Cloud-DNN: An Open Framework for Mapping DNN Models to Cloud FPGAs

Yao Chen[1], Jiong He[1], Xiaofan Zhang[2], Cong Hao[2], Deming Chen[1,2]
[1]Advanced Digital Sciences Center, Singapore
[2]University of Illinois at Urbana-Champaign, IL, USA
{yao.chen, Jiong.he}@adsc-create.edu.sg, {xiaofan3, congh, dchen}@illinois.edu

ABSTRACT

The efficacy and effectiveness of Convolutional Neural Networks (CNNs) have been proven in a wide range of machine learning applications. However, the high computational complexity of CNNs presents a critical challenge towards their broader adoption in real-time and power-efficient scenarios. FPGAs are poised to take a significant role for high-performance and energy-efficient computation of CNNs for both mobile (e.g., UAVs, self-driving cars, and IoT devices) and cloud computing domains. However, implementing an effective CNN system onto FPGAs efficiently remains problematic. The current cloud-based FPGAs with unique design constraints and architectural characteristics further increase the challenges. To address these challenges, we propose a novel open-source automated tool chain called Cloud-DNN. Our tool chain takes trained CNN models specified in Caffe as input, performs a set of transformations, and maps the model to a cloud-based FPGA. Cloud-DNN can significantly improve the overall design productivity of CNNs on FPGAs while satisfying the emergent computational requirements. Our design provides an alternative solution compared to other cloud-based options (e.g., GPUs or TPUs) while offering flexible, and high performance DNN inferences. The unique features of Cloud-DNN include the optimizations with cloud-platform characteristics and the support of easier and streamlined implementation. Experimental results demonstrate up to 104.55× performance improvement when compared to CPU implementation and comparable usability, flexibility, and strong quality compared to other state-of-the-art DNN inference implementations on standalone FPGAs.

KEYWORDS

DNN Accelerator; FPGA; High-Level Synthesis; Cloud Computing

ACM Reference Format:
Yao Chen, Jiong He, Xiaofan Zhang, Cong Hao, Deming Chen. 2019. Cloud-DNN: An Open Framework for Mapping DNN Models to Cloud FPGAs. In *The 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19), February 24–26, 2019, Seaside, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3289602.3293915

FPGA '19, February 24–26, 2019, Seaside, CA, USA

ACM ISBN 978-1-4503-6137-8/19/02...$15.00
https://doi.org/10.1145/3289602.3293915

1 INTRODUCTION

We have witnessed an increasingly growing interest in designing Deep Neural Networks (DNNs) that can perform highly accurate inference for extensive applications [1]. Conventionally, higher inference accuracy can be obtained by deeper and wider networks which include larger number of network layers and channels. Such features impose dramatic increase of computational complexity and memory demands which require sophisticated hardware accelerators to tackle the computing and memory accessing complexities. The state-of-the-art hardware accelerators for DNNs commonly exploit different resources such as CPUs, GPUs, FPGAs or ASICs to deliver sufficient performance under application-specific constraints. However, the energy-hungry CPU- and GPU-based accelerators will not meet the energy/power limits while the ASIC-based designs require long time-to-market period. Overall, FPGAs offer a promising alternative for DNN acceleration with improved latency, high energy efficiency and high flexibility.

The recent adoption of FPGA demonstrates its great capability for running DNN-related applications in both cloud servers and mobile devices [2–7]. This combination between programmable hardware and DNNs has enabled more possibilities to reshape the landscape of deep learning applications for real-time performance, high throughput, and high energy efficiency. Recent studies have reported great performance in image classification/detection [8–11], image/video description [12], speech recognition [13], and machine translation [14] using FPGAs. The FPGA-based designs also benefit by the increasingly popular design flow using high-level synthesis (HLS), where high-level programming languages are used for abstract descriptions of hardware functions instead of following register-transfer level (RTL) designs [15–17]. Since DNNs are composed of layers of regular structures, such as convolution and pooling, HLS is well suited to optimize these regular computations in DNNs [12, 14–16, 18–21]. Meanwhile, growing interest of using FPGA to accelerate DNN workloads drives the deployment of FPGAs on cloud services (e.g., Amazon AWS and Microsoft Azure) which are used by cloud customers in a pay-as-you-go scheme. The availability and flexibility of FPGAs in the cloud raise new challenges in the design and implementation of deep learning models on these platforms.

In real world applications, DNN workloads are usually compute-intensive due to the nature of the input data (e.g. streaming frames of data from sensors such as acoustics, high-definition (HD) videos and images, etc). In many cases, the entire network model may not fit into a single FPGA fabric due to large amount of neurons, weight data and intermediate results. These factors together incur a complex computational problem of massive data to be processed with increasing computations per input data entry. Naturally, cloud

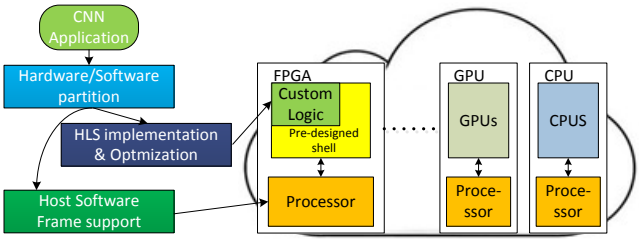

Figure 1: Mapping CNN application to cloud based FPGA.

computing platforms are poised to address this problem efficiently, as they can offer sufficient computation resources and high speed network interface to support complicated DNNs and streams of HD inputs. FPGAs offer an attractive solution operating as the backbone for such cloud platforms by providing low-latency, energy-efficient computations. Cloud FPGAs can help drive the DNN computing revolution in coming years (Figure 1). However, there are still many challenges and promising opportunities in designing an effective and efficient DNN inference accelerator on cloud FPGAs according to the following observations.

1) High performance and fast development. Cloud applications always desire fast development to support the latest DNN-based applications. Compared to conventional FPGA design flow, HLS-based design methodology provides better efficiency and design productivity, which can speed up the hardware design and verification on cloud FPGAs. However, the implementation may not be able to fully satisfy the performance requirements of the application without hardware optimization. HLS-based optimization is a practical way to improve the design quality such as specifying pipelines, unrolling loops, and handling data transfers. Therefore, a well-designed synthesizable C++ template which integrates all necessary optimizations is able to ease the development effort for users. Automatically and adaptively applying HLS optimizations for different workloads provides an easy-to-use DNN-to-Cloud implementation framework even for users who do not have domain knowledge for either FPGA or HLS tools.

2) Architectural characteristics. Contrasting with embedded platforms, resources in cloud FPGAs are usually sufficient for a large accelerator given the high capacity and capability of the FPGA chip, which is created by a manufacturing process called Stacked Silicon Interconnect (SSI) technology [22] for the Xilinx chips used in AWS. The SSI technology combines multiple Super Logic Region (SLR) components (or dies) mounted on a passive Silicon Interposer. However, this characteristic carries a trade-off between design size and operating frequency. It is very difficult to fit a design which has large amount of internal routing into such cloud FPGA without timing violations because of the long cross-die routing problem and the distributed on-chip memory. As an instance, the large convolutional accelerator shown in Figure 2 (shown with purple color) can not meet the timing due to the long crossing-die interconnections. However, the timing could easily be achieved by splitting the big accelerator into three smaller ones (shown with yellow, red and blue) with bus or FIFO connections between them. Other requirements such as bandwidth and design flexibility might be constrained by the pre-designed shell of the cloud platform. For instance, the AWS F1 platform could only provide 6.5GB/s host to FPGA memory bandwidth in the current release. Designs on

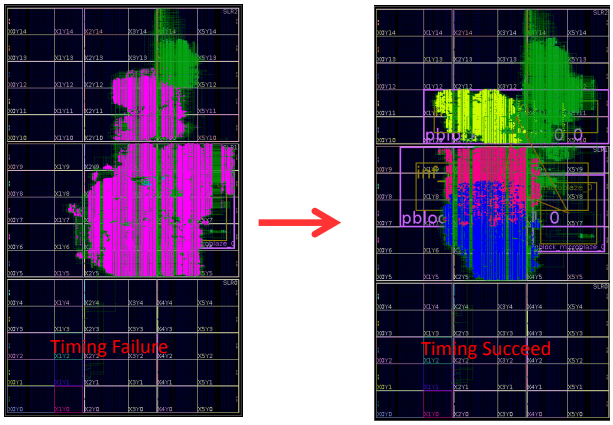

Figure 2: Cross-die routing difficulties.

AWS F1 also require the integration of the AWS Shell IP to achieve control/communication to/from host CPU. The flexibility of the system clock is also constrained by the AWS Shell IP into several limited options. Such platform constraints should be considered during DNN model implementation. In our Could-DNN flow, such platform constraints are taken into consideration during the model generation and optimization.

3) Software framework support. An efficient system in the cloud requires the scheduling of the hardware modules and the proper allocation of the tasks, and their corresponding design and optimization can be critical to the overall performance (Figure 1). Open-source software frameworks have become increasingly important for machine learning applications such as Caffe and Tensorflow. While these libraries support CPU and GPU implementations, they do not offer support for direct FPGA mapping. Our proposed framework is an open-source tool chain that can map a given trained Caffe network model to FPGA in the cloud without understanding the hardware details, while leveraging inherent FPGA advantages. With well designed hardware driver and API functions, the generated accelerators can be easily integrated into existing DNN frameworks. Our core contributions include:

- We design a fully synthesizable C++ template library with the considerations of FPGA implementation characteristics which can fit into HLS design flow.
- We propose an automated generation flow that maps DNN models in Caffe to FPGA implementations without any tedious hardware programming and verifications.
- We propose a design space exploration algorithm including task clustering and scheduling, to generate an optimized system configuration to maximize the overall performance.
- We also generate a corresponding software stack together with the DNN accelerator, to provide a complete system level solution for the users who need acceleration services.

The rest of this paper is organized as follows. Section 2 introduces the overall flow of Cloud-DNN. Section 3 presents the detailed design and model of the synthesizable library. Section 4 discusses the generated system architecture and the design methodology. Section 5 presents the details of our Cloud-DNN framework. The evaluation results and analysis are shown in Section 6. Related work is discussed in Section 7. We conclude this paper in Section 8.

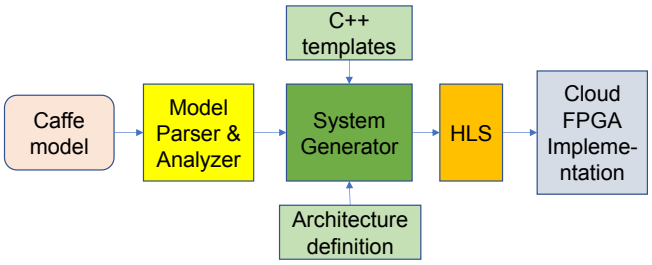

Figure 3: Cloud-DNN flow overview.

2 CLOUD-DNN OVERVIEW

Our Cloud-DNN framework takes a network model trained by Caffe as input and generates a high performance inference accelerator system as output. The overall flow is shown in Figure 3. The network model files (prototxt and caffemodel [23]) are first extracted and analyzed to obtain parameters and to reorder the weights by a model parser and analyzer. Then, the network is constructed based on a pre-designed synthesizable C++ template function library and pre-defined system architecture. The layer task allocation and the exploration of the accelerator configurations are applied during the network construction. After the C++ functions for the network model are constructed, they are passed through the HLS and design tools to generate the FPGA implementation. The corresponding host software solution is also generated automatically based on user specification and the results from model analyzer.

3 SYNTHESIZABLE LIBRARY DESIGN

Although there are various types of DNNs for a great variety of applications with entirely different topological structures, the number of basic layer types is relatively small, such as convolutional layer, fully-connected layer, pooling layer, and activation functions. With such features, DNN can be well defined based on layer types and channel numbers and eventually the required computation and memory resources can be determined before detailed implementation. HLS requires the memory allocation and interface of the targeted function to be specified statically before compilation, which is well suited to optimize DNNs. In order to satisfy this design flow, we describe two computational patterns of typical CNN operations: Tensor-to-Tensor (TT) and Element-wise (EW), as shown in Table 1.

Table 1: Two computation patterns of typical CNNs.

Tensor-to-Tensor (TT)	Conv, Pooling, FC, LRC, Unpooling, Deconv, Dropout, etc.
Element-wise (EW)	ReLU, Sigmoid, Tanh, BN, etc.

We follow the same tiling-based convolutional accelerator design strategy in existing works [20, 21], and extend their idea to support both TT and EW function template without losing flexibility. To support the emerging DNNs with constantly increasing demands on both computation and memory, we propose a highly salable and flexible solution. The proposed templates are designed to be scalable functional accelerator cores, and the neural network interfaces are considered with the memory bank design together to support multi-accelerator network construction. Before we discuss the detailed templates, we first introduce the parameter settings within layer accelerators for various functional layers.

3.1 Layer Accelerator Template

A set of parameters are used to customize the function templates to accomplish the computation of the layers as well as to meet the performance requirements after being synthesized to hardware. The element-wise functions and tensor-to-tensor functions are designed separately due to the nature of computations.

3.1.1 Element-wise functions. Typical activation functions such as $Tanh()$, $sigmod()$, $ReLU()$ of a DNN are all element-wise functions. These functions always follow the layers that process tensor data but only compute a single input for every function call. Different from activation functions, Batch Normalization (BN) is a novel approach used in recent CNNs to enable fast training convergence. The operations of BN need four pre-trained values which are constant during the inference. The BN is described as:

$$x_{out} = (x_{in} - bn_0)/bn_1 \quad (1)$$

$$y_{out} = sc_0 \times x_{out} + sc_1 \quad (2)$$

We convert BN into an EW function that computes an output element for every corresponding input data. In order to save hardware resources on constructing new layer hierarchy, we design functional templates of activation layers with customizable data types and directly in-line them into the data output module in each layer accelerator function. In this way, hardware resources can be saved by removing the need for a dedicated activation function layer within the network.

3.1.2 Tensor-to-Tensor functions. Tensor-to-Tensor functions contain numerical parameters since both the input and output data have multiple dimensions. Hence, the data pattern and the computation processes are defined by the parameters of the data dimension and patterns accordingly.

Convolutional Accelerator. A typical convolution layer is comprised of 6 levels of for-loops. We follow the design principles described in the state-of-the-art HLS convolution layer accelerator design in [20, 21], and extend it to provide a more flexible system integration support. Based on the computational nature of a convolution layer, we illustrate the computation with a list of variables including (M, N, R_{in}, C_{in}, R, C, K, S, P, act), which represent the number of output and input feature maps M and N, output and input feature size on two dimensions (R, C) and (R_{in}, C_{in}), kernel size K, stride size S, padding size P, and activation function type act, respectively. By leveraging existing convolution loop optimizations in [20, 21], layer computations are processed in multiple rounds to provide data transmission flexibility. The customizable parameters for a convolution accelerator include (T_m, T_n, T_r, T_c), where T_m and T_n represent the number of output and input feature maps, and T_r and T_c represent the width and height of output feature maps, respectively. The accelerator is designed with a 512-bit data input and output port. As shown in Figure 4 a), all the accelerator parameters are customizable so that the accelerator configuration can be tailored based on the input network model parameters.

Pooling Accelerator. The pooling layer processes the input data with a sliding window and returns the results with a selection

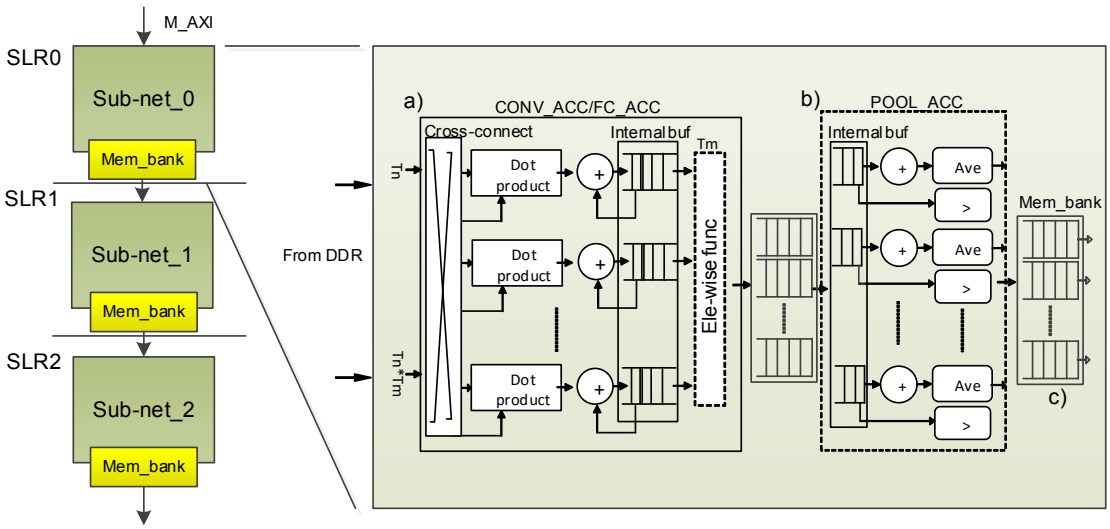

Figure 4: Sub-net, layer accelerator and memory bank templates.

method, i.e., maximum or average values. The functionality of pooling is simple, but it requires data transmission flexibility to be used in top-level application design. Therefore, we design the pooling layer as a standalone layer accelerator rather than merging it into a convolutional accelerator. Because the pooling layer does not change the input/output tensor dimension, there is only one parameter T_n for the input/output feature numbers, so the pooling accelerator parameters are denoted as (T_n, T_r, T_c), with the same definition as the convolution layer accelerator (shown in Figure 4 b)). The variables list to process a pooling layer is $(N, R_{in}, C_{in}, R, C, K, S, P, p\_act)$, correspondingly.

Fully connected Accelerator. The fully connected (FC) layer or the inner-product layer performs matrix multiplication of the input features and coefficients. Different from the convolutional accelerator, the outputs of FC accelerator are vectors, which does not contain the loops for output feature dimensions. The FC accelerator is also designed in the manner of convolutional accelerator but with the T_r and T_c equals to 1. Similarly, the FC layer parameters include (T_m, T_n) and a variable list including $(N, R_{in}, C_{in}, R, K, f\_act)$.

3.2 Sub-net Template

Large DNNs can contain many layers. Hence, in our design, the entire network is split into sub-nets based on the method introduced in Section 5.2. Since each FPGA in popular cloud platform incorporates three dies (e.g., AWS and Alicloud), in this work, we in general partition the DNN into three sub-nets. The sub-nets are mapped to different dies in an SSI-based FPGA and connected with memory banks or through external memory, so the sub-nets can execute concurently. Each sub-net contains different numbers of layer accelerators guided by a design space exploration (DSE) engine (Section 5) aiming at optimizing the performance under platform constraints.

3.2.1 In/out memory bank. The memory banks between sub-nets are used to buffer the 3-dimensional feature data passed between them. It also provides the design flexibility to meet the timing constraints for the crossing-die routing. So the size of each bank will be determined by the data storage of the sub-nets. The on/off-chip choice is decided by the storage size. We define the parameters of memory banks as $(on/off\_chip, bram\_num, depth, width)$, which represents the on/off-chip decision, the number of RAM banks, the depth of the RAM bank and the data width. The on-chip RAM memory bank is generated with a group of RAM interfaces and a group of AXI interfaces through AXI interconnection. The off-chip memory bank is instantiated on the external RAM with AXI interface. The AXI interface is used to provide crossing-die connection ability by taking advantage of the timing insensitivity of the data FIFO in the AXI interconnect.

3.2.2 Sub-net template. Each of the sub-nets contains a number of layer accelerators. The layer accelerators in a sub-net are connected through the memory bank to enable a pipelined execution scheme. The overall sub-net and layer accelerator template is shown in Figure 4. The number of accelerator instances and every parameter of the convolution/pooling kernels are determined by the algorithm presented in Section 5.

This algorithm uses the input and output values (including padding and stride) of the convolutional layers in a model and outputs the customized accelerator configurations based on the layer accelerator templates. The corresponding memory bank size is also calculated based on the model parameters under the platform constraints. The interface of the sub-net for feature data and weight is designed as AXI interface to ease the connection with external memory or previous memory bank. This also allows us to take advantage of the memory bank to overcome the difficulty of cross-die data transmission. The output data port is defined as BRAM or AXI bus interface based on the location of the following buffer, and could be directly connected to the following memory bank, as is shown in Figure 4. Also, the mismatch between the accelerator interface and the data width supported by the cloud shell system is resolved at the interface with ranging logic, to keep the unified memory mapping for all the components in the accelerator system.

3.2.3 Quantized network model support. Data quantization is a practical method for reducing both the memory footprint and computational complexity. This is accomplished by lowering the data

precision and reducing the bit-width of the data in a network model. Numerous existing studies have shown that the DNN inference will not suffer from significant accuracy drop using lower precision data if the network is well trained [13, 24, 25]. Prior designs have also shown the resource saving and performance improvement with fixed-point arithmetic for neural networks on FPGA platforms [11]. Since we are designing a generalized framework, in order to support quantized network models, all data types in the accelerator templates are specified individually, such that the data types for all accelerators involved in the network system can be specified by users during the generation process.

The parameters for different templates are shown in Table 2. The parameters for the sub-net templates are *ACC_NUM* and *Acc_type*, which specifies the number of accelerators and the types of them.

Table 2: Template Class and Parameters.

Template class	Parameters
$CONV\_ACC$	$DataType$, T_m, T_n, T_r, T_c
$POOL\_ACC$	$DataType$, T_n, Tr, T_c
FC_ACC	$DataType$, T_n, T_r, T_c
ACT	$DataType$, $ActType$
Mem_bank	on/off_chip, $bram\_num$, $depth$, $width$
$Sub\text{-}net$	ACC_NUM, Acc_type

3.3 Accelerator Model

During network construction, three groups of parameters are required: 1) the network split method and 2) parameters of the accelerators (including accelerator number) and 3) the layer tasks allocated to the accelerators. To find the optimal parameter configuration with given resource constraints, accurate resource and performance models for each layer accelerator are necessary.

3.3.1 Hardware resource cost. Adjusting the parameters of the accelerator has varying effects on resource consumption. An accurate formulation of the resource cost and parameter settings is critical for system performance optimization. Based on our experimental results and previous literature [20, 21], the LUT and FF are not the bottleneck for accelerator system generation. The DSP and on-chip RAM are potential limiting factors, and thus are carefully evaluated in our modeling process.

DSP usage. The primary use of DSP modules in a convolutional accelerator is the unrolled $T_m \times T_n$ dot-product and accumulator modules. The DSP cost is related to the processing data type. Therefore, the DSP consumption in a convolutional accelerator can be formulated as Equation 3. The data type and corresponding number of DSPs for a single dot-product and accumulator engine is shown in Table 3.

$$N_{DSPconv} = DSP_{data\_type} \times (T_n \times T_m) \tag{3}$$

The max pooling accelerator does not consume DSP resources. However, the average pooling accelerator requires DSP modules for the computation of average value output, which can be formulated based on the unroll factor (T_n) of the pooling accelerator.

$$N_{DSPpave} = 1 \times T_n + 1 \tag{4}$$

Table 3: Data type and DSP cost.

Data Type	float	fixed32	fixed24	fixed16	fixed8
DSP Cost	5	4	2	1	0.5

RAM usage. RAM resources are required by every layer accelerator for inter- and intra- accelerator data buffering. Although current cloud FPGAs provide alternative on-chip storage like Ultra RAM (URAM), the RAM consumption model is similar to BRAM. So we make use of the BRAM model in estimating the RAM usage of the accelerator system.

Except the original input data and weight data, output feature data between layer accelerators are stored in on-chip RAM as much as possible. The RAM consumption is calculated in two aspects: 1) RAM for data buffers in layer accelerators, 2) RAM for data buffers between layer accelerators. The system memory cost is the sum of all the memory cost listed above.

The internal BRAM used by the layer accelerator is affected by the tile size of the input and output features (T_{r_in}, T_{c_in}) and (T_r, T_c), the tiled input/output channel number (T_m, T_n) as well as the architectural information of (S, P) for stride and padding of the layers in the input DNN model. The weight buffer size depends on the maximum kernel size of layers allocated to the layer accelerator. Furthermore, the memory banks between layer accelerators and between sub-nets are other major RAM consuming components and are estimated with the value of M, R, C related to the output of the layers and the total instantiated number of them.

A single BRAM block is constrained to one read and one write port. RAMs in FPGA are organized as distributed blocks with fixed memory capacity, which is 18Kb block BRAM (288Kb block for URAM) in our target platforms. As a result, in our BRAM usage approximation, each of the partitioned buffers occupy at least one BRAM block. The approximated RAM consumption for the accelerator system is shown in Equation 5 and 6, where M_{acc} and M_{bank} stands for the RAM occupied by the layer accelerator and the memory bank between layer accelerators. The $factor$ refers to the $\left\lceil \frac{sizeof(datatype)}{18bit} \right\rceil$ and the $blocksize$ refers to the 1K depth of the RAM blocks in our platform. T_{r_in} and T_{c_in} are two parameters calculated with the (S, P) and (T_r, T_c) mentioned above. When the buffer size is small (less than 16), such as the weight buffers for most of the network models, we do not count them as BRAM since they are implemented with LUTRAM resources in the FPGA chip.

$$M_{acc} = \sum \begin{cases} 2 \times T_n \left\lceil \frac{T_{r\_in} \times T_{c\_in} \times factor}{blocksize} \right\rceil \\ 2 \times T_n \times T_m \times \left\lceil \frac{K_{max} \times K_{max} \times factor}{blocksize} \right\rceil \\ T_m \times \left\lceil \frac{T_r \times T_c \times factor}{blocksize} \right\rceil \end{cases} \tag{5}$$

$$M_{bank} = \sum M_{max} \times \left\lceil \frac{R_{max} \times C_{max} \times factor}{blocksize} \right\rceil \tag{6}$$

The on-chip memory size is used as one of the physical constraints during the design space exploration in Section 5 to ensure that the accelerator could fit into our device.

3.3.2 Accelerator performance model. Before constructing a network, a performance model of each layer accelerator is built to guide the selection of suitable accelerator parameters.

Convolutional accelerator. With the tiling-based convolutional accelerator design method, the cycles to compute a tiled convolution task is modeled as Equation 7.

$$Conv\_kernel_{cycles} = T_r \times T_c \times K^2 \tag{7}$$

The execution of a convolution layer is divided into multiple rounds of computation on tiled input data matrix. Since we keep the T_r and T_c as constant values for our convolutional accelerator, latency in terms of cycles for reading input (lat_{in}) and outputting results (lat_{out}) of each call of the tiled convolution is related to the interface capacity and data volumes.

We also use double buffering throughout the template library to improve the overall performance of the convolution layer design. We apply double buffer for inputs and weights but not outputs in order to simplify the data flow of the template function. With the consideration of input/output data transfer latency, we use Equation 8 to estimate the performance of running convolutional layer.

$$\begin{aligned} l_{lat} = &\left\lceil \frac{R}{T_r} \right\rceil \times \left\lceil \frac{C}{T_c} \right\rceil \times \left\lceil \frac{M}{T_m} \right\rceil \times (\left\lceil \frac{N}{T_n} \right\rceil \\ &\times max(lat_{in}, Conv\_kernel_{cycles}) + lat_{out}) \end{aligned} \tag{8}$$

Pooling layer. The pooling layer accelerator has a similar computation pattern as the convolutional accelerator. Since the input and output channel number for the pooling layer are equal, it does not take the output loop dimension change into account. Following the same method of modeling the convolution layer, the pooling layer accelerator performance could be estimated with the same equation above by simply setting both the $\left\lceil \frac{M}{T_m} \right\rceil$ and $Conv\_kernel_{cycles}$ to 1.

FC layer. Fully connected layers in convolutional neural networks are always bounded by input data transfer due to the large amount of weight data and relatively small amount of computations. We share our convolution layer accelerator design with the fully connected layer but setting the T_r and T_c to 1 to turn it into a matrix multiplication accelerator, so the performance models are also applicable.

Our accelerators are optimized with double buffering for the feature data input, so each of the computation in a layer is running in parallel with the data load to increase total throughput. Due to the varying data requirements for different layers, in order to speed up the overall execution, we balance the sub-net computation as well as the computation of the layer accelerators in a sub-net; the optimization algorithms are presented in Section 5.

4 SYSTEM ARCHITECTURE GENERATION

The complexity and variability of CNN applications result in a difficult hardware accelerator system design process, especially when the cloud FPGA platform is equipped with the SSI technology. Our design methodology incorporates a system architecture that shares hardware resources between adjacent layers flexibly. The automated system generator also requires a targeted system architecture in order to connect to the pre-designed shell that wraps the on-cloud FPGA.

Table 4: API library overview.

Class	Function	Description
Data Transfer	ExttoSysbCpy	External to system data trans
	SysbtoExtCpy	System to external data trans
	BuftoBufCpy	Buffer to buffer data trans
Accelerator Control	AccStart	Start Accelerator execution
	AccStop	Stop Accelerator execution
	AccStatusChk	Check Accelerator status
	AccInt	Accelerator Interrupt control
	ParamTrans	Transfer accelerator params
Thread Control	Set Flag	Set thread flag
	ThreadStatus	Report thread status
Data Arrangement	3DReorder	3D data re-ordering
	2DReorder	2D data re-ordering

4.1 Hardware-software Scheme

The FPGA platform is treated as a callable accelerator within our design. The control of the accelerator status is managed by the host processor, accomplished by light weight tasks on the processor. The entire DNN inference process is wrapped as a task unit and allocated to the FPGA accelerator. The accelerator is wrapped as a PCIe device and the application call sets do not need to know the details of the hardware.

The API functions required during the runtime of the accelerator system are shown in Table 4. The input data is first transferred to the pre-allocated accelerator memory space. The start of the inference process is controlled by host software, and the runtime will also be monitored. System debugging is supported by the software for convenience.

Due to the nature of CNN data access patterns, the code that runs on the host processors need to be compatible with the accelerator system. Hence, we generate the corresponding software functions that take charge of weight arrangement as well as function calls to the individual sub-nets composing the network model.

4.2 On-cloud Integration

Cloud FPGAs are designed to provide a huge amount of logic and memory resources compared to embedded platforms. In order to fully utilize the on-chip memory of the targeted FPGA, we instantiate as many memory buffers as required by the accelerator to store data on-chip to take advantage of the data locality to improve performance. The cross-die connection of the FPGA becomes a performance bottleneck. Long routing lines prevent the design from meeting timing constraints or force a lower frequency. In order to address this issue, we allocate the sub-nets to different silicon dies in the FPGA chip with additional physical constraint during the implementation process. The sub-nets transfer data through the memory banks with the help of the timing insensitive FIFOs in the AXI interconnect. Host to accelerator system communication is carried out through the pre-designed shell of the cloud platform and the accelerator system also has access to the entire off-chip memory space. Our targeted application system consists of four major components, 1) sub-net constructed with layer accelerators and corresponding control logic, 2) pre-designed shell of the platform, 3) host CPU and memory, and 4) external memory on the FPGA side. The architecture of generated system is shown in Figure 5.

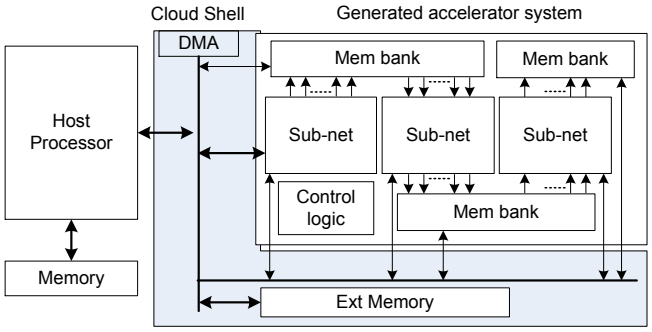

Figure 5: Targeted system architecture.

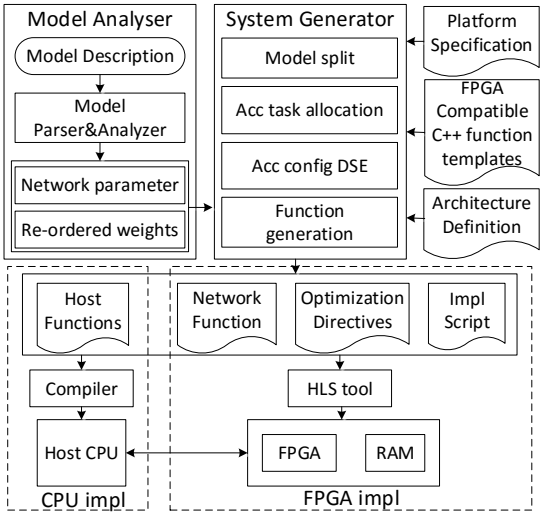

Figure 6: Cloud-DNN generation flow.

The parameters of the accelerator are customized based on the input network model during the system generation process. The detailed accelerator configuration algorithm is discussed in Section 5.

5 CLOUD-DNN FRAMEWORK IN DETAIL

The detailed functionality of our Cloud-DNN flow is presented in this section, including the detailed model analyser, the system generator, system construction and the host software generation.

5.1 Model Parser and Analyzer

By taking advantage of the flexibility of high-level programming, machine learning frameworks for CNN implementation contain detailed and accessible network parameter information, e.g. data size, intermediate result dimensions, etc. However, HLS-based hardware implementations require all the buffer sizes and dimensions to be known before compilation. In order to support existing deep neural network flows, we extract the detailed network structure description for the layers defined in the original network model. It includes all layer types and input/output feature size, stride size, padding size, etc.

5.2 Sub-net Instantiation

5.2.1 Model split. As all the multiplication and addition operations in our design are provided by the on-chip DSP resource, so the computational capacity of the FPGA is determined by the capacity of on-chip DSP. Given a NN layer, the execution latency is proportional to the DSP resource allocated to it [12]. For our current designs targeting a single FPGA chip in the cloud, there are 3 silicon dies wrapped in one FPGA chip. Based on our design methodology for the accelerator templates, most routing resources are consumed by the long interconnect in the accelerator. Considering the large amount of computation intensive layers in the original input model, in order to satisfy the computational requirement as well as reduce the timing issue caused by the crossing die routing, we split the original model into 3 sub-nets according to the number of on-chip silicon dies based on the computation requirement of the layers in a model. Also, in order to simplify the control of the accelerator, we do not violate the order and the branches in the original network model. Thus, the input model is split into 3 sub-nets with balanced computational requirement. In this way, the critical routing part for the accelerators and between accelerators remains on the same die. The data communication between the sub-nets are through external memory or across the silicon dies that handled by the bus interface with built-in buffering logics.

5.2.2 Layer accelerator task allocation. After the original DNN model is split into sub-nets, the design space exploration for the sub-nets is processed independently. There are two tasks to be accomplished: 1) accelerator task allocation and 2) accelerator configuration design space exploration. Although the configuration of a single accelerator may affect the performance of the sub-net it belongs to, the optimization target is a balanced overall throughput. With our model split method, we need to obtain an optimal configuration for all the sub-nets. Although convolutional layers are computationally intensive, the data bandwidth is also a critical requirement for optimal accelerator performance [20]. Current cloud-based FPGAs have strict data transfer bandwidth (e.g., AWS F1 provides overall read/write at 6.5GB/s from host CPU to FPGA). These metrics have been taken into consideration in our accelerator parameter computation model. Therefore, all the parameters for the accelerators that have been discussed in Section 3 will be configured during the parameter configuration procedure with the constraint of hardware resource, on/off-chip bandwidth and the characteristic of input model.

5.2.3 Problem definition and solver. Model split and accelerator allocation are both critical processes to the overall system performance. To solve it, we formulate it as the following problem.

Given the number of DSP and RAM (BRAM and URAM) resources per die, $\{DSP_i\}$ and $\{RAM_i\}$, with a *Bandwidth* limitation, maximizing the overall throughput with the layer specifications of a DNN model, where the layers are denoted as $\{L_j\}$. The accelerators in a single die as well as the sub-nets are working in a pipelined manner, so the overall latency is dominated by the maximum latency of a single accelerator in each die. Bandwidth limitation is used to verify if the current accelerator configuration could be achieved. The parameter configuration exploration is shown as Algorithm 1. Firstly, the input network model is analyzed and split into 3 sub-nets based on the computation requirements of the layers. Then a straightforward greedy search based design space exploration algorithm is illustrated for each of the sub-nets to determine the best configuration of the $(Acc\_num, T_m, T_n, T_r, T_c)$ parameters

with the additional constraints for verification. At the same time, a list of the number of accelerators for each silicon die and the label of layers in the model that are allocated for the accelerators is generated accordingly, which are Acc_num_k and l_list. The *GOP* in the algorithm posedu code denotes the computational operations required by a certain set of layers. The *DSE* is the design space exploration function that searches for the best accelerator configuration with a given number of DSP for a set of layers based on Equation 7 and 8. We also consider the DSP consumption of pooling layer when necessary as well as the computation latency of both pooling and FC layers.

Algorithm 1 Accelerator configuration search

Input: $DSP_i, RAM_i, Bandwidth, L_j$
Output: k groups of $T_m, T_n, T_r, T_c, Acc\_num, l\_list$
1: Model analysis ← L_j
2: Split L_j with GOP to sub-nets → S_i
3: **for** S_i **do**
4: Task allocation → Acc_num, l_list
5: **for** Acc_num_k **do**
6: DSE ← $min(max(latency of Acc\_num_k))$
7: Verify parameters ← Bandwidth
8: **end for**
9: Choose optimal T_m, T_n, T_r, T_c with Acc_num_k, l_list
10: **end for**

The best parameter setting is generated as a configuration file, based on which the accelerator templates are customized and then the sythesizable C++ code is generated.

5.3 System Construction

The sub-net cores and the system buffers are instantiated separately to provide flexibility to the system buffer control and the implementation flow. The targeted system includes the following three major components.

5.3.1 Sub-net core synthesis. After the C++ code is generated with the customized parameters, the accelerator code is synthesized with HLS optimization pragmas. The HLS tool converts the C++ code for the sub-nets into an exportable hardware IP.

5.3.2 System buffer generation. The system buffers are generated with the hardware synthesis tool using pre-designed Tcl scripts. With our structural design, the Tcl files only requires the generation parameters to generate the buffers. During the accelerator system construction, the flow call the buffer generation scripts accordingly to the accelerators that are generated and instantiate it as a separate module. This will also ease the physical implementation by providing clear edge for the memory module, so that the large size of sub-net core accelerators could meet the timing requirements.

5.3.3 Cloud shell integration. Generally, the cloud platforms provide a unified interface as a shell for the host CPU to communicate and manage the FPGA resources through PCIe interface. In order to provide a high data transmission speed as well as control flexibility, the input of our accelerator system is defined with AXI compatible interfaces. Specifically, AXI_Lite for accelerator control and AXI bus is for data transfer. The sub-net accelerators are attached as an AXI memory mapped device within the generated system. The cloud shell is instantiated together with an AXI Interconnect component that provides enough AXI interfaces to connect with our generated accelerator.

5.4 Host Software Generation

The corresponding software for the host processor to control and communicate with the generated accelerator is also automatically generated after the accelerator parameters have been determined. All functions in the inference process of the network model are parameterized, as presented in Section 4, and the host function refers to the parameter file generated by system generator and accomplishes the data transfer and accelerator control.

All the above processes are automated. Our flow starts from the input model analysis and proceeds to the subsequent stages until finally generating the accelerator system targeting the specific cloud platform.

6 EVALUATION

Our flow aims to provide general support for CNN model implementation on cloud FPGAs, so we test multiple CNN models with their FPGA based quantized versions. We first evaluate the system performance generated with our flow. After that, we compare it to the software version as well as state-of-the-art FPGA implementations.

6.1 Experimental Setup

The generation flow is designed with a Python interface. The accelerator core IP is generated with Vivado_HLS (v2017.4) and the corresponding memory banks are generated with Vivado in the same design package, so as the implementation of the on cloud system. We use both the AWS F1 EC2 2X.large instance (Shell version 1.4.2) and our own local FPGA cloud as our target platforms, which are all equipped with one Xilinx VU9P FPGA chip. The FPGA contains 2585K logic elements and 6840 DSPs with 75.9Mb block RAMs and 270 Mb UltraRAMs. Software implementation of the network models are running on 6 core Intel Xeon E5-2430 CPU with 15MB cache and 8 core Intel Xeon E5-2609 CPU with nVidia GPU Pascal Titan X as comparison. The network models involved in our tests are AlexNet, VGG-16 and ResNet-50.

6.2 Cloud-DNN System Performance

We first evaluate the accelerator performance generated by our flow for the given network models. The overall performance is collected by measuring the processing time and throughput for running inference on our cloud FPGA platforms. The accelerator customization and system resource consumption of the benchmark models are measured and reported in Table 5. The data types for these DNN models are all fixed16. For both AWS and local cloud FPGA platforms, we use the same accelerator configuration. The selected accelerator number and parameter settings are generated by the algorithms described in Section 5. We do not consider Flip-Flop resource since FPGA is a register-rich platform.

The generated accelerator system targets a higher utilization of the on-chip resources especially for DSPs that provide the computation capacity. For all the models involved in our experiments, the utilization of the DSP resources is higher than 78%. Different DNNs

Table 5: Generated system configuration and performance.

Model	Configuration (Acc_type, Tm,Tn,Tr,Tc) (C=Conv, F=FC)			Plat.	Resource(%)				Clock (MHz)	Perf. (ms/img)	Overall GOPS
					DSP	BRAM	URAM	LUT			
AlexNet	[C,96,3,28,28]	[C,55,13,13,13]	[C,256,5,13,13]	AWS	82.05	60.16	83.02	61.38	125	3.96	335.86
	[C,128,11,27,27]	[C,64,17,13,13]	[F,512,1,1,1]	Local	82.05	54.87	83.02	57.62	214	2.32	575.00
VGG-16	[C,32,1,32,32]	[C,37,8,28,28]	[C,128,5,28,28]	AWS	78.2	80.2	84.4	64.7	125	28.96	1068.37
	[C,64,11,32,32]	[C,86,7,28,28]	[C,128,5,28,28]	Local	78.2	74.5	84.4	58.5	214	16.92	1828.61
	[C,128,8,28,28]	[C,128,7,28,28]	[F,512,1,1,1]								
Resnet-50	[C,64,6,28,28]	[C,16,7,1,1]	[C,64,7,14,14]	AWS	80.25	83	82.01	62.4	125	13.9	721.58
	[C,64,7,28,28]	[C,128,5,14,14]	[C,64,7,7,7]	Local	80.25	76	82.01	58.9	214	8.12	1235.35
	[C,128,7,28,28]	[C,256,4,14,14]	[C,128,7,1,1]								

require different accelerator system configurations to perform optimally. The architectural differences of the input DNN models also show a different utilization on memory (BRAM and URAM) and DSP resources. VGG-16 and Resnet-50 also require the DDR memory during the runtime to temporarily store the intermediate data due to the big volume of intermediate data.

The accelerators on local FPGA cloud show better performance in terms of clock frequency than those on AWS cloud, as the clock of the local FPGA is provided by the Memory Interface Generator (MIG) module. Thus, it can be more flexibly configured to different values. However, the clock of accelerators on AWS cloud is provided by AWS F1 shell IP which aims at better programmability by sacrificing flexibility in configuring the clock frequency. Though there exist solutions such as crossing clock domain design that can achieve the same goal, it inevitably complicates the system design and makes it less robust due to the incompatibility between two clock domains, even at a great cost of engineering efforts. Thus, we only present the more dependable/trustworthy results on AWS cloud in this paper, and leave a more sophisticated implementation on AWS cloud as future work of this project. Also, our local shell is lighter than the shell IP that is provided by AWS F1, therefore, the physical constraints for our local implementation are easier to achieve compared to the AWS F1 implementation.

6.3 Comparison with Software Implementation

We compare the performance of our accelerators to the pure software counterparts. The results are shown in Table 6 where the performance of CPU is taken as baseline.

For all three models, our Cloud-DNN generated design shows better performance than CPU and GPU implementations. Our proposed design shows a similar performance trend with CPU and GPU regarding the network size and the corresponding computation requirement changes. Although both FPGA and GPU platforms can deliver great performance advantages compared to CPU, our FPGA-based design can provide much better energy efficiency by taking advantage of the high computation density and the low clock frequency.

6.4 Comparison with Prior Implementations

We compare the performance of the network models generated with our flow to the state-of-the-art published results[5, 20, 26–28], including both fully-fledged frameworks and dedicated accelerators as shown in Table 7.

Our generated systems outperform previously designs in terms of per image processing latency and system throughput. We also compare the FPGA energy efficiency to the existing designs. Our generated systems show 1.5× to 11.2× better energy efficiency in GOPS/W by taking advantage of the higher computation density on the cloud platform.

7 RELATED WORK

Among the machine learning frameworks, Caffe and Tensorflow can be considered as the most popular ones working with DNN hardware implementations since extensive number of recently published work take Caffe and Tensorflow as their front-end for DNN training [2, 3, 5, 7–9, 12, 20]. These popular frameworks support a Python/C++ interface for DNN definition and training, which can be convenient and efficient for future export to FPGA-based design flow, especially for automated tool for generating FPGA-based DNN accelerators.

The authors in [2] propose an automation tool called DNNBuilder for building DNN accelerators on FPGAs to satisfy the performance and energy efficiency demands on mobile devices and cloud severs. This tool takes Caffe and Tensorflow as front-end to define and train DNNs in software and uses pre-built RTL components as the building blocks to generate DNN accelerators. A framework called DNNWEAVER is proposed in [3] that uses hand-optimized design templates for accelerator generation with the input Caffe specifications. Same machine learning frameworks are also used in the design proposed in [27] as the front-end, but this work employs hybrid templates (RTL+HLS) to meet the tradeoff between performance and flexibility.

Previous literature also focus on building framework extensions from Caffe and Tensorflow, to support FPGA-based DNN implementation. A Caffe extension in [26] is accomplished by accelerating CNNs using a proposed uniform accelerator design. However, this universal solution also leads to low efficiency while handling different applications. Caffe framework is also modified by the author in [29] to support image classification on FPGA. They use a Winograd convolution algorithm and involve HLS as the implementation method. However, their system throughput only reaches 50 GFLOPS.

8 CONCLUSION AND FUTURE WORK

In this paper, we proposed Cloud-DNN, an open-source framework that automatically maps DNN models trained by Caffe to FPGAs in the cloud for inference acceleration. The Cloud-DNN framework provides automated structural optimizations during the FPGA implementation, and creates network description with our pre-designed C++ template library. The corresponding host code is also generated simultaneously. The DNN implementations show

Table 6: Comparison with software implementations.

Platform	CPU			CPU+GPU			Cloud-DNN-Local		
Device	E5-2430			E5-2609 + Pascal Titan X			VU118		
Model	AlexNet	VGG-16	ResNet-50	AlexNet	VGG-16	ResNet-50	AlexNet	VGG-16	ResNet-50
Data Type	float32			float16			fixed16	fixed16	fixed16
Clock(MHz)	1.9GHz			1GHz			214MHz		
Latency/Image (ms)	242.562	794.238	557.5	5.0486	25.7583	13.79	2.32	16.92	8.12
Speedup(×)	1	1	1	48.05	30.83	40.427	104.55	46.94	68.66

Table 7: Comparison with other designs.

Design	[26]	[20]	[5]	[27]	[28]	Cloud-DNN-AWS	Cloud-DNN-Local
CNN model	VGG16	AlexNet	VGG16-SVD	VGG-19	VGG-16	VGG-16	
Platform	VX690T	VX485T	XC7Z045	Str. V GSMD5	Arr. 10 GX1150	VU9P	
DSPs(used/total)	2833/3600	2240/2800	780/900	1036/1590	1518/1518	5349/6840	
Clock(MHz)	150	100	150	150	200	125	214
Data type	fixed16	float	fixed16	fixed16	fixed16	fixed16	
Power(Watt)	26	18.61	9.63	~25	-	48.62	49.25
Lat./Img.(ms)	65.13	21.61	224.60	-	42.98	28.96	16.92
Thro.(GOPS)	354	61.62	136.97	364.36	720.15	1068.37	1828.61
Eff.(GOPS/W)	13.62	3.31	14.22	14.57	-	21.97	37.13

comparable or better performance to state-of-the-art solutions on FPGAs as well as better energy efficiency compared to CPU and GPU implementations. This framework enables users to quickly create and deploy DNNs on cloud FPGAs. Thus, we provide an efficient and high-performance/energy efficiency FPGA solution for Caffe frameworks in the cloud so users have an additional choice other than always relying on CPU and GPU.

Our workflow is designed in a modular fashion which allows easy extensions for new layer types. There are some potential extensions of this work, such as supporting a wider range of DNNs. Also extending our current flow to support other frameworks like TensorFlow, MXNet and PyTorch is under exploration. We also plan to extend Cloud-DNN to utilize multiple FPGAs in the future. Our current release could be found at https://github.com/microideax/Open-Dnn.git.

ACKNOWLEDGMENTS

This work is partly supported by the National Research Foundation, Prime Minister's Office, Singapore under its Campus for Research Excellence and Technological Enterprise (CREATE) programme, and Alibaba Group through Alibaba Innovative Research (AIR) programme. It is also partly supported by the IBM-Illinois Center for Cognitive Computing System Research (C3SR) - a research collaboration as part of IBM AI Horizons Network.

REFERENCES

[1] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.
[2] Xiaofan Zhang et al. DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proc. of ICCAD*, 2018.
[3] Hardik Sharma et al. From high-level deep neural models to FPGAs. In *Proc. of MICRO*, 2016.
[4] Jialiang Zhang et al. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *Proc. of FPGA*, 2017.
[5] Jiantao Qiu et al. Going deeper with embedded FPGA platform for convolutional neural network. In *Proc. of FPGA*, 2016.
[6] Xiaofan Zhang et al. Machine learning on FPGAs to face the IoT revolution. In *Proc. of ICCAD*, 2017.
[7] Junsong Wang et al. Design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA. In *Proc. of FPL*, 2018.
[8] Huimin Li et al. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *Proc. of FPL*, 2016.
[9] Naveen Suda et al. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proc. of FPGA*, 2016.
[10] Su Liu et al. Real-time object tracking system on FPGAs. In *Proc. of SAAHPC*, 2011.
[11] Yufei Ma et al. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *Proc. of FPGA*, 2017.
[12] Xiaofan Zhang et al. High-performance video content recognition with long-term recurrent convolutional network for FPGA. In *Proc. of FPL*, 2017.
[13] Song Han et al. ESE: Efficient speech recognition engine with compressed LSTM on FPGA. 2016.
[14] Li Qin et al. Implementing neural machine translation with bi-directional GRU and attention mechanism on FPGAs using HLS. In *Proc. of ASP-DAC*, 2019.
[15] Xinheng Liu et al. High level synthesis of complex applications: An h. 264 video decoder. In *Proc. of FPGA*, 2016.
[16] Kyle Rupnow et al. High level synthesis of stereo matching: Productivity, performance, and software constraints. In *Proc. of FPT*, 2011.
[17] Deming Chen et al. Lopass: A low-power architectural synthesis system for FPGAs with interconnect estimation and optimization. *TVLSI*, 18(4):564–577, April 2010.
[18] Andrew Canis et al. LegUp: high-level synthesis for FPGA-based processor/accelerator systems. In *Proc. of FPGA*, 2011.
[19] Emanuele Del Sozzo et al. On the automation of high level synthesis of convolutional neural networks. In *Proc. of IPDPSW*, 2016.
[20] Chen Zhang et al. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.
[21] Yongming Shen et al. Maximizing cnn accelerator efficiency through resource partitioning. In *Proc. of ISCA*, 2017.
[22] Xilinx. Large FPGA methodology guide. 2012.
[23] Yangqing Jia et al. Caffe: Convolutional architecture for fast feature embedding. In *Proc. of ACMMM*, 2014.
[24] Song Han et al. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. 2015.
[25] Philipp Gysel et al. Hardware-oriented approximation of convolutional neural networks. 2016.
[26] Chen Zhang et al. Caffeine: towards uniformed representation and acceleration for deep convolutional neural networks. In *Proc. of ICCAD*, 2016.
[27] Yijin Guan et al. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *Proc. of FCCM*, 2017.
[28] Yufei Ma et al. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *Prof. of FPL*, 2017.
[29] Roberto DiCecco et al. Caffeinated FPGAs: FPGA framework for convolutional neural networks. In *Proc. of FPT*, 2016.