

# DNNExplorer: A Framework for Modeling and Exploring a Novel Paradigm of FPGA-based DNN Accelerator

Xiaofan Zhang<sup>1\*</sup>, Hanchen Ye<sup>1\*</sup>, Junsong Wang<sup>2</sup>, Yonghua Lin<sup>2</sup>, Jinjun Xiong<sup>3</sup>, Wen-mei Hwu<sup>1</sup>, Deming Chen<sup>1</sup>

<sup>1</sup>University of Illinois at Urbana-Champaign, <sup>2</sup>Easy-visible, <sup>3</sup>IBM Research

{xiaofan3, hanchen8, w-hwu, dchen}@illinois.edu, {junsong.wang, yonghua.lin}@easy-visible.com, jinjun@us.ibm.com

## ABSTRACT

Existing FPGA-based DNN accelerators typically fall into two design paradigms. Either they adopt a generic reusable architecture to support different DNN networks but leave some performance and efficiency on the table because of the sacrifice of design specificity. Or they apply a layer-wise tailor-made architecture to optimize layer-specific demands for computation and resources but loose the scalability of adaptation to a wide range of DNN networks. To overcome these drawbacks, this paper proposes a novel FPGA-based DNN accelerator design paradigm and its automation tool, called DNNExplorer, to enable fast exploration of various accelerator designs under the proposed paradigm and deliver optimized accelerator architectures for existing and emerging DNN networks. Three key techniques are essential for DNNExplorer's improved performance, better specificity, and scalability, including (1) a unique accelerator design paradigm with both high-dimensional design space support and fine-grained adjustability, (2) a dynamic design space to accommodate different combinations of DNN workloads and targeted FPGAs, and (3) a design space exploration (DSE) engine to generate optimized accelerator architectures following the proposed paradigm by simultaneously considering both FPGAs' computation and memory resources and DNN networks' layer-wise characteristics and overall complexity. Experimental results show that, for the same FPGAs, accelerators generated by DNNExplorer can deliver up to 4.2x higher performances (GOP/s) than the state-of-the-art layer-wise pipelined solutions generated by DNNBuilder [1] for VGG-like DNN with 38 CONV layers. Compared to accelerators with generic reusable computation units, DNNExplorer achieves up to 2.0x and 4.4x DSP efficiency improvement than a recently published accelerator design from academia (HybridDNN [2]) and a commercial DNN accelerator IP (Xilinx DPU [3]), respectively.

## 1 INTRODUCTION

Great successes have been achieved by deep neural networks (DNNs) in a massive number of real-life artificial intelligence (AI) applications. Such a success has been driven in part by the continuous improvement of DNN models, using deeper and more sophisticated layer interconnections, to deliver the state-of-the-art solutions [4–11]. With the impressive quality of results comes along DNN models' increasing requirements of more computation and memory. This in turn puts stringent design constraints on any domain-specific hardware accelerator designs to not only deliver high inference

accuracy but also satisfy inference speed, throughput, and energy efficiency.

There are rich studies on customized DNN accelerators that take advantage of different hardware devices, such as exploring kernel optimizations on GPUs and customizing accelerators on ASICs and FPGAs [12–16]. Because of FPGA's flexibility of meeting stringent energy constraints and fast time-to-market, FPGA-based solutions have recently become prominent choices for DNN acceleration with improved latency and energy efficiency [17–19].

By investigating recent FPGA-based solutions, we see two popular accelerator design paradigms: one adopts the generic reusable architectures for all DNN layers; while the other customizes hardware implementation for each DNN layer. When following the first paradigm, such as designs in [2, 20, 21], a generic computation unit is instantiated on FPGA where all DNN layers are processed in a recurrent manner. When adopting the second paradigm, such as designs in [1, 22, 23], a layer-wise pipeline architecture is implemented on FPGA, where each DNN layer is handled by a dedicated pipeline stage. In other words, accelerator designs have better customization for each layer. However, both paradigms may not be suitable for the emerging DNN models where more diverse layer configurations and deeper network structures are involved. The first paradigm has difficulty to optimize DNNs with diverse layers that exhibit vastly different computation-to-communication (CTC) ratios. This will degrade the accelerator performance because of the lack of fine-grained adjustments in such a generic architecture. The second paradigm also has an obvious flaw as it requires hardware instances for each pipeline stage. The more layers in the DNN models, the less resources for each stage, which eventually leads to lower performance.

To address these challenges, we propose DNNExplorer, a framework to model and explore DNN accelerator designs with the goal of producing a balanced DNN solution while keeping the advantages of the two aforementioned popular accelerator design paradigms by addressing their respective flaws. In summary, the main contributions of this paper are as follows.

- (1) We introduce a new FPGA-based accelerator design paradigm with higher-dimensional design space support, better specificity, and improved scalability to effectively address the drawbacks of existing FPGA-based DNN accelerator designs.
- (2) We propose an automation tool called DNNExplorer to enable fast architecture exploration under the proposed paradigm and deliver optimized accelerator designs in three steps as: *model analysis*, *performance modeling*, and *architecture exploration*.
- (3) We define a dynamic design space to capture more detailed configurations of the proposed accelerator paradigm. It helps generate a well-defined design space for architecture exploration to better accommodate different input DNN models and targeted FPGAs.

\*These authors made equal contributions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICCAD '20, November 2–5, 2020, Virtual Event, USA

© 2020 Association for Computing Machinery.

ACM ISBN 978-1-4503-8026-3/20/11...\$15.00  
https://doi.org/10.1145/3400302.3415609

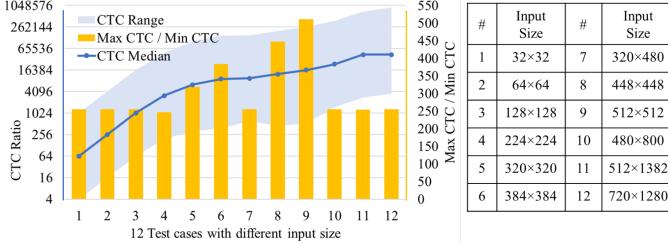


Figure 1: CTC (computation-to-communication) distribution in VGG-16 models (without FC layers) regarding 12 input resolution cases. Inputs are RGB images with 3 depth channels and height and width listed as input size.

(4) We propose a two-level (global and local) automatic design space exploration (DSE) engine to efficiently find optimized accelerator architectures by simultaneously considering both FPGAs' computation and memory resources and DNN networks' layer-wise characteristics and overall complexity.

(5) We experimentally demonstrate that DNNExplorer delivers better FPGA-based DNN accelerators with 4.2x higher throughput compared to the pure pipeline design (DNNBuilder), and 4.4x higher DSP efficiency compared to a generic design (Xilinx DPU).

## 2 RELATED WORK

There are extensive studies in customized DNN accelerator design using FPGAs. To pursue higher energy efficiency, accelerators in [17, 18] incorporate a dynamic data quantization scheme for both DNN parameters and intermediate feature maps to relax the compute and memory demands. The designs in [24, 25] support binary and ternary quantization, which intend to replace the hardware-intensive floating-point multiplications by logical operations. Fast convolution (CONV) algorithms have been investigated to lower the computation complexity and speedup DNN inference, such as using Winograd-based solutions, fast Fourier transform (FFT), and hybrid schemes with both spatial CONV and fast CONV algorithms [2, 26, 27]. A fine-grained layer-based pipeline architecture is proposed by DNNBuilder[1], which works with a resource allocation scheme to lower the end-to-end latency when running real-life DNN applications. Recently, more solutions have been developed with both hardware and software optimizations. For example, authors in [28] apply DNN compression before mapping the DNNs onto FPGAs for higher speedup, while authors in [11, 21] propose hardware-software co-design to accelerate resource-demanding DNN applications for embedded FPGAs.

Recently published literature also focuses on building automation tools for rapid performance estimation and implementation of FPGA-based DNN accelerators. The work in [29] proposes a unified representation for CONV and fully-connected (FC) layers to facilitate the accelerator modeling on targeted FPGAs. A framework proposed in [30] incorporates systolic arrays to improve computing efficiency, while the design in [2] further improves the accelerator design by proposing processing engines with both spatial- and Winograd-CONV support. To help fully explore available resources of the targeted FPGAs, authors in [1] introduce an automatic resource allocation tool to generate fine-grained parallelism guidelines for its architecture. Researchers also employ High-level Synthesis (HLS) in their tools to improve design efficiency of FPGA-based hardware designs [31–35].

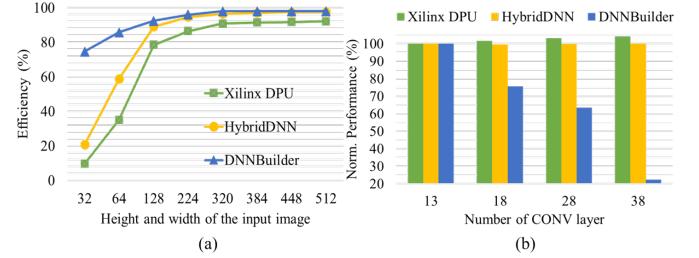


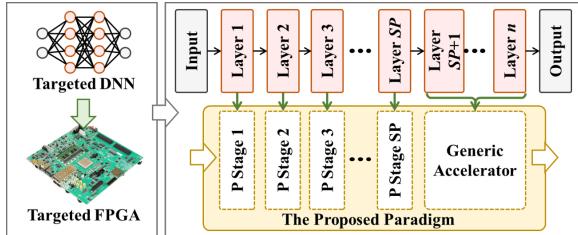
Figure 2: (a) The trends of DSP efficiency when running VGG16 with increasing input sizes in three representative FPGA-based DNN accelerators (batch size = 1); (b) Normalized throughput performance in three accelerators when running VGG-like DNNs with 3×224×224 inputs and 13~38 CONV layers. (The performance of each accelerator is normalized to their baseline cases running the 13-layer DNN.)

## 3 CHALLENGES OF CURRENT SOLUTIONS

By following the first paradigm as mentioned in Section 1, different DNN layers are handled by a generic compute unit. Although the majority of computation in DNNs comes from CONV layers, the differences among CONV layers may be significant. For example, Figure 1 shows 12 models' layer characteristics in terms of their CTC ratio distribution. The medians of CTC show an upward trend along with higher input resolutions. From 32x32 to 512x512 inputs, CTC medians rapidly increase by nearly 256 times following the blue curve, which implies significantly different computation and memory-access patterns. It is also obvious that the CTC range or variations are large even for the same model. If using a generic compute unit to process these layers, we have to accept sub-optimal solutions because of the lack of architecture specificity and limited design spaces. We use DSP efficiency (Eq. 1) to evaluate whether an accelerator is working efficiently, where  $\alpha$  represents the number of multiply-accumulate (MAC) operations handled by one DSP in one clock cycle, i.e.,  $\alpha = 2$  for 16-bit and  $\alpha = 4$  for 8-bit inputs.

$$EFFI_{DSP} = \frac{GOP/s}{\alpha \times DSP_{allocated} \times FREQ} \quad (1)$$

As shown in Figure 2 (a), both Xilinx DPU [3] and HybridDNN [2] (representing the first paradigm) suffer lower DSP efficiency (up to 64.9% and 53.7% degradation, respectively) compared to dedicated designs from DNNBuilder [1] (representing the second paradigm). For accelerators following the second paradigm, separate pipeline stages are instantiated on FPGAs for each major DNN layer and eventually all of them are combined into pipeline implementations. This allows dedicate layer designs according to layers' inherent characteristics (e.g., CTC ratio, computation and memory demands), which enables the possibility of more fine-grained hardware configurations and more in-depth design space exploration. However, its scalability may be easily restricted by the number of DNN layers that it can support, as a deeper DNN means more pipeline stages and less resources for each layer. In this case, performance degradation is expected. Some intuitive examples can be found in Figure 2 (b), where accelerators need to handle 4 VGG-like DNNs with 13~38 CONV layers. The performance of DNNBuilder decreases 77.8% on a 38-layer model compared to the shallower network with 13 CONV layers. In contrast, the generic accelerators maintain a stable performance.



**Figure 3:** The proposed accelerator paradigm to overcome existing drawbacks when mapping DNNs onto FPGAs. It includes  $SP$  pipelined stages for layer 1 ~  $SP$  and a generic accelerator for the rest of layers.  $SP$  is the split-point.

**Table 1: Ratio of CTC variances between the first half ( $V_1$ ) and the second half ( $V_2$ ) of DNNs.**

Network	Input Size	$V_1/V_2$	Network	Input Size	$V_1/V_2$
Alexnet	3x227x227	185.8	ResNet-18	3x224x224	1607.3
GoogLeNet	3x224x224	3622.8	ResNet-50	3x224x224	998.7
InceptionV3	3x299x299	6210.6	SqueezeNet	3x227x227	238.9
VGG-16	3x224x224	489.8	MobileNet	3x224x224	3904.2
VGG-19	3x224x224	552.6	MobileNetV2	3x224x224	251.5

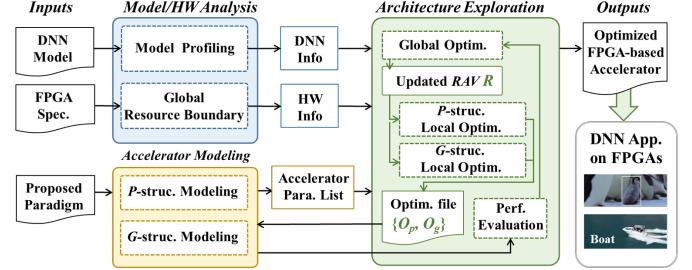
## 4 DNNEXPLORER FRAMEWORK

### 4.1 The Proposed Novel Paradigm

To overcome these challenges, we propose a new accelerator paradigm to capture the essences of both existing paradigms but address their disadvantages. We present the proposed paradigm in Figure 3, to leverage both layer-dedicated and generic reusable designs. For the first part, we implement a pipeline accelerator to generate dedicate layer stages for the first  $SP$  layers. It helps guarantee higher DSP efficiency and more fine-grained resource allocation. For the second part, we adopt a generic architecture for the rest of DNN layers so that it supports deeper DNNs with given resources.

To validate the proposed idea, we investigate 10 popular DNNs and summarize their CTC fluctuations in Table 1. Specifically, we divide every DNN into two halves: the first half covers the bottom part of layers (close to the input layer) with 50% of the total MAC operations while the second half contains the rest of layers. In each half, we calculate the average value of the squared difference between each layer's CTC and the mean CTC. We use  $V_1$  and  $V_2$  to represent these results (variances) in the first and second half, respectively. In Table 1,  $V_1$  is on average 1806.2 times higher than  $V_2$ , which means the first half have more CTC fluctuations. This is a common phenomenon during DNN inference, which causes low DSP efficiency of the FPGA-based accelerators following the first paradigm. In our proposed design, these CTC fluctuations from the first half can be properly resolved by the layer-dedicated pipeline architecture. By considering the second half, where layers have less CTC variances and share higher similarity, a generic structure can successfully eliminate the poor-scalability of the pipeline design.

The proposed design is not a simple concatenation of the two existing paradigms, as the fusion of two heterogeneous structures can directly cause an exponential increase of the design space and easily lead to tedious explorations and sub-optimal solutions. These additional design parameters include the task partitioning scheme, the batch size, and the hardware resource allocation between two structures. Therefore, we propose a highly-efficient design tool, called DNNExplorer, to perform automatic architecture exploration under the proposed paradigm and deliver optimized solutions.



**Figure 4:** The Design flow of DNNExplorer containing 3 steps to generate optimized DNN accelerators given DNN models and FPGA specifications.

### 4.2 DNNExplorer Design Flow

DNNExplorer generates optimized FPGA-based accelerators according to the input DNN model, the targeted FPGA specification, and the proposed accelerator paradigm. As shown in Figure 4, three steps are included in DNNExplorer to provide *Model/HW Analysis*, *Accelerator Modeling*, and *Architecture Exploration*.

In *Model/HW Analysis*, DNN definition files and trained parameters are passed to DNNExplorer for model profiling. The layer-wise information is extracted in this step, such as layer type, layer configuration, computation and memory demands, CTC ratio, quantization scheme, etc, and then packed as DNN info for *Architecture Exploration*. The input also includes a FPGA specification, which helps setup boundaries of available resources, such as DSP, BRAM, and external memory bandwidth.

In *Accelerator Modeling*, we use models corresponding to the pipeline ( $P$ ) and generic ( $G$ ) structure of the proposed paradigm. The goal of this step is to adopt highly-accurate pre-built analytical models for resource utilization and performance estimation. Eventually, a list of configurable accelerator parameters are collected for exploration in the next step. Also, the analytical models contribute to the performance evaluation in *Architecture Exploration*. In-depth introductions of these models are provided in Section 6.

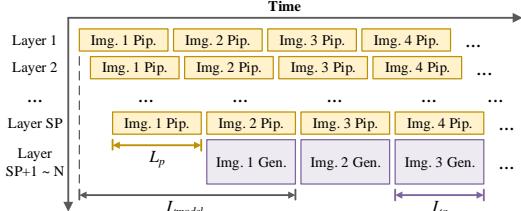
In *Architecture Exploration*, we adopt a divide and conquer strategy and design a two-level automatic DSE engine to efficiently search for optimal accelerators in the massive design space. We use Resource Allocation Vector (RAV), a 5-dim vector  $R$ , to describe resources allocated to the  $P$  and  $G$  structure.  $R$  contains the split-point ( $SP$ ), which indicates task partitioning between these two structures, the batch size ( $Batch$ ), and three major FPGA resources ( $DSP$ ,  $BRAM$ ,  $BW$ ). Given inputs from the last two steps, a global optimization is first performed to explore and update  $R$  in every iteration, providing guidelines for building  $P$  and  $G$  structure, respectively. After that, local optimizations are executed individually for  $P$  and  $G$  structure to explore the best configurations. All selected accelerator parameters are documented on an optimization file for driving the performance evaluation using analytical models in the modeling step. With the performance feedback, the global optimization algorithm can decide whether to continue exploration. Eventually, DNNExplorer generates the optimized architecture.

## 5 ACCELERATOR ARCHITECTURE

### 5.1 Accelerator Overview

In our design, the targeted DNN is mapped into hardware based on the RAV. Given a specific  $R$ , we have

$$R = [SP, Batch, DSP_p, BRAM_p, BW_p] \quad (2)$$



**Figure 5: The overall dataflow of the proposed accelerator**

Layer  $1 \sim SP$  are instantiated into a layer-wise tailor-made structure following a pipeline manner with batch size =  $Batch$  and available resource  $[DSP_p, BRAM_p, BW_p]$ . Denoting the total available global resource on the targeted FPGA is  $[DSP, BRAM, BW]$ , the rest of layers are mapped into a generic reusable structure following a recurrent manner with batch size =  $Batch$  and available resource  $[DSP - DSP_p, BRAM - BRAM_p, BW - BW_p]$ . The pipeline and generic structure share the same clock frequency. We will discuss the selection of RAV in Section 7.

Figure 6 presents the proposed accelerator design, where three major FPGA resources are utilized as computation resources (green area), on-chip memory (blue area), and external memory (orange area). Inputs are first streamed into on-chip buffer from external memory in the first pipeline stage and then intermediate results (DNN feature maps) are passed horizontally through all pipeline stages and reach the feature map buffer of the generic structure. On the other hand, trained DNN parameters are passed vertically with read-only transmission from the external memory. The overall dataflow of the proposed accelerator is shown in Figure 5. We adopt the fine-grained pipeline from [1] to reduce the initial latency when handling layer  $1 \sim SP$  in our pipeline structure. After that, the generic structure starts working on the rest of layers. To reach the maximum throughput performance, we need to balance the latency of each pipeline stage ( $L_p$ ) and the generic structure ( $L_g$ ). With a perfect load-balanced design, we are able to achieve the peak throughput performance as  $1/\max(L_p, L_g)$ . The detailed algorithms of searching such load-balanced design are introduced in Section 7.

## 5.2 The Pipeline Structure

The first part of Figure 6 covers  $SP$  dedicated pipeline stages to process major layers in targeted DNNs, such as convolutional (CONV) layers, pooling (POOL) layers, and fully-connected (FC) layers. These layers are likely to dominate most of the computation and memory resources. Other layers, such as batch normalization (BN) and activation layers are concatenated into the major ones and processed by the same pipeline stage. In each stage, we implement three main components as: (1) a computation engine to carry out DNN layer operations with dedicated parallelism design, (2) a weight buffer to pump in DNN parameters from external memory; and (3) a column/row buffer to cache columns or rows of intermediate results generated by the previous stage. Four configurable parameters are available in every stage, including the channel parallelism factor (CPF) and the kernel parallelism factor (KPF) (which are unrolling factors along input and output dimensions), the input data bit-width (DW), and the weight bit-width (WW). With these parameters, we can generate dedicate accelerator designs for different DNN layers.

**5.2.1 Computation Engine (CE).** As shown in the green parts of Figure 6, CEs handle most of the computations from DNN layers. Inside each CE, we implement a number of processing elements

(PEs) with two-dim parallelism (CPF and KPF). Since both CONV and FC layers share similar computation patterns, we leverage the same PE for these two layers. More specifically, assuming layer stage 1 in Figure 6 works for CONV layers, the CE in this stage contains  $KPF_1$  PEs, and each PE is designed to handle one  $CPF_1$ -length vector multiplication in one clock cycle (assuming  $DW_1 = WW_1 = 16$ ). Once the trained DNN parameters are ready in weight buffer, we will broadcast a  $CPF_1$ -length vector of input feature map to all  $KPF_1$  PEs, and after calculations, the accumulated results ( $KPF_1$ -length vector) will be written to the column/row buffer of the next stage.

**5.2.2 Fine-grained Pipeline & Column-based Buffer.** To ensure lower initial latency and efficient BRAM utilization of the pipeline structure, we adopt the fine-grained layer-based pipeline and column-based cache scheme from [1]. With these technologies, we no longer need to wait for the full completion of intermediate results before continuing to the next pipeline stage. Instead, as shown in Figure 5, the next pipeline stage can be launched once the first few columns or rows of input frame are ready.

## 5.3 The Generic Structure

The second part of the proposed architecture is a generic computation unit for processing layer  $SP + 1 \sim n$ . It is composed of an MAC array, a feature maps buffer, a weight buffer, an accumulation buffer, and a functional sub-module for activation and pooling operations. A controller module is included for the data and instructions transfer and management.

**5.3.1 Reusable MAC Array.** The key component of the generic structure is the callable MAC array, which contains  $CPF_g \times KPF_g$  units of MAC (implemented with DSPs on FPGAs). The MAC array is able to calculate one general matrix-vector multiplication (GEMV) between a  $CPF_g$ -length vector of input feature map and a  $CPF_g \times KPF_g$  matrix of DNN parameters in each clock cycle. The data width of the on-chip memory bus between the MAC array and weight, feature maps, and accumulation buffer is  $CPF_g \times KPF_g \times WW_g$ ,  $CPF_g \times DW_g$ , and  $KPF_g \times DW_g$  bits, respectively, where  $WW_g$  and  $DW_g$  represents the quantization width of weights and feature maps of the generic structure.

**5.3.2 On-chip Buffers Allocation.** Two types of resource on FPGAs (BRAMs and LUTs) can be utilized to allocate on-chip buffers in the generic structure. BRAMs and LUTs have vastly different characteristics in terms of memory capacity and allocation granularity, which makes them suitable for different usage scenarios. Two main strategies are found in previous literature as: (1) allocating BRAMs for feature map and accumulation buffer while LUTs for weight buffer (Xilinx DPU [3]); and (2) allocating BRAMs for all on-chip buffers (VTA [36] and HybridDNN [2]). The first strategy allocates most of BRAMs to the feature map buffer, which can effectively reduce the frequency of loading/saving intermediate feature maps from/to the external memory. Meanwhile, the first strategy allows a flexible adjustment for the depth of the feature map buffer to accommodate the BRAMs constraint of the targeted FPGA. The second strategy allocates most of BRAMs to the weight buffer, allowing controller to schedule computations with more freedom, which can potentially benefit the performance but usually consumes more BRAMs than the first strategy. In DNNEexplorer, we construct resource and performance models (Section 6) for both strategies to better explore the optimized designs.

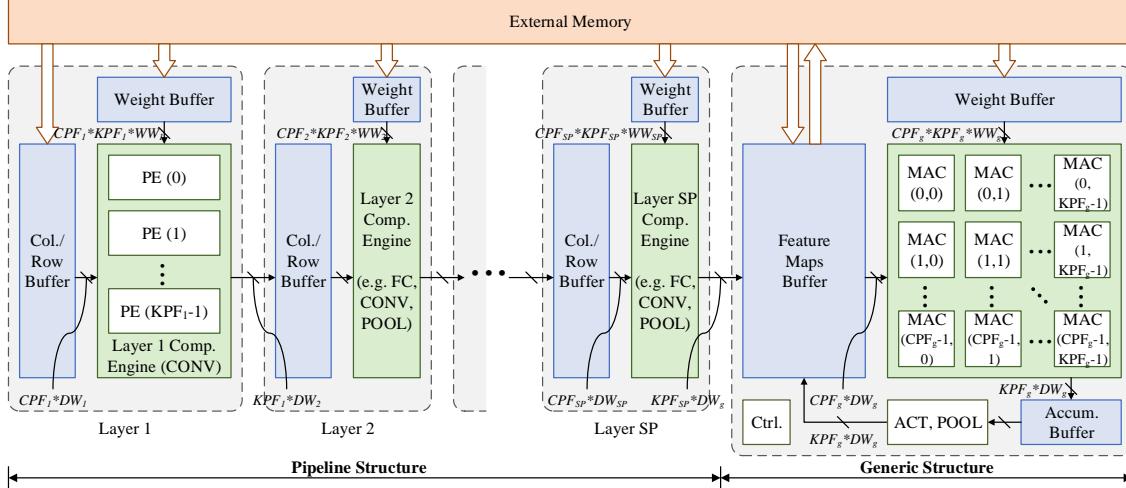


Figure 6: The proposed accelerator architecture with *SP* pipeline stages instantiated for processing the first *SP* DNN layers and a generic computation unit implemented for accelerating the rest of layers.

**5.3.3 Feature Map Partitioning.** Ideally, intermediate feature maps should be fully buffered by on-chip memory for reducing the high-cost data transfer between FPGA and external memory. However, more and more emerging DNN applications are taking high-resolution image or video inputs, demanding a large on-chip buffer capacity, which cannot be accommodated by limited BRAMs on FPGA. To support these emerging applications, we follow a line-based partitioning strategy to break down large feature maps into small groups along the dimension of height. With this strategy, the group that completes computations will be swapped out of the on-chip buffer for other new groups. However, such frequent group swapping occupies additional memory access bandwidth, which can impact the accelerator performance. In Section 6, we will introduce how we capture these negative impacts in our modeling algorithms to help deliver optimized designs with better trade-offs.

## 6 ACCELERATOR MODELING

One of the most critical part in DNNExplorer is the *Accelerator Modeling* that provides fast and accurate estimation of hardware performance and resource utilization. Only with a timely feedback can the decision-making algorithm in *Architecture Exploration* make the most informed decisions. Our modeling supports most of commonly-used DNN layers (e.g., CONV, POOL, and FC layers), and can also be extended to support new DNN layers (e.g., depth-wise CONV layers). Considering CONV is the most computation-intensive operation in modern DNNs, in this section, we will take a DNN model with  $N$  CONV layers as example to illustrate how we provide the pipeline and generic structure modeling. We assume the  $i$ -th CONV layer is calculated with a 3-dim input feature  $D$  (size  $H \times W$  with  $C$  channels) and a 4-dim kernel  $G$  (size  $R \times S$  with  $K$  output and  $C$  input channels) with clock frequency as  $FREQ$ .

### 6.1 Pipeline Structure Modeling

In the pipeline structure, we employ dedicated IPs to carry out the computation of the DNN model following a CTC-based resource allocation algorithm (with detailed discussions in Section 7). The goal of this algorithm is to ensure a load-balanced pipeline structure with perfect match between computation and memory resources for maximal performance. Since a two-dim parallelism factor ( $CPF_i$ ,

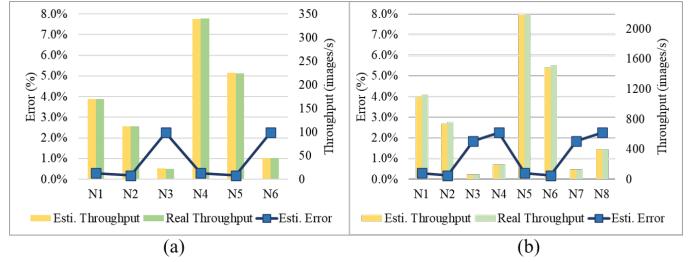


Figure 7: (a) Performance estimation errors of the pipeline structure model when mapping 6 DNNs onto Xilinx ZC706. N1~N3 represent AlexNet, ZF, and YOLO with 16-bit quantization, while N4~N6 represent the same group of networks using 8-bit quantization. (b) We swap the FPGA to Xilinx KU115 and evaluate the estimation errors on 8 DNNs where N1~N4 represent AlexNet, ZF, VGG16, and YOLO with 16-bit quantization while the rest 4 using 8-bit quantization.

$KPF_i$ ) is adopted, the latency  $L_i$  of the  $i$ -th CONV layer is:

$$L_i = \frac{H_i \times W_i \times R_i \times S_i \times C_i \times K_i}{CPF_i \times KPF_i \times FREQ} \quad (3)$$

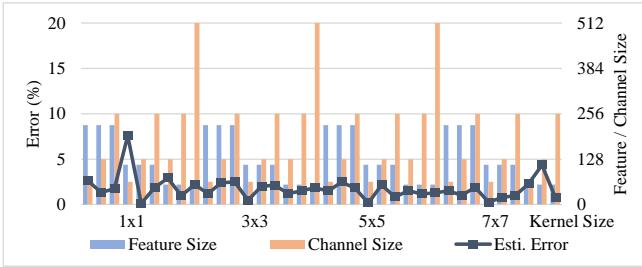
Since the throughput maximum is limited by the pipeline stage with the longest latency, we use Eq. 4 to calculate the overall throughput performance with a batch size of  $Batch$ .

$$Throughput_p = \frac{Batch}{max(L_1, L_2, \dots, L_N)} \quad (4)$$

We summarize the estimated throughput performance and the error of the proposed pipeline model in Figure 7. We evaluate 6 DNN models on an embedded FPGA (Xilinx ZC706) and 8 DNN models on a mid-range FPGA (Xilinx KU115). Results show that the average error between the estimated and the board-level performance is 1.15%, which verifies the accuracy of our proposed analytical model.

### 6.2 Generic Structure Modeling

As we discussed in Section 5, two types of on-chip buffer allocation strategies are considered here. For the first strategy, BRAMs are used for keeping feature map and accumulation buffer while for the second strategy, BRAMs are allocated for all on-chip buffers.



**Figure 8: Performance estimation errors of the generic structure when mapping 36 cases of CONV layers. Test cases cover commonly-used feature map sizes (56, 112, 224), channel sizes (64, 128, 256, 512), and kernel sizes (1, 3, 5, 7).**

**6.2.1 Allocation Strategy 1.** We follow a weight reuse scheme to perform CONV layer. The reuse factor of each weight is determined by the capacity of the accumulation buffer. Assuming the capacity (in bits) of the accumulation buffer is  $CAP_{abuff}$ , the input and output feature map will be organized as  $G_{fm}$  groups where:

$$G_{fm} = \frac{H \times W \times K \times DW_g}{CAP_{abuff}/2}. \quad (5)$$

In Eq. 5,  $CAP_{abuff}$  is divided by a factor of 2 because of using ping-pong buffers to avoid data pollution. Pixels of the feature map inside the same group can reuse the same weights which locate on chip. It means the accelerator needs to fetch weights  $G_{fm}$  times before finishing all computations in this CONV layer. The latency of computation  $L_{comp}$  and weight loading  $L_w$  can be calculated as:

$$L_{comp} = \frac{H \times W \times R \times S \times C \times K}{CPF \times KPF \times FREQ} \quad (6)$$

$$L_w = \frac{R \times S \times C \times K \times WW_g}{BW} \quad (7)$$

In Eq. 7,  $BW$  represents the external memory bandwidth allocated to the generic structure. Then, the overall latency of this CONV layer  $L_{layer}$  is:

$$L_{layer} = \max(L_{comp}, L_w \times G_{fm}) \quad (8)$$

However, due to the feature map partitioning strategy discussed in Section 5.3, the real case is more complicated than Eq. 8. When the feature map buffer is insufficient, feature maps need to be partitioned and swapped between on-/off-chip memory. To capture the impact of this mechanism, we first divide the external memory bandwidth  $BW$  into three portion,  $BW_w$ ,  $BW_{ifm}$ , and  $BW_{ofm}$ , according to the data access behaviors related to weight loading, feature map swapping in, and feature map swapping out, respectively. Then, the latency  $L_{ifm}$  and  $L_{ofm}$  can be calculated as:

$$L_{ifm} = \frac{H \times W \times C \times DW_g}{BW_{ifm}}, \quad L_{ofm} = \frac{H \times W \times K \times DW_g}{BW_{ofm}}, \quad (9)$$

while the weight loading latency  $L_w$  and the overall latency  $L_{layer}$  are updated as:

$$L_w = \frac{R \times S \times C \times K \times WW_g}{BW_w} \quad (10)$$

$$L_{layer} = \max(L_{comp}, L_w \times G_{fm}, L_{ifm}, L_{ofm}) \quad (11)$$

**6.2.2 Allocation Strategy 2.** Similar to strategy 1, to accommodate the capacity of on-chip buffers, we partition input and output feature maps into  $G_{fm}$  groups along the dimension of height, and partition weights into  $G_w$  groups along the dimension of output channels  $K$ . If we use  $CAP_{wbuf}$  to represent the capacity of weight buffer,  $G_w$  can be calculated as:

$$G_w = \frac{R \times S \times C \times K \times WW_g}{CAP_{wbuf}/2} \quad (12)$$

**Table 2: Design space of the proposed paradigm**

	Pipeline Structure	Generic Structure
Parameters	$CPF_p = \{CPF_1, CPF_2, \dots, CPF_t\}$	$CPF_g$
	$KPF_p = \{KPF_1, KPF_2, \dots, KPF_t\}$	$KPF_g$
	<i>Split-point (SP)</i>	<i>Buffer-allocation Dataflow</i>
	<i>Batch</i>	
Constraints	Available $DSP, BRAM, BW$	

Under the second allocation strategy, two types of dataflow are supported as input stationary (IS) and weight stationary (WS). For IS, the computation and memory access behaviors are similar to the case using the first strategy, and the overall latency can be calculated by Eq. 11. However, in this case, because most of BRAMs are allocated to the weight buffer,  $G_{fm}$  is usually larger than using the first strategy. It may damage the overall latency according to Eq. 11. Meanwhile, smaller feature map buffer will prevent the evolution from Eq. 11 to Eq. 8, and increase the external memory bandwidth demands compared to the first strategy. For WS, the proposed accelerator keeps weights on chip, and for each group of weights, all input feature maps need to be loaded before any computations. Since there are totally  $G_w$  groups of weights to be calculated, the overall latency is:

$$L_{layer} = \max(L_{comp}, L_w, L_{ifm} \times G_w, L_{ofm} \times G_w) \quad (13)$$

To evaluate the analytical models of the generic structure, we include 36 cases of CONV layers with different channels, feature map, and kernel configurations as benchmarks. We compare the estimated performance with the measured board-level performance using a Xilinx VU9P FPGA. Results are shown in Figure 8, where only a 2.17% average error is observed across all 36 cases.

## 7 DESIGN SPACE EXPLORATION

### 7.1 Design Space Definition

With the new accelerator design paradigm, we are allowed to explore a significantly larger design space and perform a more fine-grained hardware resource allocation. We define a dynamic design space regarding all possible accelerator design combinations in Table 2. Split-point defines the task partitioning between the pipeline and generic structure. With more layers distributed to the pipeline structure, more stages are instantiated along with higher dimensions of  $CPF_p$  and  $KPF_p$ . Given resource constraints [ $DSP, BRAM, BW$ ] of the targeted FPGA, DNNExplorer is designed to explore all design parameters and generate the best accelerators for targeted DNN applications. For the efficiency of the exploration, we adopt divide and conquer strategy and propose a two-level automatic DSE engine as the global and local optimization, which will be introduced in the next two subsections.

### 7.2 Global Optimization

Global optimization is the first task of the *Architecture Exploration* step, where updated RAV is generated to provide task partitioning and resource allocation guidelines for our unique accelerator architecture. To select the best RAV across high-dimensional design space, we employ a particle swarm optimization (PSO) algorithm to discover the most suitable combination of  $SP$ ,  $Batch$ , and hardware resource distribution between the pipeline and generic structure. The global optimization can be also extended to support other optimization algorithms in the future for different scenarios.

As shown in Algorithm 1, each RAV is considered as a particle  $P_i$ , and all of them contribute to the swarm  $\mathcal{M}$ . We use throughput

---

**Algorithm 1: Global optimization algorithm**


---

```

1: Initialize RAV with  $M$  Population:  $P(M)$ 
2: Initialize iteration number:  $N$ 
3: Initialize HW boundary:  $SP_{max}, DSP_{max}, BRAM_{max}, BW_{max}$ 
4: Evaluate each RAV:  $Fit_i = FitnessScore(P_i)$ , where  $i \in M$ 
5: Keep the local best for each RAV:  $L_i = Fit_i$ , and the global best:  $G = max(L_i)$ 
6: While  $itr < N$ :
7:   For each  $P_i$  in  $M$ :
8:     Get local and global velocity:  $V_{toLbest_i}, V_{toGbest_i}$ 
9:     Update velocity:  $V_i = w \cdot V_i + c_1 \cdot rand() \cdot V_{toLbest_i} + c_2 \cdot rand() \cdot V_{toGbest_i}$ 
10:    Update RAV:  $P_i = UpdatePos(P_i, V_i)$ 
11:    Evaluate updated RAV:  $Fit_i = FitnessScore(P_i)$ 
12:    Update local best:  $L_i$ 
13:    Update global best:  $G$ 
14: Output the best RAV

```

---

performance as the fitness index to evaluate the performance of each particle. We label the local best for particle  $i$  across all iterations as  $L_i$  and use  $G$  to represent the global best particle. The search contains  $N$  iterations and in the  $itr$ -iteration, the position of each particle is updated according to the current position and updated velocity  $V_i$ . The updated velocity is calculated based on the velocity to the local  $V_{toLbest_i}$  and global  $V_{toGbest_i}$  best position. In the update function,  $rand()$  generates random numbers between 0 to 1 and we also include the adjustable parameters as inertia weight,  $w$ , acceleration constants,  $c_1$  and  $c_2$  to fine-tune the search process. Eventually, the best particle is selected, which is the best RAV indicating the optimal task partitioning and resource allocation. For improving the search efficiency of the global optimization, we introduce an early termination feature, which will terminate the optimization if  $G$  is not improved for two continuous iterations.

### 7.3 Local Optimization

Once RAV is updated by the global optimizer, the local optimization will be launched to search for the best parameters (e.g.,  $CPF_p, KPF_p, CPF_g, KPF_g$ , and  $Batch$ ) combination given the constraint of RAV. Considering the flexible architecture of our proposed accelerator enables a huge design space, a brute-force search algorithm indicates an exponentially increasing complexity and is infeasible to find the optimal solution. To solve this problem, we propose a two-phase local optimization algorithm as shown in Algorithm 2 (phase 1) and Algorithm 3 (phase 2). In the first phase, we will calculate  $CPF$  and  $KPF$  of each layer in the pipeline structure based on workload and CTC characteristics, and ensure the computation resource and external memory bandwidth is utilized to the maximum. In the second phase, we increase  $CPF_g$  and  $KPF_g$  step by step until the performance of the pipeline and the generic structure is balanced, which means the overall latency of the generic structure is less or equal to the max latency of the pipeline structure. If the FPGA resources are exhausted under the balanced point, we will roll back and scale down  $CPFs$  and  $KPFs$  of the pipeline structure, until all resource constraints are met. In the second phase, Algorithm 3 will be executed for each on-chip buffer allocation strategy discussed in subsection 5.3.2, and the best one will become the final solution. If the second strategy is selected, the latency update in line 9 of Algorithm 3 will automatically select the better dataflow configuration (IS or WS) for each layer.

## 8 EXPERIMENTAL RESULTS

### 8.1 DSP Efficiency Comparison

To accelerate DNN applications using FPGAs, DSP is the most important and scarce computation resources. In this subsection,

---

**Algorithm 2: CTC-based local optimization algorithm for the pipeline structure**


---

```

1:  $[SP, Batch, DSP_p, BRAM_p, BW_p] = RAV$ 
2: Initialize latency and resource model,  $l_{pip}()$  and  $r_{pip}()$ , for pipeline structure.
3: Calculate operation number  $OP_i$  and computation reuse factor  $CTC_i$  for each
layer of the targeted DNN model.
4:  $BW_{total}^{norm} = \sum_{i=1}^{SP} (OP_i / CTC_i)$ 
5: for  $i$  in  $[1, SP]$ :
6:    $PF_i = [OP_i \times BW_p / BW_{total}^{norm} / FREQ]$ 
7: while  $(\sum_{i=1}^{SP} R_i \geq [DSP_p, BRAM_p])$ :
8:   for  $i$  in  $[1, SP]$ :
9:      $PF_i = max(1, PF_i / 2)$ , update  $CPF_i$  and  $KPF_i$ .
10:    Update  $L_i$  and  $R_i$  with model  $l_{pip}()$  and  $r_{pip}()$ .

```

---

**Algorithm 3: Balance-oriented local optimization algorithm for the generic structure**


---

```

1: Initialize latency and resource model,  $l_{gen}()$  and  $r_{gen}()$ , for generic structure.
2: Initialize  $R_{total}$  with the total resource boundary of the targeted FPGA.
3: while (1):
4:    $PF_g = 1, L_p^{max} = max(L_1, L_2, \dots, L_{SP})$ 
5:   while  $(\sum_{i=SP+1}^N L_i > L_p^{max} \text{ and } R_g + \sum_{i=1}^{SP} R_i < R_{total})$ :
6:      $PF_g = PF_g \times 2$ , update  $CPF_g$  and  $KPF_g$ .
7:     Update  $R_g$  with resource model  $r_{gen}()$ .
8:     for  $i$  in  $[SP+1, N]$ :
9:       Update  $L_i$  with latency model  $l_{gen}()$ .
10:       $Batch_p = (R_{total} - R_g) / \sum_{i=1}^{SP} R_i$ 
11:      if  $(\sum_{i=SP+1}^N L_i > L_p^{max} \text{ or } Batch > Batch_p)$ :
12:        for  $i$  in  $[1, SP]$ :
13:           $PF_i = max(1, PF_i / 2)$ , update  $CPF_i$  and  $KPF_i$ .
14:          Update  $L_i$  and  $R_i$  with model  $l_{pip}()$  and  $r_{pip}()$ .
15:        else : break

```

---

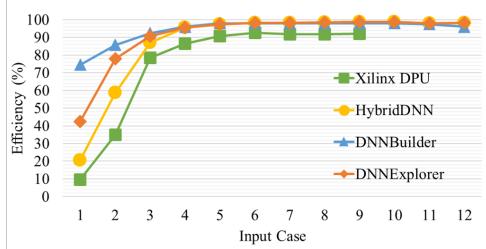
we use DSP efficiency as criteria to evaluate whether accelerators maximize the usage of allocated DSPs. We compare our design to two other automatic accelerator design frameworks, DNNBuilder [1] and HybridDNN [2], by targeting the same task: to deliver DNN accelerators on a Xilinx KU115 FPGA corresponding to 12 VGG-16 (without the last three FC layers) models with different input sizes (as the same cases in Figure 1) to simulate tasks of real-life DNN applications. In our experiments, weights and feature maps are quantized to 16-bits fixed-point in all three frameworks for equality. Results are shown in Figure 9, where DNNBuilder achieves the highest DSP efficiency especially targeting small size inputs (e.g., case 1 and 2) because of its dedicated pipeline stage design. Our proposed design is slightly behind DNNBuilder when targeting small inputs but we then reach the same efficiency level (>95%) after case 3. Compared to the generic accelerator design in HybridDNN, our design can deliver 2.0x and 1.3x higher efficiency for case 1 and 2, respectively. We also compare to Xilinx DPU [3], a commercial DNN accelerator IP, for running first 9 cases on a ZCU102 FPGA (as the last three inputs are not supported by this IP). The accelerators generated by DNNExplorer can achieve an average 1.6x higher DSP efficiency, peaking at 4.4x for case 1. As the input size increases, the efficiency gap decreases (<10% after case 5).

### 8.2 Performance Comparison

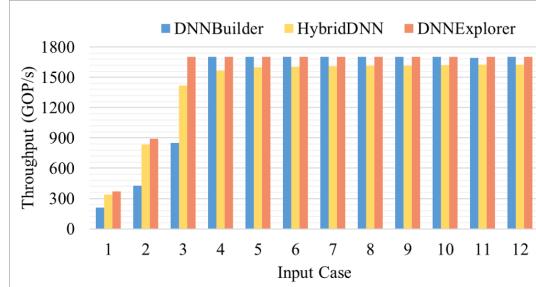
We compare the throughput (GOP/s) among accelerators generated by the proposed DNNExplorer, DNNBuilder, and HybridDNN. First, we target the same DNNs mentioned in the last subsection using Xilinx KU115 FPGA with 200MHz clock frequency. Performance results are listed in Figure 10, where accelerators generated by DNNExplorer achieve better throughput performance compared to the state-of-the art solutions. To meet resource constraints and

**Table 3: Performance and resource overhead of the DNNExplorer-generated accelerators with batch size = 1.**

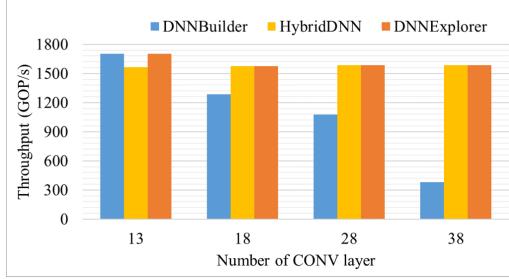
Case	Input Size	GOP/s	Img./s	R = [SP, DSP, BRAM, BW]	Total DSP	DSP Efficiency	Total BRAM	Avg. Search Time (s)
1	3x32x32	368.5	588.9	[4, 48.2%, 17.7%, 62.9%]	2268	42.3%	2326	86.6
2	3x64x64	890.8	339.1	[5, 46.5%, 13.5%, 65.9%]	2730	77.9%	2560	41.6
3	3x128x128	1702.3	169.5	[9, 50.2%, 38.9%, 54.2%]	4686	90.8%	3589	70.2
4	3x224x224	1702.3	55.4	[12, 63.6%, 53.7%, 67.3%]	4444	95.8%	3296	75.6
5	3x320x320	1702.4	27.1	[13, 73.5%, 71.4%, 63.5%]	4450	95.7%	3224	45.9
6	3x384x384	1702.4	18.8	[14, 73.2%, 70.1%, 55.0%]	4452	95.6%	3436	64.9
7	3x320x480	1702.4	18.1	[14, 78.6%, 74.6%, 58.2%]	4452	95.6%	3296	71.1
8	3x448x448	1702.4	13.8	[13, 75.8%, 72.9%, 36.1%]	4450	95.6%	3552	82.1
9	3x512x512	1702.4	10.6	[13, 80.0%, 70.0%, 80.0%]	4450	95.6%	3678	70.0
10	3x480x800	1702.4	7.2	[13, 80.0%, 72.0%, 80.0%]	4450	95.6%	3678	103.0
11	3x512x1382	1702.5	3.9	[14, 77.4%, 73.1%, 34.3%]	4452	95.6%	3792	105.5
12	3x720x1280	1702.5	3.0	[13, 82.7%, 73.4%, 84.0%]	4450	95.6%	4186	143.9



**Figure 9: DSP efficiency comparison when running VGG16 (batch size = 1) with 12 different input sizes.**



**Figure 10: Throughput comparison when running VGG16 (batch size = 1) with 12 different input sizes.**



**Figure 11: Throughput comparison when running deeper DNNs with the same  $3 \times 224 \times 224$  input.**

maximize performance, the proposed DSE engine generates configuration guidelines for implementing the proposed architecture.

Detailed results are shown in Table 3, where  $R$  indicates the most suitable task partition scheme and resources allocated to the  $P$  and  $G$  structures of the proposed accelerator. The total DSP and total BRAM represent the compute and on-chip memory utilization. To capture the runtime of searching the optimal architecture given input model and hardware constraints, we perform five independent searches using Intel i5-650 CPU for each case and list the average search time. With the divide-and-conquer idea and the early termination feature, DNNExplorer delivers the best configurations in

**Table 4: Performance and resource overhead of the generated accelerators without batch size restrict.**

Case	Input Size	Batch	GOP/s	Img./s	DSP	BRAM
1	3x32x32	8	1698.1	2712.7	4464	2228
2	3x64x64	8	1701.5	678.2	4688	3534
3	3x128x128	4	1702.4	169.5	4686	3326
4	3x224x224	2	1702.3	55.4	4686	3619

minutes regardless the high-dimensional design space. We extend the search for batch size in Table 4, where lower DNN complexity (caused by smaller inputs in case 1 to 4) creates opportunities for batch size exploration given limited available FPGA resources.

We also evaluate the scalability of DNNExplorer by targeting deeper DNNs. We prepare 4 DNNs, with 13, 18, 28, and 38 CONV layers. The 13-layer one comes directly from VGG16 by removing the last 3 FC layers. Since VGG is composed of 5 CONV groups, where each group has the same CONV configurations (e.g. number of CONV kernels), we add one CONV layer to each group (maintaining the same configurations) and get the 18-layer (13+5) model. Similarly, we add 3 and 5 CONV layers to each part for the 28- and 38-layer model, respectively. In Figure 11, our designs maintain the highest performance despite targeting deeper networks. Compared to DNNBuilder, we can deliver 4.2x higher performance for accelerating a 38-layer VGG-like DNN on the same FPGA platform.

## 9 CONCLUSION

In this paper, we presented a novel FPGA-based DNN accelerator paradigm with both dedicated pipeline structure and generic reusable structure to overcome the drawbacks of existing designs by providing improved performance, better specificity, and scalability. To achieve the full potential of the new paradigm, we proposed DNNExplorer, an automation tool to enable fast architecture exploration following the new paradigm. Novel technologies were proposed which included a dynamic design space for more fine-grained architecture configurations and a two-level automatic DSE engine for delivering optimized designs. With the above novel designs, DNNExplorer achieved 4.2x higher performance compared to the state-of-the-art design produced from DNNBuilder when targeting a 38-layer VGG-like DNN on the same FPGA. We also achieved higher DSP efficiency (up to 4.4x) compared to the latest generic architecture accelerators from academia (HybridDNN) and industry (Xilinx DPU).

## ACKNOWLEDGMENT

This work was supported in part by the IBM-Illinois Center for Cognitive Computing System Research (C<sup>3</sup>SR) – a research collaboration as part of IBM AI Horizons Network. Xiaofan Zhang is supported by a Google PhD Fellowship.

## REFERENCES

- [1] Xiaofan Zhang et al. DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, 2018.
- [2] Hanchen Ye et al. HybridDNN: A framework for high-performance hybrid DNN accelerator design and implementation. In *Proceedings of the Design Automation Conference (DAC)*, 2020.
- [3] Xilinx. Zynq dpu v3.1, 2019. Accessed: 2020-5-23.
- [4] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.
- [5] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.
- [6] Christian Szegedy et al. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2015.
- [7] Kaiming He et al. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016.
- [8] Joseph Redmon et al. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2016.
- [9] Barret Zoph et al. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2018.
- [10] Esteban Real et al. Regularized evolution for image classifier architecture search. In *AAAI conference on Artificial Intelligence (AAAI)*, 2019.
- [11] Xiaofan Zhang et al. SkyNet: a hardware-efficient method for object detection and tracking on embedded systems. In *Conference on Machine Learning and Systems (MLSys)*, 2020.
- [12] Dustin Franklin. NVIDIA Jetson AGX Xavier delivers 32 teraops for new era of AI in robotics. *NVIDIA Accelerated Computing/Parallel For all*, 2018.
- [13] Norman P Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of International Symposium on Computer Architecture (ISCA)*, 2017.
- [14] Yu-Hsin Chen et al. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *IEEE International Solid-State Circuits Conference (ISSCC)*, 2016.
- [15] Chen Zhang et al. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015.
- [16] Pengfei Xu et al. AutoDNNchip: An automated DNN chip predictor and builder for both FPGAs and ASICs. 2020.
- [17] Jiantao Qiu et al. Going deeper with embedded FPGA platform for convolutional neural network. In *Proceedings of International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.
- [18] Xiaofan Zhang et al. High-performance video content recognition with long-term recurrent convolutional network for FPGA. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2017.
- [19] Xiaofan Zhang et al. Machine learning on FPGAs to face the IoT revolution. In *Proceedings of International Conference on Computer-Aided Design (ICCAD)*, 2017.
- [20] Jialiang Zhang and Jing Li. Improving the performance of opencl-based fpga accelerator for convolutional neural network. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.
- [21] Cong Hao et al. FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge. In *Proceedings of the Design Automation Conference (DAC)*, 2019.
- [22] Huimin Li et al. A high performance fpga-based accelerator for large-scale convolutional neural networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2016.
- [23] Xuechao Wei et al. TGPA: tile-grained pipeline architecture for low latency cnn inference. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, 2018.
- [24] Ritchie Zhao et al. Accelerating binarized convolutional neural networks with software-programmable FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.
- [25] Junsong Wang et al. Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA. In *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL)*, 2018.
- [26] Qingcheng Xiao et al. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on fpgas. In *Proceedings of the Design Automation Conference (DAC)*, 2017.
- [27] Chuanhao Zhuge et al. Face recognition with hybrid efficient convolution algorithms on FPGAs. In *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI)*, 2018.
- [28] Song Han et al. ESE: Efficient speech recognition engine with sparse lstm on FPGA. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.
- [29] Chen Zhang et al. Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(11):2072–2085, 2018.
- [30] X. Wei et al. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *DAC*, 2017.
- [31] Kyle Rupnow et al. A study of high-level synthesis: Promises and challenges. In *IEEE International Conference on ASIC*, 2011.
- [32] Xinheng Liu et al. High level synthesis of complex applications: An h. 264 video decoder. In *Proceedings of International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.
- [33] Yijin Guan et al. Fp-dnn: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *Proceedings of the International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017.
- [34] Qin Li et al. Implementing neural machine translation with bi-directional GRU and attention mechanism on FPGAs using HLS. In *Proceedings of Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2019.
- [35] Yao Chen et al. Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs. In *Proceedings of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.
- [36] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, et al. A hardware-software blueprint for flexible deep learning specialization. *IEEE Micro*, 39(5):8–16, 2019.