Sparse Winograd Convolutional neural networks on small-scale systolic arrays

Feng Shi, Haochen Li, Yuhe Gao, Benjamin Kuschner, Song-Chun Zhu
University of California Los Angeles
Los Angeles, USA
{shi.feng,sczh}@cs.ucla.edu

ABSTRACT
The reconfigurability, energy-efficiency, and massive parallelism on FPGAs make them one of the best choices for implementing efficient deep learning accelerators. However, state-of-art implementations seldom consider the balance between high throughput of computation power and the ability of the memory subsystem to support it. In this paper, we implement an accelerator on FPGA by combining the sparse Winograd convolution, clusters of small-scale systolic arrays, and a tailored memory layout design. We also provide an analytical model analysis for the general Winograd convolution algorithm as a design reference. Experimental results on VGG16 show that it achieves very high computational resource utilization, $20x \sim 30x$ energy efficiency, and more than $5x$ speedup compared with the dense implementation.

KEYWORDS
FPGA, Neural networks, Winograd Convolution, systolic arrays

1 INTRODUCTION
Convolutional neural network (CNN) is a class of deep learning algorithms which has become dominant in various computer vision tasks [10, 18], so it is attracting research on acceleration for computational and power efficiencies. The core computations in the algorithm are convolution operations with multi-dimensional data, e.g. 3-D feature maps (FM) and 4-D filters, which require a high density of memory accesses and high throughput of the computation engine. One research topic emerging in recent years is to deploy the convolution operations onto FPGAs [6, 7, 9, 11], since FPGAs consist of massive compute units, e.g. DSP blocks, and storage elements interconnected by reconfigurable switch blocks. The most recent works on systolic array-based FPGA accelerators [3, 15] deliver significant performance improvement on the automata of high-level synthesis (HLS) design flow. Unlike the works [5, 15], which first construct 2-D mesh architecture for systolic array then let the loops of codes to fit on these arrays (bitstream generated once), we recursively break the memory layout down to small blocks then map these blocks onto small-scale systolic arrays to perform multiplications of submatrices, and share these submatrices among working arrays to reduce required memory bandwidth. Another performance improvement can be achieved from algorithmic perspective by applying the Winograd transform. This approach attracts more and more attention from researchers since its first GPU implementation [17]. Winograd CNN accelerators on FPGAs are also well studied recently [1, 11]; however, the greater volume after the Winograd transformation is stressing on FPGAs. To handle this issue we adopt an efficient memory layout, adopt the pruned Winograd weights [2] and their elaborate hardware, and extend the computation into 3-D. Pruning neural networks has been proven to greatly decrease both latency and energy consumption for all range of devices [13]. The major contributions are summarized in the following:

- Unified small-scale systolic arrays for both Winograd transform and matrix multiplications. We maximize the reusability of the existing design, e.g. RTL, for multiple modules. These modules share common characteristics, like matrix multiplication alike arithmetic operations.
- Efficient memory access layout. We employ a recursive memory access pattern to increase locality of buffers. This pattern significantly impacts the overall performance.
- Block-based sparse matrix compression. We employ this compression technique to adopt the above mentioned recursive memory layout.
- A comprehensive model analysis of Winograd convolution. We propose an analytical model to investigate the performance and energy consumption, and based on the analysis we use the conclusion as our design guidance.

2 BACKGROUND
2.1 Spatial Convolution
The convolution layer in a feedforward pass takes $C$ channels of $H \times W$ feature maps $D$ as input, and convolve each of $K$ filters of dimension $C \times r \times r$ with the input feature maps to produce $K$ output feature maps, $Y$, of dimension $(H - r + 1) \times (W - r + 1)$. Let $s$ be the stride and assume that the width and height of the filters are the same, then the mathematical description of the convolution is

$$Y_{k, i, j} = \sum_{t=1}^{C} \sum_{p=1}^{r} \sum_{q=1}^{r} G_{k, t, p, q} \times D_{t, i, s \cdot t + p, j, s \cdot q}$$

2.2 Winograd Algorithm
Winograd proposed an efficient algorithm for short convolutions [20] in computing of finite impulse response (FIR) filters in the signal processing field. [17] extends the Winograd algorithm to convolutional neural networks on GPU and CPU.
By applying Winograd transform to an r-tap FIR filter denoted as $F(m, r)$, which computes $m$ outputs with the filter size of $r$, the number of multiplications is reduced from $m \times r$, if through the spatial convolution, to $m + r + 1$.

2.2.1 1-D Winograd Convolution. Taking $F(2, 3)$ as an example, Winograd algorithm first transforms an input vector $d = (d_0, d_1, d_2, d_3)$ and filter $g = (g_0, g_1, g_2)$ into $j = (j_0, j_1, j_2, j_3)$ and $h = (h_0, h_1, h_2, h_3)$ respectively through

$$
j_0 = d_0 - d_2, \quad h_0 = g_0
$$

$$
j_1 = d_1 + d_2, \quad h_1 = \frac{g_0 + g_1 + g_2}{2}
$$

$$
j_2 = d_2 - d_1, \quad h_2 = \frac{g_0 - g_1 + g_2}{2}
$$

$$
j_3 = d_1 - d_3, \quad h_3 = g_2
$$

Next, element-wise multiplications are performed:

$$
c_0 = j_0 \times h_0, \quad c_1 = j_1 \times h_1, \quad c_2 = j_2 \times h_2, \quad c_3 = j_3 \times h_3
$$

Finally, the output $y = (y_0, y_1)$ can be generated via:

$$
y_0 = c_0 + c_1 + c_2, \quad y_1 = c_1 - c_2 - c_3
$$

The matrix form of the above procedure can be written as $y = A^T \left[ (Gg) \odot (B^T d) \right]$, where $\odot$ represents element-wise multiplication and

$$
A^T = \begin{bmatrix} 1 & 1 & 1 & 0 \\ 1 & -1 & -1 & 0 \end{bmatrix}, \quad G = \begin{bmatrix} 1 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{bmatrix}, \quad B^T = \begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & 1 \\ 0 & -1 & 1 \\ 0 & 1 & 0 \end{bmatrix}
$$

The element-wise product in (2) requires $m + r + 1 = 4$ multiplications, whereas the direct method does $m \times r = 2 \times 3 = 6$ multiplications.

2.2.2 2-D Winograd Convolution. The 1-D Winograd algorithm can be easily extended to 2-D or higher dimensional convolutions by being nested with itself. 2-D Winograd algorithm $F(m \times m, r \times r)$ can be formulated as follows,

$$
Y = A^T \left[ (GgG^T) \odot (B^T dB) \right] A
$$

where $d$ and $g$ are tiles of input and the filter, having size of $l \times l$ ($l = m + r - 1$) and $r \times r$, respectively. The size of the output tile $Y$ is $m \times m$.

For larger input images, the Winograd transform is performed with the overlapping of tiles, with overlapping size $r - 1$, along each dimension. When applying Winograd algorithm to a convolution layer of CNNs, the tiles along the channel dimension of this layer can be fetched simultaneously and each of them is applied with (4).

3 ALGORITHM AND OPTIMIZATIONS

This section gives an overview of our algorithm and presents several optimization methods. Fig. 1 shows the overview of our algorithm which consists of three stages of the Winograd-based convolution: input feature map and kernel transformations, matrix multiplications, and the inverse transformation of the output feature maps.

These three stages form the pipeline of the data flow of our system design.

3.1 Reduction to Matrix multiplication

By reformulating (4) with the augmentation on the channel dimension, filter $k$, tile coordinates $(\tilde{x}, \tilde{y})$, and substitution of $U = GgG^T$ and $V = B^T dB$, we get

$$
Y_{k, \tilde{x}, \tilde{y}} = A^T \left[ \sum_{c=1}^{C} U_{k,c} \odot V_{c, \tilde{x}, \tilde{y}} \right] A
$$

The summation part inside the parenthesis of (5) can be disentangled into $(m + r - 1)^2$ individual multiplications of a matrix of size $(C \times K)$ with another of size $(C \times [H/m][W/m])$.

$$
M_{k, \tilde{x}, \tilde{y}} = \sum_{c=1}^{C} U_{k,c} \odot V_{c, \tilde{x}, \tilde{y}}
$$

Another benefit of this reformation into matrix multiplications is that the number of inverse transforms has also been reduced over $C$ channels [17], since the factorization of inverse transform along channels amortizes the cost. With this reformation, the matrix multiplications are then efficiently implemented on FPGAs.

3.2 Matrix multiplications and memory access patterns

As described in section 3.1, Winograd convolution can be computed efficiently with matrix multiplications on GPUs or FPGA platforms. To optimize the performance of matrix multiplication, we employ the Z-Morton memory layout [8], which has been widely studied for the Cache oblivious algorithms on multithreaded CPUs [8, 12] and image processing on FPGAs [4]. This memory layout increases both spatial and temporal locality of memory accesses of matrix multiplication and arithmetic operations [8].
memory layout, e.g. $5$ for $B_5$. $BI$ is the list of starting indices of each block within the other three arrays, e.g. $is$ of $BI$ refers to the starting index in $AI$, $AJ$, and $AN$ of information corresponding to $B_l$. Elements in $AI$ and $AJ$ represent the row and column number of the nonzeros in its own block, respectively, and $AN$ stores the value of the corresponding nonzero. For $B_5$, the values of nonzeros are $b_{0,0}$, $b_{1,2}$, and $b_{3,1}$, the corresponding column numbers are $0$, $2$, and row numbers are $0$, $1$, $3$ in $AJ$ and $AI$, respectively. The compressed blocks are still fetched following the order determined by Z-Morton layout.

4 ARCHITECTURE DESIGN

This section discusses our implementation of accelerator for Winograd convolution. The most time-consuming parts in the computation pipeline are the Winograd transform for feature maps and matrix multiplications. In our design, we propose using unified small-scale systolic arrays, of size $I \times I$ ($I = m + r - 1$), for both these arithmetic operations.

4.1 Winograd transform by Systolic Arrays

Recall the $2$-D Winograd transform nesting $2$ transform matrices, $B^T \cdot D \cdot B$. Instead of directly computing $B^T \cdot D \cdot B$, we change it into

$$
(D^T \cdot B)^T \cdot B. Thus, we let transform matrix $B$ be stationary inside the systolic arrays. In the first iteration (1) of the Fig. 3 $D^T$ passes through systolic arrays to operate with $B$ and the output is $P = (C + D^T \cdot B)^T$ (no additional transpose needed). This intermediate result $(D^T \cdot B)^T$ feeds back to systolic arrays as *new $D^T$* in the second iteration (2). Then $P' = C' + P \cdot B = (D^T \cdot B)^T \cdot B = B^T \cdot D \cdot B$ is the final result. Note that $C$ and $C'$ are zero-matrices and there is no multiplication occurred inside these systolic arrays—the value of elements of $B$ is just used to control the adder—such as, “$1$” for addition, “$-1$” for subtraction, and “$0$” for passing by the data to next processing element (PE) inside its systolic array.

The data sharing is through the overlapping of tiles, which has been described in section 2.2.2. Fig. 3 illustrates that $(m + r - 1)$ wide data stream enter through each systolic array, and among these data, $(r - 1)$ of them travel through the current systolic array and are forwarded to the next systolic array at the same direction. The output

Z-Morton uses a *divide and conquer* approach to access the memory as in Fig. 2 (a). It is actually derived from the recursive matrix multiplication described in Algorithm 1. Compared with Strassen’s algorithm, the latter is not cache-friendly in real situations, whereas the former can provide notable improvement in performance [12]. Note, instead of implementing the algorithm exactly, we unrolled memory access order to reorganize the memory layout.

The physical memory layout in FPGAs is essentially linear, Fig 2 (a) also provides an example of translating the logical block address to physical block address. As shown in Fig. 2 (a), the address translation is easily implemented with LUTs in FPGAs by interleaving the bits of the logical column and row addresses to generate the physical address of a block.

3.3 Pruned Winograd weights and memory access patterns

After pruning the Winograd weights, we store them in a block-based sparse coordinates format (BCOO)—only those $4 \times 4$ blocks containing nonzeros will be compressed and stored. Fig. 2 (b) shows an example where the block $B_3$ is a $4 \times 4$ tile, and it has 3 nonzeros. The information of these nonzeros are stored into vectors $BN$, $BI$, $AI$, $AJ$, and $AN$. $BN$ contains the block number for each block in

Figure 2: Z-Morton memory layout for both dense and sparse matrix [4, 8]: (a) the translation from logical layout to physical layout, (b) the block-based compressed coordinates (BCOO), $l \times l$ block and $l = 4$ for our design) for pruned Winograd weights

Algorithm 1 Divide and Conquer Matrix Multiplication

1: function 
2: \hspace{1em} recursive-matmult($A$, $B$, $C$)
3: \hspace{2em} $n = A$.rows
4: \hspace{2em} if $n == l$ then $\triangleright$ $l$ is the smallest tiling size
5: \hspace{3em} $c_{1,1} = A_{1,1} \times B_{1,1}$ $\triangleright$ matrix multiply of $I \times l$ tiles
6: \hspace{2em} else
7: \hspace{3em} partition $A$, $B$, and $C$ into tiles of size $\frac{n}{2} \times \frac{n}{2}$
8: \hspace{3em} $C_{1,1} = recursive-matmult(A_{1,1}, B_{1,1})$
9: \hspace{3em} $+ recursive-matmult(A_{1,2}, B_{2,1})$
10: \hspace{3em} $C_{1,2} = recursive-matmult(A_{1,1}, B_{1,2})$
11: \hspace{3em} $+ recursive-matmult(A_{1,2}, B_{2,2})$
12: \hspace{3em} $C_{2,1} = recursive-matmult(A_{2,1}, B_{1,1})$
13: \hspace{3em} $+ recursive-matmult(A_{2,2}, B_{2,1})$
14: \hspace{3em} $C_{2,2} = recursive-matmult(A_{2,1}, B_{1,2})$
15: \hspace{2em} end if
16: \hspace{1em} return $C$
17: end function

Figure 3: Small-scale Systolic Arrays for Winograd Transform

$\begin{pmatrix} D^T \cdot B \end{pmatrix}^T \cdot B$. Thus, we let transform matrix $B$ be stationary inside the systolic arrays. In the first iteration (1) of the Fig. 3 $D^T$ passes through systolic arrays to operate with $B$ and the output is $P = (C + D^T \cdot B)^T$ (no additional transpose needed). This intermediate result $(D^T \cdot B)^T$ feeds back to systolic arrays as *new $D^T$* in the second iteration (2). Then $P' = C' + P \cdot B = (D^T \cdot B)^T \cdot B = B^T \cdot D \cdot B$ is the final result. Note that $C$ and $C'$ are zero-matrices and there is no multiplication occurred inside these systolic arrays—the value of elements of $B$ is just used to control the adder—such as, “$1$” for addition, “$-1$” for subtraction, and “$0$” for passing by the data to next processing element (PE) inside its systolic array.

The data sharing is through the overlapping of tiles, which has been described in section 2.2.2. Fig. 3 illustrates that $(m + r - 1)$ wide data stream enter through each systolic array, and among these data, $(r - 1)$ of them travel through the current systolic array and are forwarded to the next systolic array at the same direction. The output

Figure 3: Small-scale Systolic Arrays for Winograd Transform

$\begin{pmatrix} D^T \cdot B \end{pmatrix}^T \cdot B$. Thus, we let transform matrix $B$ be stationary inside the systolic arrays. In the first iteration (1) of the Fig. 3 $D^T$ passes through systolic arrays to operate with $B$ and the output is $P = (C + D^T \cdot B)^T$ (no additional transpose needed). This intermediate result $(D^T \cdot B)^T$ feeds back to systolic arrays as *new $D^T$* in the second iteration (2). Then $P' = C' + P \cdot B = (D^T \cdot B)^T \cdot B = B^T \cdot D \cdot B$ is the final result. Note that $C$ and $C'$ are zero-matrices and there is no multiplication occurred inside these systolic arrays—the value of elements of $B$ is just used to control the adder—such as, “$1$” for addition, “$-1$” for subtraction, and “$0$” for passing by the data to next processing element (PE) inside its systolic array.

The data sharing is through the overlapping of tiles, which has been described in section 2.2.2. Fig. 3 illustrates that $(m + r - 1)$ wide data stream enter through each systolic array, and among these data, $(r - 1)$ of them travel through the current systolic array and are forwarded to the next systolic array at the same direction. The output
Winograd weight blocks need to be equipped with a decompressor. To understand how this cluster works, let us examine the example.

Algorithm 1

Split into two halves since some Winograd feature maps blocks are calculated. Also the sharing of circular FIFOs reduces the memory calculations, and we need some modifications on the cluster of systolic arrays.

4.2 Matrix Multiplication by Systolic Arrays

To perform the recursive matrix multiplication Algorithm 1 with hardware, we conceive the cluster of small-scale systolic arrays. Each cluster consists of $4 \times 4$ systolic arrays ($l = 4$ for our case) and a set of shared circular FIFOs built by shift-registers, shown in Fig. 4. To understand how this cluster works, let us examine the example from Fig. 2. By unrolling the recursive code given by Algorithm 1 and using the tiles of matrices organized by Z-Morton layout, we calculate sub-matrix $C_0$ by summing up the products of submatrices $A_0 \times B_0$ and $A_1 \times B_2$. $C_4$ by sum of $A_9 \times B_4$ and $A_1 \times B_6$, and so on.

\[
\begin{align*}
C_0 &= A_0 \times B_0 + A_1 \times B_2; \\
C_4 &= A_0 \times B_1 + A_1 \times B_6; \\
C_8 &= A_3 \times B_0 + A_9 \times B_2; \\
C_{12} &= A_3 \times B_4 + A_9 \times B_6; \\
&\vdots \\
C_9 &= A_4 \times B_5 + A_{12} \times B_{10}; \\
C_4 &= A_4 \times B_1 + A_{12} \times B_{14}; \\
&\vdots
\end{align*}
\]

As shown in Fig. 4 (a), $A_0$ is shared by northwest and southwest systolic arrays, $A_9$ is shared by northeast and southeast systolic arrays, and so on. After the first iteration, the partial results of $C_0$, $C_4$, $C_8$, and $C_{12}$ are produced and stored inside the corresponding systolic arrays. In the second iteration, the blocks $A_1$, $A_9$, $B_4$, and $B_9$ get into their corresponding systolic arrays and perform the matrix multiplications, and their products are accumulated to the partial results, which still stay in their systolic arrays from iteration 1. At iteration 3 the results of $C_0$, $C_4$, $C_8$, and $C_{12}$ are spilled out, and systolic arrays continue to work on the partial results of $C_1$, $C_5$, $C_9$, and $C_{13}$. This procedure continues until all the submatrices are calculated. Also the sharing of circular FIFOs reduces the memory bandwidth requirement by 4 folds.

When the computation is comprised of sparse matrix multiplications, we need some modifications on the cluster of systolic arrays. First, each of the circular FIFOs which supply the compressed Winograd weight blocks need to be equipped with a decompressor. Second, the circular FIFOs for Winograd feature maps are virtually split into two halves since some Winograd feature maps blocks are no longer shared between the systolic arrays. The overall memory access pattern is now determined by how the sparse blocks distributed in the memory layout. Take the sparse blocks $B_2$ and $B_5$ from Fig. 2 for example; now we notice that the computation of $C_0$ becomes $A_1 \times B_2$ only, $C_8$ becomes $A_9 \times B_2$, block $B_2$ is still shared by the products of submatrices $C_0$ and $C_8$.

4.3 Extends the computation into third dimension

Whenever the computation resource is available, we can extend the computation into higher dimensions. As we have analyzed in section 3.1, there are $(m + r - 1)^2$ independent matrix multiplications, and they can be executed in parallel with several clusters of systolic arrays as demonstrated in Fig. 5. With this enhancement, the DSP utilization and throughput of the FPGA system are dramatically improved. In our design, we organize the DSPs into 8 clusters due to the limited amount of DSPs in our FPGA board.

4.4 Extension to other types of layers

In addition to convolution layers, fully-connected (FC) layers are essentially computed through matrix multiplications. Therefore, the techniques previously discussed can be also employed to FC layers. ReLU layers and Max Pooling layers are easily implemented by accompanying comparators to the output buffers.

5 DESIGN SPACE EXPLORATION

5.1 Model Analysis

A detailed study of the complexity of Winograd convolution is conducted in the following subsections, it helps us to design an optimized accelerator for both dense and sparse cases.

5.1.1 Data Layout of Winograd transform. As previously mentioned, the input feature maps are fed in system in real-time. It’s not convenient to prune them during the inference, and it will increase the difficulty in system design. Moreover, the multiplication of a sparse matrix with a dense one does not necessarily produce another sparse matrix. In such case, our analysis keeps the same characteristics of feature maps for both dense and sparse cases. The volume of $i^{th}$ Winograd convolution layer $D_{wi}$, the volume of corresponding Winograd weights $D_{wki}$ (without pruning), and the volume of the results $D_{wo}$ before the inverse Winograd transform.
can be computed as
\[
D_{wi}^i = \left[ \frac{H}{m} \right] \times \left[ \frac{W}{m} \right] \times C \times l^2 \approx \left( \frac{1}{m} \right)^2 \times H \times W \times C \tag{6}
\]
\[
D_{wo}^i = \left[ \frac{H}{m} \right] \times \left[ \frac{W}{m} \right] \times K \times l^2 \approx \left( \frac{1}{m} \right)^2 \times H \times W \times K \tag{7}
\]
\[
D_{wk}^i = C \times K \times l^2 \tag{8}
\]
The Winograd transform dilates both the input feature maps and weights by a scale factor of \( \left( \frac{1}{m} \right)^2 \), e.g., when \( m \) takes value of 2 and \( r \) of 3, the transformed feature maps and weights require not only the access speed, but also causes more energy consumption.

5.1.2 Arithmetic complexity. The arithmetic complexity greatly depends on the data layout since the volume of feature maps and weights decides how much data does the algorithm needs to process. The number of multiplications performed by Winograd convolution layer \( i \) is
\[
M_{w}^i = \left[ \frac{H}{m} \right] \cdot \left[ \frac{W}{m} \right] \cdot C \cdot K \cdot l^2 \approx H \cdot W \cdot C \cdot K \cdot \left( \frac{1}{m} \right)^2
\]
The number of additions involved in matrix multiplications is
\[
S_{W}^i = \left[ \frac{H}{m} \right] \cdot \left[ \frac{W}{m} \right] \cdot (C - 1) \cdot K \cdot l^2 \approx H \cdot W \cdot (C - 1) \cdot K \cdot \left( \frac{1}{m} \right)^2
\]
The number of additions required by Winograd transforms are \( S_{B} \) and \( S_{A} \) for \( B^{T}dB \) and \( \bar{A}^{T} \left[ M_{k,x,g} \right] A \) respectively. In most cases, Winograd transform matrices \( B \) and \( A \) are sparse, therefore, (9) and (10) utilize the operator \( \text{nnz}() \) (number of nonzeros).
\[
S_{B}^i = 2 \times \left[ \frac{H}{m} \right] \times \left[ \frac{W}{m} \right] \times C \times K \times l \times [\text{nnz} (B) - l] \tag{9}
\]
\[
S_{A}^i = 2 \times \left[ \frac{H}{m} \right] \times \left[ \frac{W}{m} \right] \times C \times K \times l \times [\text{nnz} (A) - m] \tag{10}
\]
The Winograd weights are pre-calculated and stored in memory, so the overhead of computing Winograd weights has not been taken into account.

Since the degree of parallelism and the memory access patterns are dynamic, these uncertain factors hinder accurate estimation of optimal computation time in an obvious mathematical analysis. Therefore, we focus on the analysis of achieving the optimal power consumption as the reference.

As shown in Fig. 6, the energy consumption for local (e.g., buffers, FIFOs) and external memory accesses are several times and orders of magnitude higher than arithmetic operations, respectively [14]. Let us assume for the sake of simplicity that every storage element in both local and external memory is accessed exactly once, transformed feature maps are stored in local memory after Winograd transform, and the Winograd weights are read from external memory.

Let \( E_{me} \) and \( E_{ml} \) be the unit energies consumed by an access to the external memory and an access to the local memory, respectively. Let \( E_{mul} \) and \( E_{add} \) be the unit energies consumed by a multiplication operation and an addition operation, respectively. Then the total energy consumption of layer \( i \) is
\[
E_{tot}^i = E_{ml} \cdot (D_{wo}^i + D_{wi}^i) + E_{me} \cdot D_{wk}^i + E_{mul} \cdot M_{w}^i + E_{add} \cdot (S_{B}^i + S_{A}^i) + E_{add} \cdot (S_{B}^i + S_{A}^i)
\]

Another fact derived by eq. (6) and (8) is that greater \( m \) generates less elements of the transformed feature maps but more elements of the transformed weights. This fact indicates that the pruning of Winograd weights is more efficient with greater \( m \).

After having given the above formulas and summarizations, we conduct the analysis and experiments in section 6.2.

Table 1: number of parameters in each convolution layer of different stages in \( VGG \) [19] after Winograd transform (m=2)

| Stage [19] | # of Winograd neurons | # of Winograd weights |
|------------|-----------------------|-----------------------|
| Conv1 (x2) | 12,845,056            | 65,536                |
| Conv2 (x3) | 6,422,528             | 262,144               |
| Conv3 (x4) | 3,211,264             | 1,048,576             |
| Conv4 (x4) | 1,605,632             | 4,194,304             |
| Conv5 (x4) | 401,408               | 4,194,304             |
| Conv6      | 131,072               | 4,194,304             |

6 EXPERIMENTAL EVALUATION

\( VGG \) [19] is one of the most popular and mature deep learning models which has been widely used in research and industry. In this work, we use \( VGG \)16 for our analysis and experiments.

6.1 Experiment Setup

For the CNN model part, we set the input feature map size to \( 224 \times 224 \times 3 \), which are standard input dimensions for \( VGG \) pipeline.

Table 1 shows the number of neurons and weights of each layer in different stages after the Winograd transform. For the hardware part, we evaluate our design on an FPGA board, Xilinx Virtex UltraScale XCVU095. Although it is not fabricated with the latest technologies, and equips only with a medium amount of DSPs (768 DSPs), this configuration reveals better the performance gain.
than the latest FPGAs since optimizations for FPGAs with scarce computation power is more representative.

![Image](image.png)

**Figure 7**: Energy consumption estimation and latency of Winograd convolution

### 6.2 Experiment on energy consumption analysis

Fig. 7 (d) plots the trend when different $m$ is applied. The simulations run by synthesis tools show that the design with small values of $m$ normally consume less energy. In order to simplify our design, we decide to use $m = 2$, which eventually affects the dimension of our systolic arrays, tiling size, memory access patterns of our accelerator design, and so on. Although the plot indicates that $m = 4$ might be the optimal value for the energy consumption, we are limited by other hardware resources in our FPGA system, but the situation might be different if designing with a different FPGA system. In Fig. 7 (b) we provide the latencies for the inference by VGG with different configuration of $m$ and sparsity ranging from 60% to 90%. For the best case, we achieve almost 5× speedup.

### 6.3 Results and analysis

With $m = 2$, we get the synthesized result with the resource usage as shown in Table 3. The end-to-end comparison with the state-of-art CNN FPGA accelerators is listed in Table 2. We achieve the highest DSP usage and power efficiency. Due to time limitations, we only test our design on a medium scale FPGA. In current design, we use four $4 \times 4$ systolic arrays as one cluster for one matrix multiplication, and stack 8 such clusters for eight matrix multiplications in parallel. Meanwhile, 16 $4 \times 4$ systolic arrays work on the Winograd transform. In total, all 768 PEs are used. We will try to transfer our design to the latest and most powerful FPGA board in the future, and the performance will be improved further.

### 7 CONCLUSION

In this paper we propose a design with highly efficient recursive memory access layout for both dense and sparse Winograd convolutions, unified systolic arrays for both Winograd transforms and matrix multiplications, and a three dimensional compute engine for Winograd convolution. We also provide a comprehensive algorithmic level analysis for the performance model of the Winograd convolution. We achieve high computation power usage and high power efficiency in our design. There are several aspects that we can investigate further in the future. In particular, the automation design flow will help a lot to reduce the burden of development.

And, the progress in memory technology is also a promising solution as more and more new FPGA architecture incorporate such kind of brilliant concept.

### REFERENCES

[1] Utku et al. Aydonat. 2017. An OpenCL™Deep Learning Accelerator on Arria 10. In FPGA ’17. ACM, New York, NY, USA, 55–64.

[2] Y. Choi, M. El-Khamy, and J. Lee. 2018. Compression of Deep Convolutional Neural Networks under Joint Sparsity Constraints. ArXiv e-prints (May 2018), arXiv:cs.CV/1805.08303

[3] Jason Cong and Jie Wang. 2018. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In ICCAD ’18. ACM.

[4] Prasanna Deppa and C. Vasanthanayaki. 2012. FPGA based efficient on-chip memory for image processing algorithms. Microelectronics Journal 43, 11 (2012), 916 – 928.

[5] C. Farabet et al. 2011. NeuFlow: A runtime reconfigurable dataflow processor for vision. In CVPR 2011 WORKSHOPS. 109–116.

[6] C. Zhang et al. 2015. Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks. In FPGA ’15. ACM, New York, NY, USA, 161–170.

[7] C. Zhang et al. 2016. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In ICCAD 16. 1–8.

[8] Matteo Frigo et al. 2012. Cache-Oblivious Algorithms. ACM Trans. Algorithms 8, 1, Article 4 (Jan 2012), 22 pages.

[9] Naveen Suda et al. 2016. Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks. In FPGA ’16. ACM, New York, NY, USA, 16–25.

[10] Q. Zhang et al. 2018. Interpreting CNN knowledge via an Explanatory Graph. AAAI ’18 (2018).

[11] R. DiCiccio et al. 2016. Caffeinated FPGAs: FPGA framework For Convolutional Neural Networks. In FPT ’16. 265–268.

[12] S. Chatterjee et al. 2002. Recursive array layouts and fast matrix multiplication. JCSS 63, 11 (Nov 2002), 1105–1123.

[13] S. Han et al. 2016. Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding. ICLR (2016).

[14] Vivienne Sze et al. 2017. Hardware for machine learning: Challenges and opportunities. 2018 IEEE Custom Integrated Circuits Conference (CICC) (2017). 1–8

[15] X. Wei et al. 2017. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In DAC ’17. ACM, New York, NY, USA, Article 29, 6 pages.

[16] Xilinx Inc. 2015. UltraScale Architecture. (June 2015). https://www.xilinx.com/products/technology/ultrascale.html

[17] Andrew Lavin. 2016. Fast Algorithms for Convolutional Neural Networks. (2016), 4013–4021.

[18] Wei et al. Liu. 2016. SSD Single Shot MultiBox Detector. In ECCV ’16.

[19] K. Simonyan and A. Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. CoRR abs/1409.1556 (2014).

[20] Shmuel Winograd. 1980. Arithmetic complexity of computations. CBMS-NSF Regional Conference Series in Applied Mathematics, Vol. 33. Society for Industrial and Applied Mathematics, Philadelphia.
### Table 2: Comparison with State-of-the-art implementations

| Impl. | FPGA V7 VX485T | Xilinx VC709 | Stratix-V GSD8 | Arria10 GT1150 | our impl. |
|-------|----------------|--------------|----------------|----------------|------------|
| Precision | 32 bit float | 16 bit fixed | 8-16 bit fixed | 32 bit float | 8-16 bit fixed |
| Frequency (MHz) | 100 | 200 | 120 | 221.65 | 231.85 | 150 |
| Throughput (Gops/s) | 61.6 | 354 | 47.5 | 460.5 | 1171.3 | 460.8/230.4 |
| DSP utilization | 1120/1400 | 2833/3632 | 727/1963 | 1340/1523 | 1500/3046 | 921.6 (projected, 8 bit fixed sparse) |
| Power efficiency (Gops/s/W) | 3.31 | 14.22 | 1.84 | 25.78 | 55.9 |

### Table 3: Resource usage

| Resources | LUTs | FF | BRAM | DSP |
|-----------|------|----|------|-----|
| Used      | 241,202 | 634,136 | 1,480 | 512 (arith.) + 256 (wino.) |
| Available[16] | 537,600 | 1,057,200 | 1,728 | 768 |
| Percentage | 44.9% | 60.8% | 85.6% | 67% + 33% = 100% |