Being-ahead: Benchmarking and Exploring Accelerators for Hardware-Efficient AI Deployment

Xiaofan Zhang, Hanchen Ye, Deming Chen
University of Illinois at Urbana-Champaign
{xiaofan3, hanchen8, dchen}@illinois.edu

ABSTRACT

Customized hardware accelerators have been developed to provide improved performance and efficiency for DNN inference and training. However, the existing hardware accelerators may not always be suitable for handling various DNN models as their architecture paradigms and configuration tradeoffs are highly application-specific. It is important to benchmark the accelerator candidates in the earliest stage to gather comprehensive performance metrics and locate the potential bottlenecks. Further demands also emerge after benchmarking, which require adequate solutions to address the bottlenecks and improve the current designs for targeted workloads. To achieve these goals, in this paper, we leverage an automation tool called DNNExplorer [1] for benchmarking customized DNN hardware accelerators and exploring novel accelerator designs with improved performance and efficiency. Key features include (1) direct support to popular machine learning frameworks for DNN workload analysis and accurate analytical models for fast accelerator benchmarking; (2) a novel accelerator design paradigm with high-dimensional design space support and fine-grained adjustability to overcome the existing design drawbacks; and (3) a design space exploration (DSE) engine to generate optimized accelerators by considering targeted AI workloads and available hardware resources. Results show that accelerators adopting the proposed novel paradigm can deliver up to 4.2× higher throughput (GOP/s) than the state-of-the-art pipeline design in [2] and up to 2.0× improved efficiency than the recently published generic design in [3]) given the same DNN model and resource budgets. With DNNExplorer's benchmarking and exploration features, we can be ahead at building and optimizing customized AI accelerators and enable more efficient AI applications.

1 INTRODUCTION

DNNs have been achieving great successes in facilitating a wide range of artificial intelligence (AI) applications. With the continuous improvement of neural network models, DNNs begin to adopt more sophisticated and efficient layer interconnections and deliver the state-of-the-art solutions for popular domains such as computer vision [4–11] and natural language processing [12–17]. Meanwhile, the impressive quality of results comes with the increasing compute and memory demands, which rely on building customized hardware accelerators to accommodate the DNN models and deliver high inference accuracy and satisfy inference speed, throughput, and energy efficiency.

There are rich studies on customized DNN accelerators that take advantage of different hardware architectures for emerging AI applications, such as the neural processing unit [18], DianNao series [19], Eyeriss [20], and the tensor processing unit (TPU) [21]. In addition, recently published literature also focuses on building customized accelerators on FPGAs for more flexible configurations, faster time-to-market, and improved latency and energy efficiency [2, 3, 22–25]. By investigating these designs, we see two popular architecture paradigms: as the one uses customized implementations for DNN layers and forms a layer-wise pipeline architecture with dedicated pipeline stage handling each DNN layer [2, 26, 27]; while the other adopts a generic reusable architecture that all DNN layers are processed recurrently [3, 20, 21, 28, 29].

By following different architecture paradigms and design combinations, customized accelerators may present completely different performances when handling emerging DNN models, where more diverse layer configurations and deeper network structures are involved. For example, the first paradigm has an obvious flaw as it requires dedicated hardware instances for handling each pipeline stage. The more layers in the DNN models mean the fewer resources for each stage by considering the limited resource budget. The second paradigm has difficulty optimizing DNNs with various layers that exhibit vastly different arithmetic intensity because all layers share the generic architecture.

It will be beneficial to have an efficient tool for benchmarking hardware accelerators with quantitative evaluations in the early stage of accelerator development. In this paper, we propose such a tool called DNNExplorer [1]. It enables an efficient accelerator design process through accelerator benchmarking and exploration features to accommodate demanding AI applications. Contributions of this work are summarized as follows: (1) We propose an automation tool that provides seamless connections to popular machine learning frameworks (e.g., Caffe [30] and PyTorch [31]), where DNNs developed by these frameworks can be used to benchmark customized accelerators. It can accurately evaluate the performance of accelerators following both layer-based pipeline and generic reusable architecture. (2) We introduce a novel architecture paradigm to overcome the respective drawbacks of the aforementioned popular paradigms. This unique paradigm enables high-dimensional design space as well as fine-grained adjustability and better scalability, which help deliver even better-customized accelerators. (3) To efficiently explore the high-dimensional design space, a two-level design space exploration (DSE) engine is proposed to generate optimized accelerator configurations. Results show that the proposed novel paradigm delivers up to 4.2× higher throughput than the pure pipeline accelerator [2] and up to 2.0× higher efficiency than the generic accelerator design [3]) when targeting the same DNN and hardware.

2 RELATED WORK

With the diverse DNN workloads and various hardware accelerator designs, it is critical to understand how these different workloads perform on particular hardware configurations. Building performance benchmarks is one of the most effective approaches to distinguish competing accelerator designs. To achieve this goal, microbenchmarks are developed (e.g., DeepBench [32]) to measures the performance of basic DNN operations. Further efforts have been made to deliver system-level benchmarking. For example, TensorFlow Benchmarks collect popular image classification models to evaluate the throughput performance during DNN training [33]; and Training Benchmarks for DNNs (TBD) provide GPU evaluation when handling DNNs for computer vision and natural language processing [34]. To benchmark both hardware and software, DAWNBench is built to evaluate the performance of hardware and the accuracy of DNN models [35]. Following the same idea, recently published benchmarks start leveraging more diverse tasks and developing more fair comparisons for different hardware setups. To achieve this goal, MLPerf is published through the joint efforts of academia and industry, which aims at providing unbiased evaluations of DNN training and inference performance [36].

In addition, researchers are keen on developing efficient performance modeling and hardware design methodologies to deliver desired customized accelerators for emerging AI applications. One of the popular research directions is building hardware accelerators from a higher abstraction level, such as behavioral level instead of register-transfer level to improve design efficiency [37, 38]. In [23], a customized accelerator for image captioning is implemented on FPGA using high-level synthesis (HLS). Following the HLS-based design, more accelerators have been developed to meet the needs of various AI applications, such as accelerating image classification [25, 39], face recognition [40, 41], object detection [29, 42], and language translation [43–45].

The other direction is to build automation design frameworks that provide systemic solutions to leverage the critical steps of DNN accelerator design, such as performance evaluation, architecture optimization, and fast prototyping. A framework published in [46] introduces a systolic array based accelerator design along with an analytical model to estimate performance and resource utilization. In [47], a unified representation for convolutional (CONV) and fully-connected (FC) layers is proposed for accelerator modeling. To improve accelerator design and optimization, DNNBuilder is proposed to provide an end-to-end automation tool for building and prototyping high-quality accelerators [2]. It introduces a configurable fine-grained pipeline architecture and optimization algorithms to deliver real-time DNN inference even for resource-constraint hardware. In [48], accelerators generated by the proposed framework can also target extremely low bit-width DNNs where binary and ternary quantization schemes are adopted to replace the hardware-intensive floating-point multiplications by logical operations. Recently developed frameworks include more comprehensive hardware modeling methods, such as covering both spatial- and Winograd-CONV [3], targeting accelerators for both FPGAs and ASICs [49], and supporting multi-branch DNNs for virtual reality applications [50].

Efficient approaches to benchmark and evaluate hardware accelerators also bring significant benefits to the emerging DNN-accelerator co-design strategies [11, 29, 51, 52]. For example, authors in [29] introduce a co-design method to simultaneously develop a hardware-efficient DNN model and high-performance customized accelerator given predefined applications and hardware constraints. One of the critical factors is the timely hardware performance feedback, which continuously provides design guidelines. Similarly, SkyNet is developed by fully considering the hardware constraints of both FPGAs and GPUs to deliver more efficient solutions to handle object detection and tracking [11].

3 THE OVERALL FLOW OF DNNEXPLORER

DNNExplorer is an automation framework that helps close the gap between fast DNN construction in software and slow AI hardware deployment. It can be fed in DNN models described by the high-abstraction network definition files (e.g., caffe prototxt files and PyTorch network forward functions) and deliver optimized customized accelerators by considering various user-defined constraints, such as resource budgets, accelerator architecture paradigms, and targeted hardware performance. As shown in Figure 1, DNNExplorer features three major steps as model/hardware analysis, accelerator modeling, and benchmarking and exploration.

In step 1, DNN definition files are passed to DNNExplorer as inputs for model profiling. The layer-wise information is extracted, such as layer type, layer configuration, computation and memory demands, arithmetic intensity, quantization scheme, etc. The targeted hardware specifications are also entered to help setup resource budgets. For FPGA implementation, DNNExplorer captures three major resources as DSP, BRAM, and external memory bandwidth.

During step 2, DNNExplorer adopts three accelerator models corresponding to two popular paradigms (paradigm 1: the layer-based pipeline architecture [2, 26, 27] and paradigm 2: the generic reusable architecture [3, 28, 29]) and one proposed hybrid paradigm (paradigm 3). This step aims to adopt highly-accurate pre-built analytical models for hardware resource utilization and performance estimation. These models, along with the step 1 outputs, are then passed to step 3 for architecture optimization, benchmarking, and exploration.

In step 3, optimization is performed to generate configuration guidelines. For accelerator designs following paradigm 1 or 2, respective optimization procedures (Section 4.3) are activated to configure the accelerator and attempt to achieve maximum performance given resource constraints. DNNExplorer then starts accelerator benchmarking by evaluating the optimized accelerator using the targeted DNN model. For designs following the proposed paradigm 3, both layer-based pipeline architecture and generic reusable architecture are involved, creating more diverse configurations for achieving better performance. However, it also creates an enormous design space and is challenging to search for the most suitable configuration. An architecture exploration function is proposed to address these difficulties (Section 5). This function contains a two-level automatic DSE engine following a divide-and-conquer strategy to determine task and resource partitioning schemes for both pipeline and generic architecture at the first level optimization and explore their respective optimal configuration with given resources at the second level optimization. Eventually, DNNExplorer can effectively benchmark customized accelerators and explore novel architectures to deliver improved AI acceleration on given hardware.

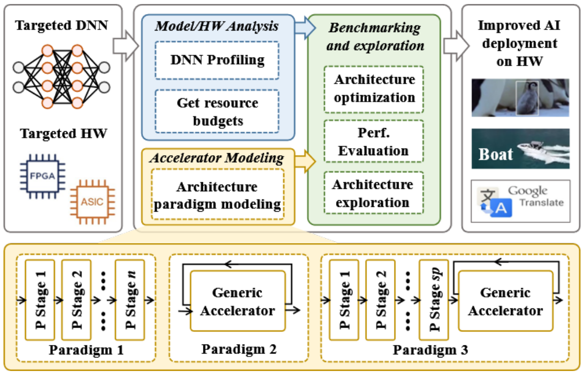


Figure 1: DNNExplorer introduces an complete flow for customized accelerator benchmarking and exploration to deliver improved AI hardware deployment.

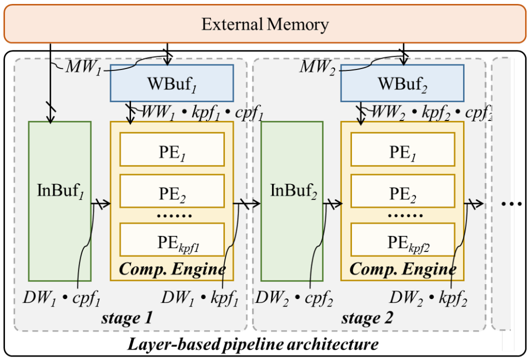


Figure 2: The layer-based pipeline architecture with dedicated pipeline stages for major DNN layers dominating computation and memory consumption.

4 DNNEXPLORER FOR ACCELERATOR BENCHMARKING

The first two paradigms shown in Figure 1 can be benchmarked after respective architecture optimizations. We adopt optimization strategies published in DNNBuilder [2], such as the fine-grained pipeline and the column-based cache scheme to ensure the pipeline architecture is fully optimized. Regarding the generic architecture, we follow the optimization designs from HybridDNN [3] to enable a hybrid CONV processing engine (for spatial and Winograd CONV) and multi-dataflow support covering input stationary (IS) and weight stationary (WS). For the third paradigm, we need one more step as architecture exploration by using the proposed two-level DSE engine to deliver optimized hardware configuration, which will be introduced in Section 5

4.1 Pipeline architecture overview

As shown in Figure 2, each stage is designed for accommodating major layers of the targeted DNN, such as CONV, Pooling (POOL), and fully-connected (FC) layers. Other layers, such as batch normalization (BN) and activation layers, are concatenated into the major ones and processed by the same pipeline stage. During implementation, three types of resources are required: the computation resources for building computation engines (CEs), the on-chip memory for implementing input and weight buffers, and the external memory for keeping DNN parameters. Configurable parameters are available in every stage, including channel parallelism factor (*CPF*), kernel parallelism factor (*KPF*), the input data bitwidth (DW), the weight bit-width (WW), and the bit-width for external memory access (MW). Among these parameters,

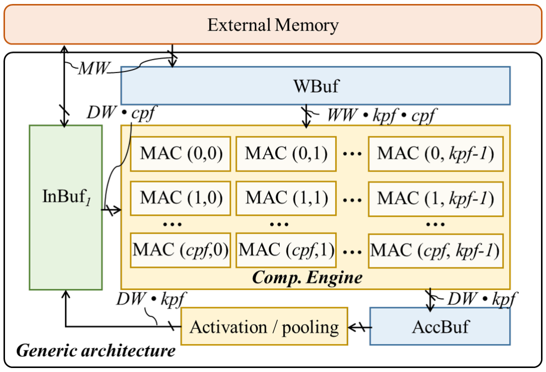


Figure 3: The reusable generic architecture adopted by DNNExplorer.

CPF and KPF are the unrolling factors along input and output dimensions, representing the number of input channels and the number of kernels being processed in parallel. CPF and KPF are calculated by the resource allocation algorithm, which will be introduced in Subsection 4.3.1.

Following this paradigm, the Computation Engine (CE) is responsible for handling most of the computations. Inside each CE, there are processing elements (PEs) with two-dim parallelism (CPF and KPF). More specifically, assuming stage 1 in Figure 2 works for the CONV layer, 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. Once the DNN parameters are ready in weight buffer, we broadcast a CPF_1-length vector of input feature map to all KPF_1 PEs. After calculations, results (KPF_1-length vector) will be written to the input buffer of the next stage.

In addition, to ensure lower initial latency and efficient on-chip memory utilization of the pipeline architecture, we adopt the fine-grained layer-based pipeline and column-based cache scheme [2]. With these technologies, we no longer need to wait for the full completion of intermediate results before continuing to the next pipeline stage.

4.2 Generic architecture overview

As shown in Figure 3, the key component is a reusable MAC array, which is composed of $CPF_g \times KPF_g$ MAC units. Three on-chip buffers (feature map buffer, weight buffer, and accumulation buffer) are allocated for pumping data to the MAC array and storing the intermediate results. Similar to the pipeline architecture, DW_g, WW_g, and MW_g denote the bit-width of input data, weight, and external memory bus. A functional module is instantiated between the accumulation buffer and feature map buffer for handling pooling and activation layers. In each clock cycle, the MAC array consumes a CPF_g-length vector of input feature map and a $CPF_g \times KPF_g$ matrix of DNN parameters and calculates one general matrix-vector multiplication (GEMV). The intermediate results are stored and accumulated in the accumulation buffer until all associated calculations are completed.

In paradigm 2, the buffer allocation strategy becomes one of the most important design choices. Allocating a large feature map buffer will effectively reduce the frequency of loading/saving intermediate feature maps from/to the external memory. Allocating a large weight buffer will provide more freedom for the computation scheduling, potentially benefiting the performance. DNNExplorer supports a flexible on-chip buffer allocation strategy to enable a comprehensive exploration in the trade-off design space. In addition, there are two supported data reuse strategies as Input Stationary (IS) and Weight Stationary (WS). For IS, input feature maps are partitioned into multiple groups along the height dimension and transferred from the external memory to the on-chip buffer by groups. One group of input feature maps will be kept on-chip and reused until all associated computations are completed. For WS, weights are partitioned along the output channel dimension, and all input feature maps are streamed in for each group of weights.

4.3 Accelerator modeling and optimization

One of the important features to enable accelerator benchmarking is the fast and accurate estimation of hardware performance and resource utilization. In this work, we adopt highly-accurate analytical models to generate estimated performance and resource utilization based on the DNN layer-wise information and optimized hardware configurations. Our tool supports most of the commonly-used DNN layers (e.g., CONV, POOL, and FC layers) and can be extended to support more emerging DNN layers. In this section, we take a n-layer convolutional neural network as an example and assume the i-th CONV layer with a 3-dim input feature D_i (size $H_i \times W_i$ with $CHin_i$ channels) and a 4-dim kernel G_i (size $R_i \times S_i$ with $CHout_i$ output and $CHin_i$ input channels). For evaluating the performance, we assume $FREQ$ as the working clock frequency.

4.3.1 The pipeline architecture. When configuring the pipeline architecture, we follow a particular resource allocation guideline for maximizing the overall hardware performance. Assuming the computation demand of DNN layer i is C_i, the increase of allocated resource R_i for layer i results in a proportional increase in parallelism (meaning larger CPF_i and KPF_i are used) and eventually lowers the latency L_i for that layer. By changing these parallelism factors, we can balance the DNN workloads by coordinating every pipeline stage and achieve the maximum throughput once the workload for each stage is well-balanced. The throughput performance can be described by Equation 1, where $Batch$ indicates

Algorithm 1 Computation resource allocation (paradigm 1)

```
1: Set available computation resource: $R_{total}$
2: Computation resource for layer $i$: $R_i = \frac{C_i}{C_{total}} \times R_{total}$
3: Initialize allocated resource for $i$-th layer (parallelism factor):
4: $R_i = 2^{\lfloor \log_2 R_i \rfloor}$
5: while $\sum_{i=1}^{n} R_i \leq R_{total}$
6:     Select layer $j$ with maximum $\frac{C_j}{R_j}$
7:     if $\sum_{i=1}^{n} R_i + 2 \times R_j \leq R_{total}$
8:         $R_j = 2 \times R_j$ //double the resource for layer $j$
9:     else break
10: $R_i = CPF_i \times KPF_i$
```

Algorithm 2 Bandwidth resource allocation (paradigm 1)

```
1: Set available memory bandwidth: $BW_{total}$
2: Set available on-chip memory for input buffers: $mem_{total}$
3: Set single DSP's bandwidth usage: $BW_R$
4: Initialize $Col_i = 1$; size of input buffer (e.g. $width_i^{rd}$, $depth_i^{rd}$,
   and $width_i^{wr}$ according to $PF_i = KPF_i \times CPF_i$)
5: Allocate bandwidth $BW_i$ for layer $i$ to best satisfy its $R_i$ demand:
   $BW_i = \frac{PF_i \times BW_R}{H_i^{out} \times Col_i}$
6: while $\sum_{i=1}^{n} BW_i \geq BW_{total}$ //CONV layer bandwidth overuse
7:     Select layer $i$ in CONV layer with maximum $BW_i$
8:     $depth_i^{rd} += \frac{H_i^{in} \times CHin_i \times Stride_i}{CPF_i}$, $depth_{i+1}^{rd} += \frac{H_i^{out} \times CHout_i}{CPF_{i+1}}$
9:     if $\sum_{i=1}^{n} f(width_i^{rd}, depth_i^{rd}, width_i^{wr}) \leq mem_{total}$
10:        $Col_i = Col_i + 1$ //Cache one more column
11:        $BW_i = BW_i \times \frac{Col_i - 1}{Col_i}$
12:    else //restore if not enough memory
13:    $depth_i^{rd} -= \frac{H_i^{in} \times CHin_i \times Stride_i}{CPF_i}$, $depth_{i+1}^{rd} -= \frac{H_i^{out} \times CHout_i}{CPF_{i+1}}$,
   break
```

the batch size and the latency for layer i can be shown in Equation 2.

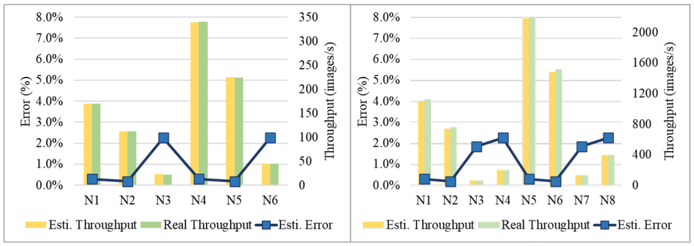
$$Throughput = \frac{Batch}{max(L_1, L_2, ..., L_n)} \tag{1}$$

$$L_i = \frac{H_i \times W_i \times R_i \times S_i \times CHin_i \times CHout_i}{CPF_i \times KPF_i \times FREQ} \tag{2}$$

To achieve the optimized performance, we follow strategies proposed in [2] to complete computation and memory resource allocation. In Algorithm 1, we allocate the computation resource to balance the latency of each pipeline stage. Since the parallelism factors must be the power of 2, we further fine-tune the allocation scheme and fills up the gap between the actual and the theoretical values. Resource allocated for layer i is represented as R_i.

Then, we adopt Algorithm 2 to allocate memory bandwidth given the constraints of total bandwidth BW_{total} and total amount of on-chip memory mem_{total} for input buffers. We initialize $Col_i = 1$ (caching one column of the input feature map according to the column-based cache scheme) and the input buffer is implemented by a dual-port RAM with the width of read/write port $width_i^{rd}$, $width_i^{wr}$, and the depth of read port $depth_i^{rd}$ in i-th layer. Algorithm 2 first satisfies the bandwidth demands of allocated computation resources (line 5). If the required memory bandwidth exceeds the total available bandwidth, we need bandwidth adjustment (starting from line 6). In this algorithm, H_i^{in}, H_i^{out} and $Stride_i$ represent the height of input and output feature maps and the stride in layer i. $CHin_i$ and $CHout_i$ represent the number of channels of the input and output feature map in layer i, respectively. Col_i represents the number of cached columns in layer i, which relates to the kernel reuse behavior and the consumption of on-chip memory mem_i.

To demonstrate the accuracy of our proposed analytical models, we compare the estimated throughput performance of the customized accelerators to their board-level implementation using FPGAs. After adopting the optimization strategies mentioned in this subsection, we evaluate accelerators on an embedded FPGA (Xilinx ZC706) and a mid-range FPGA (Xilinx KU115) and show the estimation errors in Figure 4. The average error between the estimated and the board-level performance is 1.15%. The reason for achieving such a low error rate comes from the high degree of determinism. With the dedicated compute and control units instantiated on hardware, the execution time of running targeted workloads is deterministic, which helps design highly accurate analytical models.



(a) ZC706 (b) KU115

Figure 4: Performance estimation errors of the pipeline architecture. (a) N1~N3 represent AlexNet, ZF, and YOLO (16-bit quantization) while N4~N6 represent the same group of networks using 8-bit quantization. In (b), N1~N4 represent AlexNet, ZF, VGG16, and YOLO with 16-bit quantization while the rest networks using 8-bit quantization.

4.3.2 The generic architecture. The computation latency of a DNN layer running on the generic architecture is determined by the parallel factor CPF_g and KPF_g as:

$$L_{comp} = \frac{H \times W \times R \times S \times CHin \times CHout}{CPF_g \times KPF_g \times FREQ} \tag{3}$$

To precisely estimate the memory accessing latency, we divide the external memory bandwidth BW into three portions: BW_w, BW_{ifm}, and BW_{ofm}, corresponding to the bandwidth

Algorithm 3 Generic architecture DSE (paradigm 2)

```
STEP1: Search for hardware parameters choices
Set the available resource number as $N_{dsp}$, $N_{bram}$, and $N_{lut}$
Initialize $CPF_g$, $KPF_g$, and $hwParamsList$
while (true):
    Calculate $n_{dsp}$, $n_{bram}$, and $n_{lut}$ with resource model
    if ($n_{dsp} > N_{dsp}$ or $n_{bram} > N_{bram}$ or $n_{lut} > N_{lut}$) break
    $hwParamsList$.append([$CPF_g$, $KPF_g$])
    Update $CPF_g$ and $KPF_g$

STEP2: Search for the optimal DNN mapping scheme for each hardware parameters choices
Initialize $dataflowsList$ and $latencyList$
for $hwParams$ in $hwParamsList$:
    Initialize $dataflows$ and $latency$
    for all $layer$:
        Find the best $dataflow$ and corresponding $layerLatency$
        $dataflows$.append($dataflow$)
        $latency$ += $layerLatency$
    $dataflowsList$.append($dataflows$)
    $latencyList$.append($latency$)

STEP3: Search for final solution with the lowest latency
$optLatency$, $idx$ = min($latencyList$), argmin($latencyList$)
return $hwParamsList[idx]$, $dataflowsList[idx]$, $optLatency$
```

for weight loading, feature map swapping in and out, respectively. The memory accessing latency L_w, L_{ifm}, and L_{ofm} can be calculated as:

$$L_w = \frac{R \times S \times CHin \times CHout \times WW_g}{BW_w} \tag{4}$$

$$L_{ifm} = \frac{H \times W \times CHin \times DW_g}{BW_{ifm}} \tag{5}$$

$$L_{ofm} = \frac{H \times W \times CHout \times DW_g}{BW_{ofm}} \tag{6}$$

For the IS strategy, assuming the capacity (in bits) of the accumulation buffer is CAP_{abuff}, the input and output feature maps will be partitioned into G_{fm} groups where:

$$G_{fm} = \frac{H \times W \times CHout \times DW_g}{CAP_{abuff}/2}. \tag{7}$$

In Equation 7, CAP_{abuff} is divided by a factor of 2 because the using of ping-pong buffers to avoid data pollution. Since the generic architecture needs to fetch weights G_{fm} times for IS, the overall latency of this CONV layer are:

$$L_{is} = max(L_{comp}, L_w \times G_{fm}, L_{ifm}, L_{ofm}) \tag{8}$$

Fro the WS strategy, we partition weights into G_w groups along the output channel dimension $CHout$. If we use CAP_{wbuff} to represent the capacity of weight buffer, G_w can be calculated as:

$$G_w = \frac{R \times S \times CHin \times CHout \times WW_g}{CAP_{wbuff}/2} \tag{9}$$

WS strategy keeps weights on-chip, and for each group of weights, all input feature maps need to be loaded before any computations. Since there are total G_w groups of weights to be calculated, the overall latency is:

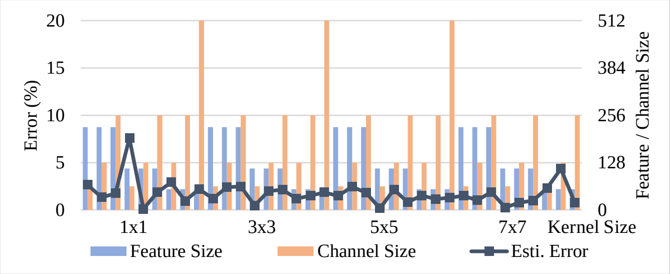


Figure 5: Performance estimation errors of the generic structure when mapping 36 cases of CONV layers, covering various feature map sizes (56, 112, 224), channel sizes (64, 128, 256, 512), and kernel sizes (1, 3, 5, 7).

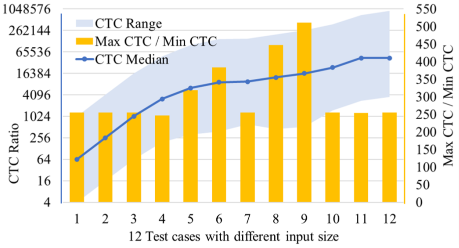
$$L_{ws} = max(L_{comp}, L_w, L_{ifm} \times G_w, L_{ofm} \times G_w) \tag{10}$$

On top of the estimation model, DNNExplorer adopts the DSE methodology from [3] to search for the optimized configuration of the generic architecture. As shown in Algorithm 3, hardware parameters and data reuse strategies are configured according to given resource budgets. To evaluate the analytical models of the generic structure, we include 36 cases of CONV layers with different channels, feature maps, 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 5, where only a 2.17% average error is observed across all 36 cases, which guarantees actuate accelerator modeling.

5 DNNEXPLORER FOR ACCELERATOR EXPLORATION

5.1 Drawbacks of paradigm 1 and 2

By applying the optimization strategies mentioned in Section 4.3, we can fully exploit the potential of accelerators following both pipeline and generic architecture under the predefined hardware constraints. However, there are still challenges preventing further performance improvements. It is hard for the generic accelerator to accommodate layers with various computation and memory demands using a generic compute unit, even though the layers from the same DNN model. To indicate the various arithmetic intensity, we present the computation-to-communication (CTC) ratios of layers from 12 DNN models in Figure 6. The medians of CTC show an upward trend along with higher input resolutions. From 32×32 to 512×512 inputs, CTC medians rapidly increase by nearly 256 times following the blue curve, which implies significantly different computation and memory-access patterns. If using a generic compute unit to process these layers,



#	Input Size	#	Input Size
1	32×32	7	320×480
2	64×64	8	448×448
3	128×128	9	512×512
4	224×224	10	480×800
5	320×320	11	512×1382
6	384×384	12	720×1280

Figure 6: CTC 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.

we have to accept sub-optimal solutions because of the lack of architecture specificity.

To obtain more intuitive data, we use DSP efficiency (defined as Equation 11) to evaluate whether an accelerator is working efficiently when prototyping in FPGA. GOP/s is a throughput performance metrics meaning Giga operations per second, while α 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. Higher DSP efficiency means better utilization of the available computation resources.

$$EFFI_{DSP} = \frac{GOP/s}{\alpha \times DSP_{allocated} \times FREQ} \tag{11}$$

We select three recently published DNN accelerators: Xilinx DPU [53], HybridDNN [3], and DNNBuilder [2] for comparison and present their DSP efficiency and throughput performance after benchmarking. In Figure 7 (a), both Xilinx DPU and HybridDNN (representing the generic architecture) suffer lower DSP efficiency (up to 64.9% and 53.7% degradation, respectively) compared to dedicated designs from DNNBuilder [2] (representing the pipeline architecture). By following the layer-based pipeline architecture, accelerators can adopt dedicated pipeline stages according to layers' inherent characteristics (e.g., arithmetic intensity, computation, and memory demands). It also enables more fine-grained hardware configurations and more in-depth design space exploration to address the low DSP efficiency issue. However, its scalability may be easily restricted by the number of DNN layers that it can support. A deeper DNN means more pipeline stages and fewer resources for each layer. In this case, performance degradation is expected as shown in Figure 7 (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, generic accelerators maintain stable performance.

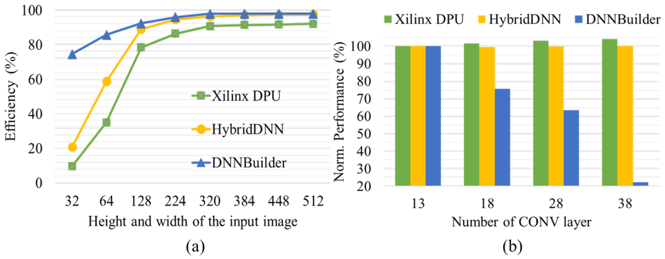


Figure 7: (a) DSP efficiency when running VGG16 with increasing input sizes in three representative FPGA-based DNN accelerators (batch size = 1); (b) Normalized throughput performance in three accelerators when running VGG-like DNNs with 3×224×224 inputs and 13~38 CONV layers. (The performance of each accelerator is normalized to their baseline cases running the 13-layer DNN.)

5.2 The proposed architecture paradigm

To overcome these challenges, we introduce a new paradigm to capture the essences of both existing paradigms but address their disadvantages. It is described as paradigm 3 in Figure 1. For the first part, we implement a pipeline accelerator to generate dedicated 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 to support deeper DNNs with given resources. The reason for such a combination comes from a common phenomenon during DNN inference, where the first half of DNN layers (close to the input layer) contains more variances of arithmetic intensity compared to the second half, which easily causes low DSP efficiency when following the generic paradigm. In our proposed design, this problem can be properly resolved by the layer-dedicated pipeline architecture. By considering the second half, where layers have fewer variances, a generic structure can be adopted to improve the design scalability. 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. Therefore, we propose an automatic architecture exploration to deliver optimized architecture configurations for accelerators following the novel paradigm.

5.3 Architecture exploration

5.3.1 Design space of the novel paradigm. With the new accelerator paradigm, we are allowed to explore significantly larger design space and perform a more fine-grained hardware configuration. We define a dynamic design space regarding all possible accelerator design combinations in Table 1. Split-point (SP) defines the task partitioning between the

Table 1: Design space of the proposed paradigm

	Pipeline Structure	Generic Structure
Parameters	$CPF_p = \{CPF_1, CPF_2, \cdots CPF_i\}$ $KPF_p = \{KPF_1, KPF_2, \cdots KPF_i\}$	CPF_g KPF_g $Buffer\text{-}allocation$ $Dataflow$
	$Split\text{-}point(SP)$	
	$Batch$	
Budgets	$C_{max}, M_{max}, BW_{max}$	

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 budgets $C_{max}, M_{max}, BW_{max}$ (corresponding to the available computation resources, on-chip memory resources, and external memory access bandwidth), DNNExplorer is designed to explore all design parameters and generate the best accelerators for targeted DNN applications. By targeting accelerator prototyping on FPGA, C_{max}, M_{max}, and BW_{max} represent the available DSPs, BRAMs, and external memory access bandwidth of the targeted FPGA. To efficiently explore the design space, we adopt a divide and conquer strategy and propose a two-level automatic DSE engine to perform global and local optimization, which will be introduced in the next two subsections.

5.3.2 A two-level DSE engine. The first-level optimization goal is to determine task and resource partitioning schemes between the pipeline and generic structure. DNNExplorer introduces RAV (resource allocation vector defined by Equation 12) to provide guidelines for mapping the targeted DNN workloads and allocating resources to the customized accelerator following the proposed paradigm. Layer 1 $\sim SP$ are mapped to a SP-stage pipeline structure with the resource budget $[DSP_p, BRAM_p, BW_p]$. Assuming the globally available resource is $[DSP, BRAM, BW]$, the rest of the layers are mapped into a generic reusable structure with the resource budget $[DSP - DSP_p, BRAM - BRAM_p, BW - BW_p]$. The pipeline and generic structure share the same batch size and working clock frequency.

$$RAV = [SP, Batch, DSP_p, BRAM_p, BW_p] \tag{12}$$

To select the best RAV across a high-dimensional design space, we employ a particle swarm optimization (PSO) algorithm to discover the most suitable combination of task and resource partitioning schemes between the pipeline and generic structure. As shown in Algorithm 4, each RAV is considered as a particle P_i, and all of them contribute to the swarm $\mathcal{M}$. We use throughput (GOP/s) as the fitness score to compare the quality of each design. To get the throughput estimation, we construct a fitness function using analytical models introduced in Section 4.3. By calling this function, the pipeline and generic structure optimization processes are individually executed to deliver the most suitable hardware configuration given the input RAV and the targeted workload. Once the configuration is ready, it is passed to analytical models for throughput (score) calculation. Based on the fitness score, 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 $\mathcal{N}$ iterations, and in the itr-iteration, each particle's position is updated according to the current position and updated velocity V_i. V_i 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.

Algorithm 4 The DSE algorithm

```
Initialize RAV with $\mathcal{M}$ Population: $P(\mathcal{M})$
Initialize iteration number: $\mathcal{N}$
Initialize HW boundary: $SP_{max}$, $DSP_{max}$, $BRAM_{max}$, $BW_{max}$
Evaluate each RAV: $Fit_i = FitnessScore(P_i)$, where $i \in \mathcal{M}$
Keep the local best for each RAV: $L_i = Fit_i$, and the global best: $G = max(L_i)$
While $itr < \mathcal{N}$:
  For each $P_i$ in $\mathcal{M}$:
    Get local and global velocity: $V_{toLbest_i}$, $V_{toGbest_i}$
    Update velocity: $V_i = w \cdot V_i + c_1 \cdot rand() \cdot V_{toLbest_i} + c_2 \cdot rand() \cdot V_{toGbest_i}$
    Update RAV: $P_i = UpdatePos(P_i, V_i)$
    Evaluate updated RAV: $Fit_i = FitnessScore(P_i)$
    Update local best: $L_i$
  Update global best: $G$
Output the best RAV
```

The second-level optimization is for individual optimization. Once the RAV is updated, pipeline and generic optimization strategies are launched to search for the best hardware configurations for both pipeline and generic structure (e.g., CPF_p, KPF_p, CPF_g, KPF_g, etc.) given the constraint of RAV. For the pipeline architecture, we follow strategies in Algorithm 1 and 2. CPF and KPF are calculated for each pipeline stage based on workload, arithmetic intensity, and given resource budget defined by the RAV. Then in Algorithm 3, DNNExplorer starts optimizing the generic structure to balance the pipeline throughput performance.

6 EXPERIMENTAL RESULTS

In this section, we demonstrate DNNExplorer for customized hardware accelerator benchmarking and exploration. The proposed tool is connected to a PyTorch framework and takes the PyTorch-developed DNNs as inputs. We select popular DNNs from the torchvision library, including models in the VGG family, the ResNet family, and the AlexNet. For the targeted hardware platforms, we choose two FPGAs as Xilinx

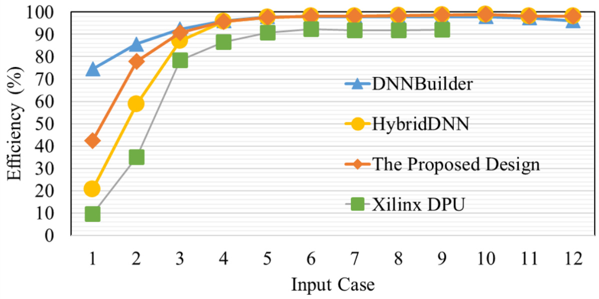
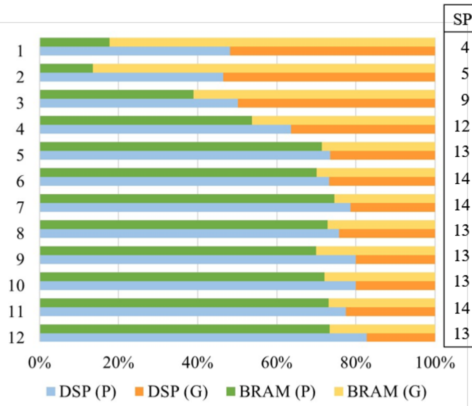


Figure 8: DSP efficiency comparison when running VGG16 (batch size = 1) with 12 different input sizes.



SP	DSP	BRAM	GOP/s
4	2268	2326	368.5
5	2730	2560	890.8
9	4686	3589	1702.3
12	4444	3296	1702.3
13	4450	3224	1702.4
14	4452	3436	1702.4
14	4452	3296	1702.4
13	4450	3552	1702.4
13	4450	3678	1702.4
13	4450	3678	1702.4
14	4452	3792	1702.5
13	4450	4186	1702.5

Figure 9: (left) Resource distribution between pipeline (P) and generic structure (G) in the proposed paradigm 3 when targeting VGG16 (batch size = 1) with 12 different input sizes. (right) Split-point (SP), utilization, and throughput of the generated accelerators.

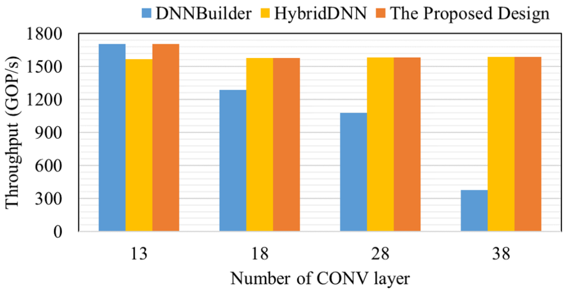


Figure 10: Throughput comparison when running deeper DNNs with the same $3 \times 224 \times 224$ input.

KU115 and Xilinx ZC706, to cover different scenarios for cloud and edge applications, respectively.

6.1 DSP efficiency

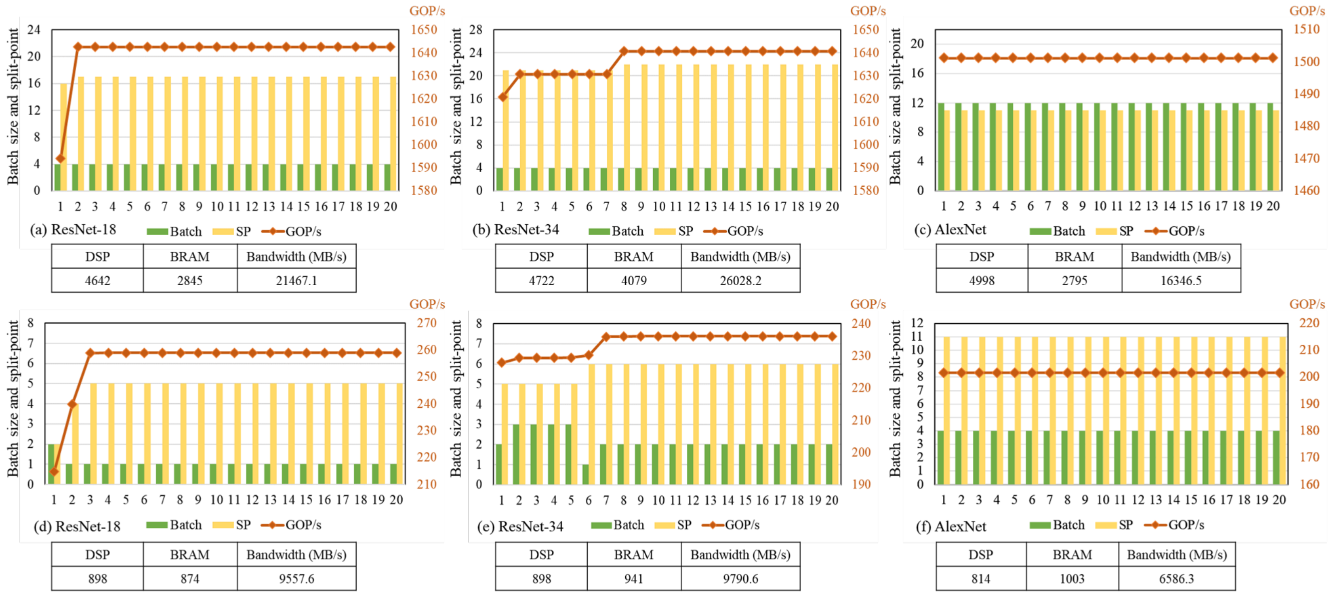
We use DSP efficiency to evaluate whether accelerators maximize the usage of allocated computation resources for FPGA implementation. We adopt DNNExplorer to evaluate all three supported accelerator paradigms mentioned in Figure 1: the pipeline architecture from [2] (paradigm 1), the generic architecture HybridDNN [3] (paradigm 2), and the proposed novel architecture (paradigm 3). All three accelerator designs are targeting the same set of DNNs: 12 VGG-16 (without the last three FC layers) models with different input sizes (which are listed in Figure 6 from #1 to #12) and the same KU115 FPGA for a fair comparison. We also quantize the DNN models to use 16-bit fixed-point data. Benchmarking results are shown in Figure 8, where paradigm 1 achieves the highest DSP efficiency, especially for small-sized inputs (e.g., case 1 and 2) because of its dedicated pipeline stage design. The proposed paradigm 3 is slightly behind when targeting small inputs but soon reaching the same efficiency level (>95%) after case 3. Compared to the generic accelerator design, paradigm 3 can deliver 2.0× and 1.3× higher efficiency for case 1 and case 2, respectively. We also compare to Xilinx DPU [53], a commercial DNN accelerator IP, for running the first nine cases on a ZCU102 FPGA (as the last three inputs are not supported yet). The proposed paradigm 3 can achieve an average 1.6× higher DSP efficiency, peaking at 4.4× for case 1. As the input size increases, the efficiency gap decreases (<10% after case 5).

6.2 Throughput performance

We present the throughput performance of the accelerators following paradigm 3 in Figure 9. These accelerators target the same DNNs (mentioned in Subsection 6.1) and the same KU115 FPGA with 200MHz working frequency. We restrict the batch size to one and create a more demanding application scenario as accelerators can not always rely on larger batch sizes to reach higher throughput. The first two cases can not provide enough workloads (due to the smaller input size), so accelerators failed to reach their peak performance. To meet resource constraints and maximize performance, the proposed two-level DSE engine generates task partitioning and resource distribution, where the SP indicates the split-point for partitioning the targeted DNN to the pipeline and generic structure and the stacked bar chart shows the resource allocation for DSP and BRAM resources. From the results, the proposed DSE is likely to allocate more tasks and resources to the pipeline structure when increasing the DNN input size from case 1 to case 12.

6.3 Scalability

To evaluate all three paradigms' scalability, we prepare four deeper DNNs, with 13, 18, 28, and 38 CONV layers. The 13-layer one comes directly from VGG16 by removing the FC layers. Since VGG is composed of five 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



(a) ResNet-18

DSP	BRAM	Bandwidth (MB/s)
4642	2845	21467.1

(b) ResNet-34

DSP	BRAM	Bandwidth (MB/s)
4722	4079	26028.2

(c) AlexNet

DSP	BRAM	Bandwidth (MB/s)
4998	2795	16346.5

(d) ResNet-18

DSP	BRAM	Bandwidth (MB/s)
898	874	9557.6

(e) ResNet-34

DSP	BRAM	Bandwidth (MB/s)
898	941	9790.6

(f) AlexNet

DSP	BRAM	Bandwidth (MB/s)
814	1003	6586.3

Figure 11: Architecture exploration for accelerators adopting paradigm 3 for ResNet-18/-34 and AlexNet. Designs in (a)~(c) adopt the resource budget of a KU115 FPGA while designs in (d)~(f) are targeting a ZC706 FPGA. Resource utilization is shown in the table below each sub-graph.

(13+5) model. Similarly, we add three and five CONV layers to each part for the 28- and 38-layer model, respectively. In Figure 10, we can clearly observe the drawback of using paradigm 1, which is hard to handle a deeper network. By contrast, accelerators following paradigm 2 and 3 can maintain peak performance despite targeting deeper networks. Compared to paradigm 1, the proposed paradigm 3 can deliver 4.2× higher performance for accelerating a 38-layer VGG-like DNN on the same FPGA platform.

6.4 Architecture exploration

To better understand the architecture exploration feature provided by DNNExplorer, we extend the experiment for customized accelerator design following paradigm 3. We lift the batch size restriction and capture the hardware configuration changes while applying the proposed two-level DSE. We target three DNNs (ResNet-18/-34 and AlexNet from torchvision) and two hardware platforms (KU115 and ZC706) for this experiment, and results are shown in Figure 11. Since we set the search iteration number to 20, each sub-figure has 20 pairs of green and yellow bars, indicating the batch size and split-point configuration. We also present the throughput of the best results for every iteration along the red curve. Resource utilization of the best accelerator after exploration is shown below the corresponding sub-graph. The proposed two-level DSE engine can soon search for the optimized hardware configuration with the throughput performance reaching the peak during the first ten iterations for all six cases. Shown in Figure 11 (a)~(c), the customized accelerators deliver 1642.6, 1640.6, and 1501.2 GOP/s using KU115 after exploration, while in Figure 11 (d)~(f), accelerators deliver 258.9, 236.1, and 201.6 GOP/s using ZC706.

To accommodate the rapid development of DNN, DNNExplorer follows the modular design strategy, which can be extended to support more emerging layers by adding corresponding layer modules. With these extended models, newly supported layers can be parsed in step 1 of the proposed flow (Figure 1) to extract layer-wise information. Still, DNNExplorer has not yet supported some new network architectures such as the Transformer [17] as it requires additional support of unique layer modules for improving the accelerator architecture paradigm. To support more network architectures will be our future work.

7 CONCLUSION

This paper presented DNNExplorer, an automation tool for benchmarking and exploring customized DNN hardware accelerators. Novel technologies included the direct support of popular machine learning frameworks for easy access to the existing or customized DNNs; highly accurate analytical models for effective hardware accelerator benchmarking; a novel accelerator paradigm to overcome drawbacks of the existing paradigms; and a two-level DSE engine for accelerator exploration to deliver optimized accelerators by considering both targeted DNN workloads and given resource budgets. With the above technologies, DNNExplorer can provide customized accelerator benchmarking and perform architecture exploration to deliver optimized solutions when targeting emerging AI applications. It can also offer DNN and accelerator design insights to enable more efficient AI deployments at the earliest design stage.

ACKNOWLEDGMENT

We want to thank Junsong Wang, Yonghua Lin, Jinjun Xiong, and Wen-mei Hwu, who contributed to an early version of the work published in ICCAD'20. This work is partly supported by IBM-Illinois Center for Cognitive Computing System Research (C3SR) and a Google PhD fellowship to Xiaofan Zhang.

REFERENCES

[1] Xiaofan Zhang et al. DNNExplorer: a framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator. In *Proc. of the International Conference on Computer-Aided Design (ICCAD)*, 2020.

[2] Xiaofan Zhang et al. DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proc. of International Conference on Computer-Aided Design (ICCAD)*, 2018.

[3] Hanchen Ye et al. HybridDNN: A framework for high-performance hybrid DNN accelerator design and implementation. In *Proc. of the Design Automation Conference (DAC)*, 2020.

[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 *Proc. of the IEEE conference on computer vision and pattern recognition (CVPR)*, 2015.

[7] Kaiming He et al. Deep residual learning for image recognition. In *Proc. 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 *Proc. 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 *Proc. 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] Yoshua Bengio et al. A neural probabilistic language model. *Journal of Machine Learning Research*, 3(2):1137–1155, 2003.

[13] Tomáš Mikolov et al. Recurrent neural network based language model. In *Proc. of INTERSPEECH*, 2010.

[14] Dzmitry Bahdanau et al. Neural machine translation by jointly learning to align and translate. In *Proc. of International Conference on Learning Representations (ICLR)*, 2015.

[15] Yonghui Wu et al. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*, 2016.

[16] Jonas Gehring et al. A convolutional encoder model for neural machine translation. In *Proc. of Annual Meeting of the Association for Computational Linguistics (ACL)*, 2017.

[17] Ashish Vaswani et al. Attention is all you need. In *Proc. of Conference on Neural Information Processing Systems (NeurIPS)*, 2017.

[18] Hadi Esmaeilzadeh et al. Neural acceleration for general-purpose approximate programs. In *Proc. of International Symposium on Microarchitecture (Micro)*. IEEE, 2012.

[19] Yunji Chen et al. DianNao family: energy-efficient hardware accelerators for machine learning. *Communications of the ACM*, 59(11):105–112, 2016.

[20] Yu-Hsin Chen et al. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. In *Proc. of International Solid-State Circuits Conference (ISSCC)*, 2016.

[21] Norman P Jouppi et al. In-datacenter performance analysis of a tensor processing unit. In *Proc. of International Symposium on Computer Architecture (ISCA)*, 2017.

[22] Jiantao Qiu et al. Going deeper with embedded FPGA platform for convolutional neural network. In *Proc. of International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[23] Xiaofan Zhang et al. High-performance video content recognition with long-term recurrent convolutional network for FPGA. In *Proc. of the International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[24] Xiaofan Zhang et al. Machine learning on FPGAs to face the IoT revolution. In *Proc. of International Conference on Computer-Aided Design (ICCAD)*, 2017.

[25] Yao Chen et al. Cloud-DNN: An open framework for mapping DNN models to cloud FPGAs. In *Proc. of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019.

[26] Huimin Li et al. A high performance fpga-based accelerator for large-scale convolutional neural networks. In *Proc. of the International Conference on Field Programmable Logic and Applications (FPL)*, 2016.

[27] Xuechao Wei et al. TGPA: tile-grained pipeline architecture for low latency cnn inference. In *Proc. of the International Conference on Computer-Aided Design (ICCAD)*, 2018.

[28] Jialiang Zhang and Jing Li. Improving the performance of opencl-based fpga accelerator for convolutional neural network. In *Proc. of the International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017.

[29] Cong Hao et al. FPGA/DNN co-design: An efficient design methodology for IoT intelligence on the edge. In *Proc. of the Design Automation Conference (DAC)*, 2019.

[30] Yangqing Jia et al. Caffe: Convolutional architecture for fast feature embedding. In *Proc. of the ACM international conference on Multimedia*, 2014.

[31] Adam Paszke et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems (NeurIPS)*, 2019.

[32] Baidu. DeepBench: Benchmarking deep learning operations on different hardware, 2017. Accessed: 2021-2-25.

[33] Google. Tensorflow benchmarks and a new high-performance guide, 2017. Accessed: 2021-2-25.

[34] Hongyu Zhu et al. Benchmarking and analyzing deep neural network training. In *Pro. of International Symposium on Workload Characterization (IISWC)*, 2018.

[35] Cody Coleman et al. Dawnbench: An end-to-end deep learning benchmark and competition. In *ML Systems Workshop of Conference on Neural Information Processing Systems*, 2017.

[36] Peter Mattson et al. MLPerf: An industry standard benchmark suite for machine learning performance. *IEEE Micro*, 40(2):8–16, 2020.

[37] Kyle Rupnow et al. A study of high-level synthesis: Promises and challenges. In *Proc. of IEEE International Conference on ASIC*, 2011.

[38] Xinheng Liu et al. High level synthesis of complex applications: An H. 264 video decoder. In *Proc. of International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

[39] Xuechao Wei et al. Overcoming data transfer bottlenecks in FPGA-based DNN accelerators via layer conscious memory management. In *Proc. of Design Automation Conference (DAC)*, 2019.

[40] Chuanhao Zhuge et al. Face recognition with hybrid efficient convolution algorithms on FPGAs. In *Proc. of Great Lakes Symposium on VLSI (GLSVLSI)*, 2018.

[41] Hao Wang et al. Hardware-software co-design for face recognition on FPGA SoCs. In *Proc. of International Symposium on Circuits and Systems (ISCAS)*, 2020.

[42] Xiaofan Zhang et al. SkyNet: A champion model for DAC-SDC on low power object detection. *arXiv preprint arXiv:1906.10327*, 2019.

[43] Qin Li et al. Implementing neural machine translation with bi-directional GRU and attention mechanism on FPGAs using HLS. In *Proc. of Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2019.

[44] Bingbing Li et al. FTRANS: energy-efficient acceleration of transformers using FPGA. In *Proc. of International Symposium on Low Power Electronics and Design (ISLPED)*, 2020.

[45] Qin Li et al. Efficient methods for mapping neural machine translator on FPGAs. *IEEE Transactions on Parallel and Distributed Systems*, 32(7):1866–1877, 2021.

[46] Xuechao Wei et al. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proc. of Design Automation Conference (DAC)*, 2017.

[47] 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.

[48] Junsong Wang et al. Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA. In *Proc. of the International Conference on Field Programmable Logic and Applications (FPL)*, 2018.

[49] Pengfei Xu et al. AutoDNNchip: An automated DNN chip predictor and builder for both FPGAs and ASICs. In *Proc. of International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020.

[50] Xiaofan Zhang et al. F-CAD: A framework to explore hardware accelerators for codec avatar decoding. *arXiv preprint arXiv:2103.04958*, 2021.

[51] Weiwen Jiang et al. Accuracy vs. efficiency: Achieving both through FPGA-implementation aware neural architecture search. In *Proc. of Design Automation Conference (DAC)*, 2019.

[52] Yuhong Li et al. EDD: Efficient differentiable DNN architecture and implementation co-search for embedded AI solutions. In *Proc. of Design Automation Conference (DAC)*, 2020.

[53] Xilinx. Zynq DPU product guide. Accessed: 2021-2-25.