Towards a Uniform Template-based Architecture for Accelerating 2D and 3D CNNs on FPGA

Junzhong Shen, You Huang, Zelong Wang, Yuran Qiao, Mei Wen, Chunyuan Zhang
National University of Defense Technology
College of Computer, National Key Laboratory for Parallel and Distributed Processing
Changsha, Hunan, China
{shenjunzhong,youhuang,wangzelong15,qiaoyuran,meiwen,cyzhang}@nudt.edu.cn

ABSTRACT

Three-dimensional convolutional neural networks (3D CNNs) are used efficiently in many computer vision applications. Most previous work in this area has concentrated only on designing and optimizing accelerators for 2D CNN, with few attempts made to accelerate 3D CNN on FPGA. We find accelerating 3D CNNs on FPGA to be challenge due to their high computational complexity and storage demands. More importantly, although the computation patterns of 2D and 3D CNNs are analogous, the conventional approaches adopted for accelerating 2D CNNs may be unfit for 3D CNN acceleration. In this paper, in order to accelerate 2D and 3D CNNs using a uniform framework, we propose a uniform template-based architecture that uses templates based on the Winograd algorithm to ensure fast development of 2D and 3D CNN accelerators. Furthermore, we also develop a uniform analytical model to facilitate efficient design space explorations of 2D and 3D CNN accelerators based on our architecture. Finally, we demonstrate the effectiveness of the template-based architecture by implementing accelerators for real-life 2D and 3D CNNs (VGG16 and C3D) on multiple FPGA platforms. On S2C VUS440, we achieve up to 1.13 TOPS and 1.11 TOPS under low resource utilization for VGG16 and C3D, respectively. End-to-end comparisons with CPU and GPU solutions demonstrate that our implementation of C3D achieves gains of up to 13x and 60x in performance and energy relative to a CPU solution, and a 6.4x energy efficiency gain over a GPU solution.

CCS CONCEPTS

• **Computer systems organization** → *Special purpose systems*;

KEYWORDS

3D CNN; Winograd Algorithm; Uniform Templates

ACM Reference Format:
Junzhong Shen, You Huang, Zelong Wang, Yuran Qiao, Mei Wen, Chunyuan Zhang. 2018. Towards a Uniform Template-based Architecture for Accelerating 2D and 3D CNNs on FPGA. In *Proceedings of 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2018).* ACM, New York, NY, USA, 10 pages. https://doi.org/http://dx.doi.org/10.1145/3174243.3174257

FPGA 2018, February 25–27, 2018, Monterey, CA, USA

ACM ISBN ISBN 978-1-4503-5614-5/18/02…$15.00
https://doi.org/http://dx.doi.org/10.1145/3174243.3174257

1 INTRODUCTION

Convolutional Neural Networks (CNNs) have been implemented with great success in the field of computer vision. However, the improvements in accuracy provided by CNNs have exacerbated the computational complexity of the computational layers. Since general-purpose CPUs have failed to provide the massive computational parallelism required by modern CNNs, many hardware accelerators (such as GPUs [8], ASICs [6] and FPGAs [2, 14, 21]) have been developed to boost CNN performance. Of these platforms, FPGAs have become a particularly attractive option due to their reconfigurability and abundant logic resources. Moreover, the availability of commercial high-level synthesis (HLS) tools greatly reduces both the programming difficulty and development time required for FPGA accelerators, meaning that FPGA-based solutions have become more popular.

Abundant studies [2, 12, 14, 20–23] have focused on accelerating 2D CNNs on FPGAs, while the accelerations of three-dimensional convolutional neural networks (3D CNNs) on FPGA are not as well-researched. However, 3D CNNs have proven to be very effective in many complicated computer vision tasks including video classification [17], human action recognition [5] and medical image analysis [7].

Our studies reveal that while the computation patterns of 2D and 3D CNNs are very similar, 3D CNNs have higher computational complexity and greater memory bandwidth demands. In addition, the design space for 3D CNN acceleration has been further expanded, making it more difficult to determine the optimal solution. More importantly, widely studied approaches in the 2D CNN field may not be a good fit for 3D CNNs. For instance, the ordinary convolutional approach adopted in [14, 20] will cause higher computation complexity when adopted in a 3D CNN acceleration application; moreover, our studies suggest that the approach of unrolling the convolution to matrix multiplication [16] could introduce high degree of data replication. Therefore, designing an efficient 3D CNN accelerator on FPGA demands more intensive design efforts than have been applied in previous works.

Motivated by the insight that the computation patterns of 2D and 3D CNNs are very similar, we here attempt to unify the 2D and 3D CNNs into a single acceleration framework rather than designing an accelerator for a given 2D or 3D CNN. In other words, we aim to use uniform templates as building-blocks to support the accelerations of both 2D and 3D CNNs. This template-based methodology, which was also adopted in [13], has a number of appealing advantages: 1.

it can make good use of the reconfigurability of FPGAs by building accelerators for complex 2D and 3D CNNs in a short period of time; 2. the scalability of the template-based design enables the generation of accelerators that match the CNN's specific need, as well as the resource constraints of the FPGA platform; 3. use of the templates makes it easy to exploit the fine-grained parallelism of CNN algorithms, which in turn contributes to high-throughput solutions.

Based on this idea, we propose a template-based design methodology for accelerating 2D and 3D CNNs on FPGA in this work. Due to its ability to reduce the complexity of convolutions, as well as its good extensibility, we first select the Winograd algorithm as the common approach to the computation of the convolutional layers in 2D and 3D CNNs. We then design uniform templates based on the common operations extracted from the 2D and 3D Winograd algorithms; these templates constitute a hierarchical architecture capable of exploiting all sources of parallelism in CNNs. Finally, we present detailed optimization strategies for improving both on-chip and off-chip memory bandwidth.

Previous design space exploration schemes [16, 20, 21] have been applicable only to 2D CNN accelerators, making them unsuitable for our architecture. Since the sizes of the design space of 2D and 3D CNN accelerators are different, determining the optimal design options for both by using two different design space exploration schemes will inevitably be costly. To resolve this issue, we propose a uniform analytical model to efficiently explore the design space of both 2D and 3D CNN accelerators. We conduct an in-depth analysis of the performance of the proposed architecture by taking the computation capacity of the computation engine and memory bandwidth into consideration. The main contributions of this work are summarized as follows:

(1) We propose a uniform template-based architecture for accelerating 2D and 3D CNNs on FPGA. With the help of uniform templates, which are based on the Winograd algorithm, we are able to build accelerators for 2D and 3D CNNs in only a short time.
(2) We develop a uniform analytical model for efficient design space explorations of both 2D and 3D CNN accelerators.
(3) As a case study, we implement two accelerators for state-of-the-art 2D and 3D CNNs: VGG16 and C3D, respectively. Experimental results show that our implementation of VGG achieves comparable performance with recent 2D CNN accelerators under conditions of low resource utilization. Furthermore, the C3D implementation achieves up to 13x and 60x gains in performance and energy over the CPU solution, as well as a 6.4x energy efficiency gain over the GPU solution.

The rest of this paper is organized as follows: Section 2 provides a background for 3D CNN and the Winograd algorithm. Section 3 presents the details of the proposed template-based architecture. Section 4 discusses the design space exploration. Section 5 shows our experimental results. Section 6 discusses related work and Section 7 concludes this paper.

2 BACKGROUND

The computation pattern of the CONV layers in 3D CNN is more complicated than that of the CONV layers in 2D CNN, which can

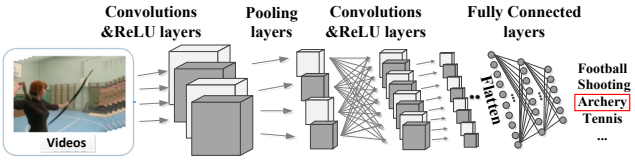


Figure 1: A real-life 3D CNN model for video classification.

Table 1: Analysis of C3D

Layers	Ops (GFLOPS)	Data Transfer(MB)		Time (ms)
		In+Out	Weights	
CONV	38.4(99.9%)	99.0(27.7%)	17.7(26.7%)	31.9(97.3%)
ReLU	0.0(0%)	96.7(27.1%)	0.0(0.0%)	0.0(2.3%)
Pool	0.0(0%)	161.0(45.1%)	0.0(0.0%)	0.0(0.2%)
FC	0.0(0.1%)	0.1(0.0%)	48.8(73.3%)	0.7(0.2%)

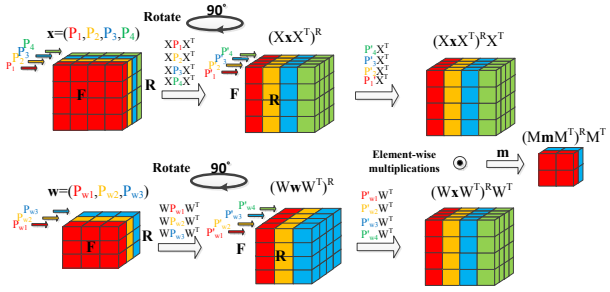


Figure 2: The process of the 3D Winograd algorithm.

be described as follows:

$$Out[m][z][r][c] = \sum_{n=0}^{N} \sum_{k=0}^{K_d} \sum_{i=0}^{K_r} \sum_{j=0}^{K_c} W[m][n][k][i][j] * In[n][S*z+k][S*r+i][S*c+j], \quad (1)$$

where In/Out and W define the three-dimensional input/output feature maps and filters, respectively. In each layer, a set of N input feature maps of size $D \times H \times W$ are convolved by M sets of $N \times Ksize$ filters ($Ksize = K_d * K_r * K_c$), yielding M output feature maps of size $Z \times R \times C$. In each of M sets, N filters slide across the corresponding input feature maps with a stride of S. It can be seen that the computational complexity of the CONV layers in 3D CNN is much higher than that of the CONV layers in 2D CNN (i.e. $2*M*N*R*C*K_r*K_c$ for 2D CNN and $2*M*N*Z*R*C*K_d*K_r*K_c$ for 3D CNN).

In addition, the computation pattern of the FC layers in 3D CNN is a dense matrix-vector multiplication, which is identical to that of the FC layers in 2D CNN. In this paper, our template-based design is targeted at the CONV and FC layers. Although the Activation and POOL layers are also supported by our proposed architecture, we omit the discussions of these layers due to their relative simplicity.

2.1 In-depth Analysis of 3D CNN

Previous studies [4, 8, 15] have demonstrated that the convolutional operations take up over 90% of computation time. Our study draws a similar conclusion with regards to 3D CNN. To better illustrate this, the profiling results of C3D are presented in Table 1 (here, the execution time is measured on a NVIDIA Tesla K40 GPU). Three observations can be drawn from Table 1: firstly, similar to 2D CNN, the CONV layers in 3D CNN are computationally-intensive and

occupy the majority of the computation time (over 97%). Secondly, the FC layers are memory-intensive, since they require 73.3% of the weights but only occupy 0.1% of the computation cost. Thirdly, the amount of intermediate data that needs to be transferred between the layers is far larger than the weights (356.7 MB vs. 66.5 MB).

It can be seen that although C3D is relatively small (66.5 MB weights), it contains a larger number of computations (38.5 GOPs) than recent large-scale 2D CNN networks such as AlexNet [8] (~240 MB weights, 1.46 GOP/image) and VGG-16 [15] (~500 MB weights, 30.9 GOPs/image). This is mainly due to the fact that the sizes of both input and output data are extended in 3D CNN. More importantly, the performance of the computation-centric CONV layers may become more sensitive to the memory bandwidth due to the huge intermediate data transfer between layers.

2.2 Winograd Algorithm Extension

The Winograd algorithm, which was proposed in [19] to reduce the arithmetic complexity of the convolutional operation, can be applied to 1D, 2D and even higher-dimensional convolutions. To simplify the discussion, we begin with an illustration of the 1D Winograd algorithm. We denote a 1D convolution as $F(m, r)$, which has an ordinary computation pattern that is given by:

$$\mathbf{y}_i = \sum_{j=0}^{r-1} \mathbf{x}_{i+j}\mathbf{w}_j, i = 0, 1, \cdots m-1. \tag{2}$$

Here, $\mathbf{x}$ is an 1D image data of size n ($n = m + r - 1$), and $\mathbf{w}$ is a filter of size r. To be more specific, we can consider the Winograd algorithm for $F(2, 3)$, which can be represented as follows:

$$\mathbf{y} = M[(X\mathbf{x}) \odot (W\mathbf{w})], \tag{3}$$

where $\odot$ is denoted as element-wise multiplication, and M, X, W are transformation matrices with constant values:

$$X = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix}, W = \begin{pmatrix} 1 & 0 & 0 \\ \frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 1 \end{pmatrix},$$
$$M = \begin{pmatrix} 1 & 1 & 1 & 0 \\ 0 & 1 & -1 & -1 \end{pmatrix} \tag{4}$$

The transformation matrices are determined for a given m and r. Note that the proof of equivalence of Equation 2 and Equation 3 is provided in [19]. The simplicity of the transformation matrices contributes to reducing the overall arithmetic complexity.

We now further consider the 2D Winograd algorithm for $F(m \times m, r \times r)$, which is given by the following equation:

$$Y = M[(X\mathbf{x}X^T) \odot (W\mathbf{w}W^T)]M^T \tag{5}$$

Here, $\mathbf{x}$ is redefined as an $n \times n$ image tile, while $\mathbf{w}$ is an $r \times r$ filter. If we regard $\mathbf{x}$ as having n column vectors of size n, then $X\mathbf{x}$ can be calculated by left multiplying all column vectors of $\mathbf{x}$ by the transformation matrix X. Similarly, $X\mathbf{x}X^T$ can be obtained by right multiplying all row vectors of $X\mathbf{x}$ by X^T, where $X\mathbf{x}$ is regarded as n row vectors of size n. When we take a similar approach to the rest of the transformation procedures in Equation 5, it emerges that the 2D Winograd algorithm can be nested by the 1D Winograd algorithm [10].

In [9], Lan et al. propose a nested technique for the 3D convolution $F(m \times m \times m, r \times r \times r)$, i.e. the 3D Winograd algorithm. In this work, we signify this algorithm using the following equation:

$$Z = (M((X\mathbf{x}X^T)^R X^T \odot (W\mathbf{w}W^T)^R W^T)M^T)^R M^T, \tag{6}$$

where $\mathbf{x}$ and $\mathbf{w}$ are extended to an $m \times m \times m$ image and $r \times r \times r$ filter, respectively. R represents the operation of rotating the transformed image or filter tiles 90 degrees clockwise. We illustrate the process of the 3D Winograd algorithm in Figure 2. Here, it can be seen that the transformations on the 3D input data can be performed by transformations on multiple 2D data plains (e.g. P_1, P_2, P_3, P_4). However, unlike the 2D Winograd algorithm, the transformation procedures employed in the 3D Winograd algorithm are asymmetric; for instance, the algorithm only applies row transformations (right multiply X^T) on the rotated image. Therefore, the 3D Winograd algorithm is the combination of the 1D and 2D Winograd algorithms. More importantly, since the 2D Winograd transformation procedure can be further broken up into 1D transformation procedures, the 3D Winograd algorithm can also be nested by the 1D Winograd algorithm. This demonstrates that the Winograd algorithm has good extensibility.

3 TEMPLATE-BASED ACCELERATOR DESIGN

We first define the key features of the uniform templates for 2D and 3D CNNs, as follows: 1. the templates can be regarded as common computational building blocks for 2D and 3D CNNs, allowing us to generate CNN accelerators with different configurations; 2. the templates can exploit the fine-grained parallelism inherent in the computation components of CNN algorithms; 3. the templates are resource-saving, in that they can be developed quickly.

3.1 Algorithms

Since CONV layers have the highest computational cost in both 2D and 3D CNNs, we primarily aim to design templates for computation of the CONV layers in this work. Our first concern is to determine the best algorithm for the uniform template-based design. We require this algorithm to be extensible, meaning that the algorithm for 2D CNN can be easily extended to accelerate 3D CNNs with minimal new operations; in this way, the computation procedures for both 2D and 3D CNNs can share a large number of common operations. In addition, the computational complexity and storage requirements of the selected algorithm must not be too high when it is adopted in 2D and 3D CNNs.

The most commonly adopted approach for computation of the CONV layers in 2D CNNs is the convolutional Matrix-Multiplication (MM), which is illustrated in Equation 1. Owing to the high computational complexity of this approach, many studies parallelize the kernel computation by combining different UNROLL strategies [12]. While the algorithm is extensible, such that the 3D convolution can be transformed into multiple 2D convolutions, the computational complexity of this approach grows cubically when it is adopted in 3D CNN; as a result, the optimization strategies employed need to become more complex if the optimal design is to be achieved.

An alternative approach is to map the convolution to matrix multiplication [16], which is also widely used in many deep-learning software frameworks [20]. However, it should be noted that the

Table 2: Comparisons of the ordinary convolution and Winograd algorithms

	$F(2,3)$		$F(2^2,3^2)$		$F(2^3,3^3)$	
Algorithms	muls	adds	muls	adds	muls	adds
Ordinary	6	4	36	32	216	208
Winograd	4	11	16	77	64	419

mapping procedure will introduce data replication for the input feature maps. Zhang et al. [20] reveal that the input feature maps in AlexNet require 25x more data when adopting this approach for a CONV layer computation. According to our study, the data replication worsens in 3D CNN applications. Theoretically, the data replication ratio of the input feature maps is $\frac{Size_{filter}*Size_{Out}}{Size_{In}}$, where $Size_{In}$, $Size_{out}$ and $Size_{filter}$ are the sizes of the input/output feature maps and filters of a given CONV layer, respectively. Therefore, using this approach for 3D CNN acceleration may stress the memory system, potentially leading to a bottleneck. More importantly, it is difficult to implement the mapping procedures in 2D and 3D CNN by means of uniform templates.

Owing to its advantageous ability to reduce the complexity of convolutions, the Winograd algorithm has attracted increasing attention in the field of CNN acceleration in recent research [2, 9, 10]. Table 2 presents the comparisons of the ordinary convolution (i.e. convolutional MM) and the Winograd algorithm with respect to the the number of arithmetic operations, demonstrating that the Winograd algorithm can efficiently reduce the number of multiplications in convolutions. While the Winograd algorithm does also increase the number of additions, the computational complexity is reduced overall. Additionally, the discussions in Section 2.2 demonstrate the good extensibility of the Winograd algorithm, which is beneficial for developing uniform templates to implement both 2D and 3D algorithms. Furthermore, use of the Winograd algorithm will not introduce any data replication. In light of the above, we here select the Winograd algorithm as the common approach for computing the CONV layers in 2D and 3D CNNs. Note that we use $F(m,r)$,$F(m^2,r^2)$ and $F(m^3,r^3)$ to represent the 1D, 2D and 3D Winograd algorithms for the rest of this paper.

3.2 Extracting Common Operations

Since uniform templates are to be used for performing the common operations of the 2D and 3D CNNs, a key step is extraction of the common operations from the procedures of these CNNs. We first apply the Winograd algorithms to the computation of the CONV layers. As shown in Figures 3(a) and 3(b), the major common operations of the algorithms can be found in the transformations, element-wise multiplications and accumulations, which are marked using dashed boxes. As discussed in Section 2.2, both the 2D and 3D Winograd algorithms can be nested by the 1D Winograd algorithm. Therefore, the 1D Winograd transformations can be used as the common operations of the 2D and 3D transformation procedures. Additionally, due to the operator-level parallelism of the element-wise multiplications, these element-wise multiplications on the 3D tiles can be performed repeatedly by multiplications on the 2D tiles. Therefore, the element-wise multiplication of a 2D input tile and a 2D filter tile can also be considered a common operation of the two algorithms. Similarly, since the accumulations of the elements

```
L1: for(row=0;row<R; row+=2){
L2:  for(col=0;col<C; col+=2){
L3:   for(m=0; m<M; m++){
L4:    for(n=0; n<N; n++){
         Load(In, n);  //4*4 input  tile
         Load(W, m ,n); //3*3 filter tile
         2D_Trans_X(In, Tin);
         2D_Trans_W(W, Tw);
         2D_Mul(Tin,Tw, P);
         2D_Trans_M(P, Tp);
         2D_Accumulate(Sum,Tp);
    }
     Send(Out,Sum,m); //2*2 output tile
}}}
```

(a) CONV layers of 2D CNN

```
L1:for(dep=0;dep<Z;dep+=2){
L2: for(row=0;row<R; row+=2){
L3:  for(col=0;col<C; col+=2){
L4:   for(m=0; m<M; m++){
L5:    for(n=0; n<N; n++){
         Load(In, n);  //4*4*4 input  tile
         Load(W, m ,n); //3*3*3 filter tile
         2D_Trans_X(In, Tin);
         2D_Trans_W(W, Tw);
         Rotate(Tin, Tin^R);
         Rotate(Tw, Tw^R);
         1D_Trans_X(Tin^R, Tin^1);
         1D_Trans_W(Tw^R, Tw^1);
         3D_Mul(Tin^1,Tw^1, P);
         2D_Trans_M(P, Tp);
         Rotate(Tp, Tp^R);
         1D_Trans_M(Tp^R, Tp^1);
         3D_Accumulate(Tp^1,Sum);
    }
     Send(Out, Sum,m); //2*2*2 output tile
}}}}
```

(b) CONV layers of 3D CNN

Figure 3: Simplified pseudocode of CONV layers in 2D and 3D CNNs with Winograd algorithms.

in one intermediate result tile are independent of each other, the accumulations of a 3D tile can be performed by the accumulations of the 2D tiles that have split from the 3D tile. Therefore, the accumulation of the transformed 2D result tile across the input channels is the common operation of the algorithms.

3.3 Template Design

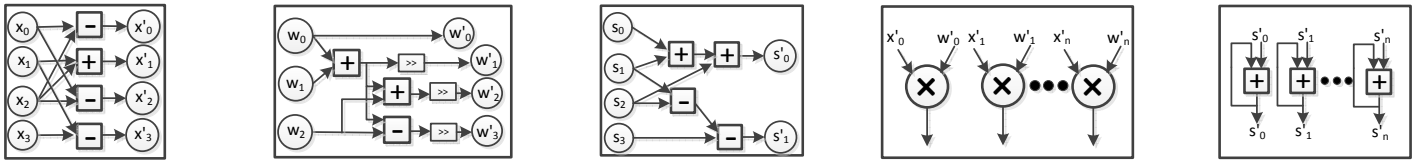
Figure 4 shows the proposed templates based on the extracted common operations, including three kinds of transformation templates (TX, TW and TM), the element-wise multiplication template and the accumulation template.

Transformation templates. The 1D Winograd transformation procedure is essentially the matrix-vector multiplications. We observe that the transformation matrices in Equation 4 contain many 1s and -1s, meaning that we can replace the multiplications here with additions or subtractions. In addition, we can utilize the sparsity of the transformation matrices to reduce the number of operations. Moreover, special multiplications/divisions (e.g. $*2$, $*\frac{1}{4}$) can be replaced by shifting operations, thus further reducing the computational complexity. Figures 4(a), (b) and (c) show the transformation templates for $F(2,3)$. It can be seen that no multipliers or dividers are required in the templates.

Element-wise multiplication template. This template is responsible for multiplying a transformed input tile with its corresponding filter tile. As shown in Figure 4(d), the template consists of a group of dot-product units that can perform multiplications independently. The maximum degrees of parallelization of the element-wise multiplications in $F(m^2,r^2)$ and $F(m^3,r^3)$ are n^2 and n^3, respectively. Therefore, n^2 dot-product units are integrated in the element-wise multiplication template to maximize throughput and minimize latency.

Accumulation template. As shown in Figure 4(e), the accumulation template offers a certain number of accumulators to iteratively sum up the intermediate transformed products. Much like the element-wise multiplication template, we integrate m^2 (i.e. the maximum degrees of parallelization of accumulations in $F(m^2,r^2)$) accumulators to achieve the highest throughput and lowest latency.

It should be noted here that, due to their simplicity, all proposed templates can be implemented easily by means of the HLS tool, which contributes to the rapid development of the accelerators.



(a) Transform template (X) (b) Transform template (W) (c) Transform template (M) (d) Multiplication template (e) Accumulation template

Figure 4: Proposed templates for 2D and 3D CNNs ($F(2,3)$).

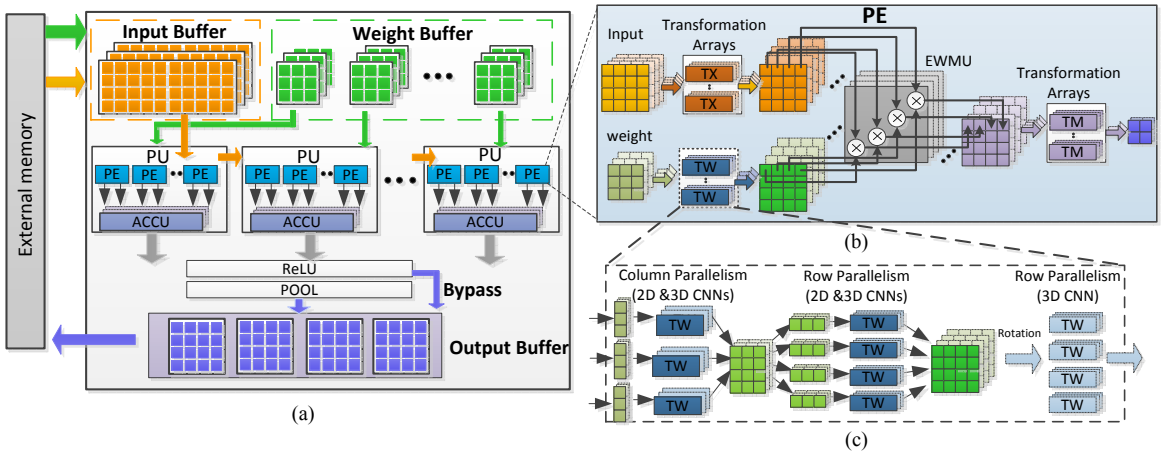


Figure 5: (a) Overview of the template-based architecture; (b) Processing element; (c) Architecture of the transformation arrays.

3.4 Template-based Architecture

```
Computation_Engine<To,Ti>(Tz,Tr,Tc):
L1:for(tz=0;tz < Tz;tz+=2){//loop flattened (only for 3D CNN)
 L2: for(tr=0;tr < Tr;tr+=2){//loop flattened
  L3:  for(tc=0;tc < Tc;tc+=2){
#Pragma HLS PIPELINE
  Load(In,Bin); //load input tiles Bin from In[Ti][Td*Th*Tw]
  Load(W,Bw); //load filter tiles from W[To][Ti][Ksize]
  L4:  for(to=0;to < To; to++){
#Pragma HLS UNROLL
    PU(Bin,Bw,Bout,to,tz,tr,tc){  //parallelism in PUs
   L5:    for(ti=0;ti < Ti; ti++){
#Pragma HLS UNROLL
      PE(Bin,Bw,to,ti); //parallelism in PEs
}
Accumulations(Bout,to,ti); }}
store(Bout,Out);//store output tiles Bout to Out[To][Tz*Tr*Tc]
}}}
```

Listing 1: Simplified pseudocode of the computation engine.

Figure 5(a) presents an overview of the proposed template-based architecture. Due to the limited on-chip memory capacity of the FPGA platform, holding all input feature maps and weights in on-chip Block RAM (BRAM) would be unrealistic; consequently, we store both the initial data and the final results in the external memory. In addition, we identify data reuse opportunities in the feature maps, as each feature map is convolved many times by M different filters; therefore, caching more filters on-chip will be beneficial for input feature data reuse. However, it is infeasible to cache all filters on-chip, since the number of filters increases significantly as the network goes deeper. Therefore, we opt to apply the tiling method on both the input/output feature maps and the filters. As shown in Figure 5(a), we manage three kinds of buffers to store the tiled data. Note that the sizes of the tiled input/output feature maps and filters are $T_i * T_d * T_h * T_w$, $T_o * T_z * T_r * T_c$ and $T_o * T_i * Ksize$ (T_d and T_z equal to 1 for 2D CNN), where T_o and T_i are tiling factors of M and N, respectively; T_d,T_h and T_w are tiling factors for the input feature maps (i.e. T_d for D, T_h for H and T_w for W), and T_z, T_r and T_c are tiling factors of output feature maps (similarly, T_z for Z, T_r for R and T_c for C).

Computation Engine. The computation engine is the kernel component of the architecture. The scalable architecture implemented has two levels of hierarchy, namely the Processing Units (PUs) and a set of Processing Engines (PEs) in each PU. The proposed templates are integrated inside the PEs. Similar to [2, 14, 20], the computation engine can be used to accelerate both the CONV and FC layers (more details in Section 3.6). As can be seen, we also map the Activation (ReLU) and pooling layers (POOL) into the architecture. In particular, the network configuration allows the pooling layer to be optionally bypassed. Consequently, the entire CNN can be perfectly mapped to our proposed architecture.

Listing1 shows the pseudocode of the computation engine. As shown in the figure above, we unroll Loops $L4$ and $L5$ in Listing1 to explore two parallelisms: 1. the inter-output parallelism, by using T_o PUs to compute multiple output feature maps in parallel; 2. the inter-input parallelism, by integrating T_i PEs in each PU to process multiple input feature maps in parallel. In addition, we also pipeline loop $L3$ in Listing1 so that the processing of different tiles can be overlapped, which leads to the computation engine having high throughput. Moreover, we also flatten $L1$ and $L2$ in Listing1, thus minimizing the latency of the computation engine.

Processing Engine (PE). PEs are the fundamental computing units that perform the major procedure of the Winograd algorithm.

The function of a PE unit is to fetch an input tile and its corresponding filter tile, then yield an intermediate result tile to be accumulated in PU. As shown in Figure 5(b), the templates presented in Figure 4 are used to build the major components of the PEs, namely the transformation arrays (TX, TW and TM) and the $EWMU$ (Element-Wise Multiplication Unit). It can be seen that the 2D and 3D Winograd algorithms are perfectly mapped to the structure of the PE. Note that the dataflow of the 2D algorithm is depicted in lighter colors, while the dataflow of 3D algorithm is depicted in darker colors. It can be seen that both of the dataflows can be organized by the proposed templates.

Figure 5(c) illustrates the micro-architecture of the transformation arrays. In order to minimize the computational latency of the transformation procedure, multiple transformation templates are integrated to support both column (transformation) and row (transformation) parallelism. Consequently, the transformation arrays are perfectly pipelined, meaning that they can process a new tile in every cycle. For 3D CNN, extra templates (as identified by dashed blocks) are required to support transformations on multiple data plains as well as the rotated transformed tiles.

Processing Unit (PU). We manage multiple identical PUs to enable re-use of the input feature maps. This means that every time the process is run, all PUs share the same input feature tiles but fetch T_o different sets of filter tiles, yielding T_o result tiles that belong to T_o output feature maps. If all PUs are connected directly to the input feature buffer, our design may experience some difficulties in making the timing closure when the clock rate is high [18]. To solve this problem, we organize the PUs according to a systolic array architecture, as in [2, 18, 20]. As shown in Figure 5, only the leftmost PU is directly connected to the input buffers, while the other PUs are connected to the adjacent PUs. Every time a PU receives T_i input tiles from the preceding PU for processing, it also delivers the data to the adjacent PU intermediately. Local buffers are required in each PU so that the the input tiles may be cached. In order to overlap computation and data delivery, we manage double buffers in each PU. The result tiles generated by the local PEs are accumulated in the $ACCU$ module of each PU, which consists of multiple accumulation templates. The temporary sum generated by the $ACCU$ will then be read to add the newly generated results until all the input feature maps have been traversed.

3.5 Memory Access Optimization

As discussed in Section 2.1, the bandwidth sensitivity of the CONV layers in 3D CNNs is exacerbated due to the cubical growth of input/output data. In addition, the improvement in computation engine throughput demands higher on-chip memory bandwidth. Optimization strategies for both off-chip DRAM and on-chip memory access are provided in this subsection so that the memory bandwidth can be matched with the computational capacity of the computation engine.

3.5.1 Optimization for the external memory bandwidth. There are two main aspects to our strategies for optimizing external memory access: reducing the number of memory transfers and increasing the efficient memory bandwidth. As shown in Table 3, we observe that the selection of tiling factors affects the effective external memory bandwidth. It can be seen that using Strategy 1 will result

Table 3: Tiling strategies

	Tiling Strategies	# of Transfers*	Burst Length
1	$T_z <= Z, T_c < C, T_r <= R$	$T_d * T_h * \frac{W}{T_w}$	$\frac{T_w * Bw_{on}}{Bw_{off}}$
2	$T_z <= Z, T_c = C, T_r < R$	T_d	$\frac{T_h * W * Bw_{on}}{Bw_{off}}$
3	$T_z <= Z, T_c = C, T_r = R$	1	$\frac{T_d * H * W * Bw_{on}}{Bw_{off}}$

* $T_w = T_c * S - S + K$, $T_h = T_r * S - S + K$, $T_d = T_z * S - S + K$.

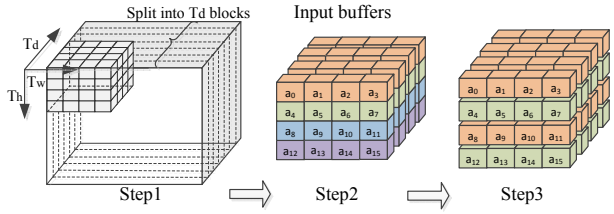


Figure 6: Step-by-step optimization for on-chip memory access.

in poor memory bandwidth efficiency, since this may result in a large number of memory transfers with short burst lengths; this is not favored by the external memory. Using Strategy 2 can result in higher effective bandwidth than Strategy 1, as it significantly increases the burst length and thereby reduces the number of memory transfers. Although Strategy 3 seems to be the best option, it requires a large amount of on-chip memory to store the data to be processed. Accordingly, we propose a flexible optimization strategy to tackle this issue. For convolutional layers where the size of the input feature maps is large, we select Strategy 2 to facilitate a good tradeoff between on-chip memory consumption and external memory bandwidth. In addition, we apply Strategy 3 only for those convolutional layers with small input feature maps.

From Table 3, it can be further concluded that the bit-widths of the external memory Bw_{off} and on-chip buffer Bw_{on} also affect the burst length. The work in [20] demonstrates that higher Bw_{off} can result in better peak bandwidth. In addition, according to [3], enlarging the data width of the on-chip buffer can lead to a larger data transfer width and thus a higher DRAM bandwidth. In this work, we increase both Bw_{on} and Bw_{off} to facilitate fast DRAM-BRAM transfer. More importantly, we also use multiple memory ports to load and store multiple input/output feature maps simultaneously.

3.5.2 Optimization for on-chip buffer access. In order to parallelize the executions of PEs in each PU, we first split the input feature buffers into T_i blocks. In this way, it is possible to read T_i input feature tiles for the corresponding T_i PEs independently. Similarly, the weight buffer is partitioned into T_o blocks to support parallelism among PUs. In a departure from prior works, we adopt the Winograd algorithm that each time a PE needs to read an input feature tile and a filter tile simultaneously. If all pixels of the input feature tile are stored in the same memory bank, the on-chip memory access conflicts can be very severe, thereby limiting the throughput of the computation pipeline. Moreover, fully splitting the memory blocks into register files is infeasible, as this may result in a great number of multiplexers which in turn cost a large number

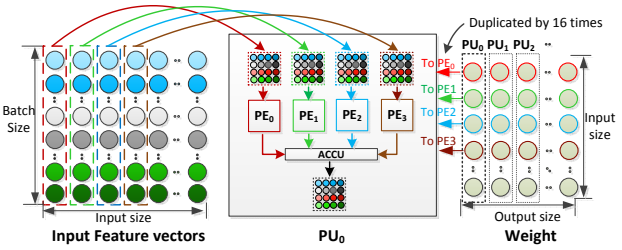


Figure 7: An illustration of FC layer mapping.

of LUTs (Loop-Up Tables). To resolve this issue, we propose a step-by-step optimization strategy that can significantly improve the on-chip memory bandwidth with moderate memory consumption. To promote better understanding, an example of this is shown in Figure 6 (For simplicity's sake, we assume $T_i = 1$ in this example).

Step 1: Each of the T_i input buffer blocks is further split into T_d partitions, as shown in Figure 6. In this way, data located at different depths of the input feature tile can be fetched simultaneously, which significantly reduces memory access conflicts by a factor of 4 (i.e. $m + r - 1$) for each PU. Note that this step can only be applied for 3D CNN.

Step 2: Further optimization is achieved by enlarging the width of the on-chip buffers. As the example shows, after this optimization method is applied, four consecutive data $a_i \sim a_{i+3}$ $(i = 0, 1, 2, 3)$ (marked as the same color) can be read simultaneously. However, it should be noted that data marked in different colors still cannot be fetched in parallel, suggesting substantial room for improvement.

Step 3: We utilize the "ARRAY_PARTITION" directive provided by the HLS tool to achieve higher bandwidth. The "cyclic" partition strategy is used in this work to split the original on-chip buffers into blocks of equal size, interleaving the elements. As the example shows, $a_0 \sim a_3$ and $a_4 \sim a_7$ are stored in different blocks that can be fetched simultaneously. Finally, we can simply use the dual-port BRAMs to implement the buffers for further improvement. Use of this method means that each buffer can handle two concurrent memory accesses. As in [21], we use double buffering in the input/output feature maps and weights to overlap data transfer time with computation, thereby further reducing the overall latency.

3.6 Fully Connected Layers

The proposed architecture can be reused to accelerate the FC layers. Since the computation pattern of the FC layers is inner-product, which mainly includes multiplications and additions, we can reuse the computation engine for FC layers by bypassing the transformation modules in PEs, such that only the *EWMU* and *ACCU* modules are used. Figure 7 depicts the mapping from an FC layer to the template-based architecture. In order to reuse the weights in FC layers thereby reducing data access, a batch of input feature vectors are organized as the input of the FC layers. The input feature tile fetched by each PE is made up of the pixels from the same location of different input feature vectors (i.e, data in the same dashed box in Figure 7 (left)). Therefore, the batch size is equal to the tile size. In addition, according to the computation pattern of the FC layers, the pixels belong to the same tile are multiplied by the same weight. As it turns out, each PE requires only a weight each time the process is run. In order to perform element-wise multiplications in each PE, each weight needs to be duplicated by 16 (i.e. the size of input tile, 16 for 2D CNN and 64 for 3D CNN) times before being sent to each PU. Subsequently, the *ACCU* in each PU accumulates the intermediate results from all local PEs. Once all the pixels of the input feature vectors are traversed, the PUs deliver the accumulated results to the output buffer. Since the size of the output feature vectors is small, these vectors are stored in on-chip buffers in order to reduce amount of the off-chip communication. Moreover, as the weights to be transferred are very large, double buffering is also adopted to overlap the computation with the memory access, similar to the process applied to the CONV layers.

4 ANALYTICAL MODEL FOR DESIGN SPACE EXPLORATION

In this section, we propose a uniform analytical model for the design space explorations of our proposed template-based architecture.

4.1 Computational Roof

To ensure uniform mathematical representation of the analytical model for both 2D and 3D CNN accelerators, we regard 2D CNN as a special form of 3D CNN with $Z = T_z = 1$. We first estimate the peak computation performance of our accelerator by calculating its computational roof (CR), which is given by:

$$Computational\ Roof = \frac{total\ computational\ operations}{execution\ cycles}. \quad (7)$$

Considering that the Winograd algorithm is essentially the fast algorithm for the convolution, the number of operations can be defined by Equation 8, which is identical to the ordinary convolutional algorithm.

$$OPs = 2 \times Z \times R \times C \times M \times N \times Ksize. \quad (8)$$

Here, $Ksize$ denotes the size of the filters. With reference to Listing1, the execution cycles can be calculated as follow:

$$EC_w = \lceil \frac{M}{T_o} \rceil \times \lceil \frac{N}{T_i} \rceil \times \frac{Z}{T_z} \times \frac{R}{T_r} \times \frac{C}{T_c} \times (\frac{T_z \times T_r \times T_c \times I}{m^{dim}} + L), \quad (9)$$

where I and L denote the iteration interval and latency of the computation pipeline, and dim is the number of dimensions of the input data (i.e., $dim = 2$ for 2D CNN and $dim = 3$ for 3D CNN). Note that for the sake of simplicity, we assume T_z, T_r and T_c are divisors of Z,R and C, respectively. Consequently, the computational roof can be calculated by:

$$CR = \frac{OPs}{EC_w} = \frac{2 \times M \times N \times Ksize \times m^{dim}}{\lceil \frac{M}{T_o} \rceil \times \lceil \frac{N}{T_i} \rceil \times I}. \quad (10)$$

Note that we omit L in Equation 10 due to its negligible impact on the execution cycles. When a comparison is drawn with the computational roof discussed in [21], it can be found that using the Winograd algorithm can allow for a higher computational roof to be reached than when the ordinary convolutional algorithm is used (given that $\frac{m^{dim}}{I} > 1$ in this work). In addition, we can observe that the computational roof is mainly determined by T_o and T_i, especially when M and N are integer multiples of T_o and T_i respectively. Increasing T_o or T_i can thus lead to a higher computational roof.

4.2 Performance Modeling

We mainly focus on modeling the execution time of a given convolutional layer. Since external memory transfer significantly affects the total execution time, we first model the transfer time of the input and output data required each time by the computation engine illustrated in Listing1:

$$V_{in} = T_i \times (S \times T_z + K - S)^{dim-2} \times (S \times T_r + K - S) \times (S \times T_c + K - S) \quad (11)$$

$$V_w = T_o \times T_i \times Ksize \quad (12)$$

$$V_{out} = T_o \times T_z \times T_r \times T_c \quad (13)$$

$$T^i_{trans} = \frac{(V_{in} + V_w) \times Data\_Width}{BW_{eff}} \quad (14)$$

$$T^o_{trans} = \frac{V_{out} \times Data\_Width}{BW_{eff}}, \quad (15)$$

where V_{in}, V_w and V_{out} denote the amount of required input/output feature maps and filters respectively. BW_{eff} is defined as the effective bandwidth of the off-chip memory, while $Data\_Width$ denotes the data widths of the input/output and weights. We define T_{com} as the computation time of *L3* in Listing1, which can be modeled as follows:

$$T_{com} = \frac{T_z \times T_r \times T_c \times I}{m^{dim} \times Freq}. \quad (16)$$

Considering the bandwidth limitation of the FPGA platform, it is not possible for the computation time to be directly calculated calculated by $\frac{EC_w}{Freq}$. Instead, given that we have adopted double buffering in the input and output to hide the transfer latency, the total execution time for a given convolutional layer can be calculated as follows:

$$T_{total} = \frac{Z}{T_z} \times \frac{R}{T_r} \times \frac{C}{T_c} \times (\lceil \frac{M}{T_o} \rceil \times \lceil \frac{N}{T_i} \rceil \times \max\{T_{com}, T^i_{trans}\} + T^o_{trans}), \quad (17)$$

where $Freq$ is the frequency of the accelerator. Note that we remove the transfer time of the output from Equation 17 due to its negligible impact on total execution time.

5 EXPERIMENTAL RESULTS

In this section, we evaluate the template-based architecture by implementing two accelerators for state-of-the-art 2D and 3D CNNs, respectively. In addition, multiple FPGAs are used to test the portability of our designs.

5.1 Experimental Setup

Benchmarks. As a case study, we evaluate our design using two representative CNN models: VGG16 and C3D. All convolutional layers of the selected CNNs have uniform $3 * 3$ and $3 * 3 * 3$ filters, which fit well with the Winograd algorithm.

CPU and GPU setup. We evaluated our FPGA implementation of C3D through comparison with other platforms , namely (1) the high-performance ten-core Intel E5 2680 v2 CPU, which operates at 2.8 GHz, (2) the NVIDIA Tesla K40m GPU with 2880 SIMD cores and (3) the NVIDIA GeForce GTX 1080 GPU with 2560 SIMD cores.

FPGA platform setup. We use two evaluation boards to evaluate our accelerators for 2D and 3D CNNs: Xilinx VC709 and S2C VUS440. The VC709 platform contains a Virtex-7 690t FPGA and two 4GB DDR3 DRAMs. Our implementations are clocked at 150MHz on this platform. To test the portability of our architecture, we implement our designs on S2C VUS440; this integrates a Xilinx VCU440 FPGA and an 8GB DDR4, which can provide a higher bandwidth. Our designs run at 200MHz on this platform.

Design tools. We use Xilinx Vivado HLS 2016.4 to implement the proposed templates, as well as to generate the template-based accelerators. All synthesized results are obtained from Xilinx Vivado 2016.4.

Table 4: Uniform cross-layer parameters

Networks	Winograd	T_i	T_o	Bw_{off}	Bw_{on}	# of Ports
VGG16	$F(2^2, 3^2)$	4	64	256bit	64bit	4
C3D	$F(2^3, 3^3)$	4	32	256bit	64bit	4

Table 5: FPGA resource utilization

Device		Resource	DSP	BRAM	LUT	FF
VC709		Available	3600	2940	433K	866K
	2D	Used	1376	1232	175K	202K
		Utilization	38%	42%	40%	23%
	3D	Used	1536	1508	242K	286K
		Utilization	42%	52%	56%	33%
VUS440		Available	2880	5040	2532K	5065K
	2D	Used	1376	1232	170K	189K
		Utilization	48%	24%	6.7%	3.7%
	3D	Used	1536	1476	209K	285K
		Utilization	53%	30%	8.3%	5.6%

5.2 Performance Analysis

In our experiment, we design an accelerator with unified unroll factors for all convolutional layers in each CNN model rather than creating an optimal design for each layer. In this way the overhead of reprogramming the FPGA for different layers is removed. Note that we primarily evaluate the $F(2^2, 3^2)$ and $F(2^3, 3^3)$ due to the associated benefits in saving DSPs as well as the simplicity of template designs. Table 4 presents the uniform cross-layer parameters for all layers of the benchmarks, including the tiling factors and memory system configurations. Although the selected uniform parameters are sub-optimal for some layers (e.g. the optimal $< T_o, T_i >$ for the Conv1 in C3D is $< 3, 32 >$ instead of $< 4, 32 >$), our results suggest that the overall performance degradation is minimal compared to the condition in which each layer uses the optimal $< T_o, T_i >$. In addition, the configurations in Table 4 also contribute to the optimal on-chip and off-chip bandwidth, according to our on-board memory test.

Table 5 presents the FPGA resource utilization of the implementations for 2D and 3D CNNs. Here, it is evident that the DSPs are no longer the limiting resource on VC709, which demonstrates the benefit of using the Winograd algorithm. Instead, LUTs, which are mainly utilized for the transformation and accumulation templates, dominated the resource consumption. Since VUS440 contains an FPGA with a larger amount of LUTs (5.8x) but fewer DSPs than VC709, it can be seen that our implementations consume over 48% of DSPs but few LUTs (< 10%) on this platform. In summary, the overall resource consumption on VUS440 is lower which contributes to the frequency improvement on this platform.

Figure 8 and Figure 9 present the evaluation results on multiples FPGA platforms for VGG and C3D, respectively. For both

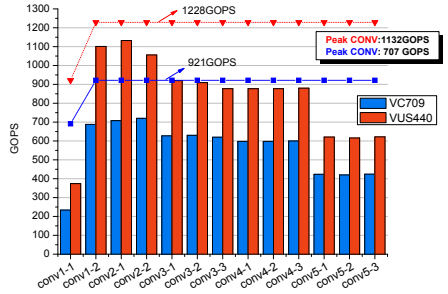


Figure 8: Evaluation results of VGG16.

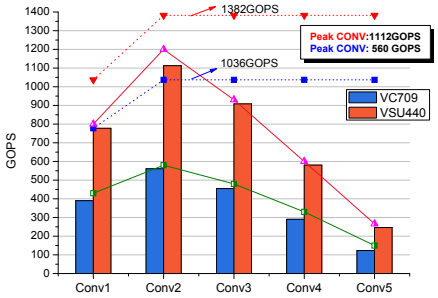


Figure 9: Evaluation results of C3D.

Table 6: Bandwidth comparison on multiple FPGA platforms for C3D

Platform	Freq. (MHz)	BW (GB/s)	C1	C2	C3	C4	C5
VC709	150	Req.	4.15	4.23	4.89	7.24	17.02
		Act.	2.47	2.27	2.14	2.02	2.01
VUS440	200	Req.	5.53	5.65	6.52	9.65	22.7
		Act.	4.98	4.55	4.29	4.05	4.04

benchmarks, implementations on VUS440 can achieve higher peak performance (1132 GOPS for VGG and 1112 GOPS for C3D) and higher overall performance of all CONV layers (902 GOPS for VGG and 940 GOPS for C3D). To explain this result, we compare the required bandwidth with the actual bandwidth of each CONV layer of C3D, with the results presented in Table 6: as it can be seen, VUS440 provides higher bandwidth than VC709 (an average of 2.0x improvement). In addition, both implementations are bounded by the memory bandwidth on the platforms. Moreover, the gap between required and actual bandwidths increases from Conv1 to Conv5 on both platforms. This is because the size of the input feature maps decreases in the latter CONV layers, resulting in inefficient memory access (short burst length).

As a result, the performance of the latter layers (especially Conv5) is poorer compared to that of the former layers. One interesting phenomenon that emerges from Figure 8 and Figure 9 is that the first layers of the benchmarks show relatively lower performance, despite having higher bandwidth; this occurs because these layers only have three input feature maps, which fail to utilize the double buffering in the input buffers ($N < T_i$). In addition, the dashed lines in Figure 8 and Figure 9 indicate the computational roofs of the CONV layers in both networks. This demonstrates that our implementations of VGG and C3D can reach up to 92% and 80%, respectively, of the computational roof on VUS440.

The solid lines in Figure 9 represent the estimated performance of each CONV layer in C3D (green line for VC709 and red line for VUS440). This shows that our analytical model perfectly matches the on-board results, in that the average error is < 5%, which is evidence of the accuracy of our model. Overall, our designs achieve high performance for both 2D and 3D CNNs, thus demonstrating the effectiveness of our proposed template-based architecture.

5.2.1 Comparison with previous work. A performance comparison between our accelerator and the previous FPGA implementations of 2D CNN is presented in Table 7. Throughput and DSP efficiency were used as performance metrics. It can be seen that our work achieves state-of-the-art performance with the second-lowest DSP utilization (38% on VC709) of the implementations listed in Table 7, after [16]), which demonstrates the advantages of utilizing the Winograd algorithms for CNN acceleration. In addition, our work outperforms most of previous work in terms of DSP efficiency except for [2] and [23]. [2] also utilizes the Winograd algorithm in its design. However, the DSPs used in [2] can perform two 16-bit floating-point multiplies and adds, while the DSPs used in this work only supports one 16-bit fixed-point multiply and add. Moreover, our experience suggests that it is difficult to reach high frequency with over 90% of DSP utilization using the Xilinx HLS tool, while [2] and [23] use Intel SDK for OpenCL, which makes this easier. Therefore, it can be concluded that differences in FPGA types and HLS tools between [2, 23] and the present work indicate that it is unfair to draw a direct comparison.

5.2.2 Comparison with SW implementation. Since there is no FPGA implementation for 3D CNN (to the best of our knowledge), we compare our implementation of C3D with the CPU and GPU solutions only. The results of this comparison are presented in Table 8. Note that OpenBLAS and CuDNN libraries are used for optimizing the CPU and GPU solutions. The results demonstrate that on VC709, our implementation achieves an end-to-end performance 7.3x greater than that of CPU and with 3.6x the energy efficiency of K40m GPU. Better results are attained on VUS440; here, our implementation achieves a speedup of 13.4x relative to the CPU solution and a 6.4x increase in energy efficiency relative to K40m GPU. In addition, it can also be seen that when compared with the state-of-the-art GTX 1080 GPU, our design on VUS440 still achieves better energy efficiency.

6 RELATED WORK

An abundance of research exists into designing FPGA-based accelerators for 2D CNN. However, the majority of these studies have only implemented the convolutional layers of CNNs. A representative work by Zhang et al. [21] proposes a roofline model to maximize computational resources on FPGA under the memory bandwidth limitations. As the convolutional layers are accelerated, the other unaccelerated layers become a performance bottleneck. To resolve this issue, several studies [14, 16, 20] attempt to accelerate the entire CNN on an FPGA. Suda et al. [16] transform the 3D convolution to 2D general purpose matrix multiplication and design a matrix multiplication accelerator for both convolutional and fully connected layers. The work by Qiu [14] presents a dynamic-precision data quantization method and uses a convolver design

Table 7: Comparison with previous implementations for 2D CNN

	[14]	[16]	[20]	[12]	[2]	[23]	Ours	
FPGA	Xilinx Zynq XC7Z045	Altera Stratix-V	Xilinx Virtex 690t	Arria10 GX1150	Arria10 GX1150	Arria10 GX1150	Xilinx Virtex 690t	Xilinx VCU440
Frequency (MHz)	150	120	150	150	303	385	150	200
CNN	VGG	VGG	VGG	VGG	AlexNet	VGG	VGG	VGG
Precision	16-bit fixed	8-16 bits fixed	16-bit fixed	8-16 bits fixed	16-bit float	16-bit fixed	16-bit fixed	16-bit fixed
DSP Utilization	780(87%)	727(37%)	2833(78%)	1518(100%)	1476(97%)	2756(91%)	1376(38%)	1376(48%)
Throughput (Gops)	137	118	354	645	1382	1790	570	821
DSP Efficiency (Gops/DSPs)	0.18	0.16	0.12	0.43	0.98	0.65	0.41	0.60

Table 8: End-to-end comparison with CPU/GPU for C3D

Platforms	CPU	GPU		FPGA	
Device	E5-2680	K40	GTX1080	VC709	VUS440
Technology	22nm	28nm	16nm	28nm	20nm
Power (W)	115	250	180	25	26
CONV (Gops)	60.3	1206.5	4375.7	474.3	940.6
CNN (Gops)	58.7	1174.0	4101.9	430.7	784.7
Latency(ms)	656.2	32.8	8.8	89.4	49.1
Speedup	1x	20.0x	69.9x	7.3x	13.4x
(Gops/W)	0.5 (1x)	4.7 (9.2x)	22.8 (44.6x)	17.1 (33.8x)	30.2 (60.3x)

for all layers in the CNN, while Zhang et al. [20] propose a uniform convolutional matrix multiplication representation to accelerate both the convolutional and fully connected layers on FPGAs. All these works mentioned above process the networks layer by layer. The works in [1, 11] take opposite approaches by making all layers working concurrently in a pipelined structure. In [1], the authors use a pyramid-shaped multi-layer sliding window to fuse the processing of adjacent CNN layers, thereby minimizing the off-chip transfer. Li et al. [11] map all the layers of the CNN on one chip and make different layers work concurrently in a pipelined structure. However, in order to store the intermediate results between layers, these works require massive memory usage for large scale CNNs in order to store the intermediate results between layers.

Currently, a few studies utilize the fast algorithms to reduce the computation complexity of CNNs. Zhang et al. [22] propose a 2D convolver in frequency domain to accelerate convolutional layers on FPGA, leaving other layers computed by CPU. However, the FFT-based approach shows less efficiency for convolutions with small filters [10]. Aydonat et al. [2] utilize the 1D Winograd algorithm for arithmetic optimization that greatly improves the DSP utilization. However, it stores all input feature maps in on-chip memory, which can only support small CNN models. To the best of our knowledge, we are the first to explore 3D CNN acceleration using the Winograd algorithm on FPGA. More importantly, we unify the 2D and 3D CNNs into a single acceleration framework.

7 CONCLUSIONS

In this work, we propose a template-based methodology for 2D and 3D CNN acceleration on FPGA. We use uniform templates to build accelerators for 2D and 3D CNNs based on the Winograd algorithm. We also develop uniform analytical model for our template-based architecture, along with performance modeling that efficiently explores the design space. We evaluate our architecture by realizing the implementations for VGG16 and C3D across multiple FPGA platforms. Experimental results show that our template-based architecture is capable of accelerating both 2D and 3D CNNs efficiently.

ACKNOWLEDGMENTS

This work was supported by National Program on Key Basic Research Project 2016YFB1000401 and 2016YFB1000403.

REFERENCES

[1] M. Alwani *et al.*, "Fused-layer cnn accelerators," In *MICRO*, pages 1–12. IEEE, 2016.
[2] U. Aydonat *et al.*, "An opencl deep learning accelerator on arria 10," *arXiv preprint arXiv:1701.03534*, 2017.
[3] J. Cong *et al.*, "Bandwidth optimization through on-chip memory restructuring for hls," *Design Automation Conference (DAC)*, pages 1–6. IEEE, 2017.
[4] K. He *et al.*, "Deep residual learning for image recognition," In *CVPR*, pages 770–778, 2016.
[5] S. Ji *et al.*, "3d convolutional neural networks for human action recognition," *IEEE transactions on pattern analysis and machine intelligence*, 35(1):221–231, 2013.
[6] N. P. Jouppi *et al.*, "In-datacenter performance analysis of a tensor processing unit," *arXiv preprint arXiv:1704.04760*, 2017.
[7] K. Kamnitsas *et al.*, "Efficient multi-scale 3d cnn with fully connected crf for accurate brain lesion segmentation," *Medical image analysis*, 36:61–78, 2017.
[8] A. Krizhevsky *et al.*, "Imagenet classification with deep convolutional neural networks," In *NIPS*, pages 1097–1105, 2012.
[9] Q. Lan *et al.*, "High performance implementation of 3d convolutional neural networks on a gpu," Inpress at https://www.hindawi.com/journals/cin/aip/8348671/.
[10] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," In *CVPR*, pages 4013–4021, 2016.
[11] H. Li *et al.*, "A high performance fpga-based accelerator for large-scale convolutional neural networks," In *FPL*, pages 1–9. IEEE, 2016.
[12] Y. Ma *et al.*, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," In *FPGA*, pages 45–54. ACM, 2017.
[13] D. Mahajan *et al.*, "Tabla: A unified template-based framework for accelerating statistical machine learning," In *HPCA*, pages 14–26. IEEE, 2016.
[14] J. Qiu *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," In *FPGA*, pages 26–35. ACM, 2016.
[15] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.
[16] N. Suda *et al.*, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," In *FPGA*, pages 16–25. ACM, 2016.
[17] D. Tran *et al.*, "Learning spatiotemporal features with 3d convolutional networks," In *ICCV*, pages 4489–4497, 2015.
[18] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," In *DAC*, page 29. ACM, 2017.
[19] S. Winograd. "On multiplication of polynomials modulo a polynomial," *SIAM Journal on Computing*, 9(2):225–229, 1980.
[20] C. Zhang *et al.*, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," In *ICCAD*, pages 1–8. IEEE, 2016.
[21] C. Zhang *et al.*, "Optimizing fpga-based accelerator design for deep convolutional neural networks," In *FPGA*, pages 161–170. ACM, 2015.
[22] C. Zhang *et al.*, "Frequency domain acceleration of convolutional neural networks on cpu-fpga shared memory system," In *FPGA*, pages 35–44. ACM, 2017.
[23] J. Zhang and J. Li, "Improving the performance of opencl-based fpga accelerator for convolutional neural network," In *FPGA*, pages 25–34. ACM, 2017.