Toward Full-Stack Acceleration of Deep Convolutional Neural Networks on FPGAs

Shuanglong Liu, Hongxiang Fan, Martin Ferianc, Xinyu Niu, Huifeng Shi, and Wayne Luk, *Fellow, IEEE*

***Abstract*— Due to the huge success and rapid development of convolutional neural networks (CNNs), there is a growing demand for hardware accelerators that accommodate a variety of CNNs to improve their inference latency and energy efficiency, in order to enable their deployment in real-time applications. Among popular platforms, field-programmable gate arrays (FPGAs) have been widely adopted for CNN acceleration because of their capability to provide superior energy efficiency and low-latency processing, while supporting high reconfigurability, making them favorable for accelerating rapidly evolving CNN algorithms. This article introduces a highly customized streaming hardware architecture that focuses on improving the compute efficiency for streaming applications by providing full-stack acceleration of CNNs on FPGAs. The proposed accelerator maps most computational functions, that is, convolutional and deconvolutional layers into a singular unified module, and implements the residual and concatenative connections between the functions with high efficiency, to support the inference of mainstream CNNs with different topologies. This architecture is further optimized through exploiting different levels of parallelism, layer fusion, and fully leveraging digital signal processing blocks (DSPs). The proposed accelerator has been implemented on Intel's Arria 10 GX1150 hardware and evaluated with a wide range of benchmark models. The results demonstrate a high performance of over 1.3 TOP/s of throughput, up to 97% of compute [multiply-accumulate (MAC)] efficiency, which outperforms the state-of-the-art FPGA accelerators.**

***Index Terms*— Convolutional neural networks (CNNs), deep learning, field-programmable gate arrays (FPGAs), hardware accelerator, layer fusion, unified architecture.**

I. Introduction

RECENTLY, large and deep convolutional neural networks (CNNs) have become widely adopted in many tasks such as image classification [1], [2], object detection [3], and semantic segmentation [4]. Particularly, they have been deployed in a variety of real-life and real-time applications, such as smart cities, cameras, and remote sensing [5]. In these applications, CNNs have shown great accuracy improvement in comparison to traditional machine learning (ML) algorithms. However, most successful CNN models exhibit very high computational complexity and require vast memory and processing power.

The operations that compose a CNN are not well-suited to the Von Neumann computer architecture at the heart of CPUs. They are better suited to hardware architectures with distributed, massively parallel computation and local memory such as graphics processing units (GPUs) or field-programmable gate arrays (FPGAs). In particular, GPUs with highly parallel architectures can achieve high throughput on CNNs by processing parallel samples in batches. The efficiency of GPUs relies largely on the regularity of data and batch size, which works well for training, but not in practice while targeting real-time inference [6]. For example, images in streaming applications arrive one by one and using batch processing can greatly increase latency, which is critical to the system's performance.

Designing dedicated hardware for accelerating CNNs requires significant investment and time to develop. However, the ML community keeps to rapidly evolve CNNs. For example, VGG16 [2] was first introduced in 2014 for object detection, one of the most popular tasks in computer vision, which employed a uniform convolutional kernel size with serial layer connectivity. A year later, CNNs have been in the trend of employing residual (ResNets [7]) and concatenative connections (GoogLeNet [8]), which introduce irregular connectivity across layers. Moreover, networks such as YOLOv3 [9] employ both types of irregular connections, making potential accelerators, e.g., for VGG16 already obsolete. These irregular connections are shown and explained in Fig. 1.

Apart from object detection, semantic segmentation has been widely studied across a variety of application domains, in order to provide pixel-wise segmented information from the image. Deconvolution layer (Deconv), also called as upsample in the literature, is hence introduced in models such as SegNet [4] or U-Net [10], in addition to classic 2-D convolution (Conv), which also employ concatenative connections. In these models, Deconv together with Conv layers constitutes the majority of computation [11]. As a result, they are far more computationally intensive than models designed for image classification or object detection. Therefore, a general customizable hardware architecture, without the need to develop dedicated circuits, with the capability to support all kinds of models mentioned above is crucial for

Manuscript received May 5, 2020; revised August 23, 2020 and November 29, 2020; accepted January 25, 2021. This work was supported in part by the National Natural Science Foundation of China under Grant 62001165; in part by the U.K. EPSRC under Grant EP/L016796/1, Grant EP/N031768/1, Grant EP/P010040/1, Grant EP/L00058X/1, and Grant EP/S030069/1; in part by the Corerain; in part by the Maxeler; in part by the Intel; in part by the Xilinx; and in part by the SGIIT. *(Corresponding author: Shuanglong Liu.)*

Shuanglong Liu is with the School of Physics and Electronics, Hunan Normal University, Changsha 410081, China (e-mail: liu.shuanglong@hunnu.edu.cn).

Hongxiang Fan and Wayne Luk are with the Department of Computing, Imperial College London, London SW7 2AZ, U.K.

Martin Ferianc is with the Department of Electronic and Electrical Engineering, University College London, London WC1E 7JE, U.K.

Xinyu Niu is with Corerain Technologies Ltd., Shenzhen 518048, China.

Huifeng Shi is with the State Key Laboratory of Space-Ground Integrated Information Technology (SGIIT), Beijing 100029, China.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNNLS.2021.3055240.

Digital Object Identifier 10.1109/TNNLS.2021.3055240

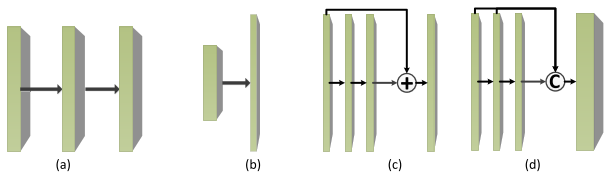


Fig. 1. Main functions and connectivity in CNNs. (a) Convolutions with serial connection. (b) Deconvolution, also known as *upsampling*, which extrapolates new information from the input feature map and is widely used in segmentation models. (c) Residual connection, also called as a *shortcut* connection, where the results of one layer skip one or more layers and then are added to the subsequent layers at different depth levels. (d) Concatenative connection, which is a utility layer that concatenates its multiple inputs to a single output. Unlike previous FPGA-based accelerators targeting CNN types shown in (a), our work aims to improve the efficiency of CNNs of all these types.

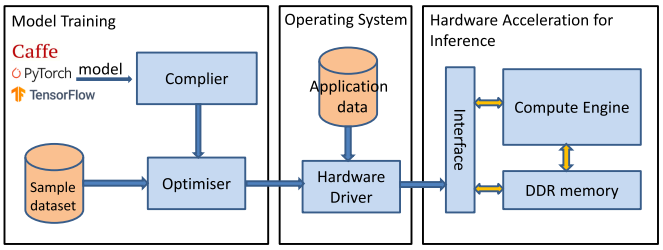


Fig. 2. Overview of the proposed optimization framework to accelerate the mainstream CNN models for low-latency inference.

rapid system development. This requires special focus and efforts on the compute efficiency[1] of both Conv and Deconv layers, as well as the irregular connections.

To address the challenges of CNNs with irregular shapes and/or Deconv layers, and adaptation to varying and evolving CNN architectures, we propose a full-stack optimization framework and develop a hardware accelerator based around FPGAs. Fig. 2 presents the workflow to incorporate the hardware engine for high-performance inference. In the training stage, the compiler tool accepts a newly designed and trained CNN model from ML frameworks, such as Tensorflow [13], PyTorch [14] or Caffe [15]. The compiler then converts the model to a streaming-graph intermediate representation and extracts model structure and coefficients. During conversion, an optimizer is used to optimize the model for efficient runtime acceleration. At the application level, the hardware driver consisting of the execution instructions is generated for the already trained CNN model. Then, the operating system makes calls to the compute engine on the FPGA to run inference. Using the provided framework, a developer can quickly transfer a CNN model to run directly on our accelerator without additional hardware development.

In the heart of our approach is a compute engine, i.e., the hardware accelerator that aims to improve the compute efficiency and reduce the inference latency for the mainstream CNNs with different topologies. It builds on top of a unified hardware architecture from our prior work [5], which maps both Conv and Deconv layers into a single hardware module. Besides, the streaming accelerator maps the irregular connections (residual and concatenative connections) with high efficiency by organizing the hardware blocks in a way where all blocks are kept busy at all times, also by using a custom-tailored design of smart cache system. In addition, the accelerator is further optimized by exploiting different levels of parallelism and fully leveraging the digital signal processing blocks (DSPs) on an FPGA. Finally, the CNN is quantized through 8-bit fixed-point (INT8) quantization scheme [16] to achieve higher performance without loss of accuracy.

The novel contributions of this work are as follows:

1) automated acceleration framework, which enables users to deploy the trained network models on FPGAs with balanced resource allocation;
2) streaming accelerator with efficient mapping of residual and concatenative connections and highly optimized with methods such as input reshaping and layer fusion;
3) latency estimation method using the Gaussian process with improved estimation accuracy without the need to run the CNN on a real FPGA, which in turn reduces the design time for better tradeoff between accuracy and hardware performance.

Leveraging all these advances into a single system, we have built an efficient CNN inference engine on an FPGA with high compute efficiency. Achieving a high compute efficiency across a wide range of CNN models is a challenge for many hardware accelerators. The high compute and energy efficiency of the proposed design is mainly due to two factors:

1) mapping the main computation operations into a singular unified architecture (Section III-A);
2) reducing the communication time by efficient execution of irregular connections (Section III-D).

As such, we can occupy the DSPs during most of the execution time (>90%). Besides, we can exploit over 97% of the DSPs in the FPGA device, due to the proposed DSP optimization technique for INT8 multiplier (Section IV-C).

II. Background and Related Work

In this section, we first review recent advances for efficient CNNs in both algorithm and hardware implementations. Then, we summarize the limitations of previous FPGA-based accelerators for CNNs in comparison to our design. Quantitative evaluation and comparison will be presented in Section VI-F.

A. CNN Layer Overview

CNNs are built of several computational operations stacked on top of each other, commonly known as layers, and most modern networks have residual or concatenative connections between them. Frequently used layers are 2-D convolutional (Conv), upsampling (Deconv), or fully connected (FC) layers. These three-layer types take up over 90% of computation in a CNN model. Besides, there are pooling and batch normalization [17] layers or nonlinear activations, such as rectified linear unit (ReLU).

As illustrated in Code 1, the Conv or Deconv receives $C \times H_i \times W_i$ sized input feature maps, and then, these inputs

[1]The compute efficiency or multiply-accumulate (MAC) efficiency is defined as the fraction of useful MAC cycles consumed by the total MAC units in the design [12], while the overall efficiency is defined as the ratio of realized performance to the theoretical peak performance of the device. Details are given in Section VI-F.

Code 1 Convolution and Deconvolution Algorithms

Input: Input feature map $\mathbf{I}$ of shape $C \times H_i \times W_i$;
Weight matrix $\mathbf{W}$ of shape $F \times C \times K \times K$;
Output: Output feature map $\mathbf{O}$ of shape $F \times H \times W$;

1: for $(f = 0; f < F; f++)$ // **filter loop**
2: for $(c = 0; c < C; c++)$ // **channel loop**
3: for $(h = 0; h < H; h++)$ // **row loop**
4: for $(w = 0; w < W; w++)$ // **column loop**
5: // Conv:

$$\mathbf{O}[f][h][w] += \sum_{i=1}^{K-1}\sum_{j=1}^{K-1} \mathbf{W}[f][c][i][j] * \mathbf{I}[c][h*S+i] \times [w*S+j]$$

6: // Deconv:
$\mathbf{O}[f] += deconv(\mathbf{I}[c], \mathbf{W}[f][c])$ // as shown in Figure 3.

are convolved or deconvolved with a kernel with the shape of $F \times C \times K \times K$. Each kernel window with the size of $K \times K$ is applied to one channel of the input ($H_i \times W_i$) by sliding the kernel with a stride of S to produce one output feature map ($H \times W$); then, the results of C channels are accumulated to produce one channel of output (channel loop in line 2). All filters of the output feature maps ($F \times H \times W$) are generated by repeating this process F times (filter loop in line 1). Line 5 of Code 1 describes the 2-D convolution. Deconv layers are implemented as transposed convolutions in CPUs or GPUs [18]. Before performing the transposed convolution, zeros need to be inserted into the original input feature maps. FC layers can be converted into a Conv layer by considering the kernel size K. For example, an FC layer with the input size of $C \times H \times W$ and the output size of $F \times 1$ can be implemented as a Conv layer with the kernel size of $F \times C \times H \times W$.

B. Efficient CNNs

It has been a general trend of increasing the depth of CNNs using residual and concatenative connections between their layers, to improve their classification accuracy as well as the speed of training [19]. As a result, the networks are gradually becoming structurally denser and thus more complex, which largely limits their application in resource-constrained settings, such as in edge devices for Internet-of-Things (IoT) applications. Therefore, many research teams have proposed methods to reduce the computation complexity of CNNs both at algorithm and hardware implementation levels.

Novel algorithms, including Winograd convolution [20] or fast Fourier transform (FFT) [21], focus on compute reduction techniques. FFT performs the convolution operation in frequency domain, and thus, it turns the originally space-domain operation into a Hadamard product between the input and the convolution kernel. Winograd convolution computes minimal complexity convolution over small tiles, which reduces the number of multiplications by a factor of approximately 2.25× using the filter $F(3\times3, 2\times2)$ [22]. Other algorithmic advances cover model compression, which shrinks model representation by channel pruning or resolution multipliers, used for example in MobileNet [23]. Another technique to compress models is to replace standard convolutions with depthwise separable Conv [24], which reduces the computation by a factor of K^2.

From the hardware level perspective, researchers [25]–[27] have proposed: 1) a quantization method, which captures the specialty of FPGAs with capability of custom precision support to save computation resources and 2) loop unrolling strategies for multiple parallel processing. A further step into improving quantization involves binarization for both weights and data while executing the CNN on FPGA since, in this case, the multiplication can be simply implemented as a XNOR gate [28]–[30], which largely relieves the use of limited DSP resources in current FPGAs. Others have proposed residual binary inputs and weights to improve binarized networks, which can improve the accuracy while still maintaining almost the same computing resources in hardware [31].

C. Related Work

Recently, various FPGA-based accelerators for CNN inference have been proposed with the key objectives of designing a system with high energy efficiency and low latency. These accelerators, however, are generally targeting relatively structurally simple networks, such as AlexNet [1] or VGG16 [2]. The common strategy used among these accelerators is to minimize the data and weight movement from the off-chip memories to the compute engine, which is implemented with FPGA's fabric. The techniques include: 1) double buffer, to overlap the computation time and the data/weight load time [25], [32], and 2) layer fusion, to process multiple CNN layers in a pipelined manner, allowing for instant use of intermediate data without external memory access [19], [27]. Wu *et al.* [12] optimized the accelerator by maximizing the operating clock frequency and compute efficiency but achieved a relatively low resource utilization. Most of these accelerators only focus on Conv layers, thus providing high efficiency only for CNNs with regular shapes.

A few works have studied the acceleration of Deconv layers and generative adversarial networks (GANs) [33]–[35]. However, these works focus specifically on accelerating Deconv in GANs that consist solely of deconvolutional layers. Therefore, they did not attempt to accelerate other models such as those used in segmentation models that employ both Conv and Deconv layers. Our prior work [11] optimized the operations of both Deconv and Conv layers for semantic segmentation. An approach was proposed to address the compute inefficiency incurred by the sparsity of Deconv when implemented as transposed Conv. However, two different hardware modules are deployed for Deconv and Conv separately in this design and their DSPs for multipliers are not shared, which caused the inefficiency of resource utilization.

Moreover, previous FPGA-based accelerators did not efficiently support models with irregular connections. Venieris and Bouganis [36] designed three separate hardware blocks for each irregular network connection for networks that they evaluated, i.e., GoogLeNet, ResNet-152, and DenseNet-161. This approach can be a solution for a reconfigurable FPGA

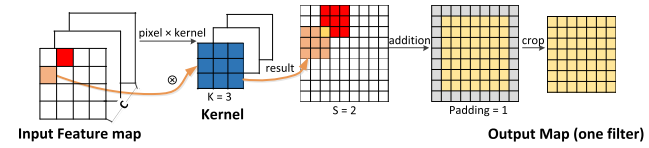


Fig. 3. Visualization of our approach to implement the Deconv layer with $K = 3$, $S = 3$, and Padding $= 1$.

design, but it still leads to low resource efficiency in execution and reduction in the design's productivity. McDanel *et al.* [19] introduced a network without any concatenative or residual connections with small accuracy loss. Although competitive performance on ImageNet [37] can be achieved in the mobile setting, it did not solve the problem from the hardware perspective, and in other applications, users are gradually more inclined toward using residual or concatenative connections.

Compared to the previous work, we first implement the main computation layers among CNN models, i.e., convolutional, deconvolutional, and FC layer, and map them into a single unified module, which improves the MAC efficiency during inference. Second, the proposed accelerator supports both residual and concatenative connections for a general set of networks with only one element-wise residual hardware block, and a high compute efficiency for these irregular structures is achieved through designing a smart memory system (see Section III-D). As a result, our approach provides high compute efficiency for both regular and irregular network structures without the need to reconfigure the FPGA fabric.

III. Streaming Accelerator Architecture

This section first proposes a unified architecture to support the main operations: Conv, Deconv, and FC layers in CNNs, which serves as the key hardware module in our compute engine. We explore different levels of parallelism for the unified architecture as well as the overall accelerator. Then, it presents the general structure of the accelerator with a smart cache design that allows the implementation of residual and concatenative connections while maintaining high efficiency for a general set of CNN models.

A. Unified Architecture

The direct mapping of CPU- or GPU-based Deconv algorithm, i.e., transposed convolution onto the FPGA, will incur the compute inefficiency due to the zero insertions leading to meaningless multiplications with zeros. In this work, an efficient 2-D Deconv approach proposed in [11] is used in our hardware implementation, as shown in Fig. 3. This approach multiplies input pixels with the corresponding weight kernel and sums the overlapping area in output maps. It improves the compute efficiency by exploiting the sparseness of transposed convolutions.

Existing FPGA-based accelerators, such as [11], [38], and [39], implement 2-D Conv by unrolling the dot-product loop in line 5 of Code 1 using $K \times K$ multipliers. However, this type of architecture cannot be reused for the Deconv approach mentioned above. Besides, it is difficult to reconfigure the compute kernel back for Conv with multiple kernel sizes (such as 3×3, 5×5, or 7×7).

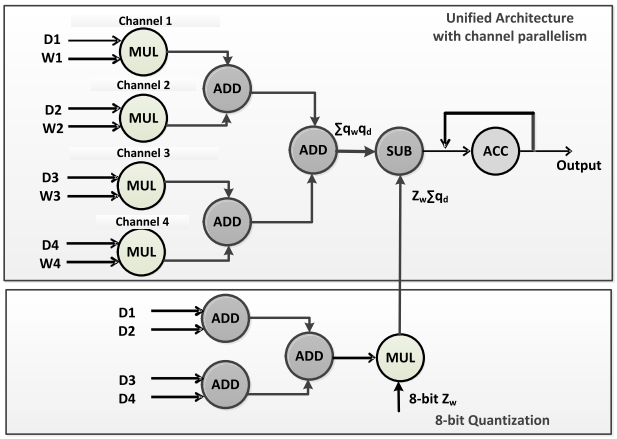


Fig. 4. Unified module proposed to map Conv, Deconv, and FC on FPGA with parallel channel processing and the 8-b quantization module to support integer-only arithmetic inference [16].

To improve the resource efficiency, we propose a unified accelerator architecture to implement both Conv and Deconv with an arbitrary kernel size. FC layers are always performed as a Conv layer without the need to introduce additional blocks and thus achieve the highest resource occupancy during runtime. In this architecture, each multiplier is responsible for computing a single output pixel such that the 2-D Conv or Deconv is performed in a single MAC unit for one output. As shown in Fig. 4, it consists of an array of multipliers that compute multiple channels of input in parallel and a quantization module that computes the sum of the input pixels, to support the 8-b linear quantization scheme proposed in [16]. The quantization scheme for CNNs achieves a very high compute density without observing loss of accuracy, as we will show in Section VI-E.

1) 8-b Quantization: The quantization scheme is a mapping of integers q to real numbers r with the form

$$r = S(q - Z) \tag{1}$$

where S and Z are constant parameters. Assume that the real and quantization numbers of inputs, outputs, and weights are (r_d, r_o, r_w), and (q_d, q_o, q_w), respectively. Conv without quantization is computed as $r_o = \sum r_w r_d$. Substituting each term with (1), we have

$$S_o(q_o - Z_o) = \sum S_w(q_w - Z_w)S_d(q_d - Z_d) \tag{2}$$

which can be rewritten as

$$q_o = Z_o + \frac{S_w S_d}{S_o}\Big(NZ_w Z_d - Z_d \sum q_w - Z_w \sum q_d + \sum q_w q_d\Big). \tag{3}$$

Note that $NZ_w Z_d - Z_d \sum q_w$ is independent of the input features, which means that it can be computed offline, and only $Z_w \sum q_d$ needs to be computed at runtime. Therefore, $\sum q_w q_d - Z_w \sum q_d$ is computed in the unified architecture, and other operations are merged with the weights and bias of each convolution layer which is done offline.

2) Conv: To compute one output pixel, the corresponding $K \times K$ input pixels of the feature maps and $K \times K$ weights are sequentially multiplied in a single multiplier. Then, the multiplied results are flowed into an accumulator (ACC shown in Fig. 4) for accumulation to compute one output pixel of the output maps of one channel. Therefore, one output of convolution requires $K \times K + L$ hardware cycles in total, where L is the total number of cycles to perform addition and accumulation.

3) Deconv: The Deconv approach shown in Fig. 3 is more complex than Conv in terms of hardware implementation. The number of MAC operations required depends on the position of the computed output pixel since there are different overlapping rows and columns presented in the output map in Fig. 3. In total, three cases need to be considered: 1) for the output with nonoverlapping rows or columns, only one input pixel is multiplied by the weights; 2) for the output with only one overlapping row or column, two adjacent input pixels in row or column dimensions are sequentially multiplied by the corresponding weights; and 3) for output with both overlapping row and column, four adjacent input pixels in row and column dimensions are sequentially multiplied by the weights. For the last two cases, the multiplied results are flowed into the accumulator for accumulation. Hence, 1, 2, or 4 clock cycles are required, respectively, to compute one Deconv output of one channel input maps in the three cases.

Therefore, the architecture can implement convolutions with any kernel size and strides as well as deconvolutions. It is also capable of supporting other convolution-based operations in CNNs such as 1-D Conv or dilated Conv, by feeding the data and weights into the multipliers in the right sequence.

B. Parallelism Exploration

We explore different levels of parallelism in order to improve the resource utilization and compute efficiency of our accelerator. Three levels of parallelism can be utilized for parallel processing in convolution-based operations: filter parallelism, channel parallelism, and data parallelism. They correspond to unrolling the loops in lines 1–3 of Code 1, respectively. Data parallelism is utilized in previous designs, such as [11]. However, the employment of data parallelism will result in computational inefficiency in practical hardware design due to the following factors.

1) *Workload Imbalance:* When performing Deconv. When employing data parallelism, it computes multiple output pixels in one row of the output feature maps in parallel. However, Deconv has three separate modes with respect to which the output in one row can be produced, just as we have mentioned above. As a result, the workload of the multipliers is imbalanced and some multipliers must be kept idle to wait for others to finish processing, resulting in low multiplier utilization.
2) *Inefficiency:* When the input width of a layer cannot be divided by the degree of data parallelism. The degree of parallelism in hardware must be a fixed number, e.g., 32. However, the layers in CNNs often have the input maps with different heights and widths, and it is impossible to have a degree of parallelism in which all the widths of layers in the network are fully supported. For example, for $W = 36$, the compute efficiency is only $((36/32)/(\lceil 36/32 \rceil)) = 56\%$. This inefficiency can be relieved by batch processing, but as previously mentioned, it increases the latency for streaming applications.

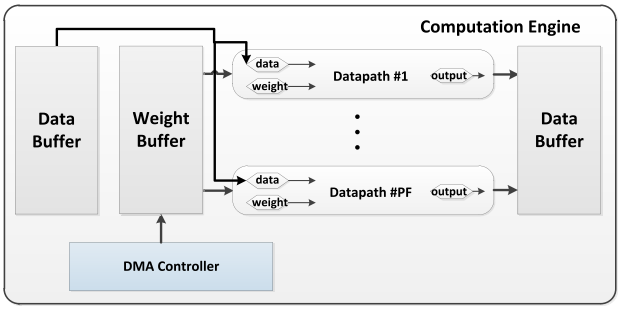


Fig. 5. Architecture of the overall accelerator that supports both channel parallelism and filter parallelism with double buffer technique employed. Weights and input image are transferred from DDR through the DMA controller. All intermediate data are processed in on-chip memories.

Therefore, instead of using data parallelism, we employ the channel parallelism in this work, as already shown in Fig. 4. Multiple channels (PC) of inputs are multiplied with the weights in parallel, and the results are then added together using an adder tree before the accumulation. The advantage of this design is that each multiplier's workload is balanced since the output pixels of different channels have the identical position in the output maps. In addition, the layers' input channel (except for the first layer) in CNNs are often a power of two or can be tuned to a power of two so that they can be divided by the degree of channel parallelism (PC) that is normally a power of two as discussed previously. Hence, the channel parallelism does not lead to any loss in the utilization of multipliers and it adapts to the algorithmic design from users. On the contrary, once the size of the first layer's input is determined, the size of all other layers is automatically decided, while the channel numbers are independent among layers in one CNN model.

Furthermore, the accelerator computes multiple filters (PF) of output in parallel. This is achieved by instantiating PF datapaths in the compute engine, where each datapath includes the unified module with channel parallelism (PC) and other hardware blocks that map a CNN onto an FPGA with parallel processing power. The overall design is shown in Fig. 5. Weights of the CNN are read from the DDR memory and cached in weight buffers using the double buffer technique to overlap the load time. Intermediate results are read from and written back to the local (on-chip) smart caches directly on the FPGA, with smart read/write controls. Note that the PF datapaths share the identical quantization module in Fig. 4 as they use the identical pixels as input data, which largely saves the hardware resources. The architecture design of the complete datapath is presented in Section III-C, while the smart cache system is described in Section III-D.

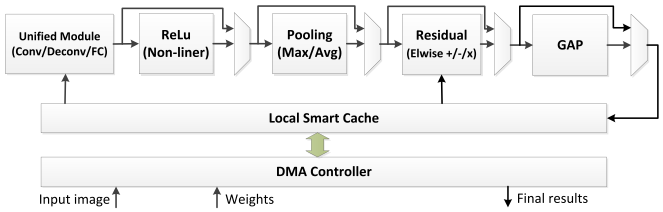


Fig. 6. Hardware blocks instantiated in our accelerator to map the mainstream CNNs on FPGA.

C. Hardware Building Blocks

Fig. 6 shows the hardware blocks of one datapath that maps a CNN model onto the FPGA. Each datapath consists of a unified module to perform Conv, Deconv, or FC operation. It is then followed by a ReLU module, which performs the nonlinear activation such as ReLU or leaky ReLU. The following pooling module is added to run average or maximum pooling on the input data. The residual block accepts one input from one of the preceding hardware blocks, which computes the results of the current layer, and another input from the local cache, which stored the result from the previous operation. It can perform element-wise operations, such as *add*, *subtract*, or *multiply*. The operations are implemented using the available DSPs and one DSP can be configured to run any of the three kinds of operations above during runtime simply by using control signals. Therefore, our design does not introduce any additional resource overhead by supporting three types of element-wise operations instead of potentially only supporting *multiply* in the residual block. The final connected block is a global average pooling (GAP) module, which is usually employed before a final FC layer in a CNN [40].

The concatenation layers do not perform any operations, and thus, no hardware block needs to be instantiated and they are actually implemented through smart cache design, as their operation is mainly dependent on routing of the incoming data.

The trick in the datapath design is that each hardware block can be bypassed through multiplexers that enable flexible layer configuration. Thus, it is capable of implementing a wide range of CNN topologies, such as GoogLeNet [8], ResNet [7], VGG16 [2], and YOLOv3 [9] by simply correctly configuring the datapath through control signals that influence the information flow. Besides, the whole datapath is run in a pipelined manner with support of intralayer pipeline using double buffer to overlap data transfer time with computation [25], as well as interlayer pipeline [41] to run all other functionalities in parallel with the unified module, thus keeping these blocks busy during most of the execution time and achieving high resource efficiency across the CNN models.

The input image and weights are stored in a DDR memory. While processing, they are first cached in the on-chip Block RAMs (BRAMs), i.e., local cache on the FPGA, and then, all the intermediate results are stored in the local cache without accessing the DDR memory, to avoid additional communication cost. The final result after execution is stored back to the DDR memory for further evaluation in the CPU. Therefore, the performance of our system is not limited by the bandwidth of the DDR interfaces.

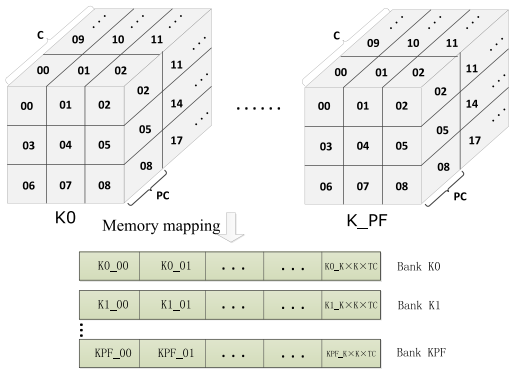


Fig. 7. Weight storing pattern in the weight buffer that consists of PF banks and each bank stores one set of filter weights with the width of PC weights ($8 \times$ PC bits) and the depth of $K \times K \times$ TC. Numbers in the figure represent the memory address of the attached data group.

D. Smart Cache Design

One of the advantages of FPGAs in comparison to GPUs and CPUs is their large on-chip bandwidth since the local BRAMs can be customized with large data width to decrease the access latency for the frequently used data. For example, in convolution, each input pixel is reused $K \times K \times F$ times, and weights are only used once. Data buffers are also needed to cache the input and output of standard convolutions, inputs of residual block, and multiple inputs of concatenative connections. Therefore, efficient utilization and management of the local caches on the FPGA are crucial to the performance of the overall system. Here, we introduce our smart cache design, in order to achieve the maximum memory utilization while maintaining parallel processing capabilities in channel and filter dimensions. Besides, we show how the local cache is divided and balanced into different parts, in order to improve the efficiency of concatenative and residual connections.

1) Data Storing Pattern in Cache: The storage data pattern in caches should mainly consider the support of parallel processing. Weights are simple and straightforward to be cached. Weight buffers are divided into PF memory banks, and each bank stores one set of filter weights, i.e., $C \times K \times K$. Each bank has a memory width of PC weights with the depth of $K \times K \times$ TC, where TC $= C/$PC. As shown in Fig. 7, the weights are stored in channel dimension first, followed by width and height dimensions. Weights of multiple filters are fed into different rows of the datapaths in parallel.

Data buffer design is much more challenging. The feature maps can be stored in the data buffer in two orders: channel-major and block-major. Both methods store the input feature maps in one single memory with the width of PC data pixels in channel dimension for implementation of channel parallelism. The illustration of both methods is shown in Fig. 8.

1) *Channel-Major:* It stores the data pixels in the order of $H \times W \times C$. The data in channel dimension are stored first and then followed by width and height.
2) *Block-Major:* It stores the feature maps block by block, as the total volume can be regarded as TC blocks in the channel dimension. Each data block is stored in the order of $H \times W \times$ TC.

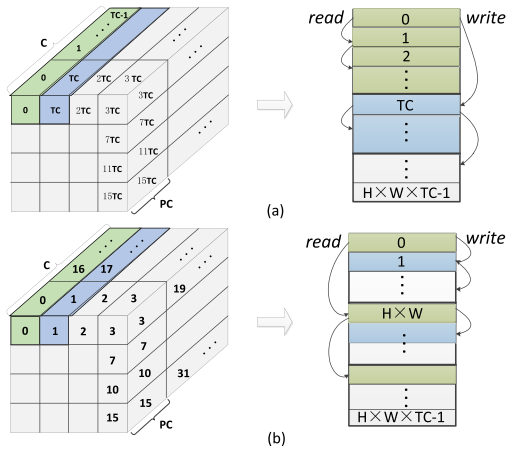


Fig. 8. Two alternatives of the data storing pattern in the data buffers and their corresponding reading and writing patterns with channel-major computing. (a) Channel-major. (b) Block-major.

In this work, we use the block-major storing method for more efficient implementation of concatenative connections, as we will discuss later.

2) Computing Pattern: Corresponding to the data storage pattern, there are two alternative ways for computing the standard Conv. For convenience, we still name them as channel-major and block-major computing patterns.

1) *Channel-Major:* It computes the final result of one data point first, and thus, it needs to access the data in one $K \times K$ window along the channel dimension until the end. This results in the datapaths reading $K \times K \times C$ input pixels in $\text{TC} \cdot K \cdot K$ cycles and then generating PF output pixels. The datapaths share the same input pixels. Every $\text{TC} \cdot K \cdot K$ cycles, the accelerator generates PF results, and in total, $\text{TC} \cdot K \cdot K \cdot H \cdot W \cdot F/PF$ cycles are needed.
2) *Block-Major:* It computes the results in width and height dimensions first instead of channel dimension, and the intermediate results of one block size need to be cached during the process. Every $K \cdot K \cdot H \cdot W \cdot \text{TC}$ cycles, it generates the results of the whole maps of PF filters, i.e., $H \times W \times$ PF output pixels. Compared to the channel-major method, it is more efficient for the DDR memory access since it generates a large volume of data consecutively and thus enables burst transfers of results to the DDR memory. However, it needs large buffers to cache the intermediate results with the size of $H \times W \times \text{PC} \times PF$, which increases the overhead of local caches. When block-major storage is used, the data are read discontinuously in the input buffer.

Nevertheless, the channel-major computation does not need any cache for intermediate results and provides us with more efficient utilization of local memories. Therefore, it better suits our architecture in which all the layers are processed by using on-chip memories. The behaviors of different combinations of storing and computing patterns in cache design are summarized in Table I. In this work, the *block-major storage* and *channel-major computation* are utilized with comprehensive consideration of design complexity and compute efficiency. Note that the output results are always produced in the block-major pattern because the results are generated in the filter dimension, so the block-major storing will lead to continuous writing behavior, as shown in Fig. 8.

TABLE I

SUMMARY OF THE BEHAVIORS OF STORING AND COMPUTING PATTERNS IN CACHE DESIGN

Behaviour: Computing \ Storing	**Channel-major**	**Block-major**
Channel-major: × DDR burst transfer √ no cache overhead	√ continuous read × continuous write × concat. efficient	**This Work:** × continuous read √ continuous write √ concat. efficient
Block-major: √ DDR burst transfer × no cache overhead	× continuous read × continuous write × concat. efficient	√ continuous read √ continuous write √ concat. efficient

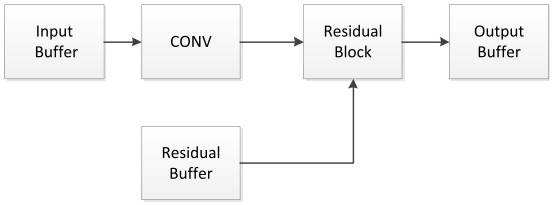


Fig. 9. Simplified illustration of the implementation of residual connections.

3) Data Buffer Organization: This section mainly considers how to manage and balance the data buffers for standard convolution, residual, and concatenative connections. When performing standard Conv, two data buffers are needed, which are input and output buffers. Residual connection has two data inputs and a single output, whereas concatenative connection can have more than two inputs and again a single output. As shown in Fig. 9, when connecting inputs in the residual layer, the input maps are stored in the input buffer and the accelerator can run standard Conv first, then, its output is connected to the residual block with the other input coming from a second data buffer—*Residual Buffer*, and finally, the residual output is stored in the output buffer. With this design, we can keep both unified module and residual block busy at the same time, which guarantees high resource occupancy.

Since there are usually more than one residual or concatenative connections in a single network, either memory buffer can be used as an input or output or residual buffer. Therefore, in this work, we customize three memory buffers to cache the data that all have the identical size and structure. When performing the concatenative connections, because the data are stored in buffers per block, they are concatenated together just by jumping to the other memory location, which stores the other input. This also works for even more intricate input patters, such as three inputs concatenated, by storing the multiple data blobs in one memory buffer. The simplicity and efficiency of implementation of the concatenative connection are owed to the chosen block-major storing pattern in data buffers.

E. Overall Accelerator

The overall system is shown in Fig. 10. It consists of the host processor, computation engine, on-/off-chip interconnect

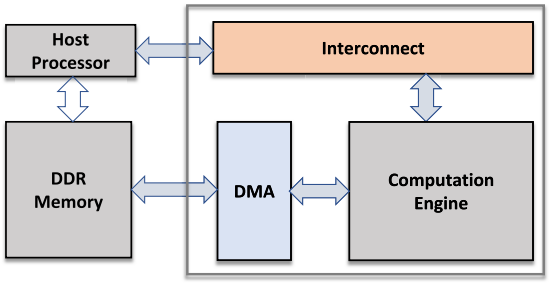


Fig. 10. Overview of the complete hardware system for FPGA-based acceleration for CNN inference.

(DMA), and off-chip DDR memory. The host processor is used to configure the parameters of layers when running the CNN model in the computation engine. All the weights of the model, the input image, and the final results of classification/detection/segmentation are stored in the DDR memory.

IV. Design Optimizations

This section presents the optimization techniques used in our approach to improve resource efficiency and compute efficiency of the proposed accelerator.

A. Layer Fusion

Layer fusion [27] is a common optimization technique to minimize data movement, which is considered in practical designs. By storing all intermediate data in on-chip memory, layer fusion processes multiple CNN layers at the same time in a pipelined manner without the need for external memory access. As shown in Fig. 5, input and output feature maps are cached in the data buffers using BRAMs during the execution. The memory size and data structure of these data buffers are the same, as described above. Before the processing of the first layer, the input data are transferred from the DDR and cached in one buffer. Then, the inputs are streamed into the datapaths, whereas the outputs are simultaneously flowing into the second data buffer. When the computation of the first convolution finishes, the second data buffer acts as the input buffer for the second convolution and the outputs will be cached in the other buffers. In the end, the final outputs are transferred back to the DDR from the data buffers. The double buffer technique is also used to cache weights, in order to overlap the weight load time with the computation time. For simplicity, it is not shown in Fig. 5.

B. Input Reshaping to Improve Utilization

For CNNs trained on ImageNet [37], nearly 10%–15% of the total computation is associated with the first convolution layer because of the large spatial size of the input image [19]. However, the computation of the first convolution layer has not been mapped well onto the previous hardware accelerators such as those based on the systolic architectures [42] because the input image only offers a small number of channels, which cannot fully utilize the input bandwidth and leads to the underutilization of the computing resources.

To solve this imbalance, we reshape the first layer [19] to improve the resource efficiency. The input maps are divided into multiple small blocks. Then, we concatenate these

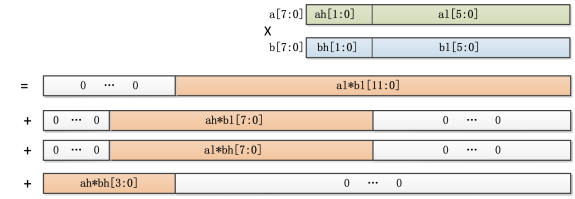


Fig. 11. INT8 multiplication is decomposed into one 6-b multiplication and other simple operations.

blocks together along the channel dimension. Correspondingly, in order to fit them into our computation engine and guarantee that the computation results are correct, we compute each set of three channels of data in a set of four multipliers in the unified module (see Fig. 4) with one multiplier idle, while the adder tree is disabled when executing the first layer. Hence, each datapath generates PC/4 output pixels in total at a time. As a result, the compute efficiency of the first layer is increased from 3/PC to 3/4 = 75%.

C. DSP Optimization for INT8 Multiplier

The maximum performance of the system depends on the number of multipliers used in our design. In FPGAs, DSPs are often used to implement multipliers, which makes them the most limiting resource for CNN acceleration. The variable-precision DSP block in Intel Arria 10 devices includes two 18×19 multipliers with variable arithmetic precision support. Without any optimization, one DSP can be configured as two INT8 multipliers. While running INT8 computations, the higher input width can carry another computation if the lower 8-b input and its 16-b results are not affected. Based on this, Xilinx proposed a method to optimize the 18×27 bit multiplier on its DSP48E2 slice for INT8 operations, which achieves a $1.75\times$ performance improvement, i.e., 1 : 1.75 DSP multiplier to INT8 MAC ratio [43]. However, it claimed that an 18×19 multiplier in the Intel DSP block is limited to a 1:1 ratio of DSP multiplier to INT8 MAC. The reason lies in that such optimization must guarantee that the upper bits should not affect the computation of the lower bits. Therefore, it requires a minimum of $16 + 8 = 24$ bits of the total input bit width.

In this work, we propose an INT8 optimization method, targeted at Intel's DSP block, to efficiently map two 8-b multiply into one 18×19 multiplier, i.e., 1:2 DSP multiplier to INT8 MAC ratio. Since the multiplier's input width is only 18 b, we first separate the inputs ($a[7:0]$) into two parts: the higher 2 b ($ah = a[7:6]$) and the lower 6 bit ($al = a[5:0]$). Then, the multiply $a \times b$ is decomposed into one 6-b multiply, three multiply with very small input bits, and one addition of the four product results, as shown in Fig. 11. Now, we can pack 6-b inputs a and b in the higher and lower 6 b of the multiplier's 18-b input port A and c and d in port B in the same manner, as shown in Fig. 12. The 36-b product result has $a \times c$ in higher 12 b and $b \times d$ in lower 12 b. As a result, two multiplication results can be separated from the 36-b product, and the other three simple operations required to generate the 8-b multiply result are implemented with logic resources.

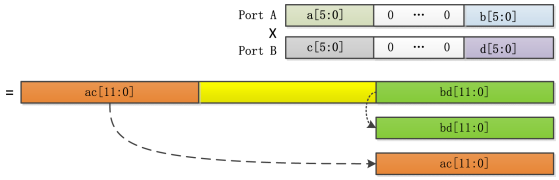


Fig. 12. 18 × 19 multiplier in Arria 10 DSP block is optimized to implement two 6 × 6 multipliers, in order to improve the performance of INT8 multiplication.

V. Latency Estimation With Gaussian Process

A. Motivation

Design space exploration (DSE) has been widely used in hardware accelerators [11], [36], [44] for CNNs, to optimize a wide range of hardware parameters in an effort to efficiently map a CNN onto the target FPGA. The DSE process usually involves two approximating models. The models are used instead of running the CNN on real hardware after each hardware iteration with different hardware parameters to collect measurements, which is very time-consuming. One model is the resource model, which models the resources for a specific architecture with given hardware parameters in the target FPGA device. The other model is the performance model, which usually estimates the corresponding system performance, e.g., latency, given the chosen hardware and fixed algorithmic properties. Then, DSE will try to find the optimal design parameters that achieve the best performance under the resource constraints for a given device.

There are several rather complicated performance estimation frameworks for FPGA-based accelerators [45], [46]. Therefore, practitioners usually resort to an analytic formulation of performance prediction that provides a rough estimate, e.g., for the latency, due to the simplicity of this prediction method. In addition, the analytic approximation can be easier to integrate into the DSE optimization loop, which is often custom to support a variety of CNNs [5], as in comparison to working with all-round simulation software such as ModelSim.

Nonetheless, avoiding the use of dedicated simulation software or complicated performance predictors and instead of using only an analytic approximation introduces several challenges. First, by formulating an analytic approximation, we usually avoid to count for scheduling, which can introduce errors in the prediction. Second, the explicit time to execute a certain operation on hardware varies by on-/off-chip communication, synchronization, control signals, I/O interruptions, and in particular for the CNN accelerators—the CNN's architecture, which cannot be covered by analytic estimation. Third, a pure analytic method is unable to account for any collected real-world performance measurements. Therefore, it is necessary to develop a performance estimation method, which provides the user with a reliable guarantee of the expected performance while not increasing the implementation effort.

B. Our Method

In this work, we propose a novel approach for accurate performance estimation of FPGA-based CNN accelerators that we used to estimate the latency of a given CNN on the accelerator. This method employs a Gaussian process regression (GPR) [47] approach coupled with the standard analytic formulation [11] and the collected measurements.

GPR is a nonparametric, Bayesian approach for regression that can embody prior knowledge/model into the target. It is specified by a mean function $m(\cdot)$ and a covariance function (kernel) $k(\cdot,\cdot)$. The mean function represents the supposed average of the estimated data. The kernel computes correlations between inputs and it encapsulates the structure of the hypothesized function.

The predictive distribution $p(y_t|\mathbf{X},\mathbf{y},\mathbf{X}_t)$ for the targets y_t given the corresponding features $\mathbf{X}_t$ and the training data $(\mathbf{X},\mathbf{y})$ is defined as a multivariate Gaussian distribution with a predictive mean $\mathbb{E}[y_t|\mathbf{X},\mathbf{y},\mathbf{X}_t]$ and a predictive variance $\mathbb{V}[y_t|\mathbf{X},\mathbf{y},\mathbf{X}_t]$, which are defined as follows:

$$\mathbb{E}[y_t|\mathbf{X},\mathbf{y},\mathbf{X}_t] = m(\mathbf{X}_t) + k(\mathbf{X}_t,\mathbf{X})(k(\mathbf{X},\mathbf{X}) + \sigma^2\mathbf{I})^{-1}(y - m(\mathbf{X})) \quad (4)$$

$$\mathbb{V}[y_t|\mathbf{X},\mathbf{y},\mathbf{X}_t] = k(\mathbf{X}_t,\mathbf{X}_t) + k(\mathbf{X}_t,\mathbf{X})(k(\mathbf{X},\mathbf{X}) + \sigma^2\mathbf{I})^{-1}k(\mathbf{X}_t,\mathbf{X})^T \quad (5)$$

where σ^2 represents the noise amplitude and $\mathbf{I}$ is the identity matrix. The detailed derivations can be found in [48].

In this work, the GP's target is to estimate the latency for a single layer based on the input features. The inputs are the features that include the model's layer parameters introduced in Section III-A and the accelerator's parameters, such as the degrees of parallelism (PC and PF), clock frequency, and data width. The output is the corresponding layer's execution time, i.e., latency. The standard analytic formulation developed in our prior work [11] is used as the mean function of the GP, with the profiling data collected by running the CNN on real hardware as the training data. Matérn 3/2 kernel [48] is chosen as the GP's kernel.

The main benefit of using a GP over other methods, such as linear regression or gradient tree boosting, which rely on a large number of collected measurements, is that it can use the previously developed analytic formulation, as prior knowledge in a form of $m(\cdot)$. Thus, it reuses any previously developed heuristics and only minimally increases the implementation effort by tuning a small number of hyperparameters while requiring a smaller number of collected measurements due to the heuristic. Moreover, it can use the previously collected measurements $(\mathbf{X},\mathbf{y})$ to learn to account for any nonlinearities such as on-/off-chip communication, synchronization, or control signals.

VI. Evaluation and Experiments

A. Benchmarks

Some typical CNNs have been tested as benchmark models, as listed in Table II. These models are widely used for tasks of classification, object detection, and segmentation. VGG16 [2] is one of the largest and computationally intensive networks, with serial layer connectivity and uniform kernel size (3×3) across its convolutional layers. ResNet-50 and ResNet-101 [7] represent the mainstream networks that contain the residual connections insides their blocks. Inception-v4 [40]

TABLE II
BENCHMARK MODELS

Category	Network	Workloads (GOPs)	Characteristic
Classification	VGG16 [2]	30.94	· serial connectivity · uniform kernel size
	ResNet-50 [7] ResNet-101 [7]	7.7 15.5	· residual connection
	Inception-v4 [40]	27.6	· concat. connection
Object Detection	SSD [49]	5.254	· branches
	YOLOv3 [9]	71.4	· up-sampling · concat. connection · residual connection
Segmentation	U-Net [10]	816.9	· up-sampling · concat. connection

has a more uniform and simplified architecture with concatenative connections compared to ResNet models. SSD [49] has the architecture that builds on VGG16, and a set of auxiliary convolutional layers were added to extract features at multiple scales and progressively decrease the size of the input to each subsequent layer. U-Net [10] is famous for the introduction of large upsampling (deconvolutional) layers for semantic segmentation and it also has concatenative connections. YOLOv3 [9] is a mainstream network with feature map upsampling and concatenation. Its feature extractor is built on Darknet-53 that is organized as a series of residual blocks. Therefore, YOLOv3 has all the characteristics of irregularities.

B. Implementation Details

Our accelerator was implemented and evaluated on the Intel's Arria 10 device that consists of a high-performance and power-efficient FPGA device, i.e., Arria 10 GX1150 (20 nm), a dual-core ARM Cortex-A9 processor (1.5 GHz), and 2-GB DDR4 memory. The ARM CPU was used to configure the layers' parameters when running each model in our accelerator. All the hardware modules are developed using Verilog HDL. The hardware system was synthesized and placed-and-routed with Quartus Prime Pro 18.1. In the target device, our accelerator achieved the optimal design parameters at PC $\times$ PF $= 64 \times 64$ and the computation engine is run at the clock frequency of 200 MHz.

C. Latency Estimation Results

The evaluation data set comprises the convolutional layers from three CNNs, i.e., 24 convolutions of SSD [49], 57 convolutions of ResNet-50 [7], and 75 convolutions of YOLOv3 [9]. Each model was executed on the implemented accelerator on Intel Arria GX1150 FPGA. For a more comprehensive evaluation, leave-one-out cross validation was used, where each time, one sample was left out and all the others were used for training. This process is then repeated for each sample in the data set. The GPR is implemented using the existing GPflow [50] library, and it was trained using an Adam optimizer with the initial learning rate 1×10^{-3} until convergence with respect to the relative error. The result is shown in Table III in comparison to the standard method in [11] using the analytic formulations.

The experiment results demonstrate the estimation accuracy improvements provided by the GPR. Compared to the standard method, it reduces the relative error from 27.6% to 8.3%, 33% to 9.2%, and 22% to 3.1% for the evaluated models, achieving a maximum of 23.8% and an average of 20.7% reduction in the errors of latency estimation. The results confirm that our method provides a very accurate estimate of latency and thus accelerates the process of CNN model tuning in order to satisfy the latency requirement for real-time applications. Therefore, the proposed method can largely reduce the design time for the tradeoff between accuracy and performance and improve the hardware design productivity.

TABLE III
LATENCY ESTIMATION WITH GPR COMPARED TO STANDARD METHOD

	ResNet-50		SSD		YOLOv3	
	standard	GPR	standard	GPR	standard	GPR
Mean absolute error per layer (ms)	0.276	0.112	0.27	0.086	0.408	0.192
Maximum absolute error per layer (ms)	1.07	0.394	1.26	0.323	1.75	0.661
Total estimated latency (ms)	3.67	4.65	2.40	3.24	38.2	47.5
Total reference latency (ms)	5.07		3.57		48.99	
Relative error	27.6%	8.3%	33%	9.2%	22%	3.1%

TABLE IV
RESOURCE UTILIZATION OF THE ACCELERATOR ON ARRIA 10 GX1150

Resources	ALMs	Registers	DSPs	M20K
Used	303,913	888,576	1,473	2,334
Total	427,200	1,708,800	1,518	2,713
Utilization	71%	52%	97%	86%

D. Resource Efficiency

Table IV shows the resource utilization of the accelerator on Arria 10 GX1150. Owning to the use of 8-b quantization and the proposed DSP optimization technique, the low-precision fixed-integer multipliers are implemented individually in soft logic or combined with other multipliers in the DSP blocks, leading to high resource utilization and great compute density. However, the result is routing congestion, which has a negative impact on the working clock frequency [51].

Fig. 13 shows a breakdown of the resources of each module in the datapath. Since the unified module (MM) is the core computation block, it has the highest utilization of ALMs, Registers, and DSPs among all the modules. Besides the unified module, the arithmetic operation inside the residual and ReLU blocks is implemented with DSPs that can be configured for element-wise *add*/*subtract*/*multiply* operation. The other two modules, i.e., GAP and Pooling use soft logic to implement the arithmetic operations, and thus, they use a relatively high percentage of ALMs and on-chip registers.

E. Compute Efficiency and Model Accuracy

Table V shows a summary of the performance, compute (MAC) efficiency, and accuracy for our benchmark models when running on Arria 10 GX1150 device. In this work, the power consumption is obtained by subtracting the idle power from the power measurement of the board due to the benchmark execution. Our accelerator achieves the

TABLE V
PERFORMANCE AND ACCURACY ON BENCHMARK MODELS

	Dataset	Network	Latency[a](ms)	Throughout[a] GOP/s	Energy Efficiency[b] GOP/s/W	MAC Efficiency	Accuracy			
							standard	This Work	FP32	diff.
Classification	Imagenet [37]	VGG16	23.9	1295	75.3	79.1%	top-1	70.23%	70.98%	-0.75%
		ResNet-50	5.07	1519	79.5	92.7%		74.73%	75.13%	-0.4%
		ResNet-101	9.79	1590	80.3	97.0%		76.72%	76.47%	+0.25%
		Inception-v4	21	1314	71.0	80.2%		80.06%	80.1%	-0.04%
Object Detection	FDDB [52]	SSD	3.57	1472	82.7	89.8%	IoU≥0.5	83.5%	83.5%	0
	COCO [53]	YOLOv3	48.99	1457	58.8	89.0%	IoU≥0.5	56.6%	57%	-0.4%
Segmentation	Cityscapes [54]	U-Net	543.19	1504	70.0	91.8%	Pixel	95.36%	95.53%	-0.17%
							Class	93.68%	93.75%	-0.07%
							IoU	85.96%	86.55%	-0.59%

[a] The batch size is set to 1.
[b] The energy efficiency is quoted in giga-operations per second per watt (GOP/s/W). Chip power is due to benchmark execution with subtracted idle power.

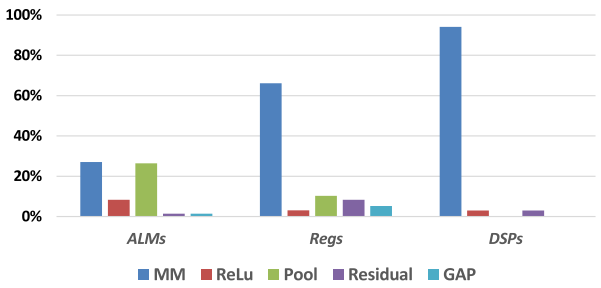


Fig. 13. Resource breakdown of each module in the datapath.

throughput of 1.30–1.59 TOP/s (teraoperations per second), which is up to 97% of the realized maximum performance.[2] As can be seen, it achieves the compute efficiency of 79.1%–97.0%, depending upon the network. The relatively low efficiency of VGG16 is due to the large computation of the first layer since it can only achieve 75% of efficiency as we have discussed in Section IV-B. Nevertheless, our accelerator achieves a high compute efficiency of more than 89% for networks with irregular types, such as ResNet and YOLOv3. Our framework employs INT8 quantization scheme in [16], and the resulting accuracy for the CNNs is almost equivalent to that of the original floating-point 32-b (FP32) model, which are within 1% point of the original FP32 accuracy without retraining.

F. Performance Comparison

1) Comparison With Embedded GPU: We compare the performance of our design with the widely used high-performance NVIDIA Tegra X1 platform. TX1 has 256 CUDA cores delivering over 1 TOP/s of peak performance with a power consumption of 10 W. NVIDIA TensorRT as supplied by the JetPack 3.1 package was run with the NVIDIA cuDNN library and FP16 precision, which enables a highly optimized execution of layers. Although a batched way of processing can fully utilize the parallelism of GPU on TX1, it is not a good choice for real-time processing because it increases latency, as discussed in Section I. Therefore, on all evaluated platforms, the benchmarks are run with a batch size of 1.

[2]The realized maximum performance is defined by multiplying the sum of the adders and multipliers used in the design by the working clock frequency.

TABLE VI
COMPARISON WITH EMBEDDED GPU TX1

	GPU TX1 (fp16)		FPGA GX1150		
	Latency (ms)	GOP/s	Latency (ms)	Speedup	
				GOP/s	GOP/s/W
VGG16	96.5	318	23.9	4.04×	2.37×
ResNet-50	53.09	145	5.07	10.5×	5.48×
ResNet-101	84.52	183	9.79	8.63×	4.39×
Inception-v4	158.00	174	21	7.52×	4.10×
SSD	21.22	247.5	3.57	5.94×	3.34×
YOLOv3	454	131	48.99	9.27×	4.48×
U-Net	2540	322	543.19	4.68×	2.17×

Performance comparison is shown in Table VI. As we can see, the GPU performance has a large divergence across the evaluated models from 131 to 322 GOP/s compared to ours of 1.3–1.59 TOP/s on FPGA. In general, GPU performs better on larger CNN models with regular shape and serial connectivity, such as in cases of VGG16 and U-Net. However, the GPU's performance decreases dramatically on smaller models or models with residual or concatenative connections. As a result, GPU TX1 has the lowest performance of 131 GOP/s for YOLOv3 among all the evaluated models. Owning to our customized and careful design for the irregular connections, our accelerator achieves an overall high compute efficiency across all benchmark models. The proposed accelerator achieves 4×–10.5× speedup in terms of the throughput of GOP/s and 2.17×–5.48× improvements on the energy efficiency of GOP/s/W compared to GPU.

2) Comparison With Previous FPGA Accelerators: Table VII shows the performance comparison of our design against prior FPGA-based accelerators. All results are based on the batch size equal to 1. The total number of DSPs in a device is used to compute the performance density (GOP/s/DSP) because the utilization of DSPs can be regarded as a metric of the quality of the hardware architecture design of FPGA-based accelerators.

For all evaluated networks, our accelerator outperforms all other accelerators in terms of both performance density (GOP/s/DSP) and energy efficiency (GOP/s/W), as shown in Table VII. Among all the accelerators, we achieve the best performance density of 1.0 GOP/s/DSP and the energy

TABLE VII
COMPARISON WITH PREVIOUS FPGA ACCELERATORS

	Ma *et al.* [26] in FPGA 2017	Aydonat *et al.* [32] in FPGA 2017	Guo *et al.* [55] in TCAD 2018	Liu *et al.* [11] in TRETS 2018	Venieris *et al.* [36] in TNNLS 2019		This Work		
Platform	Intel GX1150	Intel GX1150	Xilinx Zynq-7020	Xilinx Zynq-7045	Xilinx Zynq-7045		Intel GX1150		
Frequency (MHz)	150	303	214	200	125		200		
Bit-width	8-16 bit fixed	16-bit float	8-bit fixed	16-bit fixed	16-bit fixed		8-bit fixed		
#DSP	1518	1518	220	900	900		1518		
Logic (ALMs/LUTs)	427K	427K	53K	218K	218K		427K		
Power (W)	21.2	45[a]	3.5	9.6	4.8		17.2	19.1	21.5
CNN Model	VGG16	AlexNet	VGG16	Optimized U-Net	VGG16	ResNet-152	VGG16	ResNet-50	U-Net
Latency[b](ms)	47.97	not reported	364	58.0	249.5	156.4	23.9	5.07	543.19
Performance (GOP/s)	645.25	1382	84.3	107	124	147	1295	1519	1504
Performance Density[c] (GOP/s/DSP)	0.425	0.91	0.38	0.12	0.14	0.163	0.86	1.0	0.99
Energy Efficiency (GOP/s/W)	30.4	30.7	24.1	11.2	25.8	30.6	75.3	79.5	70.0

[a] The total board power is used in [32].
[b] All works use the batch size of 1.
[c] The performance density results are computed by dividing the total DSPs in the devices for all designs.

efficiency of 79.5 GOP/s/W. Aydonat *et al.* [32] used the Intel Xeon-FPGA Platform that targets the data center applications and achieved a similar performance density of 0.91 GOP/s/DSP to our work. However, their work only implemented AlexNet, which has a uniform and regular shape, and its performance will be impacted negatively with other CNN topology with irregular connections. Besides, Aydonat *et al.* [32] used a batch size of 1 for convolution layers and 96 for the FC layers, which increased the throughput but actually also increased the latency. Compared to the state-of-art implementation of CNNs with irregular shapes in [36], we achieve a performance density improvement of 6.13× and an energy efficiency improvement of 2.9× for VGG16 and ResNet.

3) Overall Efficiency Comparison: Here, we compare the overall efficiency of our accelerator to the state-of-the-art work presented in [12], which devoted the efforts to achieve high compute efficiency and clock frequency. The overall efficiency is defined as the ratio of the achieved performance to the peak performance of the device. The peak performance of the FPGA device is computed by multiplying the total number of multipliers and adders incorporated into the DSP blocks by the maximum clock rate [51]. Correspondingly, the realized maximum performance is defined by multiplying the sum of the adders and multipliers implemented in the design by the working clock rate. Compute or MAC efficiency also refers to the ratio of achieved performance to the realized maximum performance. Therefore, the overall efficiency can be actually computed as

$$\text{OVERALL EFF.} = \text{CLOCK EFF.} \times \text{RES. EFF.} \times \text{COMPUTE EFF.} \quad (6)$$

where clock efficiency refers to the ratio of working clock rate to maximum clock rate, resource efficiency or utilization is the ratio of multipliers and adders used in the design to that incorporated in DSPs, and compute efficiency is the fraction of useful MAC cycles. Optimizing clock rate and improving resource utilization are two competing strategies for high-performance FPGA accelerators. Note that logic resources are not considered when computing the peak performance since they make the computation very difficult and a large amount of logic is required for other functions. Besides, logic usage has a negative impact on the working clock rate [51].

TABLE VIII
OVERALL EFFICIENCY COMPARISON

Design	Clock Eff.*	Res. Eff.	Compute. Eff.◇	Overall. Eff.
[12]	92.9%	55.8%	95.9%	49.7%
Ours	44.4%	135%	90%	54.0%

* A rated speed of 450 MHz is used as the maximum clock frequency of Arria 10 DSPs [51].
◇ Average efficiency of 90% is used for evaluation.

The results are shown in Table VIII. Due to the proposed DSP optimization for INT8 multiplier, we have achieved a 1:1.35 DSP multiplier to INT8 MAC ratio of the overall compute engine, leading to a very high resource efficiency. This, in turn, limits the achieved clock rate that is only 44% of the maximum due to the resulting routing congestion in FPGA device [51], which is lower than that of [12]. Nevertheless, we achieve an overall efficiency of 54.0%, which outperforms the work in [12].

VII. CONCLUSION

This article presents an accelerating framework toward the full-stack acceleration of CNNs on FPGAs. Computational functions, such as convolutional, deconvolutional, and full-connected layers, are mapped in a unified architecture by exploiting different levels of parallelism and fully leveraging the DSPs. Besides, the proposed accelerator addresses the efficiency of the irregular connections in CNN models such as residual and concatenative connections by the smart cache design. Quantitative evaluation results demonstrate that our accelerator outperforms the performance density and energy efficiency of existing state-of-the-art FPGA-based accelerators, achieves a high compute efficiency, and therefore provides

a highly optimized, specialized hardware accelerator for ML acceleration.

REFERENCES

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1097–1105.

[2] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014, *arXiv:1409.1556*. [Online]. Available: http://arxiv.org/abs/1409.1556

[3] J. Dai, Y. Li, K. He, and J. Sun, "R-FCN: Object detection via region-based fully convolutional networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 379–387.

[4] V. Badrinarayanan, A. Kendall, and R. Cipolla, "SegNet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 12, pp. 2481–2495, Dec. 2017.

[5] S. Liu and W. Luk, "Towards an efficient accelerator for DNN-based remote sensing image segmentation on FPGAs," in *Proc. 29th Int. Conf. Field Program. Log. Appl. (FPL)*, Sep. 2019, pp. 187–193.

[6] J. Fowers *et al.*, "A configurable cloud-scale DNN processor for real-time AI," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2018, pp. 1–14.

[7] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 770–778.

[8] C. Szegedy *et al.*, "Going deeper with convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 1–9.

[9] J. Redmon and A. Farhadi, "YOLOv3: An incremental improvement," 2018, *arXiv:1804.02767*. [Online]. Available: http://arxiv.org/abs/1804.02767

[10] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional networks for biomedical image segmentation," in *Medical Image Computing and Computer-Assisted Intervention*. Cham, Switzerland: Springer, 2015, pp. 234–241.

[11] S. Liu, H. Fan, X. Niu, H.-C. Ng, Y. Chu, and W. Luk, "Optimizing CNN-based segmentation with deeply customized convolutional and deconvolutional architectures on FPGA," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, no. 3, pp. 19:1–19:22, Dec. 2018, doi: 10.1145/3242900.

[12] E. Wu, X. Zhang, D. Berman, I. Cho, and J. Thendean, "Compute-efficient neural-network acceleration," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2019, pp. 191–200.

[13] M. Abadi *et al.*, "TensorFlow: A system for large-scale machine learning," in *Proc. 12th USENIX Symp. Oper. Syst. Design Implement. (OSDI)*, 2016, pp. 265–283.

[14] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 8024–8035.

[15] Y. Jia *et al.*, "Caffe: Convolutional architecture for fast feature embedding," in *Proc. 22nd ACM Int. Conf. Multimedia*, Nov. 2014, pp. 675–678.

[16] B. Jacob *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, Jun. 2018, pp. 2704–2713.

[17] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," 2015, *arXiv:1502.03167*. [Online]. Available: http://arxiv.org/abs/1502.03167

[18] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," 2016, *arXiv:1603.07285*. [Online]. Available: http://arxiv.org/abs/1603.07285

[19] B. McDanel, S. Q. Zhang, H. T. Kung, and X. Dong, "Full-stack optimization for accelerating CNNs with FPGA validation," 2019, *arXiv:1905.00462*. [Online]. Available: http://arxiv.org/abs/1905.00462

[20] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on FPGAs," in *Proc. IEEE 25th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2017, pp. 101–108.

[21] C. Zhang and V. Prasanna, "Frequency domain acceleration of convolutional neural networks on CPU-FPGA shared memory system," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2017, pp. 35–44.

[22] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 4013–4021.

[23] A. G. Howard *et al.*, "MobileNets: Efficient convolutional neural networks for mobile vision applications," 2017, *arXiv:1704.04861*. [Online]. Available: http://arxiv.org/abs/1704.04861

[24] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 1251–1258.

[25] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2015, pp. 161–170.

[26] Y. Ma, Y. Cao, S. Vrudhula, and J.-S. Seo, "Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, New York, NY, USA, Feb. 2017, pp. 45–54, doi: 10.1145/3020078.3021736.

[27] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN accelerators," in *Proc. 49th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Oct. 2016, pp. 1–12.

[28] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "XNOR-Net: Imagenet classification using binary convolutional neural networks," in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 2016, pp. 525–542.

[29] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," 2016, *arXiv:1605.04711*. [Online]. Available: http://arxiv.org/abs/1605.04711

[30] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 4107–4115.

[31] M. Ghasemzadeh, M. Samragh, and F. Koushanfar, "ReBNet: Residual binarized neural network," in *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2018, pp. 57–64.

[32] U. Aydonat, S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu, "An OpenCL deep learning accelerator on Arria 10," in *Proc. ACM/SIGDA Int. Symp. Field-Program. Gate Arrays*, Feb. 2017, pp. 55–64.

[33] A. Yazdanbakhsh *et al.*, "FlexiGAN: An end-to-end solution for FPGA acceleration of generative adversarial networks," in *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2018, pp. 65–72.

[34] J. Yan, S. Yin, F. Tu, L. Liu, and S. Wei, "GNA: Reconfigurable and efficient architecture for generative network acceleration," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2519–2529, Nov. 2018.

[35] S. Liu *et al.*, "Memory-efficient architecture for accelerating generative networks on FPGA," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2018, pp. 30–37.

[36] S. I. Venieris and C.-S. Bouganis, "FpgaConvNet: Mapping regular and irregular convolutional neural networks on FPGAs," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 30, no. 2, pp. 326–342, Feb. 2019.

[37] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, Jun. 2009, pp. 248–255.

[38] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks," *IEEE J. Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, Jan. 2017.

[39] R. Zhao, X. Niu, Y. Wu, W. Luk, and Q. Liu, "Optimizing CNN-based object detection algorithms on embedded FPGA platforms," in *Applied Reconfigurable Computing*. Cham, Switzerland: Springer, 2017, pp. 255–267.

[40] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Proc. 31st AAAI Conf. Artif. Intell.*, vol. 31, no. 1, 2017, pp. 4278–4284.

[41] B. Liu, S. Chen, Y. Kang, and F. Wu, "An energy-efficient systolic pipeline architecture for binary convolutional neural network," in *Proc. IEEE 13th Int. Conf. ASIC (ASICON)*, Oct. 2019, pp. 1–4.

[42] X. Wei *et al.*, "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs," in *Proc. 54th Annu. Design Autom. Conf.*, Jun. 2017, pp. 1–6.

[43] Xilinx. (2017). *Deep Learning With INT8 Optimization on Xilinx Devices*. [Online]. Available: https://www.xilinx.com/support/documentation/white_papers/wp486-deep-learning-int8.pdf

[44] R. Zhao, H.-C. Ng, W. Luk, and X. Niu, "Towards efficient convolutional neural network for domain-specific applications on FPGA," in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 147–1477.

[45] S. Dai, Y. Zhou, H. Zhang, E. Ustun, E. F. Y. Young, and Z. Zhang, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," in *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr. 2018, pp. 129–132.

[46] R. Yasudo, J. Coutinho, A. Varbanescu, W. Luk, H. Amano, and T. Becker, "Performance estimation for exascale reconfigurable dataflow platforms," in *Proc. Int. Conf. Field-Program. Technol. (FPT)*, Dec. 2018, pp. 314–317.
[47] C. K. Williams and C. E. Rasmussen, "Gaussian processes for regression," in *Proc. Adv. Neural Inf. Process. Syst.*, 1996, pp. 514–520.
[48] C. E. Rasmussen, "Gaussian processes in machine learning," in *Summer School on Machine Learning*. Berlin, Germany: Springer, 2003, pp. 63–71.
[49] W. Liu *et al.*, "SSD: Single shot multibox detector," in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 2016, pp. 21–37.
[50] D. G. Matthews *et al.*, "GPflow: A Gaussian process library using TensorFlow," *J. Mach. Learn. Res.*, vol. 18, no. 1, pp. 1299–1304, 2017.
[51] Intel. (2017). *Understanding Peak Floating-Point Performance Claims*. [Online]. Available: https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01222-understanding-peak-floating-point-performance-claims.pdf
[52] V. Jain and E. Learned-Miller, "FDDB: A benchmark for face detection in unconstrained settings," Univ. Massachusetts Amherst, Amherst, MA, USA, Tech. Rep. UM-CS-2010-009, 2010.
[53] T.-Y. Lin *et al.*, "Microsoft COCO: Common objects in context," in *Proc. Eur. Conf. Comput. Vis.*, 2014, pp. 740–755, doi: 10.1007/978-3-319-10602-1_48.
[54] M. Cordts *et al.*, "The cityscapes dataset for semantic urban scene understanding," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 3213–3223.
[55] K. Guo *et al.*, "Angel-eye: A complete design flow for mapping CNN onto embedded FPGA," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 1, pp. 35–47, Jan. 2018.

Shuanglong Liu received the B.Sc. and M.Sc. degrees from the Department of Electronic Engineering, Tsinghua University, Beijing, China, in 2010 and 2013, respectively, and the Ph.D. degree in electric engineering from Imperial College London, London, U.K., in 2017.

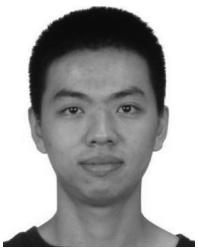
From 2017 to 2020, he was a Research Associate with the Department of Computing, Imperial College London. He is currently a Distinguished Professor with the School of Physics and Electronics, Hunan Normal University, Changsha, China. He has published over 30 research papers in peer-referred journals and international conferences. His current research interests include reconfigurable and high-performance computing for convolutional neural networks (CNNs) and statistical inference problems.

Hongxiang Fan received the B.S. degree in electronic engineering from Tianjin University, Tianjin, China, in 2017, and the master's degree from the Department of Computing, Imperial College London, London, U.K., in 2018, where he is currently pursuing the Ph.D. degree in machine learning and high-performance computing.

His current research focuses on efficient algorithm and acceleration for machine learning applications.

Martin Ferianc received the M.Eng. degree from the Department of Electronic Engineering, Imperial College London, London, U.K., in 2019. He is currently pursuing the Ph.D. degree with University College London, London.

His current research interests include convolutional neural networks and neural architecture search applied to computer vision tasks.



Xinyu Niu received the B.Sc. degree from Fudan University, Shanghai, China, in 2010, and the M.Sc. and Ph.D. degrees in computing science from Imperial College London, London, U.K., in 2011 and 2015, respectively.

He is the Co-Founder and CEO of Corerain Technologies, Shenzhen, China. His current research interest includes developing applications and tools for reconfigurable computing that involves runtime reconfiguration.

Huifeng Shi received the B.Sc. degree from the Nanjing University of Aeronautics and Astronautics (NUAA), Nanjing, China, in 1999, and the M.Sc. degree from the National University of Defense Technology, Changsha, China, in 2006.

He has been the Executive Deputy Director with the State Key Laboratory of Space-Ground Integrated Information Technology (SGIIT), Beijing, China, since 2017. His research interests include image processing, multisensor information fusion, and data mining.

Wayne Luk (Fellow, IEEE) received the doctorate degree in engineering and computing science from the University of Oxford in 1988.

He founded and leads the Computer Systems Section and the Custom Computing Group, Department of Computing, Imperial College London, London, U.K. He was a Visiting Professor at Stanford University, Stanford, CA, USA, and Queens University Belfast, Belfast, U.K. He is currently a Professor of computer engineering at Imperial College London. He has been an author or editor of six books and four special journal issues.

Dr. Luk is a member of the Program Committee of many international conferences, such as the IEEE International Symposium on Field Programmable Custom Computing Machines, the International Conference on Field-Programmable Logic and Application (FPL), and the International Conference on Field-Programmable Technology (FPT). He is a fellow of the Royal Academy of Engineering and the British Computer Society Ltd. He had 15 articles that received awards from various conferences, such as the IEEE International Conference on Application-specific Systems, Architectures and Processors, FPL, FPT, the International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation, the X Southern Programmable Logic Conference, and the European Regional Science Association. He also received the Research Excellence Award from Imperial College London in 2006. He was the founding Editor-in-Chief of the *ACM Transactions on Reconfigurable Technology and Systems*.