DNNBuilder: an Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs

Xiaofan Zhang[1]*, Junsong Wang[2], Chao Zhu[2], Yonghua Lin[2], Jinjun Xiong[3], Wen-mei Hwu[1], Deming Chen[1]
[1]University of Illinois at Urbana-Champaign, [2]IBM Research China, [3]IBM T. J. Watson Research Center
{xiaofan3,w-hwu,dchen}@illinois.edu, {junsongw,bjzhuc,linyh}@cn.ibm.com, jinjun@us.ibm.com

ABSTRACT

Building a high-performance FPGA accelerator for Deep Neural Networks (DNNs) often requires RTL programming, hardware verification, and precise resource allocation, all of which can be time-consuming and challenging to perform even for seasoned FPGA developers. To bridge the gap between fast DNN construction in software (e.g., Caffe, TensorFlow) and slow hardware implementation, we propose DNNBuilder for building high-performance DNN hardware accelerators on FPGAs automatically. Novel techniques are developed to meet the throughput and latency requirements for both cloud- and edge-devices. A number of novel techniques including high-quality RTL neural network components, a fine-grained layer-based pipeline architecture, and a column-based cache scheme are developed to boost throughput, reduce latency, and save FPGA on-chip memory. To address the limited resource challenge, we design an automatic design space exploration tool to generate optimized parallelism guidelines by considering external memory access bandwidth, data reuse behaviors, FPGA resource availability, and DNN complexity. DNNBuilder is demonstrated on four DNNs (Alexnet, ZF, VGG16, and YOLO) on two FPGAs (XC7Z045 and KU115) corresponding to the edge- and cloud-computing, respectively. The fine-grained layer-based pipeline architecture and the column-based cache scheme contribute to 7.7x and 43x reduction of the latency and BRAM utilization compared to conventional designs. We achieve the best performance (up to 5.15x faster) and efficiency (up to 5.88x more efficient) compared to published FPGA-based classification-oriented DNN accelerators for both edge and cloud computing cases. We reach 4218 GOPS for running object detection DNN which is the highest throughput reported to the best of our knowledge. DNNBuilder can provide millisecond-scale real-time performance for processing HD video input and deliver higher efficiency (up to 4.35x) than the GPU-based solutions.

1 INTRODUCTION

FPGAs have become promising candidates for DNN implementations recently [1–7]. FPGAs can be customized to implement DNNs with improved latency and energy consumption compared to CPU- and GPU-based designs. Meanwhile, FPGAs offer much more flexibility than ASICs because of their reconfigurable feature. These characteristics allow FPGAs to satisfy diverse DNN-based applications in both cloud- and edge-based computing cases. High-end FPGAs with sufficient logic, computation, and memory resources can deliver significant concurrent processing abilities for cloud services, while embedded FPGAs can provide high energy efficiency to overcome the power/energy limitations under various edge-computing scenarios [8].

Developing DNN designs on FPGAs, however, presents significant challenges: the tedious RTL programming, the intricate verification problems, and the time-consuming design space exploration process, all of which often hinder FPGA's adoption by application developers. Diverse requirements of applications and targeted FPGAs come with completely different computation and memory resources. Cloud applications require sophisticated resource allocation strategies to accommodate flexible batch-processing and meet throughput requirements with given FPGAs. Edge applications usually ask for real-time processing of streaming inputs that limit the FPGA's ability to batch the data for increased throughput, as the additional latency incurred by batch process can exceed what is allowed by real-time performance requirements. Also, most of the edge applications require processing high-definition (HD) images/videos, which generates even higher requirements for feature map storage and computation power. To address these challenges, we propose DNNBuilder for building DNN designs on FPGAs. It is an automated tool flow which can transform DNN designs from popular deep learning frameworks (such as Caffe, TensorFlow) to highly optimized board-level FPGA implementations with considerations of available computation units, on-chip/off-chip memory, and external memory access bandwidth in targeted FPGAs. To summarize, the main contributions of this paper are as follows.

(1) **An end-to-end automation tool** called DNNBuilder, which provides an integrated design flow from deep learning frameworks to board-level FPGA implementations. With DNNBuilder, users are no longer required to program in RTL or to perform manual resource allocation and optimization for deploying DNNs on FPGAs.

(2) **A flexible quantization scheme** for smooth tradeoffs between limited resources on FPGAs and desired output accuracy. Our design supports arbitrary quantization for weights and activations either within a layer or across layers in DNNs. It also supports binary and ternary networks.

(3) **A fine-grained layer-based pipeline architecture and a column-based cache scheme** can deliver high throughput (even without batch processing), low startup latency, and low on-chip memory consumption. With the proposed designs, we reduce 7.7x latency and 43x BRAM usage than the conventional structure. These features ensure the millisecond-scale response and HD input support of our design.

(4) **Highly optimized RTL network components** that can be automatically generated for building DNN layers with high quality. Commonly used loop structures in DNNs are captured by a parameterized process engine (PE), which can be configured to deliver the best performance under constraints of given FPGAs.

(5) **An automatic resource allocation management** that provides computation and memory resource allocation guidelines across network layers. It considers the external memory access bandwidth, data reuse behaviors, computation resource availability, and network complexity.

The rest of the paper is organized as follows. Sec. 2 introduces the related work while Sec. 3 summarizes the main challenges we

*This work was done while Xiaofan Zhang was an intern at IBM Research China.

ICCAD '18, November 5–8, 2018, San Diego, CA, USA

ACM ISBN 978-1-4503-5950-4/18/11…$15.00
https://doi.org/10.1145/3240765.3240801

encountered. Sec. 4 and 5 describe the design flow of DNNBuilder and the architecture of generated accelerators. The resource allocation scheme is presented in Sec. 6. Sec. 7 shows the experimental results and Sec. 8 concludes this paper.

2 RELATED WORK

For handling edge applications, a DNN is implemented in [1] which covers weight quantizations for relaxing the computational pressure in embedded FPGAs. In [2], the authors integrate conventional and Winograd-based convolution (CONV) for running DNNs in embedded FPGAs. A framework is proposed in [3] for accelerating the extreme low bit-width DNNs which supports hybrid quantization to balance the accuracy and throughput performance. To support cloud services, FPGAs with more computation and memory resources are deployed. Authors in [4] design a resource allocation algorithm and hierarchical memory system to reach the minimum DNN inference latency. In [5], a Winograd-based solution is used to reduce the required multiplications using the Arria 10 FPGA. The same FPGA is used in [6] for accelerating the VGG network. To speedup the CONV, authors in [7] evaluate the Winograd- and FFT-based algorithms and implement a face recognition accelerator on FPGA.

Previous literature also focuses on building automation tools for fast deploying DNNs to FPGAs. The authors in [9] design a framework with systolic arrays to speedup DNN inference. The framework in [10] employs an unified RTL design for CONV layers and runs software in the host CPU to ensure different network configurations. However, the Fully-connected (FC) layers are not implemented on FPGA. More frameworks are proposed to automatically map DNNs onto FPGAs using RTL [11, 12] or RTL-HLS templates [13].

Previous automation tools suffer several drawbacks: 1) relatively low hardware efficiency, e.g., the use of unified computation units (e.g., computation engine with fixed size) lowers the chance to reach the optimal design when compared to the dedicated designs for different networks, 2) limited scalability for mapping different networks onto different FPGAs, especially for the embedded FPGAs with limited computation and memory (including capacity and bandwidth) resources, and 3) no sufficient design space explorations integrated in the automation tool.

Our proposed tool can well address these drawbacks and provide a built-in design space exploration tool for high-performance and efficient designs. We introduce a fine-grained pipeline structure, a novel caching scheme between pipeline stages, and highly optimized RTL network layers with arbitrary quantizations to deliver high throughput, low latency, and desired network accuracy.

3 DESIGN CHALLENGES

The memory requirement and computational complexity of DNNs are high, which makes their deployment on FPGA a challenging and time-consuming task. There are two strategies for designing FPGA-based DNN accelerators: using either a recurrent structure or a pipeline structure. The former, such as accelerators in [5, 10, 11], constructs a unified computation unit with nearly full exploitation of the on-chip resource and shares the same unit among different DNN layers. For each layer, accelerator needs to handle required computation and save the intermediate results (feature maps) to internal (FPGA on-chip) memory or external memory (usually the DRAM). The latter, such as the accelerator proposed in [14], is a pipeline implementation in which each neural network layer is implemented as a separate pipeline stage instantiated on FPGA.

We choose the pipeline structure for our tool based on the following considerations. First, we need to balance the limited on-chip memory resource with the support of HD inputs. It is difficult to generate adaptable designs for both edge and cloud devices using recurrent structure. It requires either high memory bandwidth for swapping intermediate results to/from external memory (DRAM) or a large capacity on-chip memory for keeping all the feature maps, either of which is hard to satisfy in edge-devices. Also, the HD inputs further aggravate the resource shortage with much larger feature map generated and required memory footprint. Second, as an automation tool, DNNBuilder needs to provide flexible support of various network configurations with different computational and memory demands. A pipeline structure allows to implement dedicated design for each layer separately according to its computation and memory demands. In contrast, a recurrent structure has to choose a computation engine with uniform size, such as the 256×256 matrix multiplication unit in Google's TPU [15] or the three inner-most for-loops of CONV instantiated on FPGA as the computation engine in [10]. This means that operations in an incoming neural network need to be transformed (e.g., padding, matching, or partition) before feeding to the computation engine which reduces efficiency. Third, a pipeline architecture can well adapt to the resource allocation guidelines and deliver high throughput performance for better support of streaming inputs.

To overcome a pipeline's startup latency (the time between loading the first input at the first pipeline stage and generating the first output at the last pipeline stage) and large inter-stage cache overhead, we employ a fine-grained pipeline structure to reduce the pipeline latency and a column-based cache scheme to lower the required inter-layer memory space in full support of the HD inputs. Our design can deliver a millisecond-scale response for HD image/video inputs, which makes it possible to deploy DNNBuilder generated DNN accelerators on a latency sensitive system, such as Advanced Driver Assistance Systems for vehicle, and pedestrian and obstacle detections.

4 PROPOSED AUTOMATION FLOW

DNNBuilder produces board-level FPGA implementations in three steps: *Design*, *Generation*, and *Execution* (Fig. 1). After networks are determined, RTL codes and corresponding files for running DNN accelerators on FPGAs can be generated in seconds.

During the *Design* step, a targeted network is designed and trained using deep learning frameworks which in general employ CPUs and GPUs. After training, network definition files ("DNN def." in Fig. 1) and trained weights are passed to the next step. To ensure design freedom specified by users, the proposed flow supports arbitrary quantization schemes not only for functions within a layer (e.g., CONV, Relu), but also for inputs across layers (e.g., different weight/bias quantizations for layer i and $i + 1$) to explore tradeoffs among inference accuracy, resource utilization, performance, etc. One important feature of the *Design* step is that it receives feedbacks from performance estimation ("P. estim.") in *Generation.* If the current DNN runs slower or consumes more resources than

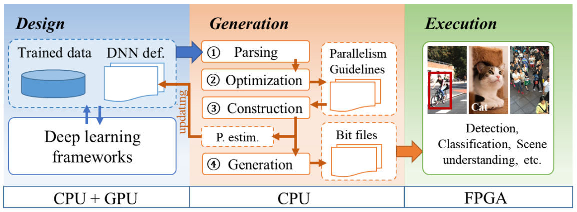


Figure 1: Design flow of using DNNBuilder containing DNN design & training (*Design*), network optimization & automated RTL code generation (*Generation*), and FPGA board-level implementation (*Execution*)

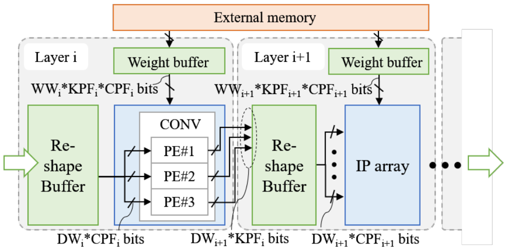


Figure 2: Accelerator architecture generated by DNNBuilder

expected, users could update their network designs, such as adjusting quantization schemes or modifying network layers to meet performance requirements. With several iterations between *Design* and *Generation*, the best network configuration can be developed for the targeted FPGA. This feature makes the hardware-software co-design possible.

In the *Generation* step, **network parsing** is the first process for decomposing targeted DNNs from input network models (which include network definitions in *.prototxt* and weight information in *.caffemodel* when using Caffe). Different network layers, e.g. CONV, Pooling, and FC layers, are decomposed and then mapped to our pre-built RTL components. The computational intensive nested loops are captured by parameterized PEs which are introduced in Sec. 5.1. **Automated optimization** works for exploring design space and balancing pipeline stages in DNNBuilder so that our design can achieve maximum throughput performance. We propose an automatic resource allocation scheme (details in Sec. 6) which generates optimization guidelines for parameter adjustment of the pre-built RTL components. Major elements in these guidelines include kernel/channel parallel factors and buffer sizes, and they can be manually modified for expert users. Following the guidelines, **network construction** is responsible for building DNN implementations with the pre-built RTL network components, dataflow controller, and memory instances, which are highly configurable to ensure the adaptability and scalability for various DNNs. **Code generation** generates accelerator related files for FPGA-based instances.

In the *Execution* step, the DNN accelerator is instantiated in FPGA hardware platforms with unified interfaces including a FIFO-like data input/output interface and a weight access interface connecting off-chip memory controller. In this final step, the DNN accelerator is ready for eventual deployment.

5 ACCELERATOR ARCHITECTURE DESIGN

DNNBuilder generates a pipeline structure where each pipeline stage corresponds to each major neural network layer, like CONV or FC layer, which dominates computation and memory consumption. The rest of layers such as batch normalization (BN), scale, and activation layers are aggregated to the major layers so that we reduce the number of pipeline stages for lower latency. In Fig. 2, we present two pipeline stages instantiated on FPGA for computing two CONV layers (i and $i+1$). This design consumes three types of FPGA resources as the computation resources (blue area), on-chip memory (green area), and external memory (orange area). Two datapathes are generated for passing input feature maps horizontally and trained weights vertically to the computation units. To maintain sufficient data supply, we setup two buffers for each pipeline stage as the reshape buffer for keeping slices of input feature map and the weight buffer for pumping in the trained weights from external memory. We define two parameters, the Channel Parallelism Factor (CPF) and the Kernel Parallelism Factor (KPF). CPF and KPF represent the number of input channels and the number of kernels, respectively, which can be processed in one IP array (a group of RTL network components) inside a pipeline stage. These two factors allow DNNBuilder to implement a two-dimensional parallelism scheme and adjust the resource utilization for each pipeline stage. CPF and KPF are calculated by our resource allocation algorithm which is discussed in Sec. 6. DNNBuilder supports flexible quantization schemes. As shown in Fig. 2, DW_i is the input data bit-width of the i-th layer while WW_i represents the bit-width of weights.

5.1 Computation Engine Design

The core functions in DNNs are carried out by the auto-generated RTL network components (e.g., CONV, FC, Pooling, BN, Relu, etc.) which are the RTL IPs for building the whole network. Since the same for-loop structure is frequently used in CONV and FC, we abstract it as a processing engine (PE), which can be unfolded in two dimensions corresponding to CPF and KPF. In our design, the CPF and KPF work for unrolling input and output channels respectively.

Fig. 3 presents a detailed structure of the PE which is designed for processing CPF number of input feature maps while the number of PEs is decided by KPF. To better explain how the PE works, we take a small-size CONV layer as a case study (notice that CPF and KPF are power of 2 in real case for efficient hardware design). Assuming there is a 4×3×3 input feature map in blue (the left side of Fig. 3 (a)), it is processed by six 4×2×2 kernels with green color (the middle of Fig. 3 (a)) with the channel/kernel parallel factors as CPF=2 and KPF=3 (total parallel: 2×3=6). Since CPF=2 and the kernel size is 2, a cube with 2 elements along X-, Y-, and Z-dimension is considered as one tile. Each tile requires four steps of processing following number ① to ④ because only one pixel in the X-Y plane is processed every step. In each step, two pieces of data (along Z-axis) from input feature maps (InFM) are collected (corresponding to the CPF) and they are simultaneously processed by the first three of the six kernels (corresponding to the KPF). In total, six multiply-accumulates are executed in parallel (which equals to CPF×KPF), and the first 3 partial sums (in orange) are generated. These partial sums still need 4 more steps to complete calculation with the next cube in input feature maps along the Z-axis (the second half along the Z-axis surrounded by the dashed line). Fig. 3 (b) shows the required input data of the PE in one step. Two elements (blue) are fetched from input feature map while six elements (green) are fetched from weights. The reshape buffer and weight buffer provide these data respectively by one memory access. In this example, the order of outputs is illustrated in Fig. 3 (c), following indexes from 1

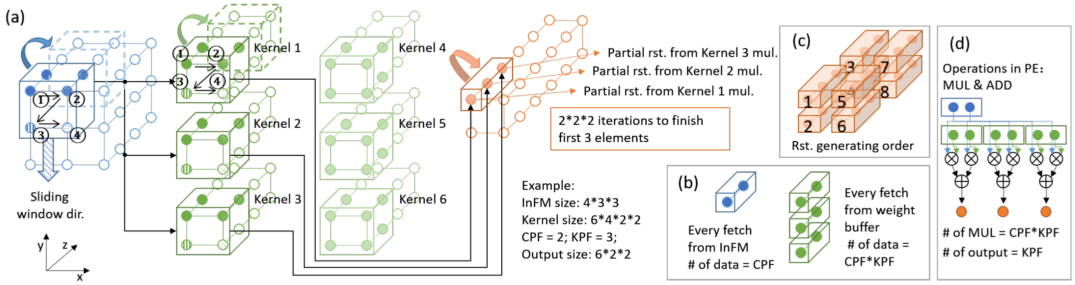

(a)
Sliding window dir.
y
z
x
1
2
3
4
Kernel 1
Kernel 2
Kernel 3
Kernel 4
Kernel 5
Kernel 6
Partial rst. from Kernel 3 mul.
Partial rst. from Kernel 2 mul.
Partial rst. from Kernel 1 mul.
2*2*2 iterations to finish first 3 elements
Example:
InFM size: 4*3*3
Kernel size: 6*4*2*2
CPF = 2; KPF = 3;
Output size: 6*2*2
(b)
Every fetch from InFM # of data = CPF
Every fetch from weight buffer # of data = CPF*KPF
(c)
1
2
3
5
6
7
8
Rst. generating order
(d)
Operations in PE: MUL & ADD
# of MUL = CPF*KPF
# of output = KPF

Figure 3: The proposed PE in DNNBuilder (aggregated functions are not shown, e.g. BN and ReLU)

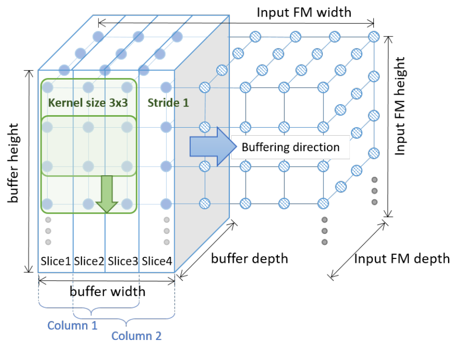


Figure 4: The proposed column-based cache scheme

to 8. The first three kernels contribute output 1, 2, 5, and 6 while the remainder three kernels generate the output 3, 4, 7, and 8. Fig. 3 (d) presents the multiply-accumulate operation in the PE with CPF=2 and KPF=3.

Following the idea above, we can design RTL IPs with high-performance and controllable resource overhead. As basic building blocks, these high-quality RTL IPs fundamentally ensure the high-performance design generated by DNNBuilder.

5.2 ON-Chip/Off-Chip Memory Management

In this section, we present two techniques to efficiently use the scarce on-chip memory in FPGAs for keeping input feature maps and buffering weight data.

5.2.1 Column-based cache scheme. For buffering input feature maps, previous designs (e.g., [14] and [16]) have stored input feature maps on chip to achieve higher throughput and avoid complicated data movement. The size of their input images is relatively small such as 256×256, 32×32, and 28×28 in ImageNet, Cifar100, and MNIST, respectively. But images captured in real-life can easily reach HD, like 1280×720. Although down-sampling may mitigate the issue somewhat, it is not always acceptable, especially for the small object detection. With the HD input, feature maps are enormous and impossible to be stored on chip entirely.

To address this problem, we propose a novel column-based cache scheme for only keeping a subset of the input feature map on chip. Fig. 4 shows a convolution with kernel size=3 and stride=1. Since slices 1~3 contribute to the first sliding window operation (from top to bottom), we name the first three slices as column 1. Similarly, column 2 represents the amount of data for the second sliding window operation, so that slices 2~4 constitute the column 2. DNNBuilder caches at least two columns before starting computing, which allows the kernel to perform the second vertical sliding window operation immediately after finishing the first one. Delay caused by data shortage will not happen by caching one more column. Meanwhile, slice 5 will start buffering to form the next column (with slices 3~5) after releasing the room taken by slice 1. In this example, the required size of the reshape buffer (shown in Fig. 2) equals to the size of two columns (four slices). Since most of the input images have fewer pixels in height than width, we buffer slices in a column way to save on-chip memory. This scheme can be easily extended to row-based processing if needed.

The column-based cache scheme can also effectively improve the performance of layers with low computation-to-communication (CTC) ratio, such as the FC layers, and the CONV layers with small input feature map size. Unlike the previous work [10], which only uses FPGA to accelerate CONV layers (with higher CTC ratio, less memory intension than the FC layers), our solution can map the whole neural network on FPGA and adaptively adjust the size of reshape buffer to cache more or fewer columns for accelerating layers with different CTC ratio. Details are described in Sec. 6.

5.2.2 Adaptive hierarchical memory system. To tolerate the delay of off-chip data access, we design an adaptive hierarchical memory system which can insert buffers between the computation-intensive IP array and the external memory (shown in Fig. 2). The weight buffers are implemented by dual-port RAMs in FPGA for continuously buffering the weights from DRAM. Also, DNNBuilder provides optional ping-pong buffers at the input of each layer. Once the required amount of weights exceed a certain threshold, these data need to be stored off-chip so that ping-pong buffers are automatically generated to overcome the data shortage problem when fetching data from external memory. Conversely, when users develop a low bit-width quantized DNN, the ping-pong will not be generated as the size of required data is below the threshold.

6 AUTOMATIC RESOURCE ALLOCATION

One of the most critical problems in FPGA-based DNN implementation is the resource allocation under constraints while seeking for the optimal performance. To address this problem, we propose an automatic resource allocator for DNNBuilder, which can generate parallel schemes (e.g., CPF and KFP for each layer) and data buffering guidelines (e.g., size of the reshape buffer) with considerations of network complexity, external memory access bandwidth, and data reuse behaviors.

6.1 Theoretical Guideline

$$L_i = \alpha \frac{C_i}{R_i}, \quad \sum_{i=1}^{n} R_i = R_{total} \tag{1}$$

$$TP = \frac{1}{max\{L_i\}} \tag{2}$$

$$\frac{C_1}{R_1} = \frac{C_2}{R_2} = ... = \frac{C_i}{R_i} \tag{3}$$

The theory for maximum throughput performance of the pipeline architecture can be found in Equation 1 to 3. L_i represents the latency of layer i, while the computation demand (computation complexity) of layer i is C_i and the computation resource consumed by that layer is R_i. Assuming the available resource is R_{total}, the

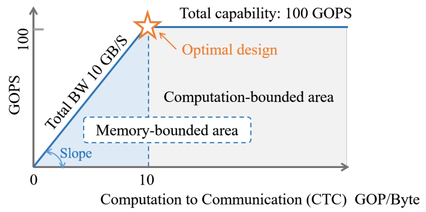
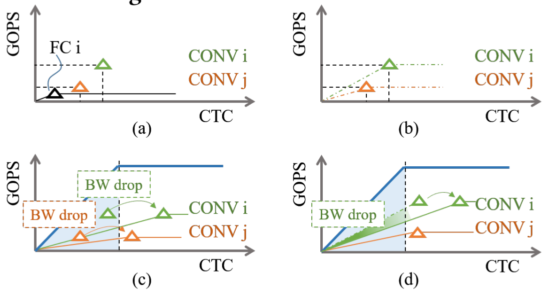


Figure 5: Basis of the roofline model



(a) (b) (c) (d)

Figure 6: Memory bandwidth adjustment

increase of allocated resource for each layer R_i results in a proportional increase in parallelism and eventually lowers the latency for that layer (α is a constant of proportionality related to the hardware working frequency). Since the generated accelerator uses a pipeline architecture, the overall throughput depends on the layer with the maximum computation time (Equation 2). We derive the upper-bound of the throughput from Equation 3, which lists the conditions for achieving the maximum throughput when workloads for all pipeline stages are perfectly balanced. It gives us a theoretical guideline to allocate computation resource for each layer.

6.2 Memory Bandwidth Adjustment

We use the roofline model [17] to intuitively illustrate the limitation of memory bandwidth. In Fig. 5, memory access bandwidth is represented by the slope. The area covered by blue is memory-bounded where performance is determined by the memory access latency under given CTC ratio. With lower CTC ratio, there is less data reuse opportunities, which means more fresh data is required to be fetched through memory interfaces to maintain GOPS. The right-side gray area is computation-bounded with larger CTC ratio, where performance is restricted by the available computation resource because memory access bandwidth is not the bottleneck. The optimal design locates at the top-left corner of the roofline which achieves the maximum throughput (GOPS) with the least bandwidth resource and CTC ratio required.

Our design intends to adjust the data reuse behavior of each layer so that we change its CTC ratio. For buffering more columns in the proposed column-based cache scheme (Sec. 5.2.1), we exploit more data reuse opportunities, which means the higher CTC ratio is achieved and it equivalently relaxes the dependency on memory access bandwidth (with a smaller required slope). Eventually, we place each DNN layer close to its optimal design spot and meet the constraints of available bandwidth and on-chip memory.

The procedure of this adjustment is shown in Fig. 6. During the first step, layer's computation demand is satisfied by a prorated allocation of the computation resource according to Equation 3, and CTC ratio is determined based on the data reuse behavior of each DNN layer. Here we mark three layers in the roofline model in Fig. 6 (a). Bandwidth resources are first allocated to FC layers for reaching their optimal design spot. We do not shrink their required bandwidth since these layers highly depend on memory bandwidth, and no data reuse can be exploited. If the bandwidth of targeted FPGA is insufficient for FC layers, we need to use more aggressive quantization (e.g., 8, 4 or even 2 bits) to represent the trained data. In this case, DNNBuilder provides feedbacks from *Generation* to *Design* step and updates the network definition, which can be found in Fig. 1 "updating" arrow. After going through the *Design* step for network retraining, we can avoid significant accuracy drop. We then allocate the remainder memory bandwidth to CONV layers as shown in Fig. 6 (b) (where the FC layer is omitted for conciseness). If CONV layers are memory-bounded as well (Fig. 6 (c)), it is necessary to cache more columns for getting higher CTC ratio so that these layers can be moved to the computation bounded area. In the other case (Fig. 6 (d)), if the CONV layers are not memory-bounded but the bandwidth is insufficient for them to reach their optimal design spots, we adjust the layer with the highest bandwidth demand (CONV i in our example) and shrink its bandwidth usage by caching more columns of the corresponding feature map even though it locates in the computation-bounded area already. As a result, the CONV i (green marker) is moved to the right and providing a bandwidth drop (smaller slope). In summary, both Fig. 6 (c) and (d) introduce bandwidth drop but they focus on two different scenarios.

6.3 Allocation Algorithm in DNNBuilder

To sum up the idea in Sec. 6.1 and 6.2, we present a resource allocation algorithm running in DNNBuilder. First, it starts allocating the computation resource shown in Algorithm 1. Since the parallelism factors must be the power of 2, DNNBuilder further fine-tunes the allocation scheme and fills up the gap between actual and the theoretical value (line 5~line 11). Resource allocated for layer i is represented as R_i, which is also the product of CPF_i and KPF_i. Eventually, DNNBuilder generates parallelism guidelines for building IP instantiations.

In step two, Algorithm 2 allocates the memory bandwidth resource given the constraints of total remainder bandwidth BW^{conv}_{total} for CONV layers (subtracted the bandwidth consumed by FC layers) and the total amount of remainder on-chip memory mem^{rb}_{total} for reshape buffers to keep feature maps (subtracted the memory occupied by weights buffers and reshape buffers in FC layers).

We initialize $Col_i = 1$ (caching one column of the input feature map) and the reshape buffer is implemented by a dual-port RAM with the width of read/write port $width^{rd}_i$, $width^{wr}_i$, and the depth of read port $depth^{rd}_i$ in i-th layer, which can be referred to Fig. 2. In line 5, Algorithm 2 first satisfies the bandwidth demands of allocated computation resources. PF_i represents the parallel factor, which is equal to R_i. If the required memory bandwidth exceeds the total available bandwidth, we need bandwidth adjustment (line 6 ~ line 16). It intends to address the problem shown in Fig. 6 (c) where the layer is in the memory-bounded area and (d) where particular layers excessively consume bandwidth resource. Since the microarchitecture of on-chip memory is significantly different between FPGAs, e.g., RAM18K/36K in Xilinx and M20K in Intel, we use a unified function f shown in line 9 to calculate the number of occupied memory blocks. In this algorithm, H^{in}_i, H^{out}_i and $Stride_i$ represent the height of input and output feature maps and the stride in layer i. C^{in}_i and C^{out}_i represent the number of channels

Algorithm 1 Computation resource allocation

```
Set available computation resource: $R_{total}$ //total DSPs
Computation resource for layer $i$ following Equations (1)-(3): $R_i = \frac{C_i}{C_{total}} \times R_{total}$ //# of DSPs for layer $i$
Initialize allocated resource for i-th layer (parallelism factor):
$R_i = 2^{\lfloor \log_2 R_i \rfloor}$
while $\sum_{i=1}^{n} R_i \le R_{total}$
    Select layer $j$ with maximum $\frac{C_j}{R_j}$
    if $\sum_{i=1}^{n} R_i + 2 \times R_j \le R_{total}$
        $R_j = 2 \times R_j$ //double the resource for layer $j$
    else break
    endif
endwhile
$R_i = CPF_i \times KPF_i$
    *CPF, and KPF are power of 2 for efficient hardware implementation
```

Algorithm 2 Memory bandwidth resource allocation

```
Set available memory bandwidth: $BW_{total}^{conv}$
Set available on-chip memory for input feature map: $mem_{total}^{rb}$
Set single DSP's bandwidth usage: $BW_R$
Initialize $Col_i = 1$; size of reshape buffer (e.g. $width_i^{rd}$, $depth_i^{rd}$, and $width_i^{wr}$ according to $KPF_i$ and $CPF_i$)
Allocate bandwidth $BW_i$ for layer $i$ to best satisfy its $R_i$ demand: $BW_i = \frac{PF_i \times BW_R}{H_i^{out} \times Col_i}$
while $\sum_{i=1}^{n} BW_i \ge BW_{total}^{conv}$ //CONV layer bandwidth overuse
    Select layer $i$ in CONV layer with maximum $BW_i$
    $depth_i^{rd} += \frac{H_i^{in} \times C_i^{in} \times Stride_i}{CPF_i}$, $depth_{i+1}^{rd} += \frac{H_i^{out} \times C_i^{out}}{CPF_{i+1}}$
    if $\sum_{i=1}^{n} f(width_i^{rd}, depth_i^{rd}, width_i^{wr}) \le mem_{total}^{rb}$
        $Col_i = Col_i + 1$ //Cache one more column
        $BW_i = BW_i \times \frac{Col_i - 1}{Col_i}$
    else //restore if not enough memory
    $depth_i^{rd} -= \frac{H_i^{in} \times C_i^{in} \times Stride_i}{CPF_i}$, $depth_{i+1}^{rd} -= \frac{H_i^{out} \times C_i^{out}}{CPF_{i+1}}$, break
    endif
endwhile
```

Table 1: Top-1 Accuracy for image classification

Network	Float32	Fix16	Fix16+f.-t. in *Design*	Fix8	Fix8+f.-t. in *Design*
Alexnet	55.7%	53.3%	55.1% (0.6% ↓)	51.6%	53.4% (2.3% ↓)
ZF	58.0%	56.3%	57.6% (0.4% ↓)	54.2%	56.2% (1.8% ↓)
VGG16	68.3%	67.0%	69.3% (1.0% ↑)	63.7%	69.2% (0.9% ↑)

Table 2: Accuracy for object detection (AP@IOU=0.5)

Network	Precision	Car	Pedestrian	Cyclist	mAP
YOLO (HD)	Float32	88.9%	64.9%	72.5%	75.5%
	Fix16+f.-t. in *Design*	88.9%	65.0%	73.2%	75.7% (0.2% ↑)
	Fix8+f.-t. in *Design*	88.9%	65.2%	72.6%	75.6% (0.1% ↑)

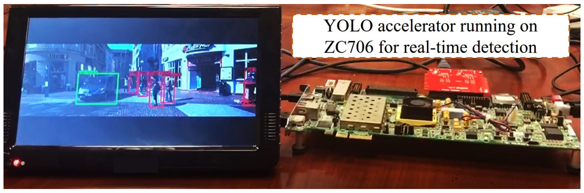


Figure 7: Accelerator for real-time car/pedestrian/cyclist detection generated by DNNBuilder

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 (CTC ratio) and the consumption of on-chip memory mem_i.

7 EXPERIMENTAL RESULTS

In this section, we demonstrate the capability and scalability of DNNBuilder by mapping four DNNs onto two FPGAs (XC7Z045 in Xilinx ZC706, KU115 in AlphaData 8K5) for running edge and cloud applications. We use a Yokogawa WT310 digital power meter to measure the power consumption.

7.1 DNN Models in Case Study

We build four DNNs using DNNBuilder, which include Alexnet [18], ZF [19], VGG16 [20] and YOLO [21]. The ZF and VGG16 are trained on ImageNet dataset [22] with input size 224×224, and Alexnet is trained on ImageNet but uses 227×227 inputs. For VGG16, to fit in our embedded FPGA (ZC706) and meet the real-time requirement, we have to cut off half of the kernels of CONV layers (except the CONV5) and half of the activations in FC layers. The weights of the FC layers are quantized to 4 bits to reserve memory bandwidth for CONV layers according to the feedbacks from *Generation* step. For the implementation in KU115 FPGA, we use the original VGG network structure without pruning. Accuracy results are shown in Table 1, where column "Float32" means using float32 model without quantization. "Fix16" and "Fix8" show the results of quantized models using 16-bit and 8-bit feature maps and weights without retraining. "f.-t. in *Design*" represents the accuracy results are collected after retraining and fine-tuning (such as adjusting the quantization following Sec. 6) in the *Design* step. The quantized models may also introduce regularizations, which causes the 1.0% accuracy increase in VGG16 even compared to the original Float32 version. Regarding the YOLO network, we modify it from the YOLO-tiny model, which is originally designed for 416×416 input resolution, and adapt it to the KITTI dataset [23] with 1280×384 HD input. Due to the hardware constraint, we change the kernel number of CONV7 and CONV8 from 1024 to 512 to target the real-time detection capability. We choose this model to demonstrate the scalability of DNNBuilder for handling HD inputs. We use 80% of the KITTI provided dataset for training set and the remainder for validation set to perform car, pedestrian, and cyclist detection and show the accuracy results in Table 2. We use IOU=0.5 as the threshold to identify true positive cases and mAP to represent the mean average precision of the car, pedestrian, and cyclist categories.

7.2 FPGA Mapping Results

These four DNNs are automatically deployed and optimized following the whole design flow of DNNBuilder. After synthesis by Vivado 2016.4, placement and routing are completed subsequently showing resource utilization and performance in Table 3 and 4. We achieve 200 MHz working frequency in the Zynq XC7Z045 (28nm) and 220-235MHz in KU115 (20nm) without any sophisticated timing adjustment. The *Batch per Die* in Table 4 shows the batch size of Fix16 version in one die of the KU115 FPGA (two dies in total) and we keep the same design in both dies. For the Fix8 version, the performance could be doubled by packing two activations together according to [24]. In summary, DNNBuilder can generate DNN hardware accelerators with performances peaking at 526 GOPS in an embedded FPGA and 4218 GOPS in a high-end FPGA.

Since DSP is one of the most critical resources of FPGA-based DNN accelerators, we need to carefully evaluate the utilization efficiency of the DSPs. Therefore, we introduce the *DSP efficiency* to exhibit the ratio between actual and theoretical maximum performance of the allocated DSPs. It is defined as:

$$DSP\_efficiency = \frac{Performance}{\beta \times DSP\_num \times freq.} \quad (4)$$

Table 3: Evaluation on Xilinx ZC706 (Zynq XC7Z045@200MHz, batch size=1 for Fix16, batch size=2 for Fix8)

Network	Utilization				Complexity	Throughput	GOPS	Throughput	GOPS	DSP
	LUT (218600)	FF (437200)	BRAM (545)	DSP (900)	(GOP)	Fix16 (img./s)	Fix16	Fix8 (img./s)	Fix8	Efficiency
Alexnet	86262(39%)	51378(12%)	303(56%)	808(90%)	1.45	170.0	247	340.0	494	76.3%
ZF	87465(40%)	50853(12%)	333(61%)	824(92%)	2.34	112.2	263	224.4	526	79.7%
VGG16 (pruned)	114521(52%)	61189(14%)	542(99%)	680(76%)	9.45	27.7	262	55.4	524	96.2%
YOLO (HD)	86103(39%)	48853(11%)	333(61%)	680(76%)	10.6	22.1	234	44.2	468	86.0%

Table 4: Evaluation on AlphaData 8K5 (Xilinx KU115 FPGA, batch size double for Fix8)

Network	Utilization				Batch	Complexity	Freq.	Throughput	GOPS	Throughput	GOPS	DSP
	LUT (663360)	FF (1326720)	BRAM (2160)	DSP (5520)	per Die	(GOP)	(MHz)	Fix16 (img./s)	Fix16	Fix8 (img./s)	Fix8	Efficiency
Alexnet	262360(40%)	177146(13%)	986(46%)	4854(88%)	3	1.45	220	1126	1633	2252	3265	76.4%
ZF	268242(40%)	186198(14%)	1162(54%)	4950(90%)	3	2.34	225	759	1776	1518	3552	79.7%
VGG16	257862(39%)	171616(13%)	1578(81%)	4318(78%)	1	30.94	235	65	2011	130	4022	99.1%
YOLO (HD)	262356(40%)	165601(13%)	1256(63%)	5286(96%)	4	10.6	220	199	2109	398	4218	90.7%

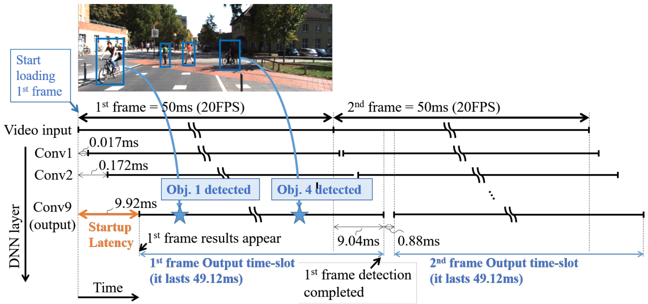


Figure 8: Startup latency analysis in pedestrian detection

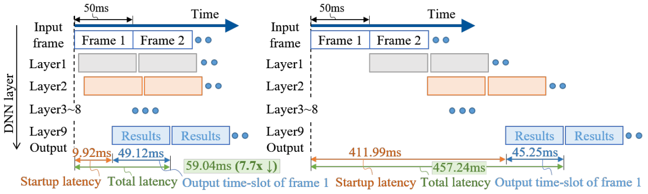


Figure 9: Latency comparison between fine-grained (left) and conventional (right) pipeline architecture while running YOLO with HD inputs

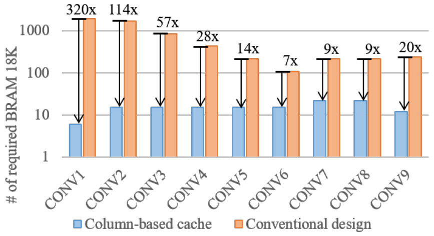


Figure 10: On-chip memory demand comparison between column-based cache and conventional design while holding feature maps in YOLO with HD inputs

Since β multiply-accumulate operations (β=2 in Fix16, β=4 in Fix8) can be handled by one DSP and corresponding logics in one cycle, the denominator equals to the theoretical best performance provided by allocated DSPs under a given frequency. The numerator means the actual achieved performance (GOPS) as shown in Table 3 and 4. Following Equation 4, VGG16 accelerator achieves the highest DSP efficiency, which is followed by designs for YOLO and ZF, while Alexnet accelerator is in the last place. The reason is that VGG16 has unified CONV (3×3 with stride 1) and pooling pattern (2×2 with stride 2), which makes Equation (3) perfectly satisfied under the constraints that R_i is a power of 2. The more balanced for the latency of each layer, the higher DSP efficiency is achieved.

7.3 Latency and On-chip Memory Analysis

We take the DNNBuilder generated YOLO accelerator (Fix16) as a case study. In Fig. 7, a real-time detection for HD video input is running on the ZC706. The input video frame with HD resolution (1280×384) is captured by a wide-angle camera and sent to the FPGA at 20 FPS (meaning the frame transmission delay is 50ms). In Table 3, this YOLO accelerator achieves a throughput at 22.1 FPS, which is enough for processing the 20 FPS video. Since the proposed fine-grained pipeline architecture and column-based cache scheme are applied, accelerator is launched once the first few columns of input frame are ready (in the reshape buffer). In Fig. 8, the startup latency is 9.92ms. After 9.92ms, CONV9 keeps generating outputs until 9.04ms after the first frame is fully loaded, so we call this period "output time-slot" which lasts 50+9.04-9.92=49.12ms. This accelerator utilizes 137 BRAMs for keeping columns of feature maps and 333 BRAMs (Table 3) for the whole accelerator. Since four pedestrians are shown in this frame, they are detected one after another from left to right during the"output time-slot".

The advantage of using the proposed fine-grained layer-based pipeline architecture is that we can hide the data transmission delay (generating outputs when the 1st frame is still loading), and deliver a small startup latency (which is 9.92ms in this case). Detailed comparisons are shown in Fig. 9. The length of each rectangle represents the latency of one major network layer. The conventional pipeline architecture (right), with the same 20 FPS overall throughput, waits for the finish of all frame and feature map transmissions at preceding stages so that it suffers a very long latency as 457.24ms for generating all results of the first frame. Conversely, our proposed design starts running layer $i + 1$ pipeline stage after we collect several columns of output feature maps of layer i. So, we deliver the latency as 59.04ms, which achieves an amazing 7.7x reduction. The advantage of using column-based cache can be shown in Fig. 10. We significantly reduce the required BRAM for keeping DNN feature maps. In total, we instantiate only 137 BRAMs in our YOLO accelerator instead of 5920 BRAMs in the conventional pipeline design where feature maps need to be stored completely using on-chip memory. The overall BRAM reduction is 5920 ÷ 137 = 43x while the best case happens in the first layer with 320x reduction achieved. The proposed column-based cache ensures the scalability of our accelerator design while using HD or even 4K inputs. It is the fine-grained layer-based pipeline structure and column-based cache that help us reduce startup latency and BRAM utilization which are common shortcomings in pipeline structures, and hide the video frame transmission time. As a result, DNNBuilder can deliver millisecond-scale response during object detection and accommodate inputs with high resolutions.

7.4 Comparison to FPGA & GPU Accelerators

We compare our design to five latest FPGA-based accelerators with classification-oriented DNNs in Table 5. For Intel FPGA, the actual DSP utilization in [5] and [10] should be twice as shown in the original paper because one variable-precision DSP block

Table 5: Comparison with existing FPGA-based DNN accelerators

Reference	[5]	[6]	[10]	DNNBuilder	[1]	[2]	DNNBuilder
Categories	Cloud-computing platforms				Edge-computing platforms		
FPGA chip	Arria10-1150	Arria10-1150	Stratix-V GXA7 + CPU	KU115	Zynq XC7Z045	Zynq XC7Z045	Zynq XC7Z045
Frequency	303 MHz	385 MHz	200 MHz & 2~3 GHz(CPU)	235 MHz	150 MHz	100 MHz	200MHz
Network	Alexnet	VGG	Alexnet	VGG	VGG	VGG	VGG
Precision	Float16	Fix16	Fix16 in FPGA	Fix16 (Fix8)	Fix16	Fix16	Fix16 (Fix8)
DSPs (used/total)	2952/3036	2756/3036	512/512 in FPGA	4318/5520	780/900	824/900	680/900
DSP Efficiency	77.3%	84.3%	-	99.1%	44.0%	69.6%	96.2%
Performance (GOPS)	1382	1790	781	2011 (4022)	137	230	262 (524)
Power Efficiency (GOPS/W)	30.7	47.8	-	90.2 (180.4)	14.2	24.4	36.4 (72.8)

Table 6: Alexnet inference comparison: GPU vs FPGA

Platform	Precision	Batch	Throughput (img./S)	Power (W)	Efficiency (img./S/W)
DNNBuilder (ZC706)	Fix16, Fix8	1, 2	170, 340	7.2	23.6, 47.2
GPU-TX2[26]	Float16	2	250	10.7	23.3
DNNBuilder (KU115)	Fix16, Fix8	3, 6	1126, 2252	22.9	49.2, 98.3
GPU-TitanX	Float32	128	5120	227.0	22.6

can simultaneously work for two 16-bit multipliers [25]. For the cloud-computing case, the DNNBuilder generated design achieves 4022 GOPS using KU115 FPGA. Our design with Fix8 quantization outperforms those in [5], [6], and [10] by 2.91x, 2.25x, and 5.15x, while our Fix16 version outperforms them by 1.46x, 1.12x, and 2.57x respectively. Although our design uses more DSPs, we deliver the highest DSP efficiency (99.1%) which allows us to slow down the clock frequency for achieving 5.88x higher power efficiency compared to [5]. The design in [10] deploys a DNN accelerator on a Xeon CPU+FPGA system, and reduces the number of computation by using frequency domain CONV. Layers with low CTC ratio (e.g., FC layers, which are limited by memory access bandwidth on FPGAs) are swapped out to CPU using QPI. Since we can not quantify the equivalent DSP utilization in CPU and the authors fail to mention any power consumptions (which should be the sum of CPU+FPGA), we leave the DSP and power efficiency blank for [10]. Major drawbacks of [10] are the large batch size requirement and the resulting high demand for FPGA on-chip memory. It requires large batch size to recover the input padding overhead and the low data reuse behavior while running CONV in the frequency domain. Design in [10] may not be feasible for using embedded FPGAs. On the contrary, DNNBuilder can deliver high-performance DNN accelerators on both high-end FPGAs and energy-efficient embedded FPGAs for cloud and edge applications. By evaluating the edge-computing ability, we use the same embedded FPGA in [1] and [2]. Our DNNBuilder generated design reaches the best performance (524 and 262 GOPS in Fix8 and Fix16) and power efficiency (72.8 GOPS/W in Fix8 and 36.4 GOPS/W in Fix16).

We extend our comparison to the latest embedded GPU (TX2) and the high-end GPU (TitanX) in Table 6. Because of the real-time requirement of edge-applications, we attempt to use the smallest batch size. However, the result of TX2 is using a batch size of 2, which is the smallest batch size implementation we could find from Nvidia's official source. Our design in ZC706 delivers higher efficiency than the TX2-based solution even without using batch processing. Our design (Fix8) in KU115 delivers 4.35x higher efficiency than the TitanX-based solution (Float32) with a much smaller batch size.

8 CONCLUSION

In this paper, we presented DNNBuilder, an automation tool for building DNN hardware accelerators on FPGAs, for delivering high performance and power efficiency. We proposed a fine-grained layer-based pipeline architecture and a column-based cache scheme for higher throughput, lower pipeline latency, and smaller on-chip memory consumption. We introduced the flexible process engine that not only provides optimal implementations of diversified DNN layers but also allows us to adjust the parallelism factors (CPFs and KPFs) to fit in the resource allocation guidelines. We designed an automatic resource allocation algorithm to enable design space exploration and generate parallelism schemes under constraints of computation resource, on-chip memory capacity, and external memory access bandwidth. Because of the above novel designs, we reached the highest throughput performance peaking at 4218 GOPS (KU115) and 526 GOPS (ZC706) compared to the existing FPGA/embedded FPGA based solutions. We also achieved higher efficiency (up to 4.35x) than the GPU based solutions.

ACKNOWLEDGMENT

This work was partly supported by the IBM-Illinois Center for Cognitive Computing System Research (C^3SR) – a research collaboration as part of IBM AI Horizons Network.

REFERENCES

[1] J. Qiu et al. Going deeper with embedded FPGA platform for convolutional neural network. In *FPGA*, 2016.
[2] Q. Xiao et al. Exploring heterogeneous algorithms for accelerating deep convolutional neural networks on FPGAs. In *DAC*, 2017.
[3] J. Wang et al. A design flow of accelerating hybrid extremely low bit-width neural network in embedded FPGA. In *FPL*, 2018.
[4] X. Zhang et al. High-performance video content recognition with long-term recurrent convolutional network for FPGA. In *FPL*, 2017.
[5] U. Aydonat et al. An OpenCL deep learning accelerator on Arria 10. In *FPGA*, 2017.
[6] J. Zhang et al. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *FPGA*, 2017.
[7] C. Zhuge et al. Face recognition with hybrid efficient convolution algorithms on FPGAs. In *GLSVLSI*, 2018.
[8] X. Zhang et al. Machine learning on FPGAs to face the IoT revolution. In *ICCAD*, 2017.
[9] X. Wei et al. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *DAC*, 2017.
[10] H. Zeng et al. A framework for generating high throughput CNN implementations on FPGAs. In *FPGA*, 2018.
[11] H. Sharma et al. From high-level deep neural models to FPGAs. In *Micro*, 2016.
[12] Y. Ma et al. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *FPL*, 2017.
[13] Y. Guan et al. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *FCCM*, 2017.
[14] H. Li et al. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *FPL*, 2016.
[15] N. Jouppi et al. In-datacenter performance analysis of a tensor processing unit. *arXiv:1704.04760*, 2017.
[16] N. Suda et al. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *FPGA*, 2016.
[17] C. Zhang et al. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *FPGA*, 2015.
[18] A. Krizhevsky et al. Imagenet classification with deep convolutional neural networks. In *NIPS*, 2012.
[19] M. Zeiler et al. Visualizing and understanding convolutional networks. In *ECCV*, 2014.
[20] K. Simonyan et al. Very deep convolutional networks for large-scale image recognition. *arXiv:1409.1556*, 2014.
[21] J. Redmon et al. Yolo9000: Better, faster, stronger. *arXiv:1612.08242*, 2016.
[22] J. Deng et al. Imagenet: A large-scale hierarchical image database. In *CVPR*, 2009.
[23] Andreas G. et al. Are we ready for autonomous driving? the KITTI vision benchmark suite. In *CVPR*, 2012.
[24] Y. Fu et al. Deep learning with int8 optimization on Xilinx devices. *White Paper*, 2017.
[25] Intel. Enabling high-performance DSP applications with Stratix V variable-precision DSP blocks.
[26] Nvidia. Nvidia Jetson TX2 delivers twice the intelligence to the edge.