8. Proposed convolution with stride = 2 for kernel = 3 × 3 × 3.

2 × 2 × 2 matrix, and the final value of this process is simply the sum of these intermediate matrices.

The number of multiplications per group is as follows: 27 multiplications for the first group; 18 multiplications in each of the second, third, and fourth; 12 multiplications in each of the fifth, sixth, and seventh; and 8 multiplications for the last. The total multiplications are 125, whereas 216 would be used with the regular convolution.

Because 5 × 5 and 7 × 7 kernels are commonly used only for the input layer (assuming an RGB color image), it is not practical to extend our stride 2 methodology to 3-D kernels other than 3 × 3 × 3; therefore, we did not implement a 5 × 5 × 3 or 7 × 7 × 3 kernel.

V. CNN Architectures With Layers Stride >1

Modern CNN architectures use convolutional layers for feature learning followed by max-pooling layers to downsample feature maps. However, max-pooling is a fixed operation that trades spatial structure knowledge for improved computational efficiency in future layers. Although spatial knowledge is less important for classification tasks, where the goal is recognizing whether a certain region of interest exists in an image, it is significant in tasks where positional information is important, such as object detection. For these tasks, strided convolution improves the computational efficiency of future layers while preserving spatial information. In [13], the pooling layers are replaced by an additional convolutional layer with stride 2; results show performance stabilization, and even accuracy improvement compared to the base model. This concept has led to the adoption of >1 stride convolutional layers in modern CNN architectures, and the use of stride 2 is a popular choice. Table I highlights the stride >1 layers in recent architectures and their utilized kernel sizes.

AlexNet [24] sparked the current DL revolution. It uses an 11 × 11 kernel with stride 4. However, with large kernels like 11 × 11, the Winograd algorithm is outperformed by other methods (e.g., FB-FFT) [12].

TABLE I
Architecture Layers Using Stride >1

Architecture	Kernels	Stride	Number of layers
AlexNet [24]	11×11	4	1
ZFNet [25]	7×7	2	1
GoogLeNet [26]	7×7	2	1
ResNet [27]	7×7, 3×3	2	1,3
SqueezeNet [28]	7×7	2	1
YOLO [29]	7×7, 3×3	2	1,1
SSD [30]	3×3	2	2
MobileNets [31]	3×3	2	6
MobileNetV2 [32]	3×3	2	5
MobileNetV3 [33]	5×5, 3×3	2	2,3

TABLE II
Comparisons of the Regular Convolution and Winograd Stride 2

Algorithms	Regular Convolution		Winograd	
	muls	adds	muls	adds
F(2,3)	6	6	5	11
F(2,5)	10	10	7	21
F(2,7)	14	14	9	27
F(2×2, 3×3)	36	36	25	77
F(2×2, 5×5)	100	100	49	137
F(2×2,7×7)	196	196	81	243
F(2×2×2, 3×3×3)	216	216	125	419

ZFNet [27], GoogLeNet [26], ResNet [27], SqueezeNet [28], and YOLO [29] use a 7 × 7 kernel with stride 2 in the first layer, allowing 1-D and 2-D Winograd to be used. Three-dimensional Winograd for a 7 × 7 × 7 kernel is unavailable for the first layer, so this has only three input channels. However, the 3-D Winograd can be used for later layers when the input depth is larger than 7.

ResNet [25], YOLO [29], SSD [30], MobileNet [31], and MobileNetV2 [32] use a 3 × 3 kernel. MobileNetV3 [33] uses three and two layers with 3 × 3 and 5 × 5 kernels, respectively. The convolutions with stride 2 can be applied using our proposed Winograd algorithms.

VI. GPU Implementation

We implemented our method for 1-D and 2-D stride 2 Winograd on an NVIDIA K20c GPU and tested it using several convolutions with 3 × 3, 5 × 5, and 7 × 7 kernel sizes. For three dimensions, we used a 3 × 3 × 3 kernel. Our implementation was compared against regular stride 2 convolutions. A comparison of the numbers of multiplications and additions for all applicable dimensions is shown in Table II.

The implemented program uses the same settings from the stride 2 layers of MobileNet (an input size of 224 × 224). The program recorded the GPU processing times and generated the average processing times for regular convolution and for our proposed algorithms. The results are shown in Table III.

Compared to regular convolution, our method is 1.44×, 2.04×, 2.42×, and 1.73× faster for the respective 3 × 3, 5 × 5, 7 × 7, and 3 × 3 × 3 kernels. Furthermore, we tested the results of our algorithms with single precision (fp32) data. The results show the same values as direct convolution, indicating

TABLE III
PERFORMANCE COMPARISON USING STRIDE 2 IN GPU

Kernel	Regular convolution	Ours	Speedup
3×3	8.09ms	5.61ms	1.44×
5×5	11.21ms	5.49ms	2.04×
7×7	13.21ms	5.46ms	2.42×
3×3×3	15.45ms	8.93ms	1.73×

TABLE IV
MODIFIED VGG-16 ARCHITECTURE WITH CONVOLUTION STRIDE 2

Layer	Type/Stride	Filter Shape	Input size
Conv1-1	Conv/S1	3×3×3×64	224×224×3
Conv1-2	Conv/S2	3×3×3×64	224×224×64
Conv2-1	Conv/S1	3×3×3×64	112×112×64
Conv2-2	Conv/S2	3×3×3×64	112×112×128
Conv3-1	Conv/S1	3×3×3×64	56×56×128
Conv3-2	Conv/S1	3×3×3×64	56×56×256
Conv3-3	Conv/S2	3×3×3×64	56×56×256
Conv4-1	Conv/S1	3×3×3×64	28×28×512
Conv4-2	Conv/S1	3×3×3×64	28×28×512
Conv4-3	Conv/S2	3×3×3×64	28×28×512
Conv5-1	Conv/S1	3×3×3×64	14×14×512
Conv5-2	Conv/S1	3×3×3×64	14×14×512
Conv5-3	Conv/S2	3×3×3×64	14×14×512
Flatten	Flatten	-	7×7×512
FC2	Dense	4096	1×25088
FC2	Dense	4096	1×4096
Predictions	Dense	1000	1×4096

that our mathematical transformations do not lose precision and therefore will affect neither multiplication accuracy nor neural network performance.

VII. CNN ACCELERATORS

A. CNN Architectures

In this article, we propose a CNN accelerator design based on strides 1 and 2 Winograd for FPGAs. The accelerator implements the VGG-16 architecture [27]. Because VGG-16 does not contain any layers of stride 2, we also implemented a modified VGG-16 architecture using a similar methodology to that implemented by Springenberg *et al.* [13] for a custom network, in which one convolution and one max pooling layer are replaced by one convolutional layer of stride 2. The modified architecture can be seen in Table IV.

The original VGG-16 model was trained on the ImageNet database [34], which contains 1M images over 1000 classes; this allows the network to learn varied and representative features for many types of images. Moreover, building and training a model without preinitialized weights is not always feasible due to the inherent computational restraints of training.

Therefore, we used the weights for the ImageNet pretrained VGG-16 model, which are available online [35], for transfer learning—where the trained model weights are reused to initialize training a model for a different task. We reused the weights of the original VGG-16 to train our modified architecture using stride 2, which has the same number of parameters as the original VGG-16.

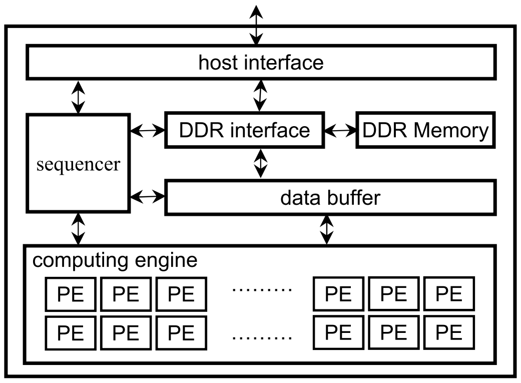

Fig. 9. Deep learning architecture.

Nonmodified layers preserve the (frozen) weights from the original model. We trained only the added convolution stride 2 layers, or 5 494 208 of 138 357 544 total parameters; thus, our model's training time is reduced. After only 200 epochs, our modified model achieved the same accuracy (91.8%) as the original VGG-16 on the Caltech-101 data set [36]. Our proposed modified VGG-16 architecture has less computational cost than the original VGG-16; this is because stride 2 convolution requires less multiplications than stride 1, and a stride 1 convolution requires an additional pooling layer for downsampling. For example, the output of layer Conv5-3 from the modified VGG-16 has a size of $7 \times 7 \times 512$, while that from the original VGG-16 is $14 \times 14 \times 512$; a max-pooling layer of stride 2 is required for the layer shape to become $7 \times 7 \times 512$ in the stride 1 case. Thus, when compared to the original architecture, four times fewer multiplications are required for the convolutional layers in our stride 2-modified VGG-16 architecture, which improves the throughput significantly whilst maintaining high accuracy.

B. FPGA Implementation

We accelerated both the original and our modified VGG-16 architectures on an Intel Arria-10 FPGA with 1150K logic elements, 1518 DSP blocks, and 2131 M20K RAMs, at a clock speed of 250 MHz. We used 16-bit fixed-point precision to evaluate our design. Fig. 9 shows the architecture which contains a host interface, double data rate (DDR) memory, on-chip buffer, sequencer, and PEs.

The host interface receives instructions and input data from and sends results to a host. The sequencer decodes instructions from which the appropriate components receive operating commands. Host data and PE results are saved in DDR memory. An on-chip buffer stores the data to be processed in the current operation. All computations are done within PEs.

C. Memory Access

Due to the limited on-chip buffer and the large amount of data required by CNNs, the accelerator is not able to transfer all the data from external memory into the on-chip buffer. That is why the input and the kernel data are divided into several groups, allowing for data reuse in both the input and the kernel. Multiple kernel groups are stored in the buffer and processed

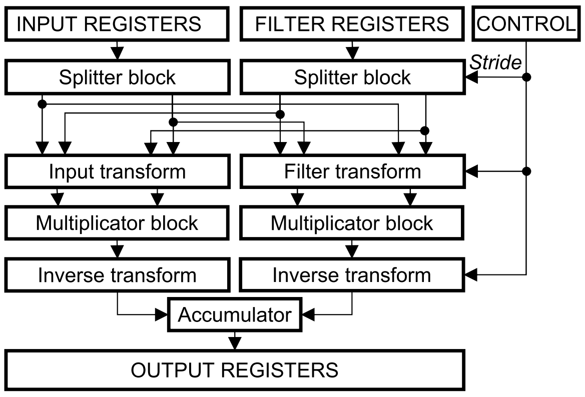

Fig. 10. Proposed PE architecture.

in parallel with the same input group. The partial results from the PEs are accumulated into the output buffer until the convolutional result is generated. Then, the results are moved to external memory. After this process, a new input group is loaded and the same kernels can be reused. In our design, double buffers are used to overlap data communication and reduce memory access delay in reading and writing data to memory.

D. Proposed PE Architecture

We designed our PEs to accommodate two Winograd $F(2 \times 2, 3 \times 3)$ stride 1 operations. As each Winograd uses 16 multiplications, 32 multipliers are required for each PE. We added some logical elements to the PE to make it compatible with our proposed Winograd stride 2, without using additional DSPs. Because VGG-16 uses the same filter size (3×3) for all convolutional layers, the PEs can be reused.

For headings 1–8 below, please refer to Section IV-B and Fig. 5 for information on Winograd stride 2 groups.

1) Input Tile From Registers: The input tile size of our design is 5×6, where the PE may process two Winograd stride 1 or one Winograd stride 2 in one clock cycle.

For Winograd stride 1, our PE allows two Winograd stride 1 in parallel, 32 multiplications are performed, and 32 multipliers are needed.

Our PE allows one Winograd stride 2, a process requiring 25 multiplications. If the PE used regular convolution stride 2 instead of our proposed Winograd, it would require 36 multipliers.

2) Splitter Block: We use the control signal "stride" to split input data based on the type of stride to process (one or two), that is, how the input is used depending on the stride (as shown in Fig. 10).

For stride 1, the block splits a 4×6 array from the input tile into two 4×4 arrays which are sent to the Transform block; this is possible because two columns have the same input for both Winograd modules.

For stride 2, a 5×5 input array is used, and the splitters split the input into four groups. Group 1 is sent to the input transform block and Groups 2–4 are rearranged to generate another input for the input transform block.

3) Input Transform Block: The input transform block contains two parts. The first part uses the 2-D transform of Winograd stride 1, which can also process the Group 1

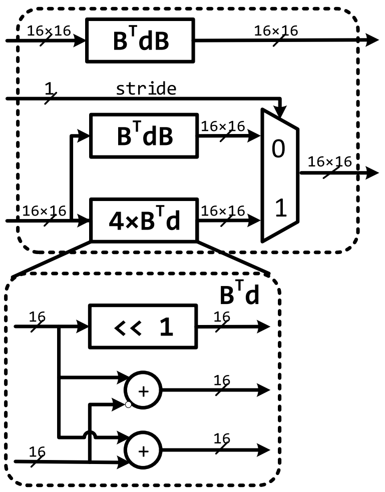

Fig. 11. Input transform block.

TABLE V
RESOURCE UTILIZATION OF DIFFERENT PEs FOR KERNEL = 3 × 3 ON INTEL ARRIA-10

Resource	LUTs	Registers	DSPs
Winograd stride two	584	304	25
Two Winograd stride one	930	487	32
Proposed method	1076	587	32

of Winograd stride 2. The second part transforms the 2-D Winograd stride 1 when the control signal "stride" is low; when the control signal is high, this part performs four 1-D Winograd transforms for Groups 2 and 3 of Winograd stride 2, and also four multiplications for Group 4. This is shown in Fig. 11.

4) Filter Transform Block: Our PE has one filter transform module controlled by the control signal "stride." If the signal level is low, a 2-D Winograd transform is processed into the kernel and the same output is sent to the DSP module for multiplications; if the signal is high, the outputs are arranged according to the kernel described in Section IV-B.

5) Multiplication Blocks: To perform the convolution operations, multiplications between the input transforms and their corresponding kernel transforms are necessary. This block requires 32 multiplications for the whole process. Similar to [17], we use an array of DSPs to perform the multiplications. The outputs of this block are two 4×4 arrays.

6) Inverse Transform: After the multiplications, an inverse transform is applied to the results. The first part uses the inverse two dimensions of Winograd stride 1. The second part transforms an inverse 2-D Winograd stride 1 when the control signal "stride" is low; when the control signal is high, this part performs four inverse 1-D Winograd transformations for Groups 2 and 3 of Winograd stride 2.

7) Accumulator Block: For stride 2, the results of the inverse transform for each Group (1–4) are summed and sent to the register. For the case of stride 1, the results skip this block and they are sent to the register directly.

Table V compares the FPGA's resource utilization for the proposed Winograd PE compatible with strides 1 and 2 against

TABLE VI
RESOURCE UTILIZATION OF CNN ACCELERATOR ON INTEL ARRIA-10

Resource	LUTs	M20K RAMs	DSPs
Available	1150K	2131	1518
Used	181K	1310	1344
Utilization	15.7%	61.5%	88.5%

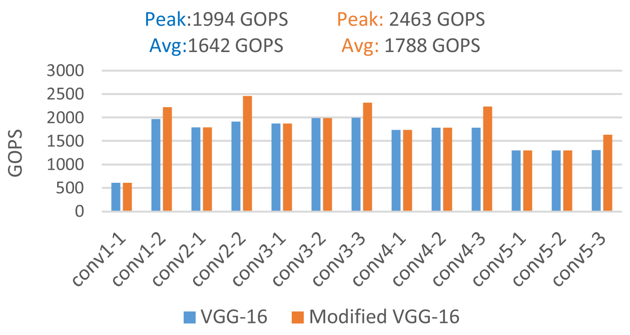

Fig. 12. Evaluation results of VGG-16 and modified VGG-16.

both a PE capable of two Winograd stride 1 operations, and a PE calculating one stride 2 Winograd operation. Although our proposed PE uses more LUTs and registers than the other Winograd modules, ours can perform Winograd operations with both strides—unlike the other PEs. Compared to the two Winograd stride 1 PEs, our PE requires only 146 and 100 more LUTs and registers, respectively. Moreover, our PEs, with the same DSP usage as that of the two Winograd stride 1, allows the additional processing of Winograd stride 2. DSPs are a critical resource for FPGAs, and our design uses only 32 DSPs; compare this to the combined 57 required by the independent two Winograd stride 1 PE and the Winograd stride 2 PE.

As demonstrated in Table V, our proposed PE can perform convolution of either stride 1 or 2, enabling its usage in a variety of architectures, including those listed in Table I. The flexibility of our proposed solution makes our PE practical for recent and upcoming novel neural network architectures that use stride 2 convolution to lower their inference processing time.

E. Parallelization

CNN architectures (including VGG-16, ResNet, MobileNet, and others) use an input image size of 224 × 224. Each network block, comprised of convolutional layers and oftentimes ending with a pooling layer, downsamples the feature map by half before outputting a new volume to be used as input to the next block. Thus, the last feature map of these networks prior to the fully connected or prediction layers has a width and height of 7 × 7.

To process the feature maps efficiently, we arrange seven PEs in a single block to parallelize convolution. This allows for a 28 × 5 input tile for each block. Inside the block, the PEs can read and share the input tile, thereby avoiding multiple memory reads for the same data. Our CNN accelerator uses six of these blocks. Although the blocks need to share an input tile and kernel data from the same buffer, connecting multiple blocks directly to the buffer may cause the system to experience issues in timing closure, especially with a high clock rate. To avoid this issue, we organize the blocks in a systolic array architecture [37]. Our design array consists of three rows and two columns, and the rows read input data while the columns read kernel data. Local buffers are used to cache the input and kernel data from each block. Thus, every time a block receives a new data tile, the previous data is also delivered. Using a double buffer in each block also allows computation and data delivery to occur simultaneously. Preliminary results from each block are accumulated in a local buffer until the entire convolution is done, after which the results can be written to external memory.

F. Results

In our implementation, we evaluated the Winograd algorithm for both strides 1 and 2 using the original and the proposed modified version of VGG-16.

Fig. 12 shows the results of all convolutional layers in the FPGA using the original and the modified VGG-16 networks. The average performance in the modified VGG-16 network is 8.9% higher than the original VGG-16, owing to higher GOPS (and a higher peak of 2463 GOPS) during stride 2 operations.

The performance of the first layer (conv1-1) is the worst. This is because the first layer has only three input channels and a large input size which is divided into many groups, taking more time to initialize the elements. Our parallelization scheme is efficient when the size of the input feature map and the output layer depth (like in the case of conv2) are well balanced. However, when the input feature maps are very small, for example, the lattermost layer (conv5), inefficient memory access patterns cause throughput reduction.

Table VI shows the resource utilization of our CNN accelerators on an Intel Arria-10 FPGA clocked at 250 MHz. The Winograd algorithms reduce the required number of DSPs per convolution; having fewer DSPs per PE allows for more PEs, which enables the possibility of greater parallelism, and multiple blocks can perform multiple convolutions in parallel.

We use 88.5% of the DSPs available on the FPGA to achieve high throughput. The Winograd algorithm uses more LUTs than regular convolution to compensate for the reduced number of multiplications. However, even with that increase, our design only used 15.7% of the FPGA's available LUTs. Because our design stores groups of input and kernel data in on-chip memory, 61.5% of the available RAM is used. In summary, by using Winograd, we can use the available DSPs efficiently to improve the performance of the accelerator.

Table VII compares our method's overall performance and DSP efficiency against previous FPGA CNN acceleration works. Because one multiplication operation consumes only one DSP and the additional operation does not use a DSP, these implementations used 16-bit fixed point data. Using 16-bit fixed-point data is more practical for an FPGA, and the results show that this numeric conversion contributes to only 0.4% accuracy loss [22].

Overall performance throughput depends on the number of resources used by the FPGA. DSP efficiency (GOPS/DSPs) is a fair comparison metric for overall performance because these works were implemented using different FPGA platforms with different numbers of available DSPs. Our modified

TABLE VII
PERFORMANCE COMPARISON WITH THE STATE-OF-THE-ART FPGA ACCELERATORS

	[22] 2016	[20] 2017	[17] 2017	[23] 2017	[18] 2018	[15] 2018	Ours	
Platform	Zynq XC7Z045	VirtexVX6 90T	MPSOC ZCU102	Zynq ZC706	VCU440	VCU440	Arria-10	Arria-10
CNN	VGG-16	VGG-16	VGG-16	VGG-16	VGG-16	VGG-16	VGG-16	Modified VGG-16
Freq(MHz)	150	200	200	100	200	200	250	250
Precision	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed	16-bit fixed
Winograd	F(2×2,3×3)	F(2×2,3×3)	F(4×4,3×3)	F(4×4,3×3)	F(2×2,3×3)	F(2×2,3×3)	F(2×2,3×3)	F(2×2,3×3)
Accuracy top-5	86.66%	N/A	84.6%	N/A	N/A	N/A	91.4%	91.4%
Logic Cell	183K	N/A	600K	156K	225K	189K	181K	180K
Used DSPs	780	2048	2304	725	756	1376	1344	1344
Performance (GOPS)	137	1467	3044	660	943	821	1642	1788
Efficiency (GOPS/DSPs)	0.18	0.716	1.32	0.91	1.2	0.6	1.22	1.33

TABLE VIII
COMPARISON WITH GPU PLATFORM

Device	TitanX	Arria-10
Precision	32 bits float	16 bits fixed
Performance (TOPS)	5.6	1.6
Power (W)	134	18
Energy efficiency (GOPS/W)	41.8	88.8

VGG-16 implementation achieves 1.33 GOPS/DSPs of DSP efficiency, which outperforms other implementations. The modified VGG-16 network uses convolutional layers of stride 2, where one stride 2 convolution is used instead of one stride 1 convolution and one max-pooling layer. The PEs in our implementation can be used for stride 1 or 2. Thus, our design has a better efficiency when it is used in architectures that contain stride 1 and 2 layers.

For standard VGG-16 implementations which contain only stride 1 convolutional layers, this work has the second highest DSP efficiency at 1.22 GOPS/DSPs, behind only [17]. However, the Winograd algorithm in [17] is $F(4 \times 4{,}3 \times 3)$ which uses an input tile of 6×6 elements; although a larger Winograd input tile leads to a higher throughput, the accuracy is reduced by 6.8% due to a truncation of intermediate values because the transformation matrices cannot be simply converted into shift operations [20]. To perform strides 1 and 2 Winograd, we used multiple Winograd $F(2 \times 2, 3 \times 3)$, which requires division by 2. This division is performed using shift operations with an additional bit; this avoids accuracy reduction when using Winograd when compared to conventional convolution.

G. Comparison With GPU

Table VIII shows a performance comparison between implementations of VGG-16 on an Arria-10 FPGA and an NVIDIA TITAN X GPU. The GPU implementation [17] uses the Caffe framework [38] with CuDNN 5.1, which includes Winograd algorithms. For a fair comparison against our Winograd FPGA implementation, all the layers in the GPU's VGG-16 architecture are forced to use Winograd.

As shown in Table VIII, the GPU implementation achieves a higher TOPS than our FPGA implementation. However, our FPGA implementation achieves much better (2.12×) energy efficiency.

VIII. CONCLUSION

We introduced new algorithms for CNNs with stride 2 based on 1-D, 2-D, and 3-D WMFAs. These algorithms separate the input and the kernel into odd-position and even-position groups, and stride 2 output is equivalent to the summation of many stride 1 convolutions. This idea allows for greater efficiency and advantages for CNN architectures that use stride 2.

These groups have different array sizes and can be computed using the popular Winograd $F(2 \times 3)$ and our proposed Winograd $F(2{,}2)$, $F(2{,}4)$, $F(2 \times 2{,}2 \times 3)$, $F(2 \times 2{,}3 \times 2)$, $F(2 \times 2{,}4 \times 4)$, $F(2 \times 2{,}4 \times 3)$, or $F(2 \times 2{,}3 \times 4)$. We determined these new Winograd versions based on the popular Winograd $F(2{,}3)$ and derived them using the Chinese Remainder Theorem, which decreases computational complexity and increases efficiency by trading expensive multiplications for cheap additions. Our GPU implementations of 1-D, 2-D, and 3-D stride 2 Winograd are tested on several stride 2 layers and show our method is 1.44×, 2.04×, 2.42×, and 1.73× faster for the respective 3×3, 5×5, 7×7, and $3 \times 3 \times 3$ kernels.

We design a Winograd PE which can process strides 1 and 2 in the same module. Our combined PE uses the same number of DSPs (32) as a PE for two Winograd stride 1 calculations, and 25 less DSPs than that required by having independent PEs to calculate two Winograd stride 1, and one Winograd stride 2, operations. This optimization allows an increase in the number of total PEs. Using a systolic array with a double buffer, greater parallelism was enabled in our design.

Stride 2 convolutional layers produce smaller output volumes while still extracting feature maps. This was demonstrated with our implementation of our proposed modified VGG-16 architecture where one stride 2 convolutional layer is used instead of one stride 1 convolutional layer followed by one max-pooling layer. Our implementation achieves DSP efficiencies of 1.22 GOPS/DSPs and 1.33 GOPS/DSPs for the original and modified VGG-16 architectures, respectively. Because our proposed design can effectively handle both strides 1 and 2 layers within the same architecture, our proposed method is a new technique for emerging architectures which contain layers using stride 1 and/or 2, and those increasing the numbers of stride two layers used.

For future work, we plan to use our modified VGG-16 architecture and its FPGA implementation as a baseline for accelerating other neural network architectures, particularly those which naturally use stride 2 convolutional layers, such as the MobileNet family.

References

[1] A. Romero, C. Gatta, and G. Camps-Valls, "Unsupervised deep feature extraction for remote sensing image classification," *IEEE Trans. Geosci. Remote Sens.*, vol. 54, no. 3, pp. 1349–1362, Mar. 2016.

[2] S.-W. Chen and C.-S. Tao, "PolSAR image classification using polarimetric-feature-driven deep convolutional neural network," *IEEE Geosci. Remote Sens. Lett.*, vol. 15, no. 4, pp. 627–631, Apr. 2018.

[3] L. Jiao and F. Liu, "Wishart deep stacking network for fast POLSAR image classification," *IEEE Trans. Image Process.*, vol. 25, no. 7, pp. 3273–3286, Jul. 2016.

[4] Y. Yu, H. Guan, and Z. Ji, "Rotation-invariant object detection in high-resolution satellite imagery using superpixel-based deep Hough forests," *IEEE Geosci. Remote Sens. Lett.*, vol. 12, no. 11, pp. 2183–2187, Nov. 2015.

[5] T.-N. Le and A. Sugimoto, "Video salient object detection using spatiotemporal deep features," *IEEE Trans. Image Process.*, vol. 27, no. 10, pp. 5002–5015, Oct. 2018.

[6] J. Han, D. Zhang, X. Hu, L. Guo, J. Ren, and F. Wu, "Background prior-based salient object detection via deep reconstruction residual," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 25, no. 8, pp. 1309–1321, Aug. 2015.

[7] B. Wu *et al.*, "An end-to-end deep learning approach to simultaneous speech dereverberation and acoustic modeling for robust speech recognition," *IEEE J. Sel. Top. Signal Process.*, vol. 11, no. 8, pp. 1289–1300, Dec. 2017.

[8] P. Zhou, H. Jiang, L.-R. Dai, Y. Hu, and Q.-F. Liu, "State-clustering based multiple deep neural networks modeling approach for speech recognition," *IEEE/ACM Trans. Audio Speech Lang. Process.*, vol. 23, no. 4, pp. 631–642, Apr. 2015.

[9] S. Park *et al.*, "An energy-efficient and scalable deep learning/inference processor with tetra-parallel MIMD architecture for big data applications," *IEEE Trans. Biomed. Circuits Syst.*, vol. 9, no. 6, pp. 838–848, Dec. 2015.

[10] M. Alam, L. S. Vidyaratne, and K. M. Iftekharuddin, "Sparse simultaneous recurrent deep learning for robust facial expression recognition," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 29, no. 10, pp. 4905–4916, Oct. 2018.

[11] S. Winograd, *Arithmetic Complexity of Computations*. Philadelphia, PA, USA: SIAM, 1980.

[12] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 4013–4021.

[13] J. T. Springenberg *et al.*, "Striving for simplicity: The all convolutional net," 2015, *arXiv:1412.6806*. [Online]. Available: https://arxiv.org/abs/1412.6806

[14] J. Hannink *et al.*, "Mobile stride length estimation with deep convolutional neural networks," *IEEE J. Biomed. Health Inform.*, vol. 22, no. 2, pp. 354–362, Mar. 2018.

[15] J. Shen, Y. Huang, Z. Wang, Y. Qiao, M. Wen, and C. Zhang, "Towards a uniform template-based architecture for accelerating 2D and 3D CNNs on FPGA," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays-FPGA*, 2018, pp. 97–106.

[16] J. Cong and B. Xiao, "Minimizing computation in convolutional neural networks," in *Artificial Neural Networks and Machine Learning—ICANN*. Springer, 2014, pp. 281–290.

[17] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr./May 2017, pp. 101–108.

[18] Y. Huang, J. Shen, Z. Wang, M. Wen, and C. Zhang, "A high-efficiency FPGA-based accelerator for convolutional neural networks using Winograd algorithm," in *Proc. J. Phys., Conf. Ser.*, May 2018, vol. 1026, no. 1, Art. no. 012019.

[19] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL deep learning accelerator on arria 10," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, 2017, pp. 55–64.

[20] J. Yu *et al.*, "Instruction driven cross-layer CNN accelerator with Winograd transformation on FPGA," in *Proc. Int. Conf. Field Program. Technol. (ICFPT)*, Dec. 2017, pp. 227–230.

[21] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Oct. 2016, p. 1–12.

[22] J. Qiu *et al.*, "Going deeper with embedded FPGA platform for convolutional neural network," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays-FPGA*, 2016, pp. 26–35.

[23] Q. Xiao, Y. Liang, L. Lu, S. Yan, and Y.-W. Tai, "Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs," in *Proc. 54th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, Jun. 2017, pp. 1–6.

[24] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[25] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *Computer Vision—ECCV*. Springer, 2014, pp. 818–833.

[26] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.

[27] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2016, pp. 770–778.

[28] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5MB model size," in *Proc. Int. Conf. Learn. Representations*, 2017, pp. 1–13.

[29] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 779–788.

[30] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Computer Vision—ECCV*. Springer, 2014, pp. 21–37.

[31] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*. [Online]. Available: https://arxiv.org/abs/1704.04861

[32] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted residuals and linear bottlenecks," 2018, *arXiv:1801.04381*. [Online]. Available: https://arxiv.org/abs/1801.04381

[33] A. Howard *et al.*, "Searching for MobileNetV3," 2019, *arXiv:1905.02244*. [Online]. Available: https://arxiv.org/abs/1905.02244

[34] *ImageNet*. Accessed: Jun. 5, 2019. [Online]. Available: http://www.image-net.org

[35] *Pre-Trained Models*. Accessed: Jun. 5, 2019. [Online]. Available: https://github.com/tensorflow/models/tree/master/research/

[36] L. Fei-Fei, R. Fergus, and P. Perona, "Learning generative visual models from few training examples: An incremental Bayesian approach tested on 101 object categories," in *Proc. Conf. Comput. Vis. Pattern Recognit. Workshop*, Apr. 2005, pp. 178–186.

[37] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. 54th Annu. Design Autom. Conf.*, 2017, p. 29.

Juan Yepez received the bachelor's degree in electronics and telecommunications engineering from Escuela Politecnica Nacional, Quito, Ecuador, in 2008, and the M.Sc. degree in electrical and computer engineering from the University of Saskatchewan, Saskatoon, SK, Canada, in 2017, where he is currently working toward the Ph.D. degree in electrical and computer engineering.

Seok-Bum Ko (SM'04) received the Ph.D. degree in electrical and computer engineering from the University of Rhode Island, USA, in 2002.

He is currently a Professor with the Department of ECE, University of Saskatchewan, Canada. His research interests include computer arithmetic, computer architecture, and biomedical engineering.

Dr. Ko is a Senior Member of the IEEE Circuits and Systems Society and an Associate Editor of the IEEE ACCESS and the IEEE TRANSACTIONS ON CIRCUIT AND SYSTEMS I: REGULAR PAPERS.