2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP) | 978-1-6654-2701-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASAP52443.2021.00045

WinoCNN: Kernel Sharing Winograd Systolic Array for Efficient Convolutional Neural Network Acceleration on FPGAs

Xinheng Liu*, Yao Chen†, Cong Hao‡, Ashutosh Dhar*, Deming Chen*,†
*University of Illinois at Urbana-Champaign, IL, USA, †Advanced Digital Sciences Center, Singapore
‡Georgia Institute of Technology, GA, USA
Email: xliu79@illinois.edu, yao.chen@adsc-create.edu.sg, callie.hao@gatech.edu, {adhar2, dchen}@illinois.edu

***Abstract*—The combination of Winograd's algorithm and systolic array architecture has demonstrated the capability of improving DSP efficiency in accelerating convolutional neural networks (CNNs) on FPGA platforms. However, handling arbitrary convolution kernel sizes in FPGA-based Winograd processing elements and supporting efficient data access remain underexplored. In this work, we are the first to propose an optimized Winograd processing element (WinoPE), which can naturally support multiple convolution kernel sizes with the same amount of computing resources and maintains high runtime DSP efficiency. Using the proposed WinoPE, we construct a highly efficient systolic array accelerator, termed WinoCNN. We also propose a dedicated memory subsystem to optimize the data access. Based on the accelerator architecture, we build accurate resource and performance modeling to explore optimal accelerator configurations under different resource constraints. We implement our proposed accelerator on multiple FPGAs, which outperforms the state-of-the-art designs in terms of both throughput and DSP efficiency. Our implementation achieves DSP efficiency up to 1.33 GOPS/DSP and throughput up to 3.1 TOPS with the Xilinx ZCU102 FPGA. These are 29.1% and 20.0% better than the best solutions reported previously, respectively.**

***Keywords*-Winograd algorithm, CNN, systolic array, FPGA, DSP efficiency**

I. Introduction

Convolution neural networks (CNN) have been playing an essential role in solving practical applications, and FPGAs have demonstrated their flexibility, efficiency, and reconfigurability as an ideal platform for CNN acceleration [1]–[5]. Many previous works have proposed different algorithms and architectures to achieve high performance for CNN acceleration on FPGAs [3]–[8]. Since DSPs in FPGAs are usually the major computational resource, the run-time DSP efficiency, defined as the average amount of effective convolution operations executed per DSP per second (GOPS/DSP), is crucial for FPGA design performance and is one of the most important factors to evaluate the design quality for FPGAs [3] [5] [9] [10].

Meanwhile, Winograd's minimal filtering algorithm has been widely adopted in CNN acceleration [11]. It trades multiplications with additions to save computational resources [11]. In FPGA, such a trade-off saves DSP resources from massive amount of multiplications in CNNs, and hence improves the concurrency and efficiency of acceleration. However, due to the inherent characteristics of the algorithm, existing Winograd convolution algorithms are usually specifically designed for a fixed convolution kernel size, e.g., 3x3 [12] [13]. When applied to other popular kernel sizes, i.e., 1x1 in light-weight CNNs, it becomes inefficient due to the overhead of kernel padding [13]. In addition, the tile-based data pattern required by the Winograd algorithm together with the concurrent processing requirement usually result in high data transmission overhead [13].

Systolic array-based accelerator architectures are considered compelling to deal with the massive amount of computations and communications required by CNNs [13]–[15], delivering the state-of-the-art performance. However, the performance of the systolic array-based architecture largely relies on the efficiency of the processing elements (PEs) inside the array, the data transmission among PEs, as well as the data access from external memory, which are all non-trivial to optimize.

In this work, to address the aforementioned issues and improve system performance and DSP efficiency for Winograd based CNN acceleration, we make the following contributions:

- We design a novel Winograd-based processing element, WinoPE, using our generalized resource sharing mechanism that supports flexible convolution kernel sizes with high DSP efficiency.
- Using the proposed WinoPEs, we construct a scalable systolic array-based accelerator WinoCNN, which supports flexible configurations with different parallelism levels honoring FPGA resource constraints.
- We design a fine-grained and highly efficient memory control system that can deal with different memory access patterns and provide tile-based data to our WinoPEs with high efficiency and throughput.
- We propose accurate models for resource and performance estimation, which guide the design space exploration for the configurable parameters of our WinoCNN accelerator.

II. Background and Design Challenges

A. Winograd Convolution on FPGA

Winograd convolution is based on Winograd minimal filtering algorithm that computes an $m\times m$ output matrix Y by convolving a $(m+k-1)\times(m+k-1)$ input matrix d with a $k\times k$ kernel g as described in Figure 1. The input size is also treated as the Winograd filter size. It reduces the number of multiplications at the cost of additions [11]. A 2D Winograd algorithm $F(m\times m, k\times k)$ includes a consecutive sequence of matrix transformation and element-wise multiplication (represented as $\odot$). The G, B, and A are constant transform matrices generated by Cook-Toom algorithm [11].

A convolution layer in CNN with $k\times k$ convolution kernel can be computed using Winograd algorithm with a configuration of $F(m\times m, k\times k)$. The computation of each output feature-map O with size $H_o\times W_o$ is divided into tiles with

978-1-6654-2701-2/21/$31.00 ©2021 IEEE
DOI 10.1109/ASAP52443.2021.00045

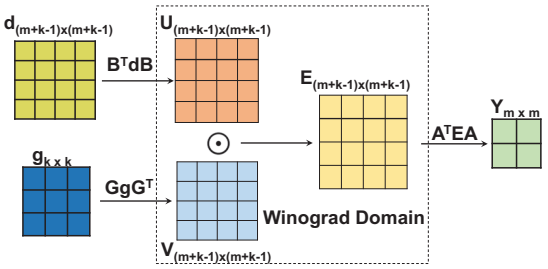

Figure 1: $F(m\times m, k\times k)$ Winograd convolution.

size $m\times m$, resulting in $\lceil H_o/m \rceil \lceil W_o/m \rceil$ tiles in each output channel. The computation of the output tile O_{o,x_o,y_o} starting at pixel (x_o, y_o) in channel o can be completed by applying Winograd algorithm on input tiles I_{i,x_i,y_i} starting at pixel (x_i, y_i) in all the C input feature-map channels with kernel $K_{o,i}$ and summing the results up, as shown in Eq. 1.

$$O_{o,x_o,y_o} = A^T(\sum_{i=1}^{C}[(B^T \cdot I_{i,x_i,y_i} \cdot B) \odot (G \cdot K_{o,i} \cdot G^T)])A \quad (1)$$

The transformation operations with B, G, A are matrix multiplications with constant element values that can be completed by add/shifting operations. So the total number of multiplications equals to the number of element-wise multiplications of U and V, which is less than the required multiplications in the conventional convolution [16]. Since the multiplications on FPGAs are conducted by DSPs, reducing required multiplications in convolution helps to improve parallelism with a given number of DSPs and hence improves computation performance.

However, there is a critical problem: The constant transformation matrices (B, G, A) for a given convolution kernel size have fixed patterns; this results in inefficient DSP utilization when using the hardware designed for one kernel size to a different kernel size, where it has to either split/pad the input data/kernels or to instantiate a new accelerator. For example, to compute 1×1 convolution kernel with a Winograd-based PE designed for 3×3 kernel, we need to pad 1×1 convolution to 3×3 with zeros, which can only achieve $\frac{1}{9}$ of the DSP efficiency of executing 3×3 convolution; or alternatively, instantiating a dedicated accelerators for 1×1 kernel only, which occupies additional resources. Hence, designing a Winograd-based PE with flexible support for different kernel sizes while maintaining high DSP efficiency is essential but remains unexplored.

B. Systolic Architecture

A systolic array [17] is typically composed of many interconnected identical PEs, where the intermediate data is computed by PEs and passed to adjacent PEs. Systolic array architectures are efficient for parallel computing and is widely adopted by FPGA accelerators for matrix multiplications and convolutions [5] [18]. One previous design [13] proposes a systolic array architecture with Winograd algorithm to accelerate sparse convolution, which achieves 5× higher performance compared to the normal dense convolution accelerator. Another work [12] proposes a systolic array architecture specifically designed for ResNet units.

In general, mapping the application to systolic array requires the data buffering in the PEs and the short PE-to-PE data transmission pattern. CNNs are not naturally providing such buffering and connections patterns, which requires careful refinement of the orders of the operations and buffering of the data.

C. Efficient Memory Access

Inefficient data access of the PEs downgrades the overall performance [5], [19], [20]. To support efficient data access with limited off-chip memory bandwidth, the memory subsystem for the accelerator must be carefully designed for specific data re-arrangements and access patterns [12], e.g., using multiple line-buffers [20]. However, it is difficult to create a universal design that would be compatible with different CNN layer configurations. In addition, the systolic array of PEs requires the memory subsystem to provide concurrent off-chip memory access and on-chip data reuse to fully utilize the computational capacity of all PEs. Winograd algorithm further complicates the memory access requirements due to the varied planar data access patterns of the PEs.

III. Design Principles

To resolve the challenges discussed in Section II, we design our WinoCNN accelerator system with the following design principles.

A. Sharing in Winograd Algorithm

As discussed in Section II, the low DSP efficiency of the Winograd algorithm for varying convolutional kernel sizes is caused by the constant transformation matrices. The key solution is to provide flexible kernel size support within the Winograd convolution PE without reloading the transformation matrix and reorganizing the computation procedure. For a Winograd convolution $F(m\times m, k\times k)$, the transformation matrices B, A, G and the intermediate Winograd filter sizes are fixed, as shown in Figure 1. The required number of element-wise multiplications equals to the size of U and V, which is $(m+k\text{-}1)\times(m+k\text{-}1)$.

The input transformation matrix B depends on the size of input tile d for the input transformation ($U=B^T dB$). For a set of Winograd algorithm configurations with a Winograd filter size ω, denoted as $F_\omega(m\times m, k\times k)$, where $\omega=m+k\text{-}1$ ($\omega \geqslant k$). As long as ω values are the same, the computation patterns of input transformation and element-wise multiplication are exactly the same. Matrices $B_\omega^T(m\times m, k\times k)$ with same ω are identical. An example for $\omega = 4$ is shown in Figure 2. Meanwhile, U and V are all $\omega\times\omega$ matrices. Therefore, the hardware resource to process $U = B_\omega^T dB_\omega$ and $E = U \odot V$ can be shared among all $F_\omega(m\times m, k\times k)$.

The transformation matrices G and A will be different for different convolutional kernel sizes under the same ω.

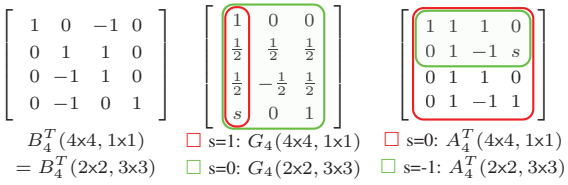

Figure 2: Winograd transformation matrix for F_4.

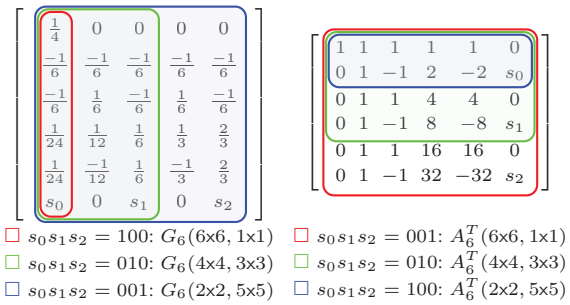

Figure 3: Winograd transformation matrix for F_6.

We observe that there are a large amount of repeated values for the G and A matrices across different m and k values when ω is the same, and the different element(s) could be used as identifier(s) for different kernel sizes and output sizes. As shown in Figure 2, a single element s could be used to identify $G_4(4\text{x}4, 1\text{x}1)$ and $G_4(2\text{x}2, 3\text{x}3)$. **Also, this sharing property of the transformation matrix A_ω and G_ω can be generalized to larger Winograd filter size ω such as F_8 and F_{10} for larger convolution kernel sizes such as** 5x5 **and** 7x7 **with multiple identifiers.** As shown in Figure 3, the transformation matrices G_6 and A_6 with three identifiers s_0, s_1 and s_2 can be shared for the convolution kernel sizes 1x1, 3x3 and 5x5. This provides us a unique opportunity to reuse the same computation resource (DSP) for different input kernel sizes using a unified PE for Winograd convolution. The design details of the PE and resource sharing are presented in Section IV-A.

B. Task Mapping For PEs

We assume a PE can perform the element-wise multiplication and output transformation $PE(U, V) = A^T(U \odot V)A$ for $F_\omega(m\text{x}m, k\text{x}k)$ Winograd convolution in one cycle. To properly map the convolution task into PEs, we partition the computation process of a convolution layer into several iterations. In each iteration, RS consecutive rows of output feature map are computed. Figure 4a shows the pseudo-code to compute the output feature map of a convolution layer with kxk kernel size using one PE. The input, weight and output are represented as C-style array `in[ID][IH][IW]`, `w[ID][OD][k][k]`, and `out[OD][OH][OW]`, respectively. However, loops shown in the Figure 4a do not have the tiled structure to target the 2D PE array. In order to map the computation to the 2D PE array and increase the parallelism of data processing, we rearrange the loop as shown in Figure 4b and introduces two levels of tiling for the

```
L0: for (r_c=0;r_c<OH;r_c+=RS)
L1: for (od=0;od<OD;od++)
L2: for (id=0;id<ID;id++)
L3: for (r=r_c;r<r_c+RS;r+=m)
L4: for (c=0;c<OW;c+=m){
    V=wtrans(w[id][od][0:k-1][0:k-1]);
    U=itrans(in[id][r:r+ω-1][c:c+ω-1]);
    Y=PE(U,V);
    out[od][r:r+m-1][c:c+m-1]+=Y; }
```

(a)

```
L0: for (r_c=0;r_c<OH;r_c+=RS)
L1: for (od=0;od<OD;od+=M)
L2: for (id=0;id<ID;id++)
L3: for (r=r_c;r<r_c+RS;r+=m)
L4: for (c=0;c<OW;c+=m*N)
L5: for (i=0;i<M;i++)
L6: for (j=0;j<N;j++) {
        cn=c+j*m;
        V=wtrans(w[id][od+i][0:k-1][0:k-1]);
        U=itrans(in[id][r:r+ω-1][cn:cn+ω-1]);
        Y=PE<i,j>(U,V);
        out[od+i][r:r+m-1][cn:cn+m-1]+=Y; }
```


(b)

Figure 4: Loops of computation process.

computation. Loop `L0` iterates through the output rows with a step of RS. Loop `L1` iterates through the output depth with a tile size of M. Loop `L2` iterates through the input depth. Loop `L3` segments the RS output rows into Winograd output tile of size m. Loop `L4` partitions the output columns into segments containing N size-m output tiles. After unrolling of `L5` and `L6`, MxN tiles of data will be processed by an MxN PE array in one cycle (as shown in Figure 4b for a 2x2 array). In this way, all WinoPEs with the same row index or column index share the same weights or the same input tile, respectively.

Note the direct mapping of the tiled computation to the PE array will generate high fanout (as shown in the embedded figure for the 2x2 array) and worsens the timing of the implementation. In order to address this issue, we schedule the computation of the PEs following the structure of an MxN systolic array. The detailed design will be presented in the Section IV-B.

C. Efficient Memory Access

Efficient execution of Winograd-based PEs requires simultaneous data access within a tile, as shown in Figure 5. This planar data access pattern (data tiles) brings in a challenge for efficient memory control and data supply for the PEs. When multiple PEs are instantiated as an array to process different tiles of a CNN layer, there are also overlaps among the data tiles required by the PEs. As shown in the example in Figure 5, the data tiles required by adjacent PEs overlap with each other (marked with purple circle). Simply assigning input buffers for all the PEs would cause high on-chip memory usage [15]. However, line buffer based design [20] faces difficulties when supplying multiple tiles for Winograd-based PEs under systolic architecture that requires varied memory access patterns, i.e., varied window moving steps. As shown in Figure 4b `L4`, the PE array

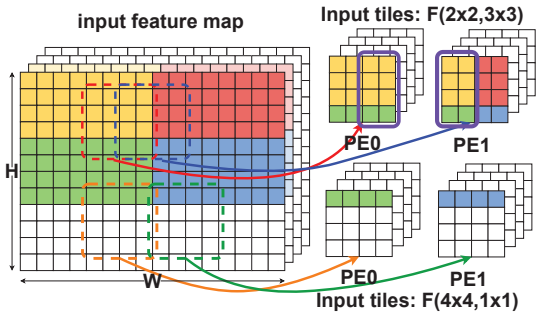

Figure 5: Planar data access pattern of PEs.

requires N input tiles with a horizontal moving step size of $N \cdot m$ each cycle. Meanwhile, the output tile size m differs according to kernel size k, leading to a varied window moving step. These motivate us to design a specialized memory system for our WinoCNN architecture.

To design an efficient PE array to work with such data access patterns, we draw three design principles: First, to improve computation efficiency with high parallelism, the data elements inside one tile must be fetched in parallel and provided to the computational unit simultaneously; and second, the overlapped data across tiles shall be fetched from external memory only once and then reused to reduce memory access overhead. Third, the memory system should be able to supply data for Winograd convolutions with different kernel sizes efficiently.

IV. Implementation

We implement our WinoPE, systolic array and memory subsystem based on the design principles from Section III to build our WinoCNN acceleration system.

A. WinoPE: PE With Multiple Kernel Support

Our WinoPE is the basic processing unit of the Winograd systolic convolution accelerator system (WinoCNN), where each WinoPE is able to complete the computation of an input kernel and a set of feature tiles in a single clock cycle. WinoPE is featured with the flexible support for different convolution kernel sizes without the DSP overhead in a unified architecture. As discussed in Section III, the Winograd algorithm with the same Winograd filter size ω can share the corresponding transformation matrices as well as the expensive dot product module. We choose the design of sharing between $F_4(4\text{x}4, 1\text{x}1)$ and $F_4(2\text{x}2, 3\text{x}3)$ as the example to present our kernel sharing mechanism. Figure 6 shows the unified architecture to process a single tile in our WinoPE. It contains an input tile register array (red block), a weight tile register array (blue block), a matrix of multipliers (purple circle), an output transformation module (green block), and an output tile (dark and light yellow) towards an output buffer.

In each working cycle, the WinoPE reads in a tile of input data and a tile of weights in parallel. Note here, the input tiles are transformed on-chip when they are fetched

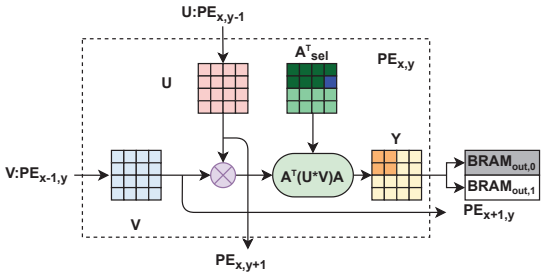

Figure 6: WinoPE processing logic.

from input buffer and the convolution kernel weights are transformed before they are stored into the on-chip memory to reduce the resource usage for transformation logic. After fetching the input and weight, the element-wise multiplication $U \odot V$ is performed. The output transformation module takes the results of $U \odot V$ and generates the output tile. The data fetching and element-wise multiplication modules can be directly shared and fully utilized by different convolution kernel sizes. To handle the different output caused by different kernel sizes, we design a selectable output transformation matrix A^T_{sel}, in which the **selection bit** s in the matrix A is used as a matrix identifier, as shown in Figure 2 (Section III). As an instance in F_4, when s is set to 0, the WinoPE performs $F_4(4\text{x}4, 1\text{x}1)$ algorithm, where the whole 4x4 matrix is the output of the WinoPE (light yellow block in Figure 6). When s is set to -1, the WinoPE performs $F_4(2\text{x}2, 3\text{x}3)$ algorithm, where the top left four elements of the result matrix is the output (dark yellow block). In this way, our WinoPE processes convolution layers with different kernel sizes without DSP overhead. Finally, the computed outputs are stored in the output buffer constructed with BRAMs. Note that such a selection bit design can be easily extended to the Winograd algorithm with larger Winograd filter sizes for larger convolution kernel sizes.

Furthermore, we partition the input tile register U and weight tile register V into individual registers that each contains a single data from the input channels, so that the multiplication for an entire tile is finished in a single clock cycle. The processing efficiency of WinoPE is further increased by instantiating Q input channels of batch size B of tile registers with the corresponding number of $U \odot V$ multiplier matrices. An adder tree constructed with LUT is used to accumulate the outputs from the multiplier matrices.

The selection bit design allows us to share the computation resources without wasting the DSPs when processing convolutions with different kernel sizes. However, it remains challenging to use Winograd convolution algorithm for large kernel convolution and irregular kernel convolution. A practical limitation is the larger Winograd filter size requires more LUT resources to conduct the addition operation during the constant matrix multiplication for Winograd convolution algorithm. Also, recent DNN models adopt irregular convolution kernels such as 1x7 and 7x1 sizes that are not well supported by the Winograd algorithm.

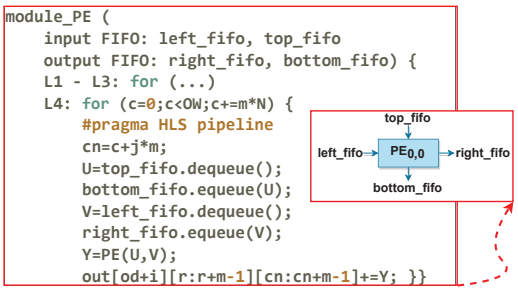

Figure 7: Loop behaviour for $PE_{i,j}$

To handle the large kernel convolution and irregular kernel convolution, we design a split mechanism that splits the target convolution kernel into supported kernel sizes as shown in the Equation 2 and 3.

$$K_s^{i,j}[h][w] = \begin{cases} 0, & ik{+}h \geq H_t \text{ or } jk{+}w \geq W_t \\ K_t[ik{+}h][jk{+}w], & otherwise \end{cases} \tag{2}$$

$$Output_{target} = \sum_{i,j} FM^{ik,jk} * K_s^{i,j} \tag{3}$$

K_t represents the target convolution kernel with size H_txW_t and K_s represents the supported convolution kernel with size kxk. The target kernel is split into $\lceil \frac{H_t}{k} \rceil$x$\lceil \frac{W_t}{k} \rceil$ supported kernels with unaligned elements padded with zeros. The split kernel $K_s^{i,j}$ is segmented from the target kernel with ik, jk offset from the top left element. The targeted convolution (denoted as $*$) is completed by applying convolution for each supported kernel $K_s^{i,j}$ on input features with the same 2D pixel offset (denoted as $FM^{ik,jk}$) and summing up the split results as shown in Equation 3.

B. Parameterized Systolic Array

Instead of sharing a single set of data tiles among different WinoPEs in the same clock cycle (as shown in the Figure 4b), we construct the WinoPEs as an MxN systolic array that shares the weight and input data among WinoPEs by shifting them PE-to-PE to further utilize the on-chip registers and reduce the high fanout and long connection caused by the flattened implementation. To achieve this, we take advantage of the insensitivity of the loop order and assign FIFOs to WinoPEs (as shown in Figure 7). Note here, the PEs are called by an outside loop as `L0` and the loops `L1`–`L4` are the same as the ones in Figure 4b; however, the input Winograd tile and the weight tile are fetched from the top and left FIFO interfaces which connect to the top and left neighbours of the WinoPEs, respectively. In the same iteration, the input and weight data are pushed into bottom and right FIFO interfaces and passed to the bottom and right neighbours after one clock cycle due to the blocking mechanism of the FIFO. Therefore, the WinoPEs are constructed as a systolic array.

With the assigned FIFOs for our WinoPEs, we could easily instantiate the systolic WinoPE array by organizing the row and column FIFOs, denoted as $row\_fifo[M][N]$ and $col\_fifo[N][M]$. The M, N parameters are configurable during the WinoCNN system generation.

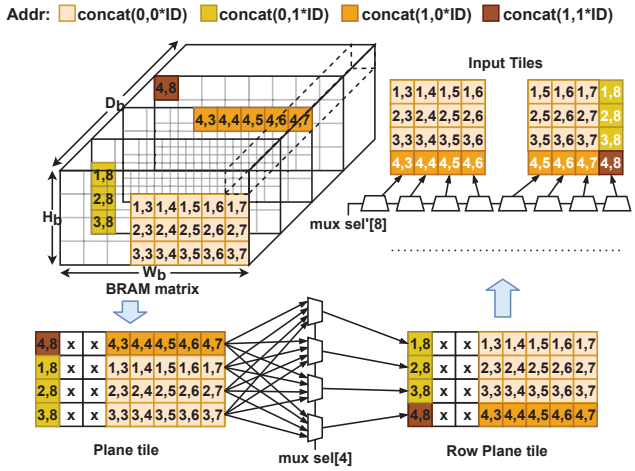

Figure 8: Hierarchical Memory Subsystem.

C. Hierarchical Memory Subsystem

To provide the required data to the WinoPE array efficiently, we propose: (1) a BRAM buffer matrix that has a unique addressing mechanism to support efficient parallel data access, and (2) a pipelined planar data control and scheduling to provide efficient on-chip data reuse and support the flexible input tile access pattern.

1) BRAM Buffer Matrix: To guarantee the parallel access of the input tiles, we fold the input feature-maps into a matrix of BRAM buffers, denoted as $BRAM_{in}[H_b][W_b][D_b]$, which consists of H_bxW_b BRAM buffer instances (or BRAM bank) of depth D_b, as shown in Figure 8. Each BRAM buffer instance has its individual address port and data port, hence total H_bxW_b entries can be accessed from the BRAM buffer matrix in each cycle at different addresses. An address mapping mechanism is designed as shown in Eq. 4 to decide the location in the buffer as $BRAM_{in}[h][w][addr]$ for a certain input pixel in the feature map $in[id][r][c]$, where ID, id, r, c represent the number of input channel for a layer, channel index, row index and column index for the pixel in the input feature map :

$$\begin{gathered} h = r\%H_b,\ w = c\%W_b \\ addr = concat(\lfloor r/H_b \rfloor, \lfloor c/W_b \rfloor \cdot ID + id) \end{gathered} \tag{4}$$

The BRAM banks in the same row share the same high address bits, while the BRAM banks in the same column share the same low address bits. The concatenation function ensures that the entries are accurately located with the given index. With the sequence of input Winograd tiles denoted as T_N, where T_N consists of N tiles with ωxω size, defined in Eq. 5, all the elements in T_N forms a continuous input data block, denoted as T_U, with size ωx$((N\text{-}1)m{+}\omega)$:

$$T_N[n]{=}in[id][r{:}r{+}\omega\text{-}1][c{+}nm{:}c{+}nm{+}\omega\text{-}1] \tag{5}$$

$$T_U{=}\bigcup_{n=0}^{N-1} T_N[n]{=}in[id][r{:}r{+}\omega\text{-}1][c{:}c{+}(N\text{-}1)m{+}\omega\text{-}1] \tag{6}$$

As an instance in Figure 8, H_b=4, W_b=8, ω=4, m=2

and N=2 and two data tiles are required to be accessed within input feature maps $in\_data[0][1{:}4][3{:}6]$ and $in\_data[0][1{:}4][5{:}8]$. The union of the two input tiles can be represented as T_U=$in\_data[0][1{:}4][3{:}8]$. According to the address mapping defined in Eq. 4, the pixels in $in\_data[0][1{:}4][3{:}8]$ are accessed from four different regions of BRAM buffer matrix in one clock cycle with different high address bits and low address bits.

2) Planar Data Access: The input tiles are then passed through a 3-stage pipeline to ensure the data reuse and to provide the planar data to the WinoPEs. As shown in Figure 8, The first stage stores the H_b×W_b output from BRAM matrix buffer into registers. The second stage ensures the row order of the planar data with a row plane multiplexer array. The third stage splits the plane to tiles by a column multiplexer array. The mux selection bits are generated on-the-fly regarding the values of r, c, and id. Since both the BRAM bank addresses and the mux selection are generated on-the-fly, the memory architecture is able to supply input tiles with varied window moving steps regarding the kernel size for the current convolution layer.

V. System Architecture and Modeling

We construct our WinoCNN system and build the performance and resource models for easy exploration of the architectural configurations.

A. WinoCNN Architecture Overview

The overall architecture of our WinoCNN accelerator system is shown in Figure 9. Note here, the flexible convolution kernel size support is provided by our WinoPEs. The convolution layers of the input models are computed in output row stationary. The input reading, computation, and output data offloading are scheduled to run in parallel.

B. System Modeling

As shown in Section IV, our system is built with performance and resource sensitive architectural parameters, the corresponding models are built for design space exploration.

1) Resource Model: **DSP usage.** The major instances of DSPs are occupied by the WinoPEs. Each WinoPE computes the element-wise sum of the product along Q input channels for input tiles $U^{\omega\text{x}\omega}$ of batch size B and weight tiles $V^{\omega\text{x}\omega}$. Thus, the total number of DSPs required by a WinoPE is $\omega^2 \cdot B \cdot Q$. The systolic WinoPE array contains M×N WinoPEs, so the total DSPs required by our WinoCNN accelerator is:

$$DSP_{use} = \omega^2 \cdot M \cdot N \cdot B \cdot Q \tag{7}$$

BRAM occupation. The BRAM resource is mainly occupied by the input, weight and output buffers. The BRAMs are in the form of 18-bit width and 1024 depth blocks.

The input buffer is a buffer matrix of size H_b×W_b with buffer depth as D_{in}. Each bank should be capable of storing

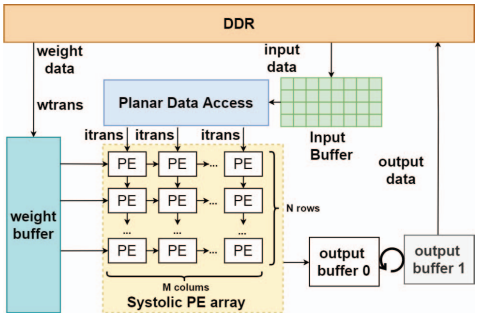

Figure 9: Overall architecture.

B input data with 8 bits. The number of BRAMs for input is $H_b \cdot W_b \cdot \lceil 8 \cdot B/18 \rceil \cdot \lceil D_{in}/1024 \rceil$.

Each row of the systolic array requires $\omega^2 \cdot Q$ transformed and quantized 16-bit weight data, thus requires $\lceil \omega^2 \cdot Q \cdot 16/18 \rceil$ BRAM blocks with a fixed depth of 1024 to provide enough weight access bandwidth. The total BRAM required for weight buffer is $M \cdot \lceil 16 \cdot \omega^2 \cdot Q/18 \rceil$.

Each WinoPE has a ω×ω buffer matrix to store 18-bit temporary output data of batch B and buffer depth D_{out}. With the requirement of 2 buffers for ping-pong access, each WinoPE needs $2 \cdot \lceil \omega^2 \cdot B \cdot 18/18 \rceil \cdot \lceil D_{out}/1024 \rceil$ BRAMs as output buffer.

The total number of BRAM is the sum of the above:

$$\begin{aligned} BRAM_{use} &= H_b W_b \lceil 8 \cdot B/18 \rceil \cdot \lceil D_{in}/1024 \rceil \\ &+ M \lceil 16 \cdot \omega^2 Q/18 \rceil + 2 \cdot MN\omega^2 B \lceil D_{out}/1024 \rceil \end{aligned} \tag{8}$$

2) Latency Model: Communication latency t_{comm} and computation latency t_{comp} in each phase of the convolution procedure are used to build the latency model. The maximum value between these two in each phase dominates the overall latency.

Since all weights are required once within each loop iteration, we have $D_{weight} = k^2 \cdot ID \cdot OD$. Each loop iteration includes a read input process and a write output process with the corresponding data transmission amount of $D_{input} = RS \cdot ID \cdot IW \cdot B$ and $D_{output} = RS \cdot OD \cdot OW \cdot B$. Neglecting the absence of output writing in the first iteration and the input reading in the last iteration, we estimate the communication latency as:

$$t_{comm} = (D_{weight} + D_{input} + D_{output})/BW \tag{9}$$

To compute RS rows of outputs in each computation process, the WinoPE array needs to sum up the convolution results through ID input planes to generate RS×OW output data for OD output planes. In each cycle, M×N WinoPEs sum up the convolution results of Q input planes for N×m×m output pixels along M depth. Considering implementation frequency f, the computation latency for each iteration is:

$$t_{comp} = \lceil ID/Q \rceil \lceil OD/M \rceil \lceil RS/m \rceil \lceil OW/(N \cdot m) \rceil / f \tag{10}$$

The overall latency is estimated as:

$$t_{loop} = \lceil OH/RS \rceil \cdot max(t_{comm}, t_{comp}) \quad (11)$$

3) Parameter Exploration: For the convenience of hardware implementation, we fix the batch size at $B = 2$. To guarantee the access of the planar data, we set H_b as 4 or 8 for F_4 or F_6 respectively and $W_b = \min 2^k$, s.t. $W_b \geq 2\omega$. Note here, the row step RS is a variable during the processing and is chosen as large as possible so that the input and output rows can fully utilize the on-chip buffers. For a given CNN model, the M, N, Q, D_{in} and D_{out} are explored targeting $min(\sum_{l \in layers} t_{loop,l})$ with the given DSP and BRAM resources on the platform.

VI. Evaluations

To validate the effectiveness of our design, we use Xilinx ZCU102 and Ultra96 boards for evaluation, where both platforms are equipped with a quad-core ARM Cortex-A53. The detailed resource specifications are shown in Table II. We use Vivado HLS design suit 2019.2 for accelerator implementation using C++.

A. WinoPE Evaluation

1) Resource effectiveness: We first compare the resource utilization of our WinoPE with the PEs without multiple kernel support, as shown in Table I. All PEs are configured with Q=4 and B=2. The same DSP utilization in each PE type ensures that the maximum parallelism of the PEs is the same. Using the same amount of DSP resources, our WinoPE consumes more LUT and FF resources than each dedicated PE but with the benefit of supporting different convolution kernel sizes without effecting the runtime efficiency.

Table I: Resource utilization of different PEs

PE type	LUT	FF	DSP	PE type	LUT	FF	DSP
F_4(2×2, 3×3)	5328	2430	128	F_6(4×4, 3×3)	21542	19235	288
F_4(4×4, 1×1)	6495	9831	128	F_6(6×6, 1×1)	24056	39126	288
WinoPE-F_4	7852	10501	128	WinoPE-F_6	33959	42793	288

2) Performance effectiveness: Since the DSP efficiency without efficient data supply will be lower than the theoretical performance, we evaluate our WinoPE together with our memory subsystem and compare the results to other PEs theoretical performance with the assumption of the data supply is perfect. We first conduct experiments of synthetic convolution layers with different kernel sizes and compare them to the theoretical performance values using the same configuration as shown in Figure 10. We measure the DSP efficiency to exclude the impact of different platforms with a system frequency at 100Mhz. The DSP efficiency of WinoPEs for different kernel sizes is measured on board, and the maximum performance for other PEs are calculated theoretically; both are shown in Figure 10. Compared to the theoretical performance of F_4 and F_6, our implementations of WinoPE-F_4 and WinoPE-F_6 under all kernel sizes achieve near-maximal theoretical performance with the proposed memory subsystem.

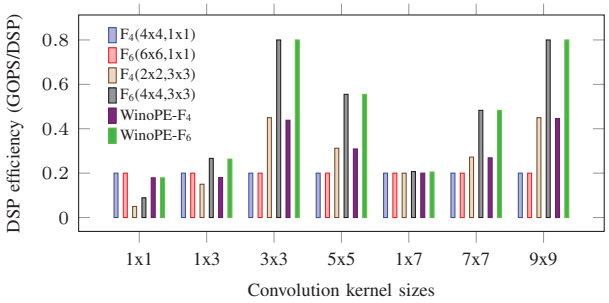

Figure 10: DSP efficiency of WinoPE to theoretical values.

B. WinoCNN Evaluation

We adopt the most representative CNN models as benchmarks to demonstrate the effectiveness of our WinoCNN system design, including VGG-16, Inception-V4 (denoted as INet-V4), and YoloV2. The non-convolution layers are executed in the processors with multi-thread optimization for end-to-end model execution. All convolutional layers are executed on the WinoCNN accelerator.

1) WinoCNN configuration: We explore the optimal WinoCNN system configurations for different platforms using our analytical model. The selected values are shown in Table II together with resource utilization and runtime performance on the different platforms for different models.

The WinoCNN accelerator configurations for different platforms and different Winograd kernel sizes vary significantly because of the different DSP and BRAM capacity of the platforms, where all configurations target to fully utilize the on-chip DSP and BRAM resources. The achievable frequency under each configuration for a certain platform is also shown together with the final performance. Our WinoCNN system naturally supports better timing due to the timing-friendly shorter data path between the WinoPEs. Notably, for the networks with homogeneous convolutional layers, i.e., VGG-16, our design achieves 3.12 TOPS throughput at 214MHz clock frequency while the performance drops to 857.23 GOPS when there are multiple divergent convolutional layer configurations in Inception-V4, i.e., 1×7 kernel. This is because of the varied efficiency of the Winograd algorithm for different convolution kernel sizes.

2) Comparison with state-of-the-art designs: We then measure the execution latency, throughput, and DSP efficiency of our implemented models and compare them with the state-of-the-art implementations, as shown in Table III.

Since all the convolution layers are executed by our WinoCNN accelerator, the DSP efficiency and latency data are calculated for the convolution layers. DSP efficiency of our design is 1.71× of the design in [15], which does not use Winograd transformation. When compared to the designs with Winograd algorithm [13] [20] together with additional

Table II: WinoCNN configuration and performance on different platforms

Platform	PE Config.					Resource Util.(% of (total))					Throughput (GOPS)		
	M	N	Q	D_{in}	D_{out}	DSP	BRAM	LUT	FF	Freq.(MHz)	VGG-16	INet-V4	YoloV2
Ultra96(WinoPE-F_4)	2	1	4	4096	1024	77.8(360)	85.9(432)	60.8(70K)	43.2(141K)	250	265	127.2	157.5
ZCU102(WinoPE-F_4)	8	2	4	8192	1024	82.8(2520)	95.5(1824)	76.3(274K)	43.4(548K)	250	1862	820.3	1241
ZCU102(WinoPE-F_6)	4	2	4	4096	1024	93(2520)	87(1824)	81(274K)	48(548K)	214	3120.3	857.23	1717.7

Table III: Comparison with state-of-the-art designs

	[20]	[15]	[12]	[13]	[21]	Vitis-AI [22]			Ours. (WinoPE-F_6)		
Platform	ZCU102	Arria10 GT1150	Stratix V GSMD8	XCVU095	Arria-10	ZCU102			ZCU102		
Model	VGG-16	VGG-16	Resnet-18	VGG-16	VGG-16	VGG-16	INet-V4	YoloV2	VGG-16	INet-V4	YoloV2
Freq. (MHz)	200	231.85	160	150	250	281			214		
Precision	16-bit fixed	8-16 bit fixed	16-bit fixed	8-16-bit fixed	16-bit fixed	8 bit			8-16 bit		
Batch size	32	-	-	-	-	2			2		
DSP Usage	2520	1500	576	768	1344	1926			2345		
Thro. (GOPS)	2601.3[1]	1171.3	233	460	1642	1225.2	1390	1008	**3120.3**	857.23	**1717.7**
Latency (ms)	10.43[1]	26.85	7.23	-	-	57.53	35.26	16	**19.67**	49.7	**13.9**
DSP Eff. (GOPS/DSP)	1.03[1]	0.780	0.405	0.599	1.22	0.636	0.722	0.523	**1.33**	0.388	**0.73**

[1]The throughput, latency and DSP efficiency are only for the convolutional layers.

model-specific optimizations, our design shows a 1.2× and a 6.78× improvement of throughput compared to that of [20] and [13], respectively. Notably, the design in [20] adopts a 32 batch size for FC layers, which is much larger than ours (fixed at 2) and leads to a long latency for a single image to be processed completely. In the comparisons, all the previous architectures containing model-specific designs can not support flexible kernel sizes, while our WinoCNN supports multiple convolution kernels without effecting the DSP efficiency. Our design also provides slightly better achievable frequency due to the efficient systolic array architecture on Xilinx platforms.

When comparing to the Vitis-AI implementations [22], our WinoCNN shows better throughput and latency for both VGG-16 and YoloV2 even with a lower clock frequency and without DSP double pumping. For the Inception-V4 model which contains unique kernel shapes, i.e., 1x7, 7x1, 3x1 and 1x3, we use the less efficient $F(4\text{x}4, 1\text{x}1)$ or $F(6\text{x}6, 1\text{x}1)$ to process them, which lead to a worse performance than the specially optimized Vitis-AI processing cores.

VII. CONCLUSION

In this work, we present a systolic array based convolution accelerator design targeting the Winograd algorithm. Our accelerator, WinoCNN, is constructed by unique Winograd convolution PEs (WinoPE) which support flexible convolution kernel sizes without sacrificing DSP efficiency. WinoCNN also has an efficient memory subsystem that is suitable for planar data access for the array of WinoPEs. Our accelerator system is configurable for different FPGA platforms with accurate resource and performance models. Overall, our accelerator delivers high throughput and state-of-the-art DSP efficiency comparing with previous accelerator implementations. Our code release can be found at https://github.com/xliu0709/WinoCNN.

VIII. ACKNOWLEDGEMENT

This work is supported in part by the IBM-Illinois Center for Cognitive Computing Systems Research (C3SR), Semiconductor Research Corporation (SRC) and is also partially supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme.

REFERENCES

[1] X. Zhang *et al.*, "Machine learning on FPGAs to face the IoT revolution," in *ICCAD*, 2017.

[2] D. Chen *et al.*, "Platform choices and design demands for iot platforms: cost, power, and performance tradeoffs," *IET Cyber-Physical Systems: Theory & Applications*, vol. 1, no. 1, pp. 70–77, 2016.

[3] C. Hao *et al.*, "FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge," in *DAC*, 2019.

[4] Y. Chen *et al.*, "Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs," in *FPGA*, 2019.

[5] X. Zhang *et al.*, "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *FPL*, 2017.

[6] X. Zhang *et al.*, "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," in *ICCAD*, 2018.

[7] H. Li *et al.*, "A high performance FPGA-based accelerator for large-scale convolutional neural networks," in *FPL*, 2016.

[8] X. Zhang *et al.*, "Skynet: a hardware-efficient method for object detection and tracking on embedded systems," 2019.

[9] Y. Chen *et al.*, "T-DLA: An open-source deep learning accelerator for ternarized DNN models on embedded FPGA," in *ISVLSI*, 2019.

[10] X. Liu *et al.*, "High level synthesis of complex applications: An h.264 video decoder," in *FPGA*, 2016.

[11] S. Winograd, *Arithmetic Complexity of Computations*. Society for Industrial and Applied Mathematics, 1980.

[12] X. Xie *et al.*, "Fast-abc: A fast architecture for bottleneck-like based convolutional neural networks," in *ISVLSI*, 2019.

[13] F. Shi *et al.*, "Sparse winograd convolutional neural networks on small-scale systolic arrays," in *FPGA*, 2019.

[14] J. Cong and J. Wang, "PolySA: Polyhedral-based systolic array auto-compilation," in *ICCAD*, 2018.

[15] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *DAC*, 2017.

[16] A. Lavin, "Fast algorithms for convolutional neural networks," *CoRR*, vol. abs/1509.09308, 2015. [Online]. Available: http://arxiv.org/abs/1509.09308

[17] H. Kung and C. E. Leiserson, "Systolic arrays (for VLSI)," in *Sparse Matrix Proceedings 1978*, vol. 1. Society for industrial and applied mathematics, 1979, pp. 256–282.

[18] U. Aydonat *et al.*, "An openCL™ deep learning accelerator on arria 10," in *FPGA*, 2017.

[19] Y. Guan *et al.*, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *FCCM*, 2017.

[20] Y. Liang *et al.*, "Evaluating fast algorithms for convolutional neural networks on fpgas," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 4, pp. 857–870, 2020.

[21] J. Yepez and S.-B. Ko, "Stride 2 1-d, 2-d, and 3-d winograd for convolutional neural networks," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 4, pp. 853–863, 2020.

[22] https://github.com/Xilinx/Vitis-AI/tree/master/models/AI-Model-Zoo.