WinoGen: A Highly Configurable Winograd Convolution IP Generator for Efficient CNN Acceleration on FPGA*

Mingjun Li+ CUHK
Pengjia Li+ CUHK-Shenzhen
Shuo Yin CUHK
Shixin Chen CUHK
Beichen Li CUHK-Shenzhen
Chong Tong CUHK-Shenzhen
Jianlei Yang Beihang Univ
Tinghuan Chen CUHK-Shenzhen
Bei Yu CUHK

Abstract

The convolution neural network (CNN) has been widely adopted in computer vision tasks. In the FPGA-based CNN accelerator design, Winograd convolution can effectively improve computation performance and save hardware resources. However, building efficient and highly compatible IP for arbitrary Winograd convolution on FPGA remains underexplored. To address this issue, we propose a novel and efficient reformulation of Winograd convolution, named Structured Direct Winograd Convolution (SDW). We further develop WinoGen, a Chisel-based highly configurable Winograd convolution IP generator. Given **arbitrary** input/output tile size and kernel size, it can generate optimized high-performance IP automatically. Meanwhile, our generated IP can be compatible with multiple kernel sizes and tile sizes. Experimental results show that the IP generated by WinoGen achieves DSP efficiency up to **3.80** GOPS/DSP and energy efficiency up to **652.77** GOPS/W while showing **2.45**× and **3.10**× improvements when processing a same CNN model compared with state-of-the-arts.

1 Introduction

With the rapid advancement of Artificial Intelligence (AI), Convolutional Neural Networks (CNNs) have emerged as one of the most prominent network architectures. To accelerate CNN computations, FPGA-based CNN accelerators have gained significant attention, as they offer the potential for high parallelism and energy efficiency [1]. Among the techniques used in these accelerators, Winograd convolution stands out for its ability to improve computation performance and conserve hardware resources [2]. This advantage becomes significant when using FPGA-based accelerators, where the usage of dedicated Digital Signal Processors (DSPs) can be minimized.

Despite the rapid pace of development in this field, the development of efficient IP for arbitrary Winograd convolution on FPGA remains a challenging task. The existing FPGA IP generation methods do not adequately support Winograd convolution. For instance, [3] does not support Winograd convolution, [4] only supports 3×3 kernel, while [5] supports 2D and 3D Winograd convolution but is limited to a single type of kernel and input tile size. [6] introduces processing units capable of handling multiple kernel sizes. However, it cannot be dynamically adjusted to be compatible with different input tile sizes. The Decomposition-based Winograd method can dynamically adapt to different kernel or tile sizes [7]. However, this method requires a complex input tiling and kernel tiling architecture, as well as a specially designed memory hierarchy, which is not suitable for IP-level generative architecture design.

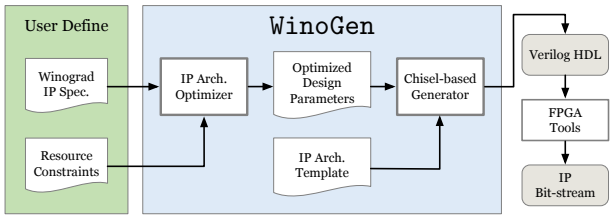

Figure 1: WinoGen automation workflow.

These issues motivate us to explore Winograd hardware IP generation in three aspects: (1) Given arbitrary input/output tile size and kernel size, how to generate Winograd IP rapidly and automatically? (2) How to make the generated IP compatible with as many types of Winograd Convolution as possible, without altering the hardware architecture? (3) How to guarantee the generated IPs maintain high DSP efficiency and energy efficiency, even for Winograd convolution under relatively large kernel or tile sizes?

In this paper, to overcome these issues, we propose WinoGen, an automatic Winograd convolution IP generator for CNN acceleration on FPGA. The main contributions are listed below:

- A Chisel-based highly configurable Winograd convolution IP generator, WinoGen, is developed via algorithm-architecture co-design approaches.
- A Structured Direct Winograd Convolution (SDW) algorithm is proposed as a novel and efficient reformulation of Winograd convolution, enabling WinoGen to generate IPs according to a given arbitrary tile size and kernel size.
- Highly paralleled and fully pipelined architecture is designed as the IP template. The generated IPs are compatible with multiple kernel sizes and tile sizes.
- An architecture optimizer is built based on resource and latency models, enabling WinoGen to generate IPs with optimal configurations.

2 WinoGen Workflow

Our automated IP generation flow WinoGen is shown in Figure 1. Users are required to specify Winograd convolution sizes (formulated as $F(k, n)$ in the following sections), input data width, and

*This work is partially supported by The Research Grants Council of Hong Kong SAR (No. CUHK14210723) and The National Natural Science Foundation of China (No. 62304197).
+Equal contributors

DAC '24, June 23–27, 2024, San Francisco, CA, USA

ACM ISBN 979-8-4007-0601-1/24/06.
https://doi.org/10.1145/3649329.3657392

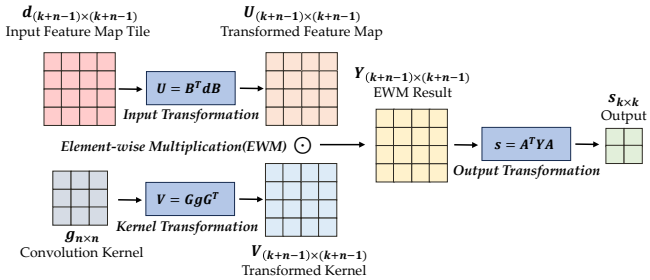

Figure 2: Winograd convolution flow defined by $F(k, n)$.

output data width. They can specify how many input channels an IP can compute simultaneously, or leave the decision to `WinoGen`. In addition, users can provide hardware resource constraints for IP. Then `WinoGen` optimizer will carry out Design Space Exploration (DSE) to determine optimal design parameters. `WinoGen` and the generated IPs have remarkable static and dynamic configurability.

- **Static configurability** is WinoGen's ability to generate IP compatible with arbitrary forms of Winograd convolution for acceleration.
- **Dynamic configurability** indicates the generated IP's versatility, which supports multiple convolution kernel sizes and input/output feature tile sizes under a unified architecture.
- **WinoGen Optimizer** is constructed with resource and latency modeling, enabling DSE for IP architecture configuration.

3 Structured Direct Winograd Convolution

In this section, we will illustrate how to construct the Winograd used in `WinoGen` in a hardware-friendly manner. Based on that, Structured Direct Winograd Convolution (`SDW`) is proposed as a novel and efficient Winograd convolution reformulation.

3.1 Winograd Convolution Construction

A typical Winograd convolution algorithm is depicted in Figure 2. The two-dimensional Winograd defined by $F(k, n)$ is described as

$$s = A^\top \left[B^\top dB \odot GgG^\top \right] A, \tag{1}$$

where s is a $k\times k$ output tile, g is an $n\times n$ kernel and d is a $(k+n-1)\times(k+n-1)$ input feature map tile. We denote the input tile size $\omega = k+n-1$ as the Winograd filter size. For the CNN inference stage, usually, k is even and n is odd. A Winograd convolution consists of the input transformation ($U = B^\top dB$), the kernel transformation ($V = GgG^\top$), the element-wise multiplication ($Y = U \odot V$) and the output transformation ($s = A^\top YA$). The B, G and A are corresponding transformation matrices, which are constant and determined by Winograd parameters k, n and ω.

The transformation matrices (B, G and A) are generated by Cook-Toom algorithm [8]. A generation process (using $F(4, 3)$ as an example) has been provided in [9]. In this section, we will give a more general derivation to show how we construct the Winograd convolution implementation in `WinoGen`, given arbitrary k and n.

When constructing Winograd convolution defined by $F(k, n)$, firstly we need to select ω interpolation points α_i and construct a polynomial sequence:

$$m_i(x) = x + \alpha_i, \quad i = 0, 1, \cdots, \omega - 1. \tag{2}$$

Usually we choose $m_0(x) = x$ and $m_{\omega-1}(x) = x-\infty$, using a notation similar to [9]. $(x-\infty)$ denotes a special remainder term. In our design,

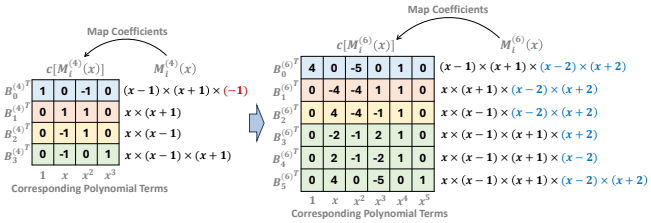

Figure 3: Input Transformation Matrix $B^\top$ for $F(\omega = 4)$ and $F(\omega = 6)$. The blue terms in $M_i^{(\omega)}(x)$ are additional polynomials $m_\Delta(x)$.

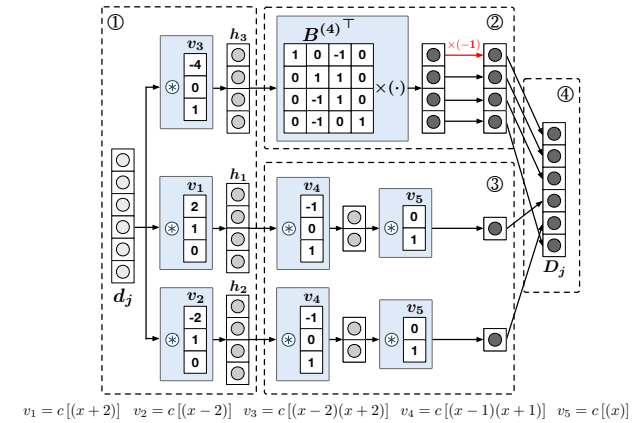

Figure 4: Recursive Input Transformation for $F(\omega = 6)$. The ①②③④ correspond to ❶❷❸❹ of *RIT procedure* in Section 3.2. For simplicity, only $D_j = B^{(6)\top} d_j$ is shown.

we select interpolation points that are all powers of 2 (for hardware-friendly consideration), and obtain:

$$m_i(x) = \begin{cases} x, & i = 0, \\ x + (-1)^i \times 2^{\lfloor \frac{i-1}{2} \rfloor}, & 0 < i < \omega - 1, \\ x - \infty, & i = \omega - 1, \end{cases} \tag{3}$$

where $\lfloor \cdot \rfloor$ is the floor function. For example, for $F(4, 3)$, the sequence will be: x, $(x-1)$, $(x+1)$, $(x-2)$, $(x+2)$, $(x-\infty)$.

Now we are ready to elucidate our **Structured Direct Winograd Convolution** (`SDW`) method. It is composed of **Recursive Winograd Input Transformation** (`RIT`) and **Blockwise Winograd Output Transformation** (`BOT`). The kernel transformation is not considered because for hardware accelerators it can be precomputed offline. `SDW` provides a unified approach to transform arbitrary forms of Winograd input transformation and output transformation into addition, subtraction, and shift operations. It does not need decomposition-based methods used in [7, 10] when handling large kernels or tiles, thus avoiding complex input tiling and kernel tiling.

3.2 Recursive Winograd Input Transformation

The key idea of `RIT` is computing the Winograd input transformation recursively. It is based on our observation that given two forms of Winograd convolution $F(k_1, n_1)$ and $F(k_2, n_2)$, if $(k_1 + n_1 - 1) < (k_2 + n_2 - 1)$, i.e., $\omega_1 < \omega_2$, then the input transformation of $F(k_2, n_2)$ can be computed incrementally based on the computation procedure of $F(k_1, n_1)$.

The input transformation matrix B is decided by the value of ω. Therefore we denote it as $B^{(\omega)}$ in this section. To get $B^{(\omega)}$, we first

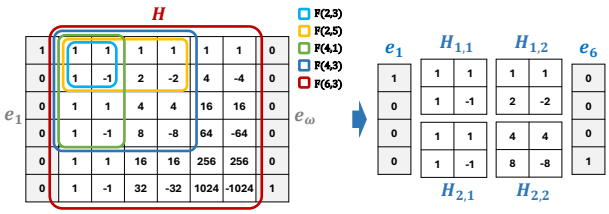

Figure 5: Sharing mechanism and block partition of $A^\top$ (using $F(6,3)$ and $F(4,3)$ as examples).

define polynomial sequence $M_i^{(\omega)}(x)$:

$$M_i^{(\omega)}(x) = \begin{cases} \frac{1}{m_i(x)} \prod_{j=0}^{\omega-2} m_j(x), & 0 \le i < \omega - 1, \\ \prod_{j=0}^{\omega-2} m_j(x), & i = \omega - 1. \end{cases} \tag{4}$$

Then $B^{(\omega)\top}$ can be derived by mapping coefficients of $M_i^{(\omega)}(x)$ to the i-th row of $B^{(\omega)\top}$, denoted as $B_i^{(\omega)\top}$:

$$B_i^{(\omega)\top} = \begin{cases} \boldsymbol{c}\left[M_i^{(\omega)}(x) \times (-1)^{\frac{\omega-2}{2}}\right] & i = 0 \\ \boldsymbol{c}\left[M_i^{(\omega)}(x)\right] & 0 < i \le \omega - 1 \end{cases} \tag{5}$$

where $\boldsymbol{c}\left[M_i^{(\omega)}(x)\right]$ means the coefficient vector of polynomial $M_i^{(\omega)}(x)$. Figure 3 shows an example of $F(\omega = 4)$ and $F(\omega = 6)$, where $F(\omega)$ is used to denote the set of $F(k,n)$ with a same ω.

For $F(\omega_1)$ and $F(\omega_2)$, when $\omega_2 - \omega_1 = 2$, the polynomial sequence $m_i(x)$ of $F(\omega_2)$ will have 2 additional terms compared with $F(\omega_1)$: $m_{\omega_2-3}(x)$ and $m_{\omega_2-2}(x)$. And $B^{(\omega)\top}$ will also expand accordingly, as shown in Figure 3.

Additionally, based on the relationship between convolution and polynomial multiplication, we can get the relationship:

$$c\left[M_i^{(\omega)}(x) \cdot m_\Delta(x)\right] = \boldsymbol{c}\left[M_i^{(\omega)}(x)\right] * \boldsymbol{c}\left[m_\Delta(x)\right], \tag{6}$$

where $m_\Delta(x)$ denotes any of the additional polynomial or their product, and $*$ denotes the convolution of two discrete sequences (distinct from the "convolution" used in CNN). Furthermore, based on the properties of convolution and cross-correlation, we have the following relationship:

$$\left\{\boldsymbol{c}\left[M_i^{(\omega)}(x)\right] * \boldsymbol{c}\left[m_\Delta(x)\right]\right\} \cdot \boldsymbol{d}_j = \boldsymbol{c}\left[M_i^{(\omega)}(x)\right] \cdot \left\{\boldsymbol{d}_j \circledast \boldsymbol{c}\left[m_\Delta(x)\right]\right\}, \tag{7}$$

where $\boldsymbol{d}_j$ denotes the j-th column of the input tile $\boldsymbol{d}$, $\cdot$ denotes the inner product of vectors, and $\circledast$ denotes cross-correlation. Combining Equations (6) and (7), we can get the recursive relationship:

$$B_{i_2}^{(\omega)\top} \cdot \boldsymbol{d}_j = B_{i_1}^{(\omega-2)\top} \cdot \left\{\boldsymbol{d}_j \circledast \boldsymbol{c}[m_\Delta(x)]\right\}, \tag{8}$$

and (-1) will be additionally multiplied when $i = 0$, according to Equation (5). The correspondence between i_2 and i_1 is determined by the remaining polynomials after extracting $m_\Delta(x)$. As the example illustrated in Figure 3, the rows with the same color indicate correspondence.

Based on the relationships above, we propose the RIT method. When computing the vector dot product between $B_i^{(\omega)\top}$ and $\boldsymbol{d}_j$, we can extract the additional polynomials and first calculate cross-correlation between their coefficient vectors and $\boldsymbol{d}_j$. Remaining computations can be done by multiplying the intermediate result vectors with $B_i^{(\omega-2)\top}$, or continue calculating cross-correlation by further decomposing $B_i^{(\omega-2)\top}$.

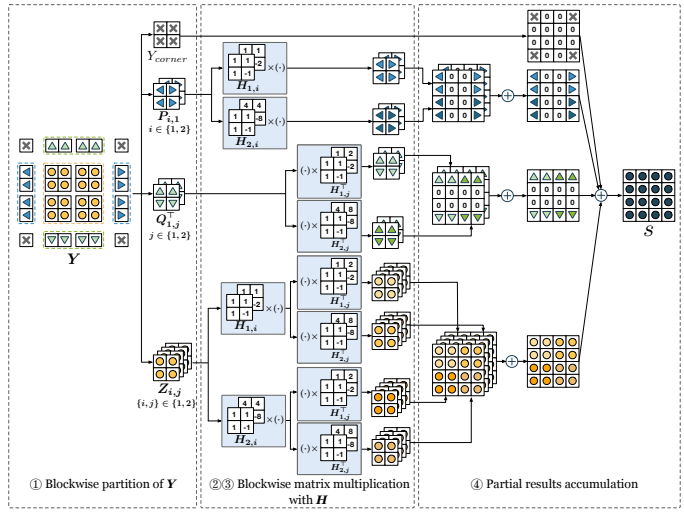

Figure 6: Blockwise Output Transformation for $F(4,3)$. The ①②③④ correspond to ❶❷❸❹ of *BOT procedure* in Section 3.3.

RIT procedure. For $F(\omega)$, the computation of $D_j = B^{(\omega)\top} d_j$ can be done recursively: ❶ perform correlation of $\boldsymbol{d}_j$ with the coefficient vectors of $m_{\omega-2}(x)$, $m_{\omega-3}(x)$ and $m_{\omega-2}(x)m_{\omega-3}(x)$ to get result vectors $\boldsymbol{h}_1$, $\boldsymbol{h}_2$ and $\boldsymbol{h}_3$; ❷ multiply the $B^{(\omega-2)\top}$ with $\boldsymbol{h}_3$; ❸ continue to perform correlation operations on $\boldsymbol{h}_1$ and $\boldsymbol{h}_2$; ❹ concatenate the results to get D_j. As for the final result, the input transformation can be obtained similarly through $U = (B^\top D)^\top$. An example of $F(\omega = 6)$ is shown in Figure 4. For simplicity, only the process of $D_j = B^{(6)\top} d_j$ is shown.

When using RIT to design hardware computing units, it is possible to preserve all the hardware structures of $F(\omega')$ (where $\omega' < \omega$) while implementing the input transformation functionality of $F(\omega)$.

3.3 Blockwise Winograd Output Transformation

The key idea of BOT is decomposing matrix $A^\top$ and matrix Y into 2×2 sub-matrices. It enables the output transformation to be performed using block matrix computation. Moreover, the sharing mechanism of $A^\top$ and the numerical distribution pattern of its internal elements are exploited to unify different output transformations. This approach also transforms associated multiplication operations into addition, subtraction, and shift calculations.

The sharing mechanism of $A^\top$ has been observed in [6], indicating repeated values in $A^\top$ across different $F(k,n)$ with same ω. We further exploit it to any $F(k,n)$, as shown in the left of Figure 5. When k or n increases, the center part of $A^\top$ will expand accordingly in units of 2×2 cells. The first/last column of $A^\top$ has a fixed pattern: the value at the first/last row is 1, while all other positions are 0s.

To exploit these characteristics, we divide $A^\top$ to a center matrix H and two edge vectors $\boldsymbol{e}_1$, $\boldsymbol{e}_\omega$, and further divide H into 2×2 sub-matrices. According to our Winograd polynomial construction strategy in Equation (3), the value distribution pattern of each sub-matrix can be summarized as:

$$H_{i,j} = 2^r \begin{bmatrix} 1 & 1 \\ 2^t & -2^t \end{bmatrix}, \tag{9}$$

where $r = 2(i-1)(j-1)$ and $t = (j-1)$. An example of the blockwise breakdown and the value distribution pattern is shown in Figure 5.

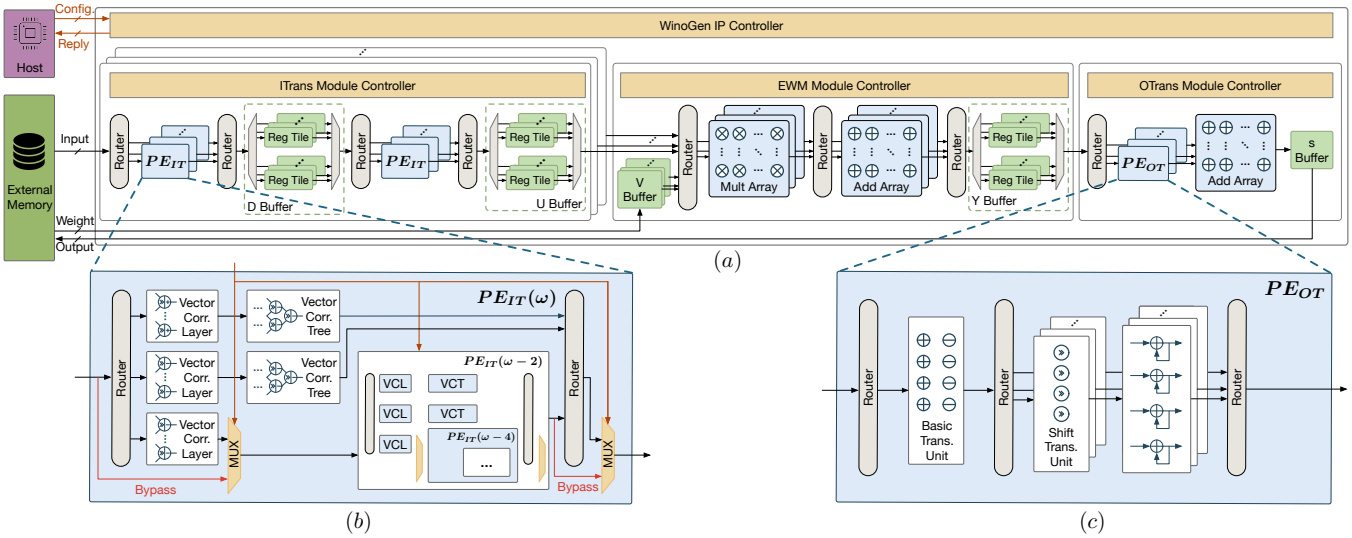

Figure 7: Architecture of the Winograd acceleration IP generated by WinoGen: (a) the overall architecture of WinoGen IP; (b) the detailed architecture of PE_{IT}; (c) the detailed architecture of PE_{OT}.

Based on the observations above, we divide $\boldsymbol{A}^\top$, $\boldsymbol{Y}$ and $\boldsymbol{A}$ into 2×2 sub-matrices to perform block matrix computations. The Winograd output transformation is therefore reformulated as:

$$\begin{aligned}\boldsymbol{A}^\top\boldsymbol{Y}\boldsymbol{A} &= \begin{bmatrix}\boldsymbol{e}_1 & \boldsymbol{H} & \boldsymbol{e}_\omega\end{bmatrix} \times \begin{bmatrix} Y(1,1) & \boldsymbol{q}_1^\top & Y(1,\omega) \\ \boldsymbol{p}_1 & \boldsymbol{Z} & \boldsymbol{p}_\omega \\ Y(\omega,1) & \boldsymbol{q}_\omega^\top & Y(\omega,\omega)\end{bmatrix} \times \begin{bmatrix}\boldsymbol{e}_1^\top \\ \boldsymbol{H}^\top \\ \boldsymbol{e}_\omega^\top\end{bmatrix}^\top \\ &= \begin{bmatrix} Y(1,1) & 0 & Y(1,\omega) \\ 0 & 0 & 0 \\ Y(\omega,1) & 0 & Y(\omega,\omega)\end{bmatrix} + \boldsymbol{H}\begin{bmatrix}\boldsymbol{p}_1 & 0 & \boldsymbol{p}_\omega\end{bmatrix} + \begin{bmatrix}\boldsymbol{q}_1^\top \\ 0 \\ \boldsymbol{q}_\omega^\top\end{bmatrix}\boldsymbol{H}^\top + \boldsymbol{H}\boldsymbol{Z}\boldsymbol{H}^\top.\end{aligned} \tag{10}$$

The $\boldsymbol{Y}$ matrix is divided into four parts: the four corners, the first row $\boldsymbol{q}_1^\top$ and the last row $\boldsymbol{q}_\omega^\top$ (excluding corners), the first column $\boldsymbol{p}_1$ and the last column $\boldsymbol{p}_\omega$ (excluding corners), and the central part $\boldsymbol{Z}$. We combine the four corners into a single 2×2 matrix, combine $\boldsymbol{q}_1^\top$ and $\boldsymbol{q}_\omega^\top$ into several 2×2 matrices, and combine $\boldsymbol{p}_1$ and $\boldsymbol{p}_\omega$ into several 2×2 matrices. We then divide $\boldsymbol{Z}$ into several 2×2 matrices. Subsequently, we perform blockwise computation between the partitioned $\boldsymbol{Y}$ and the partitioned $\boldsymbol{H}$ and accumulate the results onto corresponding positions of the output matrix $\boldsymbol{s}$. An example of $\boldsymbol{F}(4,3)$ is shown in Figure 6.

The calculations required in `BOT` for sub-matrices from the four parts of $\boldsymbol{Y}$ are respectively: remain unchanged, left-multiplied by $\boldsymbol{H}_{i,j}$, right-multiplied by $\boldsymbol{H}_{i,j}^\top$, left-multiplied by $\boldsymbol{H}_{i,j}$ and right-multiplied by $\boldsymbol{H}_{i,j}^\top$. The multiplication between $\boldsymbol{H}_{i,j}$ and the sub-matrices of $\boldsymbol{Y}$ can be divided into a set of add/sub-operations and a set of shift operations. Taking $\boldsymbol{Z}_{i,j}$ as an example, it can be seen that the add/sub-operations between the elements and the shift operations for each output value can be done separately:

$$\begin{aligned}\boldsymbol{H}_{i_1,j_1}\boldsymbol{Z}_{i,j}\boldsymbol{H}_{i_2,j_2}^\top &= 2^{r_1}\begin{bmatrix}1 & 1 \\ 2^{t_1} & -2^{t_1}\end{bmatrix} \times \begin{bmatrix}a & b \\ c & d\end{bmatrix} \times 2^{r_2}\begin{bmatrix}1 & 2^{t_2} \\ 1 & -2^{t_2}\end{bmatrix} \\ &= 2^{(r_1+r_2)}\begin{bmatrix}(a+b+c+d) & 2^{t_2}(a-b+c-d) \\ 2^{t_1}(a+b-c-d) & 2^{t_1+t_2}(a-b-c+d)\end{bmatrix}.\end{aligned} \tag{11}$$

BOT procedure. Overall, we can summarize the procedures of `BOT` as follows: ❶ divide $\boldsymbol{Y}$ to 2×2 sub-matrices, determine the type of blockwise computation required by each sub-matrix, and identify corresponding $\boldsymbol{H}_{i,j}$ matrices; ❷ conduct addition and subtraction operations according to the type of blockwise computation required by the sub-matrix; ❸ conduct shift operations according to $\boldsymbol{H}_{i,j}$; ❹ accumulate computation results onto corresponding positions of $\boldsymbol{s}$.

4 WinoGen IP Architecture

Based on our `SDW` algorithm, we propose a powerful and efficient hardware architecture as a template for `WinoGen` to generate IPs. The overall architecture is shown in Figure 7(a). It can be hosted and dynamically configured by an outside processor or controller, and takes inputs and weights from external memory and return outputs.

Inside the `WinoGen` IP, *Input Transformation Module*, *Element-Wise Multiplication Module* and *Output Transformation Module* are exploited to compute `RIT`, element-wise multiplication and `BOT`. The IP built under the static configuration of $\boldsymbol{F}(k,n)$ can be dynamically configured to support arbitrary $\boldsymbol{F}(k',n')$, as long as $\omega' \le \omega$ and $k' \le k$, which will be explained in the following subsections.

4.1 Input Transformation Module

The Input Transformation Module (ITrans Module) computes Winograd input transformation using `RIT` method. It takes feature map tile $\boldsymbol{d}$ as an input and outputs transformed feature map $\boldsymbol{U}$. In our design, we use PN to denote parallel numbers. In a single `WinoGen` IP, PN_C ITrans Modules are deployed in parallel to simultaneously compute PN_C input channels. PN_C can be defined by the user or decided by `WinoGen` optimizer, varying from 1 to 4.

$\mathbf{PE_{IT}}$ is the processing element (PE) deployed in the ITrans Module. The detailed architecture is shown in Figure 7(b). Following the mechanism of `RIT` in Section 3.2, PE_{IT} has a nested structure. We denote the top layer of PE_{IT} as $PE_{IT}(\omega)$, and it recursively contains $PE_{IT}(\omega-2)$, $PE_{IT}(\omega-4)$, ..., until reaches $PE_{IT}(2)$, whose outputs directly equals to inputs. After taking in a column or a row of the input matrix, three Vector Correlation Layers conduct the ❶ procedure of `RIT`, the $PE_{IT}(\omega-2)$ conducts the ❷ procedure, two Vector Correlation Trees conduct the ❸ procedure. The nodes in Vector

Correlation Layers and Vector Correlation Trees can perform shift and add/sub operations on input values, thus realizing vector cross-correlation. The Vector Correlation Layers contain $(\omega - 2)$ nodes, while the Vector Correlation Trees are constructed as reduction trees with $(\omega - 4)$ layers. The data path of PE_{IT} is fully pipelined, thus enabling it to process one column of $\boldsymbol{d}$ in one clock cycle.

There are two groups of PE_{IT} in the ITrans Module, each containing PN_{IT} PEs. The first group conducts $\boldsymbol{D} = \boldsymbol{B}^\top \boldsymbol{d}$ and the second group conducts $\boldsymbol{U} = (\boldsymbol{B}^\top \boldsymbol{D}^\top)^\top$, thus realizing the reuse of PE_{IT} architecture. The parallel number PN_{IT} (varying from 1 to ω) is decided by `WinoGen` optimizer, taking into account the resource constraints and the throughput of the whole IP.

Buffers in the ITrans Module are built with registers and constructed by double-buffering. For D Buffer and U Buffer, all have two groups of Register Tiles, while each tile consists of ω registers to store a column or row of the matrices. For each group of D Buffer, the tile number is fixed to ω due to the requirement of matrix transpose. For Y Buffer, the tile number equals PN_{IT}.

Compatibility of the ITrans Module (constructed under $\boldsymbol{F}(k, n)$) lies in that it supports any $\boldsymbol{F}(k', n')$ input transformation as long as $\omega' \leq \omega$. This is realized by the **Bypasses** in PE_{IT}, linking all the nested PEs, as shown in Figure 7(b). The bypass behavior can be dynamically configured by the ITrans Module Controller.

4.2 Element Wise Multiplication Module

The Element Wise Multiplication Module (EWM Module) computes the element-wise multiplication in Winograd convolution and conducts channel-wise accumulation. It takes transformed feature map $\boldsymbol{U}$ and transformed kernel $\boldsymbol{V}$ from PN_C input channels as inputs, and outputs EWM result $\boldsymbol{Y}$ of an output channel.

Multiplier Array performs the element-wise multiplication. PN_C Multiplier Arrays are deployed in parallel, each having $\omega \times PN_{EWM}$ multipliers. The multipliers are pipeline multipliers with a stage number of T_{mult}, decided by `WinoGen` optimizer.

Adder Array performs channel-wise accumulation. There are $(PN_C - 1)$ Adder Arrays, each having $\omega \times PN_{EWM}$ adders implemented by combinational logics.

Buffers in the EWM Module are divided into V Buffer and Y Buffer. V Buffer comprises PN_C tiles of $\omega \times \omega$ registers to store $\boldsymbol{V}$ matrix. Y Buffer is double-buffered, with two groups of $\omega \times PN_{EWM}$ register tiles.

Compatibility of the EWM Module is the same as the ITrans Module. By selecting part of the outputs from the Multiplier Arrays and Adder Arrays, any $\boldsymbol{F}(k', n')$ element-wise multiplication is supported when $\omega' \leq \omega$. When $\omega' < \omega$, only the first ω' outputs in each row of the arrays are valid, and stored in Y Buffer under the control of the EWM Module Controller.

4.3 Output Transformation Module

The Output Transformation Module (OTrans Module) computes Winograd output transformation using `BOT` method. It takes EWM result $\boldsymbol{Y}$ as an input and outputs Winograd convolution result $\boldsymbol{s}$.

$\mathbf{PE_{OT}}$'s architecture is shown in Figure 7(c). Following the rule of `BOT` in Section 3.3, PE_{OT} is designed as a three-stage structure. Each PE_{OT} takes 2×2 sub-matrices of $\boldsymbol{Y}$ as input. The Basic Transformation Unit, which contains 4 adders and 4 subtractors and several MUXers, conducts the ❷ procedure of `BOT`. The Shift Transformation Unit, organized in groups of 4 shifters, conducts the ❸ procedure of `BOT`. The Accumulator Unit, organized in groups of 4 accumulators, conducts the ❹ procedure of `BOT`. For each $\boldsymbol{Z}_{i,j}$, there are $(k/2)$ types of $\boldsymbol{H}_{i_1,j_1}$ be left-multiplied and $(k/2)$ types of $\boldsymbol{H}^\top_{i_2,j_2}$ be right-multiplied to it. Combining Equation (11), each PE_{OT} is equipped with 1 Basic Trans. Unit, $(k^2/4)$ groups of Shift Trans. Unit, and $(k^2/4)$ groups of Accumulator Unit. PE_{OT} is fully pipelined with $T_{PE_{OT}} = 3$ stages. The parallel number of PE_{OT} is defined as PN_{OT}, varying from 1 to $(\omega^2/4)$.

The OTrans Module Controller and Routers handle the tasks of 2×2 partition of $\boldsymbol{Y}$, including providing the parameters used in ❶ procedure of `BOT`. The Adder Array accumulates the outputs from PE_{OT}s and saves the final result in the s Buffer.

Compatibility of the OTrans Module (constructed under $\boldsymbol{F}(k, n)$) lies in that it supports any $\boldsymbol{F}(k', n')$ output transformation as long as $k' \leq k$. This is because the hardware architecture of the OTrans Module is only affected by k, while different n affects the number of blockwise computations and the values in $\boldsymbol{A}^\top$. Thus, it can be dynamically adjusted by the controller. And the output transformation of $\boldsymbol{F}(k', n')$ when $k' < k$ can be done by blocking the output from part of the accumulators in PE_{OT}, similar to the idea in [6].

4.4 WinoGen Optimizer

Due to the `WinoGen` flexibility, we can navigate the design space of generated IPs, tailoring them to fit specific requirements. The `WinoGen` Optimizer models the utilization of DSP and LUT as resource constraints, and models the throughput of IP as optimization target.

DSP utilization is derived by calculating the number of multipliers in the EWM Module. LUT utilization is derived by calculating the combinational operators including adders, subtractors and shifters in each module. And IP throughput is derived by calculating the expected clock cycles to process a certain number of input tiles (e.g. 1000 tiles).

With the estimation of resource overhead and latency in place, the design space exploration of IP architecture can be formulated as an Integer Linear Programming problem. Given the resource constraints of DSP and LUT, the throughput is maximized under the constraints by setting the IP hardware parameters PN_{IT}, PN_{EWM}, PN_{OT} and PN_C.

5 Experiments

For experimental evaluations, we generate IPs with `WinoGen`, using Xilinx Vivado 2023.2 to implement and test them on ZCU104.

IP-level Evaluation. We firstly carry out IP-level evaluation to test the operator-level performance of generated IPs, while testing the functionality of the `WinoGen` optimizer. We select three fundamental types of IP: $\boldsymbol{F}(4, 1)$, $\boldsymbol{F}(4, 3)$ and $\boldsymbol{F}(6, 3)$ for generality. For each fundamental IP type, we each test three configuration cases: (a) "PN_{min}" means that when constructing the IP, the parallel numbers are set to the minimum; (b) "Constrained" means a set of hardware resource constraints is given, and the parallel numbers are decided by `WinoGen` optimizer; (c) "PN_{max}" means the parallel numbers are set to the maximum.

The configurations, resource utilization, throughput, and compatibility of the IPs are shown in Table 1. For extremely limited

Table 1: Optimized WinoGen IP under different resource constraints (* is used to test the throughput of the IP)

IP Type	Config. Type	IP Config.				Resource Util. (Constraint)			Thro. /GOPS				Supported Winograd Type
		PN_{IT}	PN_{EWM}	PN_{OT}	PN_C	DSP	LUT	FF	1×1 Kernel	3×3 Kernel	5×5 Kernel	7×7 Kernel	
F(4,1)	PN_{min}	1	1	1	1	4	1689	2191	1.308	5.559	-	-	F(2,3)*, F(4,1)*
	Constrained	4	4	4	1	16(20)	2267(2400)	2901	5.232	22.236	-	-	
	PN_{max}	4	4	4	4	64	4858	7579	36.624	92.868	-	-	
F(4,3)	PN_{min}	1	1	1	1	6	2441	3587	1.308	9.883	7.121	-	F(2,5)*, F(4,3)*; F(2,3), F(4,1)*
	Constrained	2	2	3	4	48(64)	8267(8400)	11678	12.208	123.824	86.764	-	
	PN_{max}	6	6	9	4	144	17472	18476	36.624	371.472	260.292	-	
F(6,3)	PN_{min}	1	1	1	1	8	3757	5703	1.308	12.508	16.023	7.930	F(2,7)*, F(4,5)*, F(6,3)*; F(2,5), F(4,3), F(6,1)*; F(2,3), F(4,1)
	Constrained	8	8	16	2	128(128)	31899(40000)	25307	35.316	412.02	517.968	255.06	
	PN_{max}	8	8	16	4	256	41093	40238	82.404	835.812	1041.168	511.428	

Table 2: WinoGen evaluation and comparison with state-of-the-art designs.

	[3]		[5]	[11]	Vitis-AI[12][6]		[6]		WinoGen IP-F(4,3)×12			WinoGen IP-F(6,3)×5		
Platform	KU115		VCU118	VU9P	ZCU102		ZCU102		ZCU104			ZCU104		
CNN Model	VGG16	AlexNet	VGG16	VGG16	VGG16	YOLOv2	VGG16	YOLOv2	VGG16	AlexNet	YOLOv2	VGG16	AlexNet	YOLOv2
Freq. (MHz)	235	220	200	167	281		214		317			317		
Supported Precision	8 bit / 16 bit	8 bit / 16 bit	16 bit	12 bit	8 bit		8-16 bit		8-16 bit			8-16 bit		
DSP (% of total)	4318 (78%)	4318 (88%)	3224 (47%)	5163 (75%)	1926 (76%)		2345 (93%)		1728 (100%)			1280 (74.1%)		
LUT (% of total)	257862 (39%)	257862 (40%)	-	-	-		-		209664 (91%)			205465 (89.2%)		
Power (W)	22.3	22.9	-	45.9	-		-		8.25			7.45		
Thro. (GOPS)	4022	3265	3772	3775.7	1225.2	1008	3120.3	1717.7	4453.2	4190.8	2408.6	4170.7	4863.1	2640.8
DSP Eff. (GOPS/DSP)	0.991	0.764	1.17	0.65	0.636	0.523	**1.33**	0.73	2.58	2.43	1.39	**3.26**	**3.80**	2.06
Energy Eff. (GOPS/W)	**180.4**	98.3	-	73.5	-	-	-	-	539.78	507.98	291.95	**559.83**	**652.77**	354.47

hardware resources, `WinoGen` can generate IPs with minimal resource consumption. In cases where hardware resources are abundant, `WinoGen` can create IPs with ultra-high computational performance (over **1000** GOPS). For other scenarios, `WinoGen` can utilize its optimizer to generate optimized IPs, aiming for maximum throughput while adhering to the given resource constraints.

System-level Evaluation. We then conduct a system-level evaluation to test the network-level performance of a system equipped with multiple `WinoGen` IPs. We select two types of IP: $\boldsymbol{F}(4,3)$ and $\boldsymbol{F}(6,3)$, each is built under PN_{max} configuration. Then we separately implement these two IPs on ZCU104, maximizing hardware resource utilization.

The evaluation results and comparison with other state-of-the-art designs are shown in Table 2. The selected related works are all IP-based or template-based. The throughput, DSP efficiency and energy efficiency are only for the convolution layers, except [3]. For [3], only the metrics for 8-bit configuration are listed here.

For our implementation, the DSP efficiency reaches **3.80** GOPS/DSP and energy efficiency reaches **652.77** GOPS/W when deploying IP-F(6,3) to compute AlexNet. Considering the first layer of AlexNet is not yet supported by `WinoGen` IPs due to its non-1 stride, we use the metrics for VGG16 to conduct an apple-to-apple comparison with other works. When computing VGG16, our design achieves DSP efficiency up to **3.26** GOPS/DSP and energy efficiency up to **559.83** GOPS/W, thus showing **2.45**× and **3.10**× improvements compared with state-of-the-arts [3, 6].

6 Conclusion

In this paper, we propose an efficient IP generation method for arbitrary Winograd convolution on FPGA. By introducing `SDW` method and leveraging Chisel for algorithm-architecture co-design, we have successfully developed a highly configurable Winograd convolution IP generator `WinoGen`. Based on our resource and latency models, we optimize the generated IP configurations. Experimental results demonstrate the effectiveness of `WinoGen`, showing significant improvements in DSP efficiency and energy efficiency compared to state-of-the-art methods.

References

[1] A. N. Mazumder, J. Meng, H.-A. Rashid, U. Kallakuri, X. Zhang, J.-S. Seo, and T. Mohsenin, "A survey on the optimization of neural network accelerators for micro-ai on-device inference," *IEEE JETCAS*, vol. 11, no. 4, pp. 532–547, 2021.

[2] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. CVPR*, 2016.

[3] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proc. ICCAD*, 2018.

[4] Y. Liang, Q. Xiao, L. Lu, and J. Xie, "FCNNLib: A flexible convolution algorithm library for deep learning on fpgas," *IEEE TCAD*, vol. 41, no. 8, p. 2546–2559, 2022.

[5] J. Shen, Y. Huang, M. Wen, and C. Zhang, "Toward an efficient deep pipelined template-based architecture for accelerating the entire 2-D and 3-D CNNs on FPGA," *IEEE TCAD*, vol. 39, no. 7, pp. 1442–1455, 2019.

[6] X. Liu, Y. Chen, C. Hao, A. Dhar, and D. Chen, "WinoCNN: Kernel sharing winograd systolic array for efficient convolutional neural network acceleration on FPGAs," in *Proc. ASAP*, 2021.

[7] C. Yang, Y. Wang, X. Wang, and L. Geng, "WRA: A 2.2-to-6.3 TOPS highly unified dynamically reconfigurable accelerator using a novel winograd decomposition algorithm for convolutional neural networks," *IEEE TCAS I*, vol. 66, no. 9, pp. 3480–3493, 2019.

[8] S. Winograd, *Arithmetic complexity of computations*. Siam, 1980, vol. 33.

[9] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. CVPR*, 2016.

[10] D. Huang, X. Zhang, R. Zhang, T. Zhi, D. He, J. Guo, C. Liu, Q. Guo, Z. Du, S. Liu *et al.*, "DWM: A decomposable winograd method for convolution acceleration," in *Proc. AAAI*, vol. 34, no. 04, 2020.

[11] H. Ye, X. Zhang, Z. Huang, G. Chen, and D. Chen, "HybridDNN: A framework for high-performance hybrid DNN accelerator design and implementation," in *Proc. DAC*, 2020.

[12] Xilinx, "Vitis-ai model zoo," https://github.com/Xilinx/Vitis-AI/tree/master/model_zoo, 2023.