Towards Efficient Convolutional Neural Network for Domain-Specific Applications on FPGA

Ruizhe Zhao*, Ho-Cheung Ng*, Wayne Luk* and Xinyu Niu†
*Department of Computing, Imperial College London, London, United Kingdom
{*ruizhe.zhao15, h.ng16, w.luk*}*@imperial.ac.uk*
†Corerain Technologies Ltd., Shenzhen, China. *xinyu.niu@corerain.com*

***Abstract*—FPGA becomes a popular technology for implementing Convolutional Neural Network (CNN) in recent years. Most CNN applications on FPGA are domain-specific, e.g., detecting objects from specific categories, in which commonly-used CNN models pre-trained on general datasets may not be efficient enough. This paper presents TuRF, an end-to-end CNN acceleration framework to efficiently deploy domain-specific applications on FPGA by transfer learning that adapts pre-trained models to specific domains, replacing standard convolution layers with efficient convolution blocks, and applying layer fusion to enhance hardware design performance. We evaluate TuRF by deploying a pre-trained VGG-16 model for a domain-specific image recognition task onto a Stratix V FPGA. Results show that designs generated by TuRF achieve better performance than prior methods for the original VGG-16 and ResNet-50 models, while for the optimised VGG-16 model TuRF designs are more accurate and easier to process.**

I. Introduction

There has been much recent work on developing FPGA implementations of *Convolutional Neural Networks* (CNNs). While significant progress has been made in optimising the inference process of general CNN models on FPGAs, training and optimising CNNs for various *domain-specific applications* remain a demanding task. CNN models for domain-specific applications only need to detect or classify objects from a narrow range of classes. Recent discovery in *transfer learning* [1] — a research topic focusing on exploiting features reusable from one task to another — shows that CNN models that are pre-trained on general datasets can be efficiently *fine-tuned* [2] for specific domains. This approach works well for medical image analysis: a pre-trained CNN with adequate fine-tuning can outperform or perform as well as training from scratch [3].

While the transfer learning approach is promising, the challenge is to exploit it for domain-specific applications on FPGA, where efficient processing is vital. For tasks in a specific domain, standard convolution layers dedicated to extracting general features are over-parameterised and can be replaced by efficient *convolution blocks*, which consist of multiple small convolution layers with much fewer parameters. Example blocks are *bottleneck* [4], *depthwise separable* [5], and *separable bottleneck* [6]. They can reduce computational redundancy while maintaining a satisfactory accuracy. Meanwhile, since a layer-replaced model normally can be easily fine-tuned, the cost of layer replacement is minor. However, they are rarely explored and implemented in any of the previous work on FPGA acceleration of CNNs.

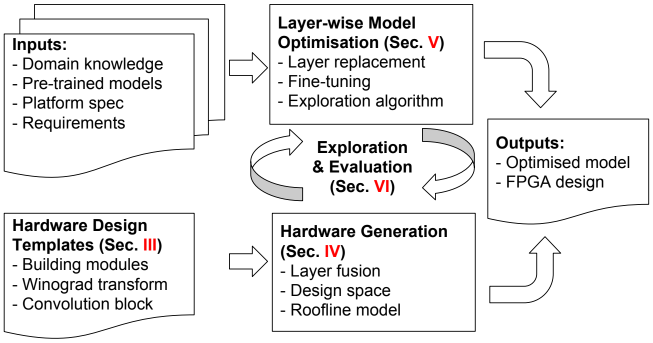

Fig. 1. TuRF design flow. Corresponding sections are marked.

This paper proposes *TuRF*, a novel framework that generates efficient CNN models on FPGA for domain-specific applications (Fig. 1). TuRF accepts a CNN model pre-trained from a large-scale dataset, replaces its selected standard convolution layers with various convolution blocks, fine-tunes and evaluates the layer-replaced model, and outputs an efficient FPGA design in the end. To efficiently process convolution blocks, TuRF generates FPGA designs that *fuse* their inner convolution layers. The major contributions are as follows:

1) A design template that supports efficient CNN models and convolution blocks with Winograd [7], and also layer fusion optimisation (Section III and IV).
2) Characterisation of the design space of CNN model regarding domain-specific applications and a transfer learning inspired layer-wise optimisation that replaces standard convolution layers by blocks with fine-tuning (Section V).
3) Evaluation of our framework with flower classification [2], a domain-specific application for transfer learning evaluation. Results show that on Stratix V 5GSD8, our framework can generate both efficient hardware and better CNN models for a given application (Section VI).

II. Motivations and Background

Implementing CNN onto FPGA for domain-specific applications is challenging. Most of the previous efforts focus on data quantisation and binarisation [8], [9], arithmetic transformations [10], or exploiting model sparsity with pruning [11]. However, it is difficult to apply these methods from a domain-specific perspective because their evaluations and results are based on specific CNN models, and are not guaranteed to be reproducible using other models. Also, no clear correlation between accuracy and data representation or sparsity has been discovered yet. Finally, a sparse CNN model is much harder to process on FPGA and different sparsity patterns can result in very different performance.

The motivation of the proposed framework rests on the recent trend in efficient CNN architecture design [4]–[6]. Essentially, the redundancy in CNN is sometimes architectural and can be reduced by modifying the standard convolution layer with efficient *convolution blocks* as shown in Fig. 2. A convolution block is a set of convolution layers that extracts features as a whole. More importantly, a convolution block generally consumes fewer resources than its equivalent standard convolution layer. Compared to quantisation and pruning which rely on statistical properties of pre-trained CNN models, this approach is more generic and has been evaluated for different domain-specific applications, as shown in [5]. It is one of the major tasks for TuRF to explore the optimisation opportunity of switching from convolution layers to blocks. Nevertheless, there is no dedicated FPGA implementation for convolution blocks, and therefore, TuRF also aims at accelerating CNN for domain-specific applications by exploring the FPGA implementation of convolution blocks.

A. Convolution Layer and Convolution Block

A convolution layer correlates an input feature map $\mathbf{D}$ and a filter $\mathbf{G}$ together. Suppose $\mathbf{D}$ is an image with C channels and spatial dimensions $H \times W$, and $\mathbf{G}$ is a 4D filter that consists of F output channels, C input channels, and kernels of size $K \times K$, the resulting feature map $\mathbf{Y}$ is defined as (1), where $*$ is the spatial convolution operator.

$$\begin{aligned}\mathbf{Y}_f &= \sum_{c=1}^{C} \mathbf{D}_c * \mathbf{G}_{f,c} \\ \mathbf{Y}_{f,x,y} &= \sum_{c=1}^{C}\sum_{h=1}^{K}\sum_{w=1}^{K} \mathbf{D}_{c,x+h,y+w} \times \mathbf{G}_{f,c,h,w}\end{aligned} \qquad (1)$$

A standard convolution layer can be replaced by *convolution block* to improve efficiency. There are basically four types:

1) Stacked Block: It simply stacks two standard convolution layers together and reduces the number of channels. Its input and output are connected by a *shortcut* connection. Please refer to the model ResNet-34 [4] for more details.

2) Depthwise Separable: It is proposed in [5], [12], [13], where the *spatial* and *cross-channel* correlation is studied separately using *depthwise* and *pointwise* convolution respectively. The depthwise convolution only performs spatial convolution in each channel of the input feature map, and the pointwise convolution is a special case of the standard convolution by setting K to 1. Assume that $\widehat{\mathbf{G}}$ is the 3D depthwise filter and $\mathbf{G}$ is the 2D pointwise filter, (2) defines the depthwise separable convolution layer as described in [5].

$$\mathbf{Y}_{f,x,y} = \sum_{c=1}^{C} \left(\mathbf{D}_c * \widehat{\mathbf{G}}_c\right) \times \mathbf{G}_{f,c} \qquad (2)$$

3) Bottleneck Block: A recent trend of constructing CNN is the prevailing use of *bottleneck* block as demonstrated in ResNet-50 [4] which is economical and easy-to-train for deeper networks. A bottleneck block consists of a stack of 1×1, 3×3, and 1×1 convolution layers, in which the first and last one reduce and increase the number of channels respectively. The input of the bottleneck block is connected to the output of the previous stack where there also exists a residual connection performing element-wise addition.

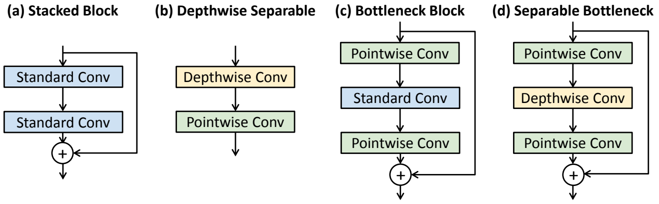

Fig. 2. Efficient convolution blocks that aim at improving the efficiency.

4) Separable Bottleneck: Evolved from the original bottleneck block, *linear bottleneck* is proposed and used in MobileNet V2 [6] for efficiency improvement. Compared to the original, the middle bottleneck convolution is replaced with its depthwise version and the activation layer is removed after the last convolution, so as to combine the efficiency of depthwise separable with the effectiveness of bottleneck block.

B. Winograd Algorithm

As an arithmetic optimisation method dated back to 1980s, the Winograd's minimal filtering algorithm [14] is still proven to be powerful in optimising convolution layer processing based on recent research [7]. An instance of 2D Winograd algorithm, denoted by $F(m \times m, r \times r)$ where m is the 2D tile size and r is the filter kernel size, can be formulated as (3),

$$\mathbf{Y} = \mathbf{A}^\mathsf{T}\left[\left[\mathbf{G}g\mathbf{G}^\mathsf{T}\right] \odot \left[\mathbf{B}^\mathsf{T} d\mathbf{B}\right]\right]\mathbf{A} = \mathbf{A}^\mathsf{T}\mathbf{X}\mathbf{A} \qquad (3)$$

where $\odot$ is the Hadamard product. $\mathbf{G}$, $\mathbf{B}$, $\mathbf{A}$ are three transformation matrices with $(m+r-1)\times r$, $(m+r-1)\times(m+r-1)$, $(m+r-1)\times m$ in shape. g is an r^2 filter kernel and d with the size $(m+r-1)\times(m+r-1)$ is a tile of the input feature map. A widely used configuration is $F(4^2, 3^2)$, and compared to standard convolution with $4^2 3^2 = 144$ multiplications to produce 16 output elements, 2D Winograd based convolution reduces the computation complexity to $6^2 = 36$ multiplications, which is equivalent to $144/36 = 4$x speed-up. For details about the Winograd Algorithm, please refer to [7].

The utilisation of Winograd for CNN acceleration on FPGA is discussed in [10], [15]–[17]. $F(4^2, 3^2)$ is applied in [10], [15] and $F(2^2, 3^2)$ in [16], [17]. $F(4^2, 3^2)$ can achieve higher speed-up, but it contains constant factors that are not 2^n. In the following discussion, we use $F(4^2, 3^2)$ but our approach can be configured to support $F(2^2, 3^2)$.

C. Efficient CNN Models

In this paper, we mainly study three efficient CNN models: ResNet-50 [4], MobileNet V1 [5] and V2 [6], to gain research insight for our hardware template and to make a comparison to our generated CNN model for domain-specific application. Default configurations are used for these networks. As shown in Fig. 3, convolution blocks are responsible for most of the

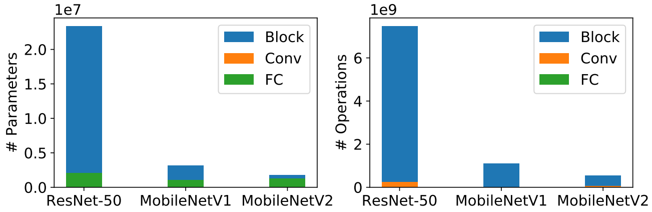

Fig. 3. A comparison of the number of parameters (left) and operations (right) within the convolution block, single convolution layer and FC layer between three efficient CNN models. Convolution blocks are dominating.

TABLE I. CNN MODELS STATISTICS.

Model	Block[1]	Ops (GOPS)	Params (M)	Top-1 (%)
VGG-16	—	30.95	138.3	71.5
ResNet-50	(c)	7.72	24.3	75.2
MobileNetV1	(b)	1.14	4.01	70.9
MobileNetV2	(d)	0.61	3.31	71.9

[1] (b), (c), (d) are symbols that denote convolution blocks in Fig. 2.

operations and parameters. These models are also compared to VGG-16 [18] as shown in Table I. ImageNet top-1 accuracy of each model is listed in the same table as well.

III. Hardware Design Template

TuRF proposes the use of a scalable hardware design template as a fundamental component in our framework. The template enables generation of optimised CNN hardware by supporting various convolution types. Particularly, the convolutional blocks discussed in Section II are the primary research focus in our template development process. Winograd transformation is also used to accelerate spatial convolution.

A. Design Template Overview

Our design template can be configured to support all the layers utilised in recent efficient CNN models. We focus on convolution layer and convolution blocks, which are the most time-consuming parts in these models. An accelerator for convolution layer or block can be constructed by basic *building modules* in our template. Each module can be configured regarding the level of parallelism or computation sequence. Our design is by default implemented with *fixed-point* representation, and its configuration is decided by the data range.

Similar to [19], a design module is described as a tuple $\langle cfg, in, out\rangle$ in which cfg is a set of module configuration and in, out specify the width of input and output streams respectively. The module configurations can be described with:

1) *Tile shape*: T_h, T_w, T_c denote the height, width, and channels of the input and T_f represents the output channels.
2) *Level of parallelism*: P_c, P_f, P_h, P_w represent the number of elements to process in parallel along the input and output channels, height, and width axis.
3) *Layer specifics*: K denotes the kernel size and r mentioned in Section II is replaced by K.

B. Basic Building Modules

1) Line Buffer: Given by (4), it is a module that creates *sliding windows* over an input feature map. We use K' to denote either convolution kernel size K or the Winograd input tile size $(m+K-1)$. This module is implemented using *shift registers* organised into K' rows.

$$\langle\{P_c, P_h, P_w\}, P_cP_hP_w, (K'+P_h-1)(K'+P_w-1)\rangle \quad (4)$$

2) Input and Output Buffers: Buffers are implemented as on-chip memory to exploit the locality of the computation. An input buffer caches input feature map to be reused throughout the computation, and an output buffer stores and accumulates temporary results. See (5) and (6) for descriptions.

$$\langle\{P_c, P_w\}, P_c \times P_w, P_c \times P_w\rangle \quad (5)$$

$$\langle\{P_f, P_w\}, P_f \times P_w, P_f \times P_w\rangle \quad (6)$$

3) Winograd Transformation: The Winograd algorithm is applied to standard and depthwise convolution to reduce the computation complexity. According to (3), three transformation modules are required to process a Winograd convolution. Let T_k be the Winograd tile size $(m+K-1)$, (7), (8), (9) illustrate the configurations and interfaces of the transformation modules for input feature map $\mathbf{B}^\intercal d\mathbf{B}$, weights $\mathbf{G}d\mathbf{G}^\intercal$, and output $\mathbf{A}^\intercal\mathbf{X}\mathbf{A}$ respectively. Each transformation consists of two multiplications between an input and a constant matrix, which are implemented with either multipliers with LUTs or shift operators (for 2^n constants) to save resources.

$$\langle\{P_c, K\}, P_cT_k^2, P_cT_k^2\rangle \quad (7)$$

$$\langle\{P_c, P_f, K\}, P_cP_fK^2, P_cP_fT_k^2\rangle \quad (8)$$

$$\langle\{P_c, P_f, K\}, P_cP_fT_k^2, P_cP_fm^2\rangle \quad (9)$$

4) Arithmetic Module: Most of the arithmetic computations in a typical CNN workload is dot-product, which is employed in the spatial and cross-channel convolution and also in the fully-connected (FC) layers. Each dot-product module consists of an array of multipliers followed by an adder tree. The dot-product modules are further organised into a higher-level array for parallelisation. This module can be shared among convolution and FC layers when necessary.

5) Other Design Modules: An *element-wise addition* module performs addition of two identically sized feature maps. An *activation* module implements non-linear activation functions. A *normalisation* module normalises its input by Batch Normalisation. We omit details about these modules because they are simple and have limited impact on the overall performance.

C. Implementation of a Single Layer

An accelerator can be constructed from building modules to perform the computation, which can only perform the computation of one layer at one time, in contrast to the fused layer design (Section IV). Fig. 4 shows the system diagram when a single layer is implemented. It can support different types of convolution such as depthwise or pointwise and fully-connected layers by efficiently sharing dot-products in the arithmetic module. Modules are connected using data-flow streams with the same input and output width. Outputs from building modules should be consumed immediately to avoid congestion. A global state controller together with counters are utilised for each design to assign addresses for buffers

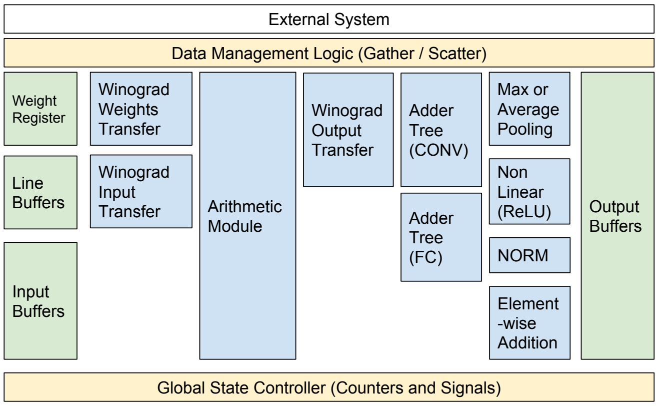

Fig. 4. Diagram of our proposed configurable system architecture (NORM stands for normalisation).

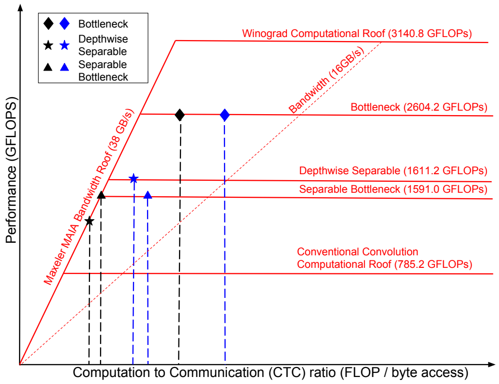

Fig. 5. We use the roofline model to evaluate the benefits of layer fusion for different convolution blocks. The baseline is marked in black colour and the fused design is marked blue colour. The target FPGA is Stratix-V 5SGSD8 on Maxeler MPC-X node.

and to enable/disable read/write actions and control data-flow directions. Multiplexers are implicitly inserted between the columns in the figure to control computation. The arithmetic module is shared by convolution with or without Winograd transformation, and fully-connected layer.

The computation sequence of convolution, which is a permutation of interchangeable nested loops indices (f, c, i) based on (1), has a large impact on the architectural structure. The impact of such permutation for individual convolution layer has been extensively studied in recent research studies [20], [21]. Therefore, we only discuss two computation sequences, *filter-major* (f, c, i) and *channel-major* (c, f, i), and present their impacts on buffer sizes and pipeline in the rest of this paper. The size of the buffer for the major index is linear to its parallel factor. For example, the output buffer, which is iterated with the f index, is of size $P_f HW$ and is linear to P_f. The pipeline behaviour between two adjacent layers can be different if their computation sequences are configured in different ways (6). Further discussions in the next section.

IV. Fused Convolution Blocks

Convolution blocks consume most of the operations according to Fig. 3 and should be well-optimised for performance improvement according to Amdahl's law. Similar to the previous work [22], a baseline accelerator for the convolution block is mainly based on a layer-by-layer execution. This approach incurs significant off-chip data transfer and consequently cannot fully exploit the potential of pipelining CNN layers.

To overcome this drawback, we propose a fused accelerator for the convolution block that enables the computation of all layers to complete in one launch. The benefits of layer fusion are explained by the *roofline model* [23] and Fig. 5 illustrates the possible benefits from layer fusion in different convolution blocks. We extend the analysis from [10] by adding computational roofs for three convolution blocks. We have to note that the bandwidth of the evaluation system (Maxeler MPC-X Node) is so large that only depthwise separable blocks can take advantage of layer fusion. However, all convolution blocks can benefit from layer fusion if the bandwidth is decreased to 16 GB/s or smaller which is common for commonality FPGA devices.

The idea of layer fusion is inspired by [24], [25]. These works only fuse the standard convolution with uniform kernel size and target high-end FPGAs. Yet layer fusion is more difficult in our case because there are other convolution variants, and FPGAs do not always have sufficient resources to fully place the fused block. Basically, we improve the previous approaches to address the following *new* challenges:

1) The fusion method can support various convolution types.
2) There are many options to explore when the layers are fused, such as buffer size and computation sequence.
3) Tiling must be considered to support small FPGAs.

A. Fusion Method for General Layer

We propose a method that can generate the fused design for typical convolution blocks automatically using the following steps. 1) The hardware implementation of convolution layer is selected layer by layer. 2) The input and output buffers between adjacent layers are combined. 3) The final configurations are aggregated and decided by the predicted latency.

The first step can be easily implemented because our design template can support convolution types in any known blocks. In the second step, we decide the buffer usage between two adjacent layers by the layer types and performance requirements. If two adjacent layers are standard and pointwise convolution, the use of a single buffer can minimise the area cost but may incur stalling of the entire pipeline, as the previous layer can only write into the buffer when the subsequent layer finishes the computation. Doubling the buffer can eliminate this issue with the increase of the area cost.

The third step determines the final configurations of the fused accelerator which includes: the level of parallelism, buffer sizes, and computation sequences. Suppose there are N layers in a given module. Let $\langle P_h^i, P_w^i, P_c^i, P_f^i \rangle$ be the parallelisation parameters of layer i. To represent the computation sequences, we use $Seq^i \in \{\mathtt{FM}, \mathtt{CM}\}$ to denote whether layer i is filter-major `FM` or channel-major `CM`.

TABLE II. BUFFER SIZES UNDER DIFFERENT CONFIGURATIONS.

(Seq^{i-1}, Seq^i)	L_{i-1} **Output**	L_i **Input**	**Double Buffering**
$(\texttt{FM}, \texttt{CM})$	$P_c^iT_h^iT_w^i$	$T_c^iT_h^iT_w^i$	$2 \times P_c^iT_h^iT_w^i$
$(\texttt{CM}, \texttt{FM})$	$T_c^iT_h^iT_w^i$	$T_c^iT_h^iT_w^i$	$2 \times T_c^iT_h^iT_w^i$
$(\texttt{FM}, \texttt{FM})$	$P_c^iT_h^iT_w^i$	$T_c^iT_h^iT_w^i$	$2 \times P_c^iT_h^iT_w^i$
$(\texttt{CM}, \texttt{CM})$	$T_c^iT_h^iT_w^i$	inefficient	$2 \times T_c^iT_h^iT_w^i$

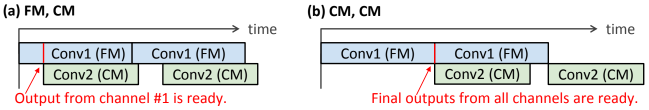

Fig. 6. Two example pipelines of a fused design with different sequences for a stacked block: $\langle \texttt{FM}, \texttt{CM} \rangle$ and $\langle \texttt{CM}, \texttt{CM} \rangle$. Each rectangler box represents a complete execution of a tile. Double buffering is applied in (a).

1) Parallelisation Parameters: The parameters of a fused design should satisfy constraints in (10), which ensures the widths between all the input and output ports along the design modules are the same. Derived from (10), parallelisation parameters of a fused design can be simplified as $\langle P_h, P_w, P_c^1, P_c^2, \dots, P_c^N, P_f \rangle$.

$$\forall i \in \{2, \dots, N\} \; P_h^i = P_h^{i-1} \wedge P_w^i = P_w^{i-1} \wedge P_c^i = P_f^{i-1} \quad (10)$$

2) Buffer Size: The size of a buffer depends on the sequence of layers that it connects to. The first input and the last output buffers are similar to the ones in Section III-B, while other intermediate buffers are more complicated to analyse. The size of buffer B_i, which is connected to layer L_{i-1} and L_i, depends on Seq^{i-1}, Seq^i, and the following options: 1) the same size as the buffer in L_i input or L_{i-1} output; 2) double buffering to avoid stalling of the pipeline.

Table II lists the size of the intermediate buffer B_i under different configurations. The first column is the sequence of the two layers connected using a buffer. Double buffering is only applied when it is indeed beneficial for improving the pipeline performance. A configuration is *inefficient* if the buffer is too small to store the required input or output.

3) Computation Sequence and Pipeline: The fused design is a streaming architecture and the computation of all layers are pipelined. We notice that computation sequence of each layer can affect the pipelining as illustrated in Fig. 6. $\langle \texttt{FM}, \texttt{CM} \rangle$ has a lower latency than $\langle \texttt{CM}, \texttt{CM} \rangle$ (used in [25]) since the first output finalised by layer 1 can be immediately consumed by layer 2. For more complicated cases, we implement a cycle-accurate simulator to obtain all combinations of computation sequences and evaluate their latency. Apart from latency, we also consider buffer sizes since they are affected by computation sequences as shown in Table II and Fig. 7 shows the exploration result for the bottleneck and stacked blocks.

4) Tiling: A convolution block can be tiled into smaller pieces when the on-chip resources are limited. Specifically, for a convolution block with N layers, a *tile* can be defined as $\langle T_h, T_w, T_c^1, \dots, T_c^N, T_f \rangle$. Unlike tiling a convolution layer which mainly introduces an off-chip transfer overhead, tiling a convolution block can incur much redundant computation as well. Therefore, tiling configurations should be carefully explored to avoid such cases.

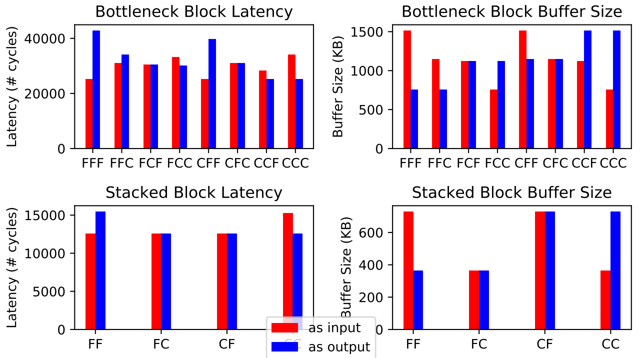

Fig. 7. Comparison of *pipeline latency* and *buffer size* with different computation sequences (x-axis) and buffer options (bar colouring). In the x-axis, 'F' refers to filter and 'C' refers to channel, and hence 'FFF' means that every layer within the bottleneck block uses a filter-major sequence. Moreover, A red bar refers to the adaption of the buffer size used in the previous layer while a blue bar refers to that used in the next layer.

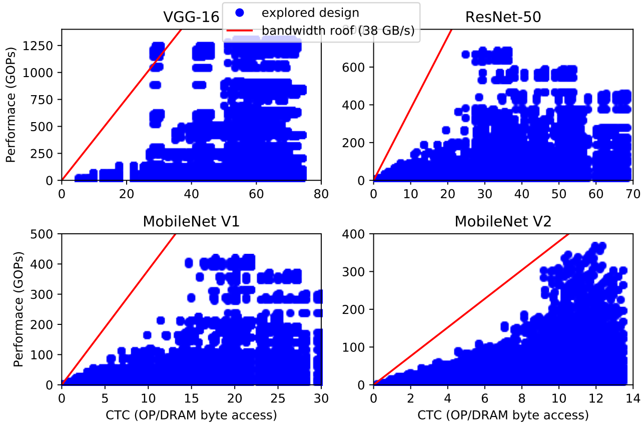

Fig. 8. Design space exploration of four CNN models targeting Stratix-V 5SGSD8 on Maxeler MPC-X node. The computational roofs are not shown because they are above the upper limit of the y-axis.

B. Design Space Exploration

When the previously discussed configurations are combined, we can characterise the hardware design space of a convolution block by $\langle T_h, T_w, \{T_c^i\}, T_f, P_h, P_w, \{P_c^i\}, P_f, \{Seq^i\} \rangle$. Winograd based design is used by default. The performance and area cost are evaluated in two steps: a cycle-accurate simulator is used to find the best computation sequence and its latency, and the latency can provide the performance numbers; and the resource consumption can be computed by a linear prediction model built upon synthesised designs. Finally, the roofline model is used to find the best design under resource constraints. Fig. 8 presents the exploration results for three efficient CNN models and a baseline VGG-16 model using our hardware template. We notice that the performance in GOPs for the efficient models is generally smaller. However, fewer operations are also required for these networks, and hence the overall inference time is still shorter (Section VI).

TABLE III. Evaluation of Efficient CNN Models and Different VGG-16 Variants on Stratix V.

	Efficient CNN Models			**VGG-16 Variants**[1]			
	ResNet-50	MobileNet V1	MobileNet V2	VGG-16	VGG-16 (1)	VGG-16 (2)	VGG-16 (5)
# Ops (GOP)	7.74	1.14	0.611	30.95	26.29	19.36	3.82
# Param. (M)	25.5	4.21	3.47	138.3	132.1	129.83	125.3
Clock Freq. (MHz)	200	200	200	200	200	200	200
Bit width	16 bit	16 bit	16 bit	16 bit	16 bit	16 bit	16 bit
DSP usage	1680	1664	1856	1738	1872	1536	1680
Latency (ms)	7.95	0.884	1.02	14.5	10.3	9.65	8.42
Throughput (GOPS)	973.2	1287.2	592	1928.4	2561.5	2007.0	453.6
Top-1 Accuracy	93.5	88.3	87.5	90.5	93.5	92.75	84.75

[1] The n in VGG-16 (n) denotes a VGG-16 variant that has n number of layers replaced by depthwise separable convolution.

V. Layer-wise Model Optimisation

The objective of TuRF is to find an efficient CNN model and its corresponding design on FPGA for a given domain-specific application. Section III and IV discuss how to map efficient CNN models on FPGA designs. Moreover, in this section, we look into the design space exploration of CNN model, which is inspired by transfer learning for layer-wise optimisation, and the final tool-flow for TuRF.

A. CNN Model Selection and Optimisation

CNN model optimisation is about searching for the most efficient network under pre-defined accuracy requirements. If the hardware factor is put aside for now, we can define model efficiency as the number of parameters and operations required to achieve a certain accuracy, and the remaining challenges are the characterisation and exploration of the model design space.

1) Model Design Space: A typical CNN model is a sequence of cascading layers with convolution layers. To restrict the scale of the design space, we only explore models that are *grown* from either VGG-16 or ResNet-50. To further limit the design space for feasible exploration, a model originated from VGG-16 or ResNet-50 can have its convolution layers replaced only by a particular *separable* convolution block as shown in (11). Such replacements are also the partial motivation for the design of MobileNet. We represent a model in our design space as $\langle M, L^1, \ldots, L^N \rangle$, in which $M \in \{\text{VGG-16, ResNet-50}\}$ is the base model with N convolution layers and $L^i \in \{\texttt{ORIGIN}, \texttt{SEPARABLE}\}$ indicates replacement.

$$\begin{aligned} \textit{standard convolution} &\longrightarrow \textit{depthwise separable block} \\ \textit{bottleneck block} &\longrightarrow \textit{separable bottleneck block} \end{aligned} \tag{11}$$

2) Exploration Method: In most cases we do not have a sufficient budget for training every possible model. Therefore, we devise the following optimisation approach, inspired by the principles of transfer learning. The input to our exploration procedure can be any models pre-trained based on ImageNet, which supposedly is general and consists of removable redundancies regarding the targeting application. We intend to achieve the required accuracy by *fine-tuning* the input model, in which only top layers are trained and others are fixed. We also assume that replacing top convolution layers are more beneficial than bottom ones. This is based on [26] explaining the mechanism of CNN for computer vision, that convolution layers closer to the bottom extract lower-level features such as edges and shapes, and those closer to the top understand the higher-level features such as faces and eyes. Hence, our assumption is mostly valid because the model is now focusing high-level domain knowledge.

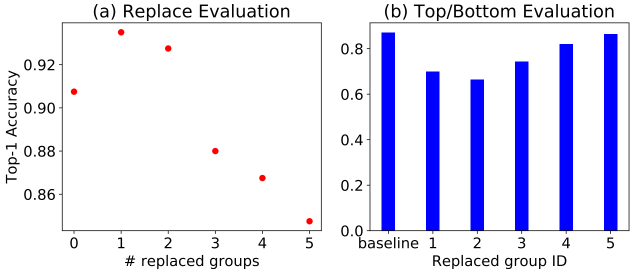

Fig. 9. (a) evaluates of accuracy by the number of replaced groups in VGG-16, and (b) presents the impact of replacement positions.

As such, we propose a heuristic, greedy algorithm to explore model design space. It starts with a pre-trained model and tries to replace layers from the top. In each iteration, this algorithm fine-tunes the model candidate for a fixed number of steps. The procedure terminates once the algorithm fails to satisfy the accuracy requirement. Note that when the budget is sufficient, we need not stop and can continue searching.

3) Evaluation: We evaluate our method on a *flowers* classification problem [2] where the original model is a pre-trained VGG-16. The convolution layers in VGG-16 are replaced by their groups in this case. Figure 9 presents the evaluation results in two aspects. The left figure shows the final exploration results: each point illustrates the accuracy and size of an explored model, and the most efficient model is the one with the top convolution group replaced (the second point from left), which is even better than the one with no layer replacement (the leftmost point). Layers are consecutively replaced from top to bottom. The right figure evaluates our assumption that replacing the top convolution layers is more beneficial than the bottom ones. In the figure, a replaced group is closer to the top if its ID is bigger. The rightmost column with the topmost group replaced achieves almost the same top-1 accuracy as the baseline model. This evaluation is a proof-of-concept. We will further evaluate this approach for other models and applications in future work.

Algorithm 1 Pseudocode of the proposed framework.

```
1:  procedure FRAMEWORK($\mathcal{D}$, $\mathcal{R}$, $\mathcal{P}$, $\mathcal{M}$)
2:      $m \leftarrow$ MODELGEN($\mathcal{M}$)                          ▷ Initial model $m$
3:      $m^*, p^* \leftarrow m, 0$                                  ▷ Initialise record
4:      while VALID($m$) $\wedge$ ACC($m$) $\geq \mathcal{R}_{acc}$ do     ▷ Check model accuracy
5:          $d \leftarrow$ DESIGNGEN($m$, $\mathcal{P}$)               ▷ Optimised design $d$
6:          $p \leftarrow$ PERF($d$, $m$, $\mathcal{P}$)               ▷ Evaluate performance
7:          if $p \geq \mathcal{R}_{perf} \wedge p > p^*$ then
8:              $m^*, p^* \leftarrow m, p$                          ▷ Update the best record
9:          end if
10:         $m \leftarrow$ MODELGEN($\mathcal{M}$, $m$, $p$)            ▷ Next model
11:     end while
12: end procedure
```

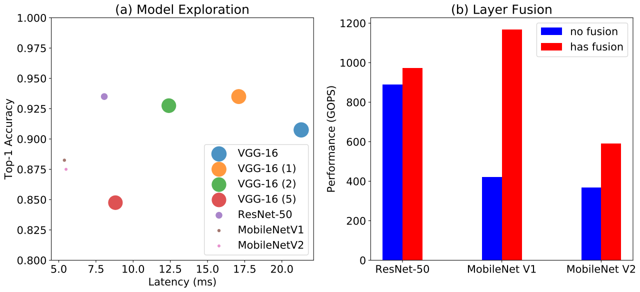

Fig. 10. (a) visualises explored models and (b) shows the improvement of layer fusion on efficient CNN models. The size of a point in (a) shows its relative model size.

B. Final Toolflow

Combining the model optimisation procedure described and the hardware optimisation and generation method in Section III and Section IV, we can deduce the final toolflow for our framework as illustrated in Algorithm 1. This algorithm can jointly explore the design space of CNN model and hardware for efficient inference.

Given $\mathcal{D}$ is the domain-specific dataset, $\mathcal{R}$ are requirements, $\mathcal{P}$ is platform specification, and $\mathcal{M}$ are pre-trained models. This algorithm is driven by the MODELGEN procedure in line 10, which can generate new models from pre-trained models and information from the current iteration, such as the performance p of the intermediate model m. Basically, DESIGNGEN automatically explores the hardware design space regarding m and $\mathcal{P}$ and generates an optimised design d. This design is then evaluated to get performance metrics p. The best record is updated if the p is better than the performance requirement $\mathcal{R}_{perf}$ and the current best p^*. The algorithm terminates when the accuracy is worse than the accuracy requirement $\mathcal{R}_{acc}$. In case we have sufficient training budget, we can loosen the terminating condition in line 4 by removing the accuracy requirement and checking the accuracy until all possible models are searched.

VI. EVALUATION

In this evaluation, we look at the capability of TuRF by generating hardware design for typical efficient CNN models. Then, we evaluate TuRF in terms of model transformation and optimisation by accepting conventional model VGG-16 pre-trained based on a large dataset and generating a set of smaller models with different number of groups replaced. Finally, we compare our approach with previous work.

A. Experimental Setup

The domain-specific application that we select in this evaluation is the flowers classification problem [2] mentioned above. The data representation in the generated hardware designs is quantised to be 16 bit fixed-point, which does not hurt the accuracy of the evaluated application. All the CNN models evaluated here are built, trained and evaluated using the latest TensorFlow (v1.6). Pre-trained models are downloaded directly from TF-Slim. The experimental FPGA platform is Stratix-V 5SGSD8 on a Maxeler MPC-X node, which contains 262.4K adaptive logic modules (ALM), 1963 variable-precision DSP blocks, and 2567 BRAM (M20K). The bandwidth of off-chip data transfer is 38 GB/s. The hardware template prototype is implemented in OpenSPL [28]. MaxCompiler (v2016.1.1) synthesises generated designs.

B. Performance of Efficient CNN Models

We first evaluate the performance of three popular efficient CNN models: ResNet-50, MobileNet V1 and V2 generated by our framework and the results are shown in Table III. Each model is fine-tuned to find the highest attainable top-1 accuracy for flower classification. From the table, ResNet-50 can achieve the best accuracy but it suffers from the worst performance in latency. The network size is also substantially larger when compared to the others. On the other hand, MobileNet V1 is better than the V2 in terms of latency and accuracy with just a minor increase in network size.

We also study the benefits of layer fusion for convolution blocks by analysing the performance in GOPS. The layers within the convolution block are fused in each efficient model. Fig. 10 (b) compares performance, showing the fused designs always outperform the implementations without layer fusion. Layer fusion is particularly effective for MobileNet V1, revealing that depthwise separable blocks can be fused more effectively. It also explains the compelling performance of MobileNet V1 as shown in Table III when compared to the V2. Our framework allows users to choose which models to use, based on their requirements. This involves a repeated execution of the exploration procedure and the model with higher satisfiability for the given requirements will be chosen.

C. Evaluation on Model Optimisation

A pre-trained VGG-16 is used as an input to our framework so as to evaluate its capability to perform model optimisation. The accuracy requirement supplied to the framework is gradually adjusted to generate implementations with different number of groups replaced. This enables us to understand the implications of replacing the standard convolution layer with various types of convolution block in conventional CNN model. Table III shows the results where VGG-16 (1), (2) and (5) imply one, two and five groups are replaced respectively. Essentially, VGG-16 (1) and (2) perform better than the original model in flowers classification regarding the accuracy

TABLE IV. Comparison of VGG-16 and ResNet-50 Performance with prior works

	VGG-16						ResNet-50			
	[9]	[27]	[22]	[10]	[17]	Ours	[22]		Ours (Plain)	Ours (Fused)
Year	2016	2016	2017	2017	2018	2018	2017	2017	2018	2018
FPGA board	ZC706	KU060	GX1150	ZCU102	VCU440	5GSD8	GXA7	GX1150	5GSD8	5GSD8
Tech.	28nm	20nm	20nm	16nm	16nm	28nm	28nm	20nm	28nm	28nm
Clock Freq. (MHz)	150	200	200	200	200	200	150	200	200	200
Bit width	16bit	16bit	16bit	16bit	16bit	16bit	16bit	16bit	16bit	16bit
Max DSP blocks	900	2760	1518	2520	2880	1963	256	1518	1963	1963
Perf. (GOPS)	137.0	266.0	720.2	2941	821.0	1928	250.75	619.13	890.5	973.2

and hardware efficiency. The VGG-16 (5) only showcases minor performance gain with an enormous accuracy drop.

Furthermore, Fig. 10 (a) demonstrates the accuracy versus the latency and size among all models. MobileNet is more suitable for performance-aware applications while ResNet-50 and VGG-16 (2) are more appropriate for accuracy-aware applications. Yet, the model size cannot drop significantly for VGG-16 because most parameters are occupied by FC layers.

D. *Comparison with Previous Work*

To demonstrate the performance of our hardware template, we make a comparison to prior works related to automatic CNN accelerator generation on FPGA. The original pre-trained CNN models, VGG-16 and ResNet-50, are used in this experiment. The convolution layer of our VGG-16 accelerator is not replaced by any convolution blocks to ensure a fair comparison. The Winograd algorithm is applied to reduce the computation complexity. Table IV shows that our approach is better than most of the previous work and is still competitive with [10] in the same technology. Our performance normalised by 16 nm technology (3374 GOPS) is higher than [10] (2941 GOPS). Moreover, to show that layer fusion can be beneficial for efficient convolution blocks, we evaluate our accelerator on ResNet-50. As shown in Table IV, our implementations outperform the ones given in [22], and the fused design can achieve the finest performance. Here the *plain* design is generated with only the Winograd algorithm, and the *fused* design performs layer fusion for all bottleneck blocks.

VII. Conclusion

This paper proposes TuRF, a new CNN optimisation framework inspired by efficient CNN architectures and transfer learning, which supports domain-specific optimisations. The novel aspects include a design template for various convolution blocks, a layer fusion method, and a model optimisation technique which allows layer replacement and fine-tuning of pre-trained CNNs. The proposed approach is capable of producing some of the fastest CNN designs targeting FPGA implementations. Further research includes design space exploration with functional evaluation tools, such as ADAM [29], and extending our approach to support various applications.

Acknowledgements

The support of the United Kingdom EPSRC (grant numbers EP/I012036/1, EP/L00058X/1, EP/L016796/1, EP/N031768/1 and EP/K034448/1), European Union Horizon 2020 Research and Innovation Programme (grant number 671653), Corerain, Intel, Maxeler and the Lee Family Scholarship is gratefully acknowledged.

References

[1] Y. Bengio, "Deep Learning of Representations for Unsupervised and Transfer Learning," *JMLR*, 2011.
[2] "Fine-Tuning TensorFlow Flowers Dataset." [Online]. Available: https://www.tensorflow.org/tutorials/image_retraining#training_on_flowers
[3] N. Tajbakhsh *et al.*, "Convolutional Neural Networks for Medical Image Analysis: Full Training or Fine Tuning?" *IEEE Trans. on Medical Imaging*, 2017.
[4] K. He *et al.*, "Deep Residual Learning for Image Recognition," in *CVPR*, 2016.
[5] A. G. Howard *et al.*, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv:1704.04861*, 2017.
[6] M. Sandler *et al.*, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," *arXiv:1801.04381*, 2018.
[7] A. Lavin and S. Gray, "Fast Algorithms for Convolutional Neural Networks," in *CVPR*, 2016, pp. 4013–4021.
[8] Y. Umuroglu *et al.*, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *FPGA*, 2017, pp. 65–74.
[9] J. Qiu *et al.*, "Going Deeper with Embedded FPGA Platform for Convolutional Neural Network," in *FPGA*, 2016, pp. 26–35.
[10] L. Lu *et al.*, "Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs," in *FCCM*, 2017, pp. 101–108.
[11] S. Han *et al.*, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in *FPGA*, 2017, pp. 75–84.
[12] F. Mamalet and C. Garcia, "Simplifying ConvNets for Fast Learning Simplifying ConvNets for Fast Learning," in *ICANN*, 2012, pp. 58–65.
[13] F. Chollet, "Xception: Deep Learning with Depthwise Separable Convolutions," *arXiv:1610.02357*, 2016.
[14] S. Winograd, *Arithmetic complexity of computations*. Siam, 1980, vol. 33.
[15] U. Aydonat *et al.*, "An OpenCL(TM) Deep Learning Accelerator on Arria 10," in *FPGA*, 2017, pp. 55–64.
[16] J. Yu *et al.*, "Instruction Driven Cross-Layer CNN Accelerator with Winograd Transformation on FPGA," in *ICFPT*, 2017, pp. 227–230.
[17] J. Shen *et al.*, "Towards a Uniform Template-based Architecture for Accelerating 2D and 3D CNNs on FPGA," in *FPGA*, 2018, pp. 97–106.
[18] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," in *ICLR*, 2015, pp. 1–14.
[19] S. I. Venieris and C. S. Bouganis, "fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," in *FCCM*, 2016, pp. 40–47.
[20] C. Zhang *et al.*, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *FPGA*, 2015, pp. 161–170.
[21] Y. Ma *et al.*, "Optimizing Loop Operation and Dataflow in FPGA Acceleration of Deep Convolutional Neural Networks," in *FPGA*, 2017, pp. 45–54.
[22] ——, "An Automatic RTL Compiler for High-Throughput FPGA Implementation of Diverse Deep Convolutional Neural Networks," in *FPL*, 2017, pp. 1–8.
[23] S. W. Williams *et al.*, "Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures," *Commun. ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.
[24] A. Manoj *et al.*, "Fused-Layer CNN Accelerators," in *MICRO*, 2016, pp. 1–12.
[25] Q. Xiao *et al.*, "Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs," in *DAC*, 2017, pp. 1–6.
[26] M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks," *ECCV*, vol. 8689, pp. 818–833, 2014.
[27] C. Zhang *et al.*, "Caffeine: Towards Uniformed Representation and Acceleration for Deep Convolutional Neural Networks," in *ICCAD*, 2016, pp. 1–8.
[28] O. Consortium *et al.*, "OpenSPL: Revealing the Power of Spatial Computing," Tech. Rep., 2013.
[29] H.-C. Ng *et al.*, "ADAM: Automated Design Analysis and Merging for Speeding Up FPGA Development," in *FPGA*, 2018, pp. 189–198.