An FPGA Overlay for CNN Inference with Fine-grained Flexible Parallelism

ZIAUL CHOUDHURY, SHASHWAT SHRIVASTAVA, LAVANYA RAMAPANTULU, and SURESH PURINI, International Institute of Information Technology, Hyderabad, India

Increasingly, pre-trained convolutional neural networks (CNNs) are being deployed for inference in various computer vision applications, both on the server-side in the data centers and at the edge. CNN inference is a very compute-intensive task. It is a challenge to meet performance metrics such as latency and throughput while optimizing power. Special-purpose ASICs and FPGAs are suitable candidates to meet these power and performance budgets simultaneously. Rapidly evolving CNN architectures involve novel convolution operations such as point convolutions, depth separable convolutions, and so on. This leads to substantial variation in the computational structure across CNNs and layers within a CNN. Because of this, FPGA reconfigurability provides an attractive tradeoff compared to ASICs. FPGA-based hardware designers address the structural variability issue by generating a network-specific accelerator for a single network or a class of networks. However, homogeneous accelerators are network agnostic and often sacrifice throughput and FPGA LUTs for flexibility.

In this article, we propose an FPGA overlay for efficient processing of CNNs that can be scaled based on the available compute and memory resources of the FPGA. The overlay is configured on the fly through control words sent by the host on a per-layer basis. Unlike current overlays, our architecture exploits all forms of parallelism inside a convolution operation. A constraint system is employed at the host end to find out the per-layer configuration of the overlay that uses all forms of parallelism in the processing of the layer, resulting in the highest throughput for that layer.

We studied the effectiveness of our overlay by using it to process AlexNet, VGG16, YOLO, MobileNet, and ResNet-50 CNNs targeting a Virtex7 and a bigger Ultrascale+VU9P FPGAs. The chosen CNNs have a mix of different types of convolution layers and filter sizes, presenting a good variation in model size and structure. Our accelerator reported a maximum throughput of 1,200 GOps/second on the Virtex7, an improvement of 1.2× to 5× over the recent designs. Also, the reported performance density, measured in giga operations per second per KLUT, is 1.3× to 4× improvement over existing works. Similar speed-up and performance density is also observed for the Ultrascale+VU9P FPGA.

CCS Concepts: • **Computer systems organization → Reconfigurable computing;**

Additional Key Words and Phrases: FPGAs, convolutional neural networks, accelerators

ACM Reference format:
Ziaul Choudhury, Shashwat Shrivastava, Lavanya Ramapantulu, and Suresh Purini. 2022. An FPGA Overlay for CNN Inference with Fine-grained Flexible Parallelism. *ACM Trans. Archit. Code Optim.* 19, 3, Article 34 (May 2022), 26 pages.
https://doi.org/10.1145/3519598

Authors' address: Z. Choudhury, S. Shrivastava, L. Ramapantulu, and S. Purini, International Institute of Information Technology, Hyderabad, Gachibowli, Hyderabad, Telangana, India, 500032; emails: {ziaul.c, shashwat.shrivastava}@research.iiit.ac.in, lavanya.r@gmail.com, suresh.purini@iiit.ac.in.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.

1544-3566/2022/05-ART34 $15.00
https://doi.org/10.1145/3519598

1 INTRODUCTION

Since the early 2010s, **Convolutional Neural Networks (CNNs)** have found applications in many computer vision tasks such as image classification and segmentation, motion analysis, three-dimensional (3D) scene reconstruction, and so on. Often CNNs are used for automatic feature extraction followed by the application of conventional machine learning models. A CNN consists of multiple convolutional, max-pooling, and thresholding layers. The difference between one CNN and another is their architecture, which entails the number and choice of layers and how they are interconnected. Even the simplest CNN has a high model complexity, which means that the number of parameters to learn is large, hence the associated training time. However, once the model parameters are learned, the model can be deployed to perform inference on real data. A learned model is often deployed in memory and power-constrained devices, for example, in a mobile robot for object detection. Efficient implementations of CNNs on multi-core CPUs and GPUs have been reported in the literature. For example, the TITAN GPU from Nvidia is tailor-made for processing of CNNs. ASICs [6, 19] provide significant gains in performance and power efficiency for CNNs but they may not cope with the ever-evolving CNN models due to the long design cycles and high engineering costs. FPGAs are highly configurable and provide a good tradeoff between the high-cost ASICs and low-performance general-purpose processors for specialized applications such as CNNs. Therefore, they prove to be an attractive solution for accelerating CNN inference.

Many works have proposed various FPGA-based CNN accelerators that outperform CPUs and GPUs on the performance-per-watt metric. These architectures can be broadly categorized into two classes. In the first class, for a given CNN, a custom accelerator is synthesized using one of the available parametric templates [14]. This requires expensive design synthesis and FPGA reflashing. In many edge and deeply embedded applications, this may not be feasible. Further, these days cloud companies are offering Machine Learning as a service. These services are supported by a large cluster of custom accelerators at the backend. The CNN-specific hardware accelerators severely constrains the scheduling of machine learning workloads and results in hardware resource under utilization.

In contrast to the CNN-specific architectures, the second class consists of *overlay architecture* [51], which is synthesized and flashed on the FPGA once, but is flexible enough to process a broad class of CNNs through soft reconfiguration. This addresses the aforementioned problems due to custom CNN accelerators. There are few design approaches for overlay architectures. For example, we can have an overlay architecture that resembles a processor controlled through an instruction set [35]. The same architecture can process all the layers of a network, and also different networks, without the need for re-synthesis. However, such a *homogeneous* approach to process different CNN layers with varying input and output feature map shapes, kernel sizes, and so on, result in sub-optimal utilization of the available FPGA resources such as DSPs and/or LUTs. The work in [34] simulated various shapes of a systolic array to show that the throughput gain is sensitive to the shape of the array for various workloads.

To solve this problem of homogeneous overlay accelerators, a heterogeneous design methodology is proposed in several previous works [46]. A heterogeneous overlay accelerator contains multiple homogeneous units. Each of these is optimized specifically for a set of workload characteristics, for example, input data shape, kernel size, and so on. A heterogeneous design mainly aims to optimize throughput by concurrently processing multiple images over the different accelerator units in a pipelined fashion. While this approach increases the throughput, the latency will be longer. Also, as the variations in network layer architectures grow, so does the complexity of these heterogeneous architectures.

The primary contribution of this work is an overlay architecture that can process a variety of CNN architectures with varying computational structures, kernel sizes, and strides. The overlay

parameters are chosen based on the available FPGA memory and compute capacity. Unlike other works that schedule instructions on the overlays, we orchestrate the computations associated with a CNN layer by configuring the overlay at runtime. Our overlay exploits all forms of parallelism present within a CNN layer. Further, the configuration parameters include parallelism factors such as surface and filter parallelism. A constraint system determines these factors to achieve optimal performance. We keep the design complexity of our overlay low, in terms of FPGA LUT consumption, without sacrificing its flexibility. Our overlay's host interface is comparatively simple, wherein the host sends a few control words to configure the accelerator, followed by the required input data. We demonstrate the effectiveness of our overlay by processing the well-known AlexNet and VGG16 CNNs. Our overlay also supports the processing of three relatively new CNNs, namely YOLO, MobileNet, and ResNet. Our hardware is synthesized for a mid-sized Virtex7 and a bigger Ultrascale+VU9P FPGA to illustrate the range of FPGAs our overlay can target effectively.

The rest of this article is organized as follows. In Section 2 and Section 3, we present related work and the necessary background on CNNs; in Section 4, we discuss how we schedule the computations of a CNN layer over our accelerator; in Section 5, we present our overlay architecture; in Section 7 we present the experimental results, and, finally, we conclude in Section 8.

2 RELATED WORK

The hardware generated by the existing CNN-to-FPGA frameworks can be categorized as either a streaming architecture or a single computation engine-based architecture [43]. A streaming architecture typically consists of dedicated hardware modules for each layer of the CNN, connected in a pipeline. All the layers are processed simultaneously by streaming data across the pipeline. Fpga-ConvNet [42] is designed on this principle. It supports multi-bit-stream design via complete FPGA reconfiguration, where different hardware architectures, matching the layer workloads, are used to process different layers of the CNN. DeepBurning [45] used a library-based approach. Based on the layer functionality, hardware building blocks are instantiated from a repository and interconnected to form the network. Each block is configured using fixed tiling parameters, calculated from a heuristic search, and is time-shared across the network layers. Haddoc [2] generates its architecture by modeling the target CNN as a dataflow graph of actors and directly mapping each actor to a dedicated compute unit. AutoCodeGen [23] includes parameterized hardware blocks for each CNN layer, instantiated using a high-level analytical performance and resource model, with the convolution blocks executing in a fully unrolled manner. The streaming design principles favor customization over flexibility, where a single accelerator gets tightly coupled with a specific CNN. Also, it becomes hard to map all the CNN layers to resource-constrained FPGAs, which is when the CNN is processed in a time-multiplexed fashion over a generic accelerator architecture leading us to the idea of single-engine architectures. Single engine architectures comprise a single computation engine that executes the CNN layers sequentially. The accelerator processes each layer at its maximum throughput. This design is a derivative of the well-studied systolic array structures [28]. AngelEye [32] comprises an array of homogeneous processing elements. Each contains a bank of convolvers, a summing tree, and a pooling logic that are instantiated using a throughput maximization heuristic, which uses a set of loop unroll factors. DnnWeaver [35] contains parametric hardware templates arranged in a similar array of PEs. The configuration of each PE is found through a search-based heuristic, which is later used to synthesize the accelerator. Once synthesized, this configuration remains the same throughout the processing of the network. Caffeine [52] consists of a systolic array of PEs that perform multiplication operations configured using a roofline model on the hardware design space. Snowflake [10] employs a hierarchical hardware structure that is designed to be controlled by software, with complex control logic and is CNN agnostic depending only on the available FPGA resources. Despite the flexibility gains in

this design principle, inefficiencies are introduced due to control mechanisms that resemble those of an instruction-based processor [16]. Moreover, the uniform unroll factors applied to the entire processing array can reduce the performance of CNNs with varying workload characteristics.

In another classification, the state-of-the-art CNN accelerator designs can be divided into two categories: uniform/homogeneous and heterogeneous. The homogeneous design methodology uses a uniform architecture to process all layers of a CNN model in order. For a single layer's inference, a homogeneous design first divides an input feature map into tiles. Then it repeatedly loads the tiles one after another from off-chip memory to on-chip memory and then processes the tiles in sequence by a jointly optimized accelerator design for all convolutional layers. Such designs have been described in [3, 8, 9, 27, 54]. However, different layers in a CNN model have different input data shapes. As a result, the same tiling factors in a homogeneous design may cause dynamic resource inefficiency for some layers. To solve this problem, a heterogeneous design methodology is proposed in several previous works [21, 36, 36, 40, 53]. A heterogeneous design incorporates multiple accelerators on a single FPGA. Each of them is optimized specifically for one or a set of layers. These architectures concurrently process multiple input images, by pipelining them on the different accelerators. The work in [47], supported pipelined execution of different tiles from a single input image on multiple heterogeneous accelerators. As the range of supported networks grows so does the complexity of managing the different accelerators on the FPGA. Overlay architectures[1, 5, 25, 38], combine ideas from both heterogeneous and homogeneous designs. They operate as a single uniform architecture with the flexibility to adjust to different tile sizes during runtime. The work in [50] proposed an FPGA overlay with software-like programmability for CNN end users. The overlay operate via an Instruction Set Architecture with complicated functions with variable runtimes but a uniform length. The granularity of instruction is optimized to provide good performance and sufficient flexibility. Overlay designs tend to consume more FPGA logic resources and often fail to exploit all forms of parallelism within CNN layer. In this work, the proposed overlay architecture operates with fewer FPGA LUTs, which can be attributed to a simple **Processing Engine (PE)** structure and exploits all forms of parallelism within a CNN layer.

3 BACKGROUND ON CNNS

This section provides a brief introduction to the structure of CNNs. The anterior part of a CNN consists of a series of convolutional and pooling layers, whereas the posterior part can contain zero, one, or multiple **Fully Connected (FC)** layers.

At each convolutional layer of a CNN, an input volume[1] V of dimensions (IL, IL, ID) is convolved with a set of F filters $\{F_i \mid 1 \leq i \leq F\}$, each of dimensions (K, K, ID), to generate an output volume V' of dimensions (OL, OL, OD). If S is the stride of the filter application, then $OL = \frac{IL-K+1}{S}$. The depth slices of the input volume are called input feature maps and are denoted by $IFMAP_i, 1 \leq i \leq ID$. The dimensions of an input feature map $IFMAP_i$ are $IL \times IL$. Similarly, the depth slices of the output volume are called output feature maps and are denoted by $OFMAP_i, 1 \leq i \leq OD$. Note that $OD = F$. The dimensions of an output feature map $OFMAP_i$ are $OL \times OL$.

An output feature map $OFMAP_i$ is rendered by applying a 2D convolution of filter F_i with the input volume V. We call this a 2D convolution, since the filter moves only in the length and breadth directions but not along the depth. Each three-dimensional filter F_i is an array of two-dimensional kernels $F_i^j, 1 \leq j \leq ID$. Then the surface convolution between an input feature map $IFMAP_j$ and a filter kernel F_i^j is defined as

$$OFMAP_i^j = IFMAP_j \circledast_s F_i^j. \tag{1}$$

[1]In this article, input volume refers to the input of the first CNN layer and subsequent intermediary layers, too.

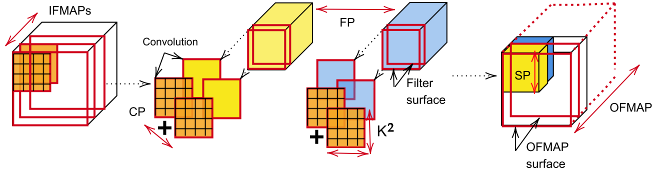


Fig. 1. An schematic overview of the types of parallelism exploited in our overlay.

Given that, the 2D convolution of the input volume with a filter is algebraically equivalent to the summation of the surface convolutions between their corresponding depth slices. Thus we have the following:

$$OFMAP_i = \sum_{j=1}^{ID} IFMAP_j \circledast_s F_i^j = \sum_{j=1}^{ID} OFMAP_i^j. \quad (2)$$

An activation function such as Rectified Linear Unit is applied on each pixel of the output volume before it is fed as input to the next layer. Usually, a pooling layer follows a convolution layer. In a pooling layer, `max` or `average` filters are applied on each surface of the input volume. So, when a 2×2 max-pooling filter with a stride 2 is applied on an input volume, then the ouput volume dimensions are $OL = IL/2$ and $OD = ID$. Thus the pooling layers help in reducing the surface dimensions of an input volume. Notice that there is no change in the depth dimension.

The FC layers resemble the conventional neural networks in which every neuron from a layer is connected with every other neuron from the previous layer. Due to this, the number of weights per layer will be huge. So as to contain them, FC layers start after the input volume dimensions are substantially reduced by the preceding convolutional and pooling layers.

Apart from the canonical full-depth convolution described above, there are other novel convolutions such as point and depth-separable convolutions like in YOLO and MobileNet. In point convolutions, the filter dimensions are $(1, 1, ID)$, which leaves the output volume's surface dimensions the same as the input volume. In a depth-separable convolution layer, the depth of each filter is only 1, and hence the output feature map $OFMAP_i$ is obtained by doing a mere surface convolution between the input feature map $IFMAP_i$ and filter f_i, i.e., $OFMAP_i = IFMAP_i \circledast_s f_i$.

3.1 Parallelism and Data-Reuse

From the Equation (2), we can infer the following four kinds of available parallelism while rendering an output volume, see Figure 1.

(1) **Filter Parallelism (FP):** Each OFMAP can be calculated in parallel. This is because an OFMAP depends only on the input volume and corresponding filter bank. Let the parameter FP denote the number of OFMAPs generated in parallel.
(2) **Surface Parallelism (SP):** While applying a surface convolution, it is possible to compute each value in the OFMAP in parallel, and there are OL^2 such values. Let the parameter SP denote the number of surface convolutions computed in parallel.
(3) **Channel Parallelism (CP):** From Equation (2), we can see that an OFMAP is obtained by the summing of the ID surface convolutions. Each of these surface convolutions can be computed in parallel. Let the parameter CP denote the number of surface convolutions computed in parallel across the input volume's ID channels.
(4) **Kernel Parallelism (K^2):** A single value of an OFMAP is created by applying a filter with a kernel of size K over an IFMAP. A total of K^2 multiply-and-accumulate operations are required to compute this value and all of them can be executed in parallel.

Data reuse of an input data point refers to the number of computations it contributes to in the overall processing of a layer. Each filter weight, in any filter, is used to generate OL^2 pixels in an OFMAP, and hence the corresponding reuse factor is OL^2. While rendering an OFMAP, each value in an IFMAP is used $(K/S)^2$ times, where S is the stride. There are in total F OFMAPs. Therefore the overall reuse factor for a single IFMAP value is $F \times (\frac{K}{S})^2$.

Proposed Approach: It is possible to exploit all the aforementioned kinds of parallelism by suitably orchestrating computations on our overlay. The type of parallelism exploited has a direct impact on the data reuse. We usually exploit filter, surface, and kernel parallelism to saturate the DSP utilization. Utilizing channel parallelism puts stress on the available memory bandwidth. The canonical approach we adopt is to fetch a filter only once from off-chip memory and completely reuse it before discarding it. Those fetches are equivalent to compulsory misses in the cache parlance of CPUs. This means we achieve maximal data reuse with respect to filter coefficients. An input volume pixel, once fetched, is completely reused while rendering FP OFMAPs and then discarded, to be re-fetched again while rendering the next batch of FP OFMAPS. Thus a value from an IFMAP is fetched F/FP times from off-chip memory leading to a reuse of $FP \times (K/S)^2$. In the next section, we present how we orchestrate the layer computations over our overlay to exploit different parallelism types while maximizing data reuse.

4 SCHEDULING COMPUTATIONS ON OUR OVERLAY

Multiply-and-accumulate (MAC) operations dominate the computations in a CNN. Let M^{mac} be the total number of MAC operations from all the CNN layers put together. These operations are executed on the physical FPGA DSP blocks. If there are N^{dsp} blocks on the FPGA, then the peak achievable compute throughput, measured in MACs/cycle, is N^{dsp}. If hardware is synthesized at a certain clock speed, then the effective compute throughput achievable depends on two factors: the off-chip memory bandwidth and the number of MAC operations performed for every input fetched from the off-chip memory (data reuse factor). A single inference pass of a CNN requires at least $\frac{M^{mac}}{N^{dsp}}$ clock cycles. To approach this theoretical lower bound, and in general, to minimize the end-to-end latency of a design, we have to arrive at an overlay architecture and a suitable way to schedule the layer computations on the overlay such that the data reuse is maximized.

In the rest of this section, we first describe the execution model of our overlay architecture. Then we show how we batch the computations within a layer using a system of constraints and schedule them on the overlay while maximizing parallelism and data reuse to achieve peak DSP utilization.

4.1 Execution Model

We use a single-engine accelerator design approach for our overlay. A single processing pipeline computes all the layers of the network. The output volume rendered while processing layer i is streamed out to external memory and later loaded as input when computing the next layer $i + 1$. With respect to the Equation (2), rendering the output volume is equivalent to computing the output feature maps $OFMAP_i$ for $1 \leq i \leq F$. These output feature maps are computed in batches iteratively. The number of OFMAPs computed in each batch, which is equal to FP, can vary and is determined by the scheduling algorithm described in Section 4.3.

We compute convolutions in a streaming fashion, wherein the input volume is streamed, and the necessary filter coefficients are pre-fetched and stored in registers. The pre-fetch of filter coefficients overlaps with computation to avoid stall cycles.

Each input feature map $IFMAP_j$ of the input volume is surface convolved with the corresponding depth slice F_i^j where filter F_i belongs to the current batch. This computation starts only after the

corresponding depth slices of all the current batch filters are available on the FPGA registers. Then each input feature map $IFMAP_j$ is streamed in a row major fashion to compute the corresponding surface convolution with respect to each filter. The computed surface convolution is accumulated in respective output buffers. The number of surface convolutions done in parallel depends on the parallelism factor SP. While the surface convolutions are being computed, the next depth slice's filter coefficients are streamed using a double buffering technique.

Once the minimum number of input pixels required for carrying out the filter convolutions have accumulated in the input buffers, surface convolutions can be computed at a steady throughput determined by FP and SP. At any point, the accelerator stores a set of rows from an input slice and multiple partially rendered output volume slices. The line buffering of the input ensures that only a working set of rows from the input slice is required to sustain a fixed number of surface convolutions every cycle.

Once a filter batch is done, i.e., all the filter depths have convolved with the corresponding input volume depths, the output slices stored in the buffers are drained out to external memory one at a time. This drainage of output corresponding to a filter batch is overlapped by executing parallel convolutions corresponding to the next batch of filters. To do this, a second set of buffers are used for accumulating the output of the next batch. Note that the input volume has to be streamed again at this point. This double buffering scheme ensures maximum compute to memory overlap during the processing of multiple filter batches,

$$\texttt{DSP constraint} \rightarrow \quad FP \times SP \times K^2 \leq N^{dsp}, \tag{3}$$

$$\texttt{DRAM bandwidth constraint} \rightarrow \quad SP \times CP \leq \frac{B_e}{S^2}, \tag{4}$$

$$\texttt{Number of BRAM blocks constraint} \rightarrow \quad 2 \times FP \times SP \leq N^{bram}, \tag{5}$$

$$\texttt{Compute/Memory overlap constraint} \rightarrow \quad \frac{OL^2}{SP} \geq \frac{FP \times K^2}{B_w}, \tag{6}$$

$$\texttt{Output flush constraint} \rightarrow \quad \frac{OL^2 \times ID}{SP} \geq \frac{FP \times OL^2}{B}. \tag{7}$$

4.2 Constraint System

The runtime configuration of our overlay is characterized by the type of operation (convolution, max pooling, etc.) and the parallelism mix with which it is executing that operation. We set constraints on the parallelism factors, such that the processing pipeline in our overlay operates with zero or minimum stalls. The constraint system is discussed in the rest of the section. At first, we recall the CNN layer and FPGA parameters used in the constraint system.

- **Layer Parameters:** At each layer of a CNN, an input volume of dimensions (IL, IL, ID) is convolved with a set of F filters, each of dimension (K, K, ID), to generate an output volume of dimensions (OL, OL, F).
- **FPGA Parameters:** Let the number of DSP and BRAM blocks on the FPGA be N^{dsp} and N^{bram}, respectively, and $2B$ bytes/cycle be the combined read and write memory bandwidth to external memory.

If $FP \times SP$ surface convolutions are computed per cycle and the filter dimensions are (K, K), then the number of MACs per cycle are $FP \times SP \times K^2$. The maximum number of MACs is bound by the number of available DSPs (N^{dsp}) on the FPGA yielding the DSP constraint (3). Our overlay is configured to process different types of convolutions. One such type are depthwise convolutions prevalent in sparse CNNs like MobileNet. These special convolutions are computed with extra DSP

resources. The DSPs used for computing the MACs in the depthwise computations are very few compared to the DSPs used during the normal convolutions. Because of the low number, these DSPs have not been taken into the DSP constraint (3).

If B_e is effective off-chip memory bandwidth, since surface convolutions are computed in a streaming fashion, then every surface convolution requires S^2 additional inputs, corresponding to an IFMAP, from off-chip memory, where S is the filter stride. Hence, to compute SP convolutions in parallel, $SP \times S^2$ inputs have to be fetched from off-chip memory. For parallel convolutions across the channel dimension, every cycle CP inputs corresponding to an CP different IFMAPs have to be fetched. Combining the above two factors, leads to the DRAM bandwidth constraint (4). Notice that the off-chip memory bandwidth has to be split between the surface and channel type of parallelism.

The number of available BRAM blocks on the FPGA impacts the parallelism factors FP and SP. To facilitate SP parallel convolutions for each of the FP output volume slices from the current batch, we store each output volume slice in SP BRAM blocks in an interleaved fashion. Thus, we need $FP \times SP$ BRAM blocks. Since we are doing double buffering to overlap computation and communication, we need BRAM blocks twice that number. This leads to the BRAM size/bandwidth constraint (5). The input buffer utilizes very few BRAM blocks compared to the output buffers. Empirically, we use a $16 \times L$ line buffer, where L is the row length of the largest IFMAP. The buffer just utilizes 16-20 BRAM blocks. Therefore, we keep the BRAM count from the input buffer out of the constraint system. The channel parallelism factor CP does not contribute toward BRAM usage, since all the CP convolutions reduce to a single value. We resort to channel parallelism only for pointwise convolutions where $K = 1$.

We view a BRAM block as an array of fixed-point numbers of type (IB, FB) where IB and FB denote integral and fractional bit widths. When we say that a BRAM block's size is A, it can hold A pixels of a given fixed-point type. For many CNNs, in the first few layers, the output dimension OL^2 could be much greater than $A \times SP$ limiting the available filter parallelism. This limitation is relaxed by tiling the input volume across its surface into vertical cubes of dimensions $IL \times OL' \times ID$.

As we move from one input channel to another, we have to pre-fetch the corresponding filter weights for the respective channel. During these cycles, there is no DSP utilization. Alternatively, we can use part of the bandwidth B to overlap fetching of filters from the next channel while computing convolutions on the current channel, i.e., $B_e = B - B_w$. Here B_w is the part of the input bandwidth that is used for fetching of filter weights, and it varies across layers. In our accelerator design, we use this approach, which leads to a small decrease in the available bandwidth B for the input volume pixels but, at the same time, prevents the DSPs from stalling for filter data. For a given SP, $\frac{OL^2}{SP}$ is the number of cycles it takes to render FP partial output surfaces, for a filter batch, in parallel. To hide filter pre-fetch latency, $FP \times K^2$ weights corresponding to the next batch of a filter from the respective channel should also be fetched within this interval using a part of the DRAM bandwidth. This compute/memory overlap constraint to hide the communication latency is given in Equation (6).

As the current batch of OFMAPs is computed, the previous batch of OFMAPs is written to off-chip DRAM. The number of cycles required for computing a batch of FP OFMAPs is given by $(FP \times OL^2 \times ID)/(FP \times SP)$, which is equal to $(OL^2 \times ID)/SP$. The size of the output volume generated in each step is $FP \times OL^2$. Assuming that the write bandwidth is equal to the read bandwidth of the DRAM, the number of clock cycles required to spill the entire output buffer is $(FP \times OL^2)/B$. To minimize the interference between computing and communication and reduce the potential stall cycles as the output buffer is not yet completely spilled, the output flush constraint (7) has to hold good and as tight as possible. If this constraint gets violated, then the correctness may

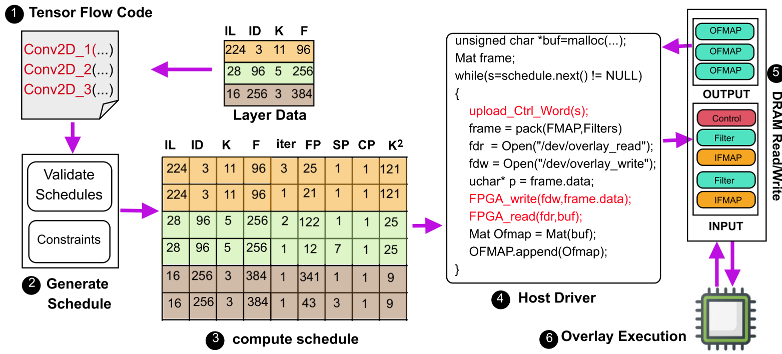


Fig. 2. The workflow for processing a CNN using our overlay. The tensor flow specification is lowered to a set of accelerator calls using a high-level language (Python). Communication with the overlay is done through a driver API written in C++. Computations of a layer are broken into multiple batches/iterations, in the form of a compute schedule, and executed on the overlay. The hardware is configured on the fly toward processing a compute batch using control words. A compute schedule for three layers of AlexNet is shown in the figure.

not be affected, but there will be an imbalance between computation and communication cycles, due to which stalls will be incurred. In other words, there will be cycles where the DSP blocks are idle.

The constraint system is part of the CPU host logic written in a high level language like python. These constraints are used by a scheduler, that generates per layer accelerator configurations for processing the current layer.

4.3 Layer Scheduling

Figure 2 shows the overall workflow of executing a CNN on the overlay. The host side code for CNN inference can be written using any of the available libraries such as TensorFlow, PyTorch, and so on. In this work, we used TensorFlow-Python to test our overlay. The host code communicates with the overlay running on the FPGA through a C++-based driver code.

The host code pre-processes each CNN layer to identify the feasible parameters (FP, SP) satisfying the set of Constraints (3) to (7). Note that if the overlay is configured to compute FP output feature maps in parallel with a surface parallelism factor SP, then $FP \times SP$ convolutions will be computed per cycle. Hence, to maximize the DSP utilization, among the feasible parameters, those parameters (FP, SP) that maximize $FP \times SP$ are filtered. From among them, a parametric pair with maximal FP is chosen as it minimizes the number of times the FPGA is invoked from the host side. The F output feature maps are computed in $\lceil \frac{F}{FP} \rceil$ batches with each batch computing FP output feature maps in parallel. The last batch may not be full and hence to increase the DSP utilization we increase the surface parallelism factor SP subject to the satisfaction of the constraints. The input volume will be re-streamed from DRAM while computing a batch of output feature maps. Our strategy not only maximizes DSP utilization but also maximizes data reuse at the same time.

The host configures the overlay to compute a batch of output feature maps by sending a set of suitable control words (refer Figure 2). The overlay uses the control words for hardware reconfiguration. The overlay acknowledges the completion of the configuration phase to the host. At this point, the host packs the input feature maps and filter weights into a data stream. This data stream is communicated to the overlay through the host driver. The processed OFMAPs from overlay are transferred back to the host code via the driver. The OFMAPs are stored in the host memory to be used as input to the subsequent layers.

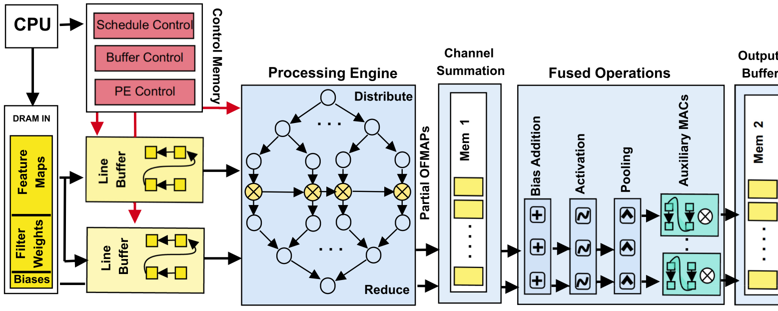


Fig. 3. The block diagram of our overlay. The control words dynamically configure the memory and the compute modules. These configurations determine the runtime behavior of the overlay. The control words for each module are stored in a register-based memory.

Overall, the host processes a CNN layer by layer. Each layer is further divided into batches. This batching is done so as to maximize DSP utilization, data reuse and reduce the number of FPGA invocations. The overlay is soft-reconfigured using control words to process a batch with a certain filter and surface parallelism parameters.

5 OVERLAY MICRO-ARCHITECTURE

This section describes the micro-architecture of our overlay as shown in Figure 3. The host operates the overlay through control words. A register-based control memory contains control words that define the runtime configuration of the overlay. A dedicated set of control words configure the overlay's memory and processing engine modules depending on the compute schedule. Loading up of the control memory marks the beginning of a new computation batch. Input to the overlay is made up of filter weights and feature map values. The host streams this input data on a PCIe-DRAM interface using the C++ driver. IFMAPs are buffered inside the overlay using line buffers. Filter weights are stored using registers. In a steady-state, the line buffer generates *SP* surface windows over an IFMAP. These windows are convolved with *FP* pre-fetched filter windows. The resulting $FP \times SP$ convolutions are computed in parallel over the overlay's PE. The output data from the PE are packed into a $SP \times FP$ sized vector.

The partial OFMAPs rendered by the PE are accumulated in a memory block (Mem 1 in Figure 3), built from the FPGA BRAMs. The channel summation module uses this memory to sum up, all the partial OFMAPs corresponding to the input volume channels to produce the final output feature map. The final OFMAP is passed through a set of stages representing the operations that typically follow a convolution. The fusing of these operations greatly cuts down the need to re-fetch the processed output volume. Depending on the network specification, these stages can be selectively disabled, in which case they pass the input to the next stage without any modification. The final output is stored in a memory block (Mem 2 in Figure 3) from where it is written to the FPGA DRAM to be read by the CPU host driver.

In the following sections, we discuss the line buffer and PE modules in our overlay. We focus on the reconfigurability aspect of each of these modules adding to the reconfigurablity of the overall design.

5.1 Programmable Line Buffer

Exploiting kernel level parallelism is essential toward increasing the data reuse per input fetched from the external memory. It is non-trivial to exploit full kernel level parallelism, amounting from

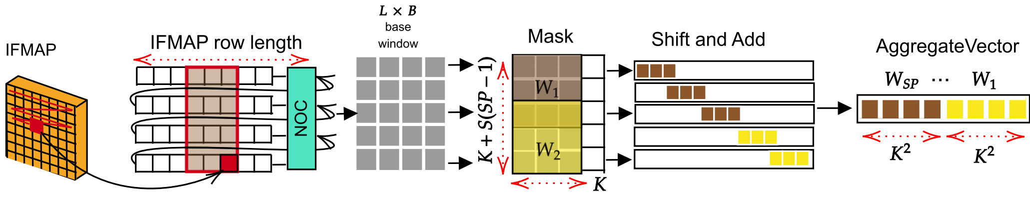


Fig. 4. The line buffer pipeline inside our overlay that generates stencil windows over a streaming input feature map.

a single K^2 sized window, in overlay architectures, due to the variation in kernel sizes across different CNN layers.

Previous approaches provide line buffers that are either limited to a single window per cycle [7, 13], or support a specific window and/or image size. The work in [39], proposed a scalable line buffer design that can generate multiple windows of arbitrary size. The design was purely register based and had no support for windows moving with a stride greater than 1. We propose a novel programmable line buffer architecture for our accelerator, that can generate multiple variable sized windows over a streaming input source moving with stride values greater than 1. Our novel hybrid design optimally uses both BRAM and registers to improve throughput as shown in Figure 4. The generation of square windows by the line buffer facilitate full exploitation of kernel level parallelism.

5.1.1 Buffer Architecture. Every cycle, the line buffer generates SP convolutional windows from the IFMAP each of dimension $K \times K$. The input feature map is streamed to the line buffer in a row-major fashion. The SP windows are generated along the vertical direction over the IFMAP. We extract the surface windows in only one direction as it simplifies the flexible line buffer design. This also relaxes constraint (4) from section 4.2. In a steady-state, our line buffer reads in $S \times SP$ new inputs along $S \times SP$ rows of the IFMAP. The rows are buffered using an array of FIFOs. We use a hybrid FIFO implementation. In each FIFO, the first B values are stored in registers and rest of the row is stored in a BRAM block. Each FIFO is read from and written to independently. The connections between the FIFOs is programmable through a flexible interconnection network. With L FIFOs, $l_0 \cdots l_{L-1}$, the line buffer can hold a maximum of L IFMAP rows at any time.

The line buffer needs to buffer a total of Z rows and K columns ($Z = K + S \times (SP - 1)$) to generate SP, vertical surface windows, moving with stride S. There is an overlap of $K - S$ rows between two vertically adjacent windows. The IFMAP is loaded in two phases. In the initial loading phase, the rows are streamed sequentially until the first Z FIFOs of the line buffer are filled. After this, the lateral loading phase starts, wherein the remaining rows are streamed. In this phase, every cycle, $S \times SP$ values (one value from a different but adjacent row) are fed to the FIFOs l_{K-S} though l_Z. The FIFOs support parallel read/writes. With every en-queue operation, the FIFO is also de-queued. As the new data are en-queued to FIFO L_i, the old data from L_i are en-queued to FIFO $L_{i-(S \times SP)}$. This pattern ensures that the overlapping rows between the vertically adjacent convolutional windows are preserved inside the line buffer. These connections vary with the value of S and SP, and are enabled by setting the control words configuring the programmable all-to-all network between the FIFOs. For example, in Figure 5(a), the new data are en-queued to FIFOs 2 and 3. The old data from FIFO 2 and 3 are en-queued to FIFO 0 and 1, respectively. This replaces rows 0 and 1 in the line buffer with rows 2 and 3.

In the lateral loading phase, with data in Z FIFOs, the line buffer outputs a $Z \times B$ window every cycle. This is done by shifting out a column and shifting in a new column from across the Z FIFOs. A total of B values can be read per FIFO because of the register storage. From the $Z \times B$ base window,

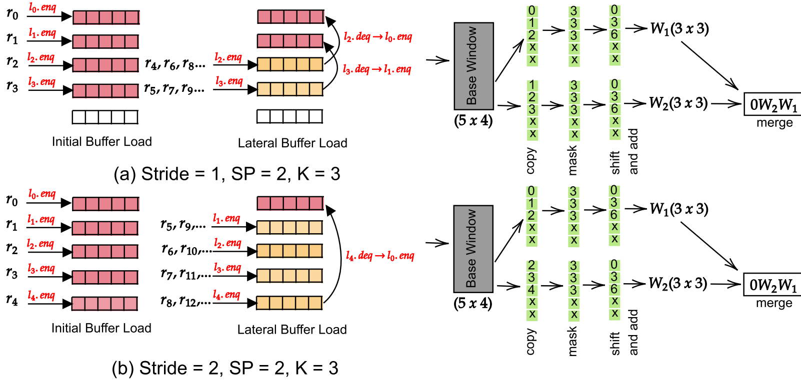


Fig. 5. The line buffer is operating in two configurations generating 3 × 3 windows with different stride values. The connection of the input to the FIFOs and connections between the FIFOs change with the stride factor.

the sub-windows required to perform the convolutions are extracted. A set of copy registers, selects the rows required per window. For example, in Figure 5(a), since two, 3×3 convolution with stride 1 needs to be executed, the first two copy registers are used. The first one selects rows 0, 1, and 2 from the base window and the second selected rows 2, 3, and 4. The selected rows are copied into a vector register that is then passed through a dedicated masking register, that zeros out the extra columns. The masked out rows within the vector register are appropriately shifted and later added to construct a $1 \times K^2$-dimensional vector W_l. We call W_l a *window vector* to mean that it is logically a window but is physically stored in a vector register. All the input vectors required to perform SP convolutions are aggregated/merged into a single vector $W = [W_{SP}, \ldots, W_1]$ with $SP \times K^2$ elements in it. The vector thus constructed is passed to the PE.

The copy/mask/shift values and the FIFO inter-connections represent the dynamic configuration of the line buffer. Knowing the kernel size and the SP value for a given computation batch, the host logic generates these values. It passes them to the overlay as control words during the configuration phase.

5.2 PE Architecture

The PE forms the core of our overlay, where the convolutions are processed. The PE receives SP windows from the line buffer that are convolved in parallel with FP pre-fetched filter windows of the same dimension. Depending on the computation batch, the PE can be configured to process any mix of FP and SP values. Recall, that the windows emanating out of the line buffer are aggregated into a vector $W = [W_{SP}, \ldots, W_1]$ and forwarded as input to the PE.

The PE is a fully pipelined structure. It consists of a distribution tree, a multiplier array, and a reduction tree. The vector W first enters the distribution tree, which creates FP replicas of the same and distributes them over the multiplier array. The size of the multiplier array characterizes the size of the PE. The PE design is entirely scalable depending on the DSP resources of the underlying FPGA. A multiply unit multiplies an IFMAP value with a filter weight. Unlike the IFMAP values, the filter weights are fed directly to the multipliers, bypassing the distribution tree. Using shift logic, the PE loads filter weights from external memory into a register array. Each register in this array is connected to an individual multiplier. The PE starts operating after loading the first FP filters. As the PE convolves the IFMAPs with the current filter set, it pre-fetches the next set in the same fashion, ensuring a compute-to-memory overlap. With the arrival of the IFMAP values, the

multiplications are performed. The products are passed down to an additive binary reduction tree, where they are combined to produce the convolution output.

A binary reduction tree can only reduce window vectors whose size is a power of 2. But such window sizes rarely occur in CNNs. The problem can be addressed by zero padding the window vectors. This approach leads to sub-optimal utilization of PE compute resources. In our case, we achieve optimal PE throughput by orchestrating the distribution of the products (multiplier outputs), corresponding to the windows vectors, over a complete binary tree in a clever fashion without any padding, thus maximizing the compute utilization. We explain this data orchestration next.

Data Orchestration: The aggregate vector W is replicated FP times to facilitate convolution with FP filters in parallel. This is logically equivalent to replicating each window vector W_i, $1 \leq i \leq SP$, into FP copies, W_i^j, $1 \leq j \leq FP$. Thus, overall, a total of $FP \times SP$ window vectors are created. Each window vector is further broken into smaller vectors whose size is a power of 2. For example, a 3×3 vector is broken into two smaller vectors of size 8 and 1 each. In general, a K^2 sized window is broken into at most w partitions, P_0 through P_w, where $w = \lfloor \log_2(K^2) \rfloor - 1$. The partition $P_{i,j}^m$ is related to the jth copy of the ith window vector and is of size m. This partition constitutes a window vector if the $\log m$th bit in the binary representation of the vector size is set. Note that there will be a total of $FP \times SP$ partitions of a given size. In the proposed orchestration strategy, the same-sized partitions from all the windows are packed adjacent to each other. The partition groups are arranged in the decreasing order of their sizes from left to right. The filter windows are pre-arranged by the host in the same data layout mentioned here and fed to the multipliers directly during runtime. When $SP > 1$, SP copies of a partition from the filter window is made using a Shift-and- Or logic and propagated internally between the multipliers. Once the copies are propagated, the next partition belonging to the same filter window is loaded from the host. A partition P^m of size m is mapped to m multipliers. The multiplication of the filter weights and IFMAP values corresponding to this partition creates a partial product vector of the same size. Recall that the multiplier array forms the base of a complete binary tree, used for reduction. This partial product vector is reduced to a single value using an appropriate subtree of the reduction tree. Using this computational layout, there is no necessity for zero padding while computing convolutions, thus optimal DSP utilization is achieved. For example, in a 3×3 convolution window, the partitions of size 8 and 1 are reduced by subtrees, that are complete binary trees, with 15 and 1 nodes, respectively. Since each multiplier is connected to a leaf node of the reduction tree, the subtree mapping is done automatically based on where the partition lies within the layout.

5.2.1 Distribution Tree. The above data orchestration is realized in hardware, inside the PE, with a distribution tree. The structure is organized as a complete binary tree. The number of leaves is greater than or equal to the number of multipliers. The IFMAP values inside the aggregate vector W, are routed through the distribution tree nodes to the multipliers through a sequence of shift operations. Every tree node has two shift units and is configured with a specific shift value. Each node performs a copy-and-shift operation wherein it makes two copies of the received vector, right shifts each copy by its corresponding shift amount, and forwards the modified vectors across the edges to both the children nodes. Note that a shift value of 0 on an edge results in copying the input vector without shift. This is the default configuration of the distribution tree. Overall, there can be a maximum of H vector transformations on a root to a leaf path. Here H is the height of the distribution tree. Finally, each multiplier reads the first entry of the leaf-node vectors as input.

We further explain the details of the distribution tree logic using a running example, see Figure 7(a), where $K = 3$, $SP = 3$, and $FP = 1$. The aggregate vector contains 27 IFMAP values distributed across three window vectors W_1, W_2, and W_3, each of size 9. Each window vector is broken into 2 partitions of size 8 and 1. The data orchestration rearranges the window vector

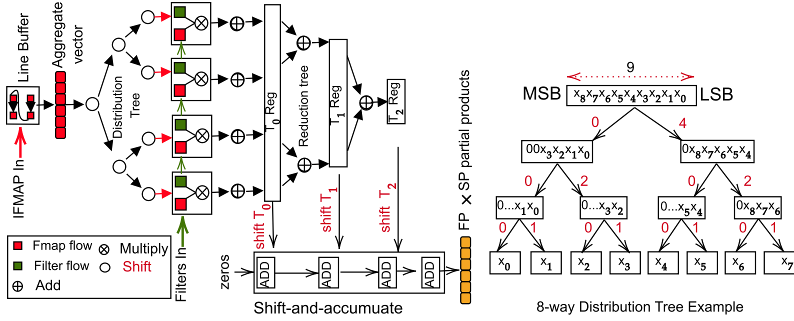


Fig. 6. The micro-architecture of the Processing engine. An example is shown on the right of how the distribution tree distributes a window vector over 8 multipliers.

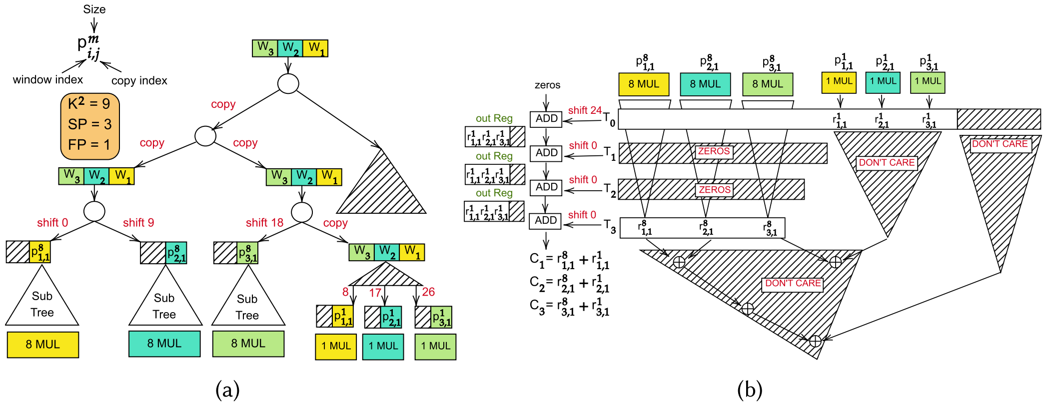


Fig. 7. (a) The distribution tree configuration for processing 3 convolutions with $SP = 3$, $K^2 = 9$, and $FP = 1$. (b) The reduction tree configuration for processing 3 convolutions with $SP = 3$, $K^2 = 9$, and $FP = 1$. The copy index is 1 throughout, since $FP = 1$.

partitions, $W = P^1_{3,1}P^8_{3,1}P^1_{2,1}P^8_{2,1}P^1_{1,1}P^8_{1,1}$, and puts them in the decreasing order of their sizes ($P^8_{1,1}$ occupies the least significant bits of W) resulting in a transformed aggregate vector W', where $W' = P^1_{3,1}P^1_{2,1}P^1_{1,1}P^8_{3,1}P^8_{2,1}P^8_{1,1}$. Each partition in W', from right to left, is mapped to a group of multipliers/DSP. For example, the partition $P^8_{1,1}$ is mapped to the first 8 multipliers and so on. This mapping/data orchestration is achieved by using sub trees from the distribution tree. A group of m multipliers is assigned a subtree, with its root located at level $H - \log m$ within the distribution tree of height H. The root-node of the subtree receives a vector with partition $P^m_{i,j}$ at the lowest significant position, from the tree nodes above. The subtree is configured to route the elements/values of the partition to the appropriate multipliers at the leaf level. Figure 6, shows a subtree distributing a 8 sized vector partition over 8 multipliers.

At every node, the input vector is copied across the left edge and right-shifted across the right edge. The shift amounts are adjusted such that desired value appears at index 0 of the vector in the leaf node. The aggregate vector W is routed to the parent node of the subtree through the default copy operation. It is right shifted, across the edge connecting the subtree, such that the partition mapped to the subtree appears at the lowest significant position of the resultant vector. In Figure 7(a), the partition $P^8_{2,1}$ is mapped to the second subtree from the left. This partition appears

after the partitions $P^8_{1,1}$ and $P^1_{1,1}$ inside W at the parent node of the subtree. Thus, W is right-shifted by $|P^8_{1,1}| + |P^1_{1,1}| = 9$ units so that $P^8_{2,1}$ appears at the lowermost eight positions of the resultant vector at the root of the subtree.

Setting up the shift values corresponding to the subtrees and their parent nodes completes the configuration of the distribution tree. The other nodes are left in the default copy propagation mode. Using the K, SP, and FP values, the host sets up the shift values using control words, thereby configuring the distribution tree for a new compute batch.

5.2.2 Reduction Tree. The partitions corresponding to a window are reduced to single values inside a binary reduction tree. The reduced value $r^m_{i,j}$ corresponding to a partition $P^m_{i,j}$ is generated at the $\log m$th level. For example in Figure 7(b), the computations from the 8 sized partitions $P^8_{1,1}$,$P^8_{2,1}$,$P^8_{3,1}$ are reduced to the single values $r^8_{1,1}$,$r^8_{2,1}$,$r^8_{3,1}$ at the third ($\log 8 = 3$) level of the reduction tree. All the reduced values corresponding to a window vector must be aggregated to produce the final convolution output. This cannot be done inside the reduction tree, since the reduced values lie at different levels, and no adder path exists between them. The reduced values from different levels are latched onto registers and passed through a shift and accumulate hardware that performs the final aggregation.

5.2.3 Shift and Accumulate. The per-partition reduced values at various levels of the reduction tree are latched on vector registers. The size of these vectors is set to the maximum number of reduced values produced at any given level. These vectors are pairwise added, using a sequence of shift-and-accumulate units, to create the final convolution outputs.

The shift-and-accumulate units are connected in a pipeline. A unit i adds the output of the previous unit with the vector T_i. Here T_i stores the reduced values from the ith level of the reduction tree. The location of the reduced values inside the vector depends on the tree level. For example, in Figure 7, the reduced values in the first level vector T_0 lie at positions 24, 25, and 26 respectively. These values need to be aligned before the pairwise addition operation with the vector T_3, containing the reduced values corresponding to the eight sized partitions at positions 0, 1, and 2, respectively. Consequently, T_0 is right-shifted by 24 units. The shift-and-accumulate sequence adds both T_0 and T_3 to produce the final convolution outputs. For vectors that do not contain any reduced values (T_1 and T_2 in the example) the corresponding shift-and-accumulate, performs a zero addition, thereby forwarding the input unmodified to the next unit. The adder units are connected in a pipeline through registers. In general, for the pairwise addition to happen correctly, with respect to two vectors T_i and T_j, $j > i$, T_i needs to be right-shifted by $p(i) - p(j)$ units. Here $p(i)$ and $p(j)$ are the starting location of the reduced values within T_i and T_j, respectively.

5.2.4 PE Configuration. The shift values of the distribution tree and the shift-and-accumulate stages represent the dynamic configuration of the PE. The host writes these values to control words dedicated to the PE inside the control memory of the overlay. Therefore, configuring the PE at runtime for processing a compute batch with a specific value of FP, SP, and K.

6 HANDLING SPECIAL CONVOLUTIONS

We optimize the processing of different types of convolutions by altering the basic processing flow of our accelerator.

6.1 Pointwise Convolution Layer

Pointwise convolutions occur in MobileNet, ResNet, and YOLO. Since the kernel size is 1, every convolution results in a single MAC operation. To achieve a high compute utilization for such layers, the FP/SP value has to be increased. Recall that the output vector generated by the PE has

$FP \times SP$ elements, and each value is written in parallel to the PE memory. Therefore, an increment in either FP or SP increases the compute utilization and the required memory bandwidth of the PE linearly. This is different for layers with $K > 1$, where a single unit increment in FP/SP results in quadratic (K^2) improvement in the compute utilization but increases the memory bandwidth requirement linearly. To achieve a similar utilization pattern, we exploit depth parallelism, CP, instead of surface parallelism, for pointwise layers. With every unit increase in FP, the compute utilization increases by a factor of CP and the bandwidth requirement increases by 1 (the values from CP channels adds up to a single output).

We exploit channel parallelism by feeding the PE with an input vector of size CP. The vector contains data from CP adjacent IFMAPs of the input volume. Every cycle, the multiplier array inside the PE produces $FP \times CP$ values. The CP values produced per convolution are added inside the merge network of the PE to generate only one value that needs to be written to the PE output memory. Therefore, the overlay increases the computations by a factor of CP with every increase in FP.

6.2 Depthwise Convolution Layer

Depthwise convolutions are found in sparse networks like MobileNet. In such a layer, each depth of the input volume is convolved with a separate filter.

In such a scenario, the FP value is upper bounded by 1, as this is the maximum number of filters that can be applied over a single surface, severely impacting the data reuse factor and bringing down the achievable throughput. Our overlay overcomes this limitation by executing parallel filter convolutions over different IFMAPs of the input volume. But the basic processing flow of our overlay is geared toward processing one IFMAP at a time. To achieve channel parallelism, we exploit the fact that, in general, a depthwise layer occurs after a regular convolution layer. Our overlay executes both the layers in a pipelined fashion.

In this flow, the overlay retains the output surfaces rendered after the execution of FP filters from the convolution layer, instead of flushing them to DRAM. The retained surfaces are passed through a set of auxiliary MAC units. Each such unit is made up of a small line buffer and a multiplier array. The filters from the next depthwise layer are applied over these surfaces, inside the auxiliary MAC units, whose output is later flushed to the DRAM. After this, the accelerator resumes the execution of the convolution layer with the leftover filter batches, followed by the depthwise layer, continuing the zigzag pattern of execution.

6.3 Elementwise Addition Layer

The Elementwise layer occurs in networks like ResNet, Inception, and so on. In such a layer, the outputs from two or more previous layers are combined using simple elementwise operations like addition. These layers are mostly memory bound, since computationally they are simple but consumes at-least twice the input compared to a normal convolutional layer. Also, in most settings, the participating convolution layers are similar in structure concerning their kernel sizes. This is true for ResNet, which has a branching factor of 2 and the elementwise addition layer receives input from two pointwise convolution layers, each at a different branch. To increase the overall throughput and reduce external memory traffic, both the input convolution layers are executed simultaneously over the PE. Their outputs are later added inside the reduction tree. The simultaneous processing is enabled in our overlay with an extra line buffer for streaming input to the other convolution branch. The parallelism factors, FP, SP, and CP, are equally distributed between the branches.

7 EXPERIMENTS

We evaluate our overlay by synthesizing it for two Xilinx FPGAs, Virtex7-690t, and Ultrascale+ VU9P. We use 16-bit fixed-point precision for the accelerator. We employ the synthesized hardware

Table 1. FPGA Resource Consumption of Our Accelerator on Both the FPGAs

	% KLUTs					Total KLUTs	Kilo FF	BRAM (36 Kb)	DSP
	Distribution Tree	Reduction Tee	Input Logic	Output Logic	Misc				
Virtex7-690t	29.5%	40.1%	5.2%	10.1%	14.8%	223/693	405/866	911/1470	3072/3600
Ultrascale+VU9P	30%	40.3%	4.8%	10.5%	14.2%	448/1182	1112/2364	1836/2210	4096/6840

The % utilization of the FPGA LUTs, out of the total LUTs consumed, across different hardware modules of our overlay is shown.

to process five networks namely, AlexNet [20], VGG16 [37], YOLO [33] MobileNet [18], and ResNet-50 [17]. We have reported throughput (in GOps per second) for two cases: with and without fully connected layers. When considering FC layers for overall throughput calculation, VGG-16 and AlexNet undergo a reduction in throughput (only these two networks have FC layers among other networks mentioned above). This happens because of the overlay's restriction to process one batch (B = 1).

Experimental set-up: Our overlay is synthesized at 166 MHz and operates with a data bit rate of 16 bytes/cycle in the read and write direction. The combined bit rate of our overlay is 32 bytes/cycle. Therefore the overlay operates at a bandwidth of 10.6 GB/s (5.3 GB/s in each direction). We run our overlay on two FPGAs, a Virtex7-690t and an Ultrascale+VU9P FPGA that vary with respect to the number of resources and the operational framework.

The Virtex7-690t FPGA is a standalone FPGA with 3600 DSP blocks and 6.4 MB of on-chip BRAM. It is connected to an Intel Core-i5 processor through a PCIe-8x link. The code running on the CPU streams input using PCIe to our overlay using the Xillybus PCIe core (http://xillybus.com/doc/revision-b-xl). The ideal data bandwidth of the core operating on the Virtex-7 device with 8× Gen3 lanes, utilizing the Gen3 Integrated Block for PCI Express v3.0, can be expected to reach 6.4 GB/s each direction (read from host and write to host), see http://xillybus.com/doc/xillybus-bandwidth.

The Ultrascale+VU9P FPGA is present on the **Amazon Web Services (AWS)** EC2 F1 instance and has 6840 DSP blocks and 75.9 MB of BRAM. The overlay is wrapped in the standard AWS F1 Verilog wrapper that interacts with the AWS F1 shell to retrieve data from DDR4 memory. The host code uses OpenCL to send data to the FPGA DDR memory and from there it is read over the memory interface that provides a combined bandwidth of 16 GB/s.

We use Bluespec System Verilog [31] to design our hardware. All the reported hardware characteristics of the design are obtained Post Place and Route in the Vivado Design Suite, Refer to Table 1.

7.1 Performance Comparison with State-of-the-Art

We compare the performance of our accelerator with other recent works. As listed in Table 2, the accelerator frameworks chosen for comparison come under two categories. In the *Specific* category, the accelerators are tailor made for processing a specific network or a class of networks. In the *Generic* category, the accelerators are designed as overlays and are network agnostic.

As can be seen in Tables 3, Table 4, Table 5, Table 6 and Table 7, our overlay outperforms most of the frameworks across both the categories. With respect to the throughput metric in GOps per second, we outperform generic/overlay accelerators by a factor of approximately 1.2× to 5× on both the FPGAs. The highest speed up is observed for AlexNet and VGG-16, since these are regular CNNs with canonical convolution layers. FPGA 2020 [48] achieves higher throughput, approximately 1.2× higher. This can be explained by the fact that it uses a low precision (8-bit) floating-point quantization method to quantize both weights and activations. Our overlay, however, uses a 16-bit fixed-point representation. We perform at-par compared to the work in [44], which uses inter-layer parallelism on an FPGA cluster to do the processing. Our overlay operates at a slightly

Table 2. Classification of the Reference Works Into Network Specific and Generic Processing Architectures

	FPGA 2020 [48]	FGPA 2020 [51]	ICET 2020 [55]	VLSI 2020 [50]	IEEE Trans 2020 [44]	FCCM 2019 [24]	FPL 2019 [49]	VLSI 2019 [30]	VLSI 2019 [22]	IPSJ 2019 [38]	FPL 2018 [56]	FPGA 2018 [29]	VLSI 2018 [26]	IEEE Trans 2018 [4]	IEEE Trans 2018 [41]	ICCAD 2018 [47]
Network Specific		✓	✓				✓	✓		✓	✓	✓		✓	✓	✓
Generic/ Overlay	✓			✓	✓	✓			✓				✓			

Table 3. Performance Comparison of Our Overlay for AlexNet with Other Recent Works

	FPGA 2020 [48]	FCCM 2019 [24]	ICCAD 2018 [47]	IEEE Trans. 2020 [44]	IEEE Trans. 2018 [11]	IEEE Trans. 2018 [41]	**Ours**	**Ours**
Architecture	Common, B = 1	Without FC	Common, B = 4	Common, runs on 15 FPGAs	Specialized P.E. for different layers	Without FC	**Common, B = 1**	**Common, B = 1**
FPGA	XC7K325T	Zynq ZCU102	Xilinx VU9P	Xilinx XC7VX690T	Virtex 7 690t	Zynq 7045	**Virtex 7 690t**	**Ultrascale+ VU9P**
Throughput GOps/s (GOPS)	1066.4	223.4 (C)	1432	1157 (Per FPGA)	910.2	197.4(C)	**1030 / 1200 (C)**	**1356.57 / 1581 (C)**
Precision	8b floating	16b fixed	16b fixed	single floating point	16b fixed	16b fixed	**16b fixed**	**16b fixed**
Performance Density GOPS/KLUTs	6.89	0.405 (C)	3.06	—	2.98	—	**4.62 / 5.38 (C)**	**3.03 / 3.53 (C)**
Performance Density GOPS/DSP	1.388	0.195 (C)	0.317	—	0.305	0.220 (C)	**0.33 / 0.39 (C)**	**0.29 / 0.34 (C)**
Frequency MHz	200	200	200	—	150	125	**166**	**166**

Common signifies same architecture used for convolution and fully connected layer. B is an acronym for batch size. C represents throughput for only convolutional layers.
The text in bold represent our numbers.

Table 4. Performance Comparison of Our Overlay for ResNet-50 with Other Recent Works

	FPGA 2020 [48]	FCCM 2019 [24] Resnet	FPL 2018 [56]	VLSI 2018 [26]	VLSI 2018 [26]	**Ours**	**Ours**
FPGA	XC7K325T	Zynq ZCU102	Stratix-V 5SGSD8	Intel Stratix-V GXA7	Intel Arria 10 GX	**Virtex 7 690t**	**Ultrascale+ VU9P**
Throughput GOps/s (GOPS)	1101.9	291.4	973.2	243.3	611.4	**902**	**1003**
Precision	8b floating	16b fixed	16b fixed	16b fixed	16b fixed	**16b fixed**	**16b fixed**
Performance Density GOPS/KLUTs	7.13	0.53	—	1.38	2.76	**4.04**	**2.24**
Performance Density GOPS/DSP	1.435	0.255	0.579	0.950	0.408	**0.29**	**0.22**
Frequency MHz	200	200	200	150	200	**166**	**166**

reduced throughput for networks with special convolutions, i.e., ResNet, YOLO, and MobileNet. We outperform the RTL-based **overlay processor (OPU)** in [50], which utilizes a fine-grained instruction control mechanism to process the individual layers using different parallelism mixes.

OPU is an 8-bit accelerator. All networks are quantized to 8-bit precision for both kernel weights and feature maps. We can perform two 8-bit MAC operations using a single DSP against a single 16-bit MAC operation. The effective memory bandwidth also increases by two times, keeping the DSPs busy in computations. Further, the capacity of other resources such as BRAMs, flip-flops, and so on, effectively doubles, and the placement/routing complexity reduces non-linearly. All these factors contribute to higher throughput and a GOPs/DSP ratio. As can be seen in Tables 5 and 6, the GOPs/KLUT of OPU (refer to the Table 3 of the OPU paper) is 4.1 and 3.8, which is comparable to our 16-bit design. This highlights that our accelerator consumes less FPGA resources, which can be attributed to the simple PE design and the control logic. OPU is more of an instruction

Table 5. Performance Comparison of Our Overlay for VGG-16 with Other Recent Works

	FPGA 2020 [48]	IEEE Trans. 2020 [44]	IPSJ Trans. 2019 [38]	VLSI 2020 [50]	FCCM 2019 [24]	VLSI 2019 [22]	ICCAD 2018 [47]	FPL 2018 [56]	**Ours**	**Ours**
Architecture	Common, B = 1	Common, runs on 15 FPGAs	Without FC	Common, B = 1	Without FC	Different	Common, B = 4	Different	**Common, B = 1**	**Common, B = 1**
FPGA	XC7K325T	Xilinx XC7VX690T	Arria10 GX1150	XC7K325T	Zynq ZCU102	XC7VX690T	Xilinx VU9P	Stratix-V 5SGSD8	**Virtex 7 690t**	**Ultrascale+ VU9P**
Throughput GOps/s (GOPS)	1086.8	1197 (per FPGA)	960(C)	354 / 397 (C)	309(C)	760.83	1510	1928	**834.6 / 1025 (C)**	**1223 / 1503 (C)**
Precision	8b floating	single floating point	16b fixed	8b fixed	16b fixed	8b floating	16b fixed	16b fixed	**16b fixed**	**16b fixed**
Performance Density GOPS/KLUTs	7.03	—	3.99 (C)	3.73/4.1 (C)	0.860 (C)	—	3.06	—	**3.74 / 4.60(C)**	**2.73 / 3.35(C)**
Performance Density GOPS/DSP	1.415	—	1.533 (C)	0.69 / 0.77 (C)	0.270 (C)	0.741	0.369	1.1	**0.27 / 0.33(C)**	**0.26 / 0.33(C)**
Frequency MHz	200	—	200	200	200	200	210	200	**166**	**166**

Different signifies different architecture used for convolution and fully connected layer.

Table 6. Performance Comparison of Our Overlay for YOLO-v2 with Other Recent Works

	FPGA 2020 [48] Tiny YOLO-v2	IEEE ICET 2020 [55]	VLSI 2020 [50]	IEEE Trans. 2018 [12]	VLSI 2020 [50] Tiny YOLO	FPGA 2018 [29] Lightweight YOLO-v2	VLSI 2019 [30] Tiny YOLO-v2	VLSI 2019 [30] Slim YOLO-v2	**Ours**	**Ours**
FPGA	XC7K325T	Zynq Ultrascale +	XC7K325T	XC7Z020	XC7K325T	Ultrascale +	Virtex-707	Virtex-707	**Virtex 7 690t**	**Ultrascale+ VU9P**
Throughput GOps/s (GOPS)	1095.4	289	391	62.9	366	610.9	464.7	1877	**1075**	**1649**
Precision	8b floating	16b fixed	8b fixed	8b fixed	8b fixed	(1-32,1-32) fixed	(1,6) fixed	(1,6) fixed	**16b fixed**	**16b fixed**
Performance Density GOPS/KLUTs	7.08	3.04	4.1	2.11	3.8	4.52	5.40	12.11	**4.82**	**3.68**
Performance Density GOPS/DSP	1.426	0.47	0.758	0.331	0.709	1.620	2.766	6.901	**0.35**	**0.36**
Frequency MHz	200	300	200	214	200	300	200	200	**166**	**166**

Table 7. Performance Comparison of Our Overlay for MobileNet v2 with Other Recent Works

	FPGA 2020 [51]	ArXiv 2020 [15]	FPL 2019 [49]	FPL 2019 [49]	FPL 2018 [56]	IEEE Trans. 2018 [4]	**Ours**	**Ours**
FPGA	XC7K325T	Stratix-10 2800	Xilinx ZU9EG	ZU2EG	Stratix-V 5SGSD8	Arria 10 SoC	**Virtex 7 690t**	**Ultrascale+ VU9P**
Throughput FPS & GOps/s	325.7 FPS —	4539 FPS —	809.8 FPS —	205.3 FPS —	— 592 GOPS	266.2 FPS 170.6 GOPS	**272.87 FPS** **830 GOPS**	**275.15 FPS** **948 GOPS**
Precision	8b fixed	16b fixed	8b fixed	8b fixed	16b fixed	16b fixed	**16b fixed**	**16b fixed**
Performance Density GOps/KLUTs	—	—	—	—	—	2.08	**3.72**	**2.12**
Performance Density GOps/DSP	0.14	—	—	—	0.319	0.133	**0.27**	**0.21**
Frequency MHz	200	390	333	430	200	133	**166**	**166**

set-based accelerator, while we use a simple set of control words that are driven by the host to process the computations in different configurations. The Light-OPU [51] paper is a variant of the original OPU paper that handles only light convolutional networks like MobileNet. In our case, we can handle both dense nets like VGG-16 and light CNNs like MobileNet using the same micro-architecture. LightOPU on MobileNet has a GOPS/DSP value of 0.14 while we achieve a value of 0.27 (refer to Table 7).

Overlay designs warrant high FPGA resources. Performance density with respect to the area of an accelerator is measured in GOps per KLUTs. Our accelerator achieves better performance density by almost a factor of 1.3× to 4×. A similar trend can be seen for performance density values with respect to the DSP resources. This shows that compared to other overlay designs, our design achieves higher throughput at lower resource consumption. Frameworks, like Reference

[48], employing 8-bit inferencing has higher performance density values because of the smaller data type. Compared to the Virtex7 FPGA, the performance density of our hardware is less on the VU9P board. This is due to the inefficient PCIe bandwidth utilization using the Xillybus protocol leading to the under-utilization of DSP blocks.

In the network-specific category, our overlay performs at par or better, compared to the other accelerators. On the VU9P FPGA, we slightly outperform the TGPA heterogeneous accelerator [47], running with batch size four, both in the throughput and performance density metric. We achieve approximately 1.4× higher throughput compared to the work in [11], which proposed a pipelined heterogeneous accelerator with different layers of a CNN mapped to different chip units within the FPGA. The work in FPL 2018 [56] outperforms our overlay for VGG-16 as it uses an optimized model for the VGG-16 network along with different hardware architectures for convolution layer and FC layer. In the case of ResNet-50, our overlay has comparable performance with others.

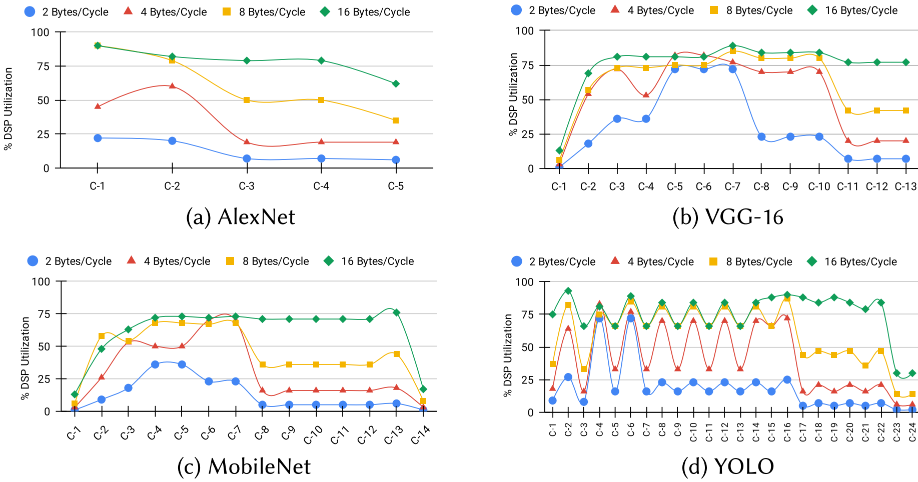
Among all the CNNs, our accelerator reports the lowest throughput for MobileNet on both the FPGAs. This is because of the presence of the depth-separable convolution layers. Although we optimize the processing of MobileNet using the Zig-Zag processing flow, the alteration of the pointwise and depthwise layers lowers the overall throughput. All the accelerators processing MobileNet, considered in our comparison, fall under the network-specific category. These employ techniques to optimize the sparsity factor of MobileNet, especially the depth-separable layers. Our overlay achieves a throughput of 830 and 948 GOps/s on Virtex7 and VU9P FPGA, respectively. As can be seen from Table 7, our overlay operates at par with the 16-bit accelerators but is outperformed by the 8-bit accelerators. When comparing the results of both FPGAs, the number of multipliers increases by 1.3×, resulting in 1.46× and 1.53× performance increase for VGG-16 and YOLO, respectively. Recall from Section 5, that at the end of processing a batch, all the computational pipelines will be flushed, and at the beginning, the pipelines will again be filled before a steady state is achieved. These overheads will be reduced as the number of batches decrease. Hence more speed up compared to the increase in the multipliers is observed.

The results presented in this section are for a 16-bit version of our accelerator. Our hardware design is parameterized to handle different fixed-point precision. For an 8-bit version, from a theoretical perspective, our accelerator reports 1.5× higher throughput than the 16-bit version. For example, for AlexNet, the theoretical peak throughput on the Virtex7 FPGA is 1853 GOPs.

7.2 Architecture Analysis

In this section, we do an architectural analysis of our overlay. First, we observe how our overlay reacts to changing **External Memory Bandwidths (EMBs)**. Figure 8, plots the percentage DSP utilization for each convolutional layer of AlexNet, VGG-16, YOLO, and MobileNet CNNs for different EMBs. We calculate the dynamic DSP utilization of the overlay. It is calculated by taking a weighted average across the DSP utilization of all the compute batches of a layer. Note that, our overlay employs greater number of DSP units, compared to the state of the art. Thus, maximizing DSP utilization over other research works. In total, 360 DSPs are reserved for depthwise convolutions appearing in MobileNet. These DSPs will be idle for regular convolutions and contribute to 10% of the total number of DSPs used in the overlay. Figure 8, provides a detailed DSP utilization plot of the 3,072 DSPs, barring the 360 DSPs.

Based on the utilization plots in Figure 8, it can be seen that a CNN layer is either sensitive or insensitive toward bandwidth change. Bandwidth-insensitive layers are those where our overlay manages to maintain a comparable DSP utilization across different EMBs. Layers `C6` from VGG-16 and layers `C4`, `C6` from YOLO CNNs are examples of such layers. The insensitivity of these layers can be attributed to the similarity in their compute schedules generated. Table 8 enumerates the compute schedules for a few representative CNN layers across different EMBs. Notice that



(a) AlexNet (b) VGG-16

(c) MobileNet (d) YOLO

Fig. 8. Variation in DSP utilization with changing external memory bandwidth. The bandwidth is reported as Bytes/Cycle fetched from external DRAM. The DSP utilization for MobileNet combines a depthwise and canonical convolution layer under a single layer, since they are processed in a pipelined fashion.

Table 8. Enumerating Compute Schedules of Our Overlay against Different Convolutional Layers of AlexNet, VGG-16, YOLO, and MobileNet CNNs

	2 Bytes/Cycle				**8 Bytes/Cycle**				**16 Bytes/Cycle**			
	Schedule = iter x (FP, SP/CP*)				Schedule = iter x (FP, SP/CP*)				Schedule = iter x (FP, SP/CP*)			
	Alex	**VGG**	**YOLO**	**Mobile**	**Alex**	**VGG**	**YOLO**	**Mobile**	**Alex**	**VGG**	**YOLO**	**Mobile**
C3	13x(28,1) 1x(20,1)	1x(128,1)	1x(128,1*)	1x(64,1)	1x(199,1) 1x(185,1)	1x(113,3) 1x(15,7)	1x(128,7*)	1x(56,6) 1x(8,7)	1x(341,1) 1x(43,6)	1x(85,4) 1x(31,11) 1x(12,15)	1x(128,15*)	1x(34,10) 1x(26,13) 1x(4,15)
C4	13x(28,1) 1x(20,1)	1x(128,1)	1x(256,1)	1x(128,1)	1x(199,1) 1x(185,1)	1x(113,3) 1x(15,7)	1x(170,2) 1x(85,4) 1x(1,7)	1x(113,3) 1x(15,7)	1x(341,1) 1x(43,6)	1x(85,4) 1x(31,11) 1x(12,15)	1x(170,2) 1x(85,4) 1x(1,15)	1x(31,11) 1x(85,4) 1x(12,15)
C6		1x(256,1)	1x(341,1) 1x(171,1)	2x(87,1) 1x(82,1)		1x(170,2) 1x(85,4) 1x(1,7)	1x(341,1) 1x(170,2) 1x(1,7)	1x(170,2) 1x(85,4) 1x(1,7)		1x(170,2) 1x(85,4) 1x(1,15)	1x(341,1) 1x(170,2) 1x(1,15)	1x(170,2) 1x(85,4) 1x(1,15)
C14			5x(100,1) 1x(12,1)	2x(81,1) 4x(13,1)			1x(341,1) 1x(170,2) 1x(1,7)	2x(81,1) 1x(38,1) 1x(14,2)			1x(341,1) 1x(170,2) 1x(1,15)	12x(81,1) 1x(38,2) 1x(14,4)

The * mark in a schedule signifies the channel parallelism factor.

the bandwidth-insensitive layers highlighted in the table have similar compute schedules. They have a high degree of filter parallelism for the starting batches. For the final batch, the degree of surface parallelism varies depending on the bandwidth. For example, the schedule generated for layer C6 of VGG varies only in the final iteration, wherein the surface parallelism is maximized.

Our scheduling logic compensates for the bandwidth deficiency by increasing the degree of filter parallelism. But this can only be leveraged for layers with a high compute to memory overlap. As can be seen from the compute-to-overlap constraint in Equation (6) from Section 4, the higher degree of filter parallelism is balanced by a higher output dimension, OL^2 value, in the inequality. Therefore, bandwidth-insensitive layers occur at the front of a CNN where the surface dimensions are comparatively larger. As we move toward the end layers, where the input dimensions are small, the compute to memory ratio decreases, and the layers become sensitive toward bandwidth

Table 9. The Table Lists the Compute Cycles (CC), Memory Cycles (MC), and Pipeline Efficiency (PEF) for Processing Various CNNs

Virtex7 690-t				
	CC ($\times10^3$)	MC ($\times10^3$)	PEF ($\times10^3$)	Latency (ms)
AlexNet	1068.4	210	82	6.5
VGG-16	6698.7	976.2	78	40.2
ResNet-50	3157.8	744.8	55	19.1
MobileNet	2892.5	1034.8	60	17.2
YOLO-V2	10133.1	2077.3	79	60.8

Table 10. Comparison of Static and Dynamic Scheduling in Our Accelerator

	Static Scheduling			**Dynamic Scheduling**	
Network	**Config (FP, FS)**	**Throughput (GOps/s)**	**Calls**	**Throughput (GOps/s)**	**Calls**
AlexNet	(77,4)	1098	27	1200	18
VGG-16	(32,10)	995	141	1020	30
YOLO-V2	(256,1)	820	111	1075	72
Resnet-50	(128,2)	710	264	902	85
Mobile-Net	(86,3)	720	114	830	61

change. This can be clearly observed for all the networks in Figure 6. Also, as can be seen from Table 8, the sensitive layers, for example, `C14`, have very different compute schedules for different memory bandwidths. Kernel size also affects the compute to memory ratio, thereby affecting the utilization. For example, all the pointwise layers in the YOLO CNN are bandwidth sensitive. The channel parallelism factor, see layer `C3` of YOLO CNN in Table 8, varies with the bandwidth in the compute schedules of the pointwise layers.

Recall from the earlier discussions that the compute schedules for our overlay are generated such that the pipeline stalls are minimized. If the average DSP utilization of the design is A operations per cycle, then the pipeline efficiency is defined as A/N^{DSP} where N^{DSP} is the number of DSPs on the FPGA. N^{DSP} operations per cycle is the theoretical maximum operations throughput achievable. The average DSP utilization A is computed by dividing the total number of operations in the CNN by the total execution cycles (compute and memory). As can be seen from Table 9, the overlay sustains a pipeline efficiency of approximately 80% for AlexNet, YOLO, and VGG-16 CNNs. For the smaller networks, ResNet-50 and Mobile-Net, we see a dip in the efficiency, which can be attributed to the presence of depthwise sparse convolution and element wise convolution layers in these networks. With respect to the FPGA, the pipeline efficiency is slightly lower for the VU9P board. This is due to the inefficient PCIe bandwidth utilization leading to the under-utilization of DSP blocks.

We next contrast the effectiveness of a static schedule, prevalent in most accelerator designs, against a dynamic schedule (proposed in this work) in processing CNNs. We run our accelerator in static mode for this experiment. For a given CNN, a fixed schedule is used to process all the layers in the static version. The fixed schedule is generated by first choosing the $SP/CP/FP$ value that maximizes the throughput of a single layer. Then, from among all the CNN layers, the schedule that maximizes the total throughput of the entire network, i.e., all the layers combined, is chosen. As can be seen in Table 10, there is a throughput drop of 100 to 200 GOps/s between both the schedules. The lowest throughput drop is seen for VGG-16, as it is a relatively uniform CNN, and the same schedule suffices to extract fairly good throughput for all the layers. The throughput drop is more significant for non-uniform networks like ResNet/Mobile-Net that vary in kernel dimensions and convolution types. Another essential comparison metric is the number of accelerator invocations/calls that the host makes to process a complete CNN. As can be seen, compared to the dynamic schedule, the static schedule results in 1.3× to 3× higher accelerator invocations. Higher invocations result in more processing latency due to the long communication setup time between the host and the FPGA. Compared to the static schedules, the dynamic schedule can better adapt to varying computation shapes across the layers, which enables it to process more computations per batch, resulting in fewer batches.

8 CONCLUSION

In this work, we presented an FPGA overlay for CNN inference. Apart from regular convolutional layers, our overlay can handle point and depth separable layers. The synthesized architecture is dependent on the available FPGA resources alone and is agnostic to any specific CNN. The host machine configures the overlay through control words on a per layer basis so as to maximize the throughput based on the layer characteristics. These configuration parameters, surface and filter parallelism degrees, are determined through a constraint satisfaction problem.

The performance of the overlay for various CNNs are thoroughly analyzed and their performance is compared against the theoretical limits obtained using hardware performance counters in the design. For future work, we plan to extend our synthesis framework to include emerging convolutional layer types such as grouped and shuffled grouped convolutions.

REFERENCES

[1] Mohamed S. Abdelfattah David Han, Andrew Bitar, Roberto DiCecco, Shane OConnell, Nitika Shanker, Joseph Chu, Ian Prins, Joshua Fender, Andrew C. Ling, and Gordon R. Chiu. 2018. DLA: Compiler and FPGA overlay for neural network inference acceleration. arXiv:1807.06434. Retrieved from http://arxiv.org/abs/1807.06434.

[2] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Sérot, François Berry, Cédric Bourrasset, and Jean-Charles Quinton. 2017. Hardware automated dataflow deployment of CNNs. arXiv:1705.04543. Retrieved from http://arxiv.org/abs/1705.04543.

[3] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL deep learning accelerator on Arria 10. arXiv:cs.DC/1701.03534. Retrieved from https://arxiv.org/abs/1701.03534.

[4] L. Bai, Y. Zhao, and X. Huang. 2018. A CNN accelerator on FPGA using depthwise separable convolution. *IEEE Trans. Circ. Syst. II: Expr. Briefs* 65, 10 (2018), 1415–1419. https://doi.org/10.1109/TCSII.2018.2865896

[5] Srimat Chakradhar, Murugan Sankaradas, Venkata Jakkula, and Srihari Cadambi. 2010. A dynamically configurable coprocessor for convolutional neural networks. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*. ACM, New York, NY, 247–257. https://doi.org/10.1145/1815961.1815993

[6] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA'16)*. IEEE Press, Los Alamitos, CA, 367–379. https://doi.org/10.1109/ISCA.2016.40

[7] Yazhuo Dong, Yong Dou, and Jie Zhou. 2007. Optimized generation of memory structure in compiling window operations onto reconfigurable hardware. In *Reconfigurable Computing: Architectures, Tools and Applications*, Pedro C. Diniz, Eduardo Marques, Koen Bertels, Marcio Merino Fernandes, and João M. P. Cardoso (Eds.).

[8] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs. In *Proceedings of the Design Automation Conference (DAC'17)*. ACM, New York, NY. https://doi.org/10.1145/3061639.3062207

[9] Jeremy Fowers, Kalin Ovtcharov, Michael Papamichael, Todd Massengill, Ming Liu, Daniel Lo, Shlomi Alkalay, Michael Haselman, Logan Adams, Mahdi Ghandi, Stephen Heil, Prerak Patel, Adam Sapek, Gabriel Weisz, Lisa Woods, Sitaram Lanka, Steven K. Reinhardt, Adrian M. Caulfield, Eric S. Chung, and Doug Burger. 2018. A configurable cloud-scale DNN processor for real-time AI. In *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA'18)*.

[10] V. Gokhale, A. Zaidy, A. X. M. Chang, and E. Culurciello. 2017. Snowflake: An efficient hardware accelerator for convolutional neural networks. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS'17)*. 1–4. https://doi.org/10.1109/ISCAS.2017.8050809

[11] L. Gong, C. Wang, X. Li, H. Chen, and X. Zhou. 2018. MALOC: A fully pipelined FPGA accelerator for convolutional neural networks with all layers mapped on chip. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 37, 11 (2018), 2601–2612. https://doi.org/10.1109/TCAD.2018.2857078

[12] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang. 2018. Angel-eye: A complete design flow for mapping CNN onto embedded FPGA. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 37, 1 (2018), 35–47. https://doi.org/10.1109/TCAD.2017.2705069

[13] Zhi Guo, Betul Buyukkurt, and Walid Najjar. 2004. Input data reuse in compiling window operations onto reconfigurable hardware. *SIGPLAN Not.* 39, 7 (June 2004), 249–256. https://doi.org/10.1145/998300.997199

[14] Zhi Guo, Walid Najjar, and Betul Buyukkurt. 2008. Efficient hardware code generation for FPGAs. *ACM Trans. Archit. Code Optim.* 5, 1, Article 6 (May 2008), 6:1–6:26 pages.

[15] Mathew Hall and Vaughn Betz. 2020. HPIPE: Heterogeneous layer-pipelined and sparse-aware CNN inference for FPGAs. arXiv:2007.10451. Retrieved from https://arxiv.org/abs/2007.10451.

[16] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10).* ACM, New York, NY, 37–47. https://doi.org/10.1145/1815961.1815968

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. arXiv:1512.03385. Retrieved from http://arxiv.org/abs/1512.03385.

[18] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for Mobile vision applications. *CoRR* abs/1704.04861.

[19] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA'17).* ACM, New York, NY, 1–12. https://doi.org/10.1145/3079856.3080246

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems–Volume 1 (NIPS'12).* Curran Associates Inc., 1097–1105. http://dl.acm.org/citation.cfm?id=2999134.2999257.

[21] Huimin Li, Xitian Fan, Li Jiao, Wei Cao, Xuegong Zhou, and Lingli Wang. 2016. A high performance FPGA-based accelerator for large-scale convolutional neural networks. In *Proceedings of the 26th International Conference on Field Programmable Logic and Applications (FPL'16).* 1–9. https://doi.org/10.1109/FPL.2016.7577308

[22] X. Lian, Z. Liu, Z. Song, J. Dai, W. Zhou, and X. Ji. 2019. High-performance FPGA-based CNN accelerator with block-floating-point arithmetic. *IEEE Trans. VLSI Syst.* 27, 8 (2019), 1874–1885. https://doi.org/10.1109/TVLSI.2019.2913958

[23] Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. 2016. Automatic code generation of convolutional neural networks in FPGA implementation. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'16).* 61–68.

[24] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang. 2019. An efficient hardware accelerator for sparse convolutional neural networks on FPGAs. In *Proceedings of the IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'19).* 17–25. https://doi.org/10.1109/FCCM.2019.00013

[25] Sangkug Lym and Mattan Erez. 2020. FlexSA: Flexible systolic array architecture for efficient pruned DNN model training. arXiv:2004.13027. Retrieved from https://arxiv.org/abs/2004.13027.

[26] Y. Ma, Y. Cao, S. Vrudhula, and J. Seo. 2018. Optimizing the convolution operation to accelerate deep neural networks on FPGA. *IEEE Trans. VLSI Syst.* 26, 7 (2018), 1354–1367. https://doi.org/10.1109/TVLSI.2018.2815603

[27] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17).* Association for Computing Machinery, New York, NY, 45–54. https://doi.org/10.1145/3020078.3021736

[28] John McCanny, John McWhirter, and Earl Swartzlander, Jr. (Eds.). 1989. *Systolic Array Processors.* Prentice-Hall, Inc., Upper Saddle River, NJ.

[29] Hiroki Nakahara, Haruyoshi Yonekawa, Tomoya Fujii, and Shimpei Sato. 2018. A lightweight YOLOv2: A binarized CNN with a parallel support vector regression for an FPGA. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18).* Association for Computing Machinery, New York, NY, 31–40. https://doi.org/10.1145/3174243.3174266

[30] D. T. Nguyen, T. N. Nguyen, H. Kim, and H. Lee. 2019. A high-throughput and power-efficient FPGA implementation of YOLO CNN for object detection. *IEEE Trans. VLSI Syst.* 27, 8 (2019), 1861–1873. https://doi.org/10.1109/TVLSI.2019.2905242

[31] Rishiyur S. Nikhil and Arvind. 2009. What is bluespec? *SIGDA Newsl.* 39, 1 (Jan. 2009), 1–1. https://doi.org/10.1145/1862876.1862877

[32] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. 2016. Going deeper with embedded FPGA platform for convolutional neural network. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'16).* ACM, New York, NY, 26–35. https://doi.org/10.1145/2847263.2847265

[33] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2016. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)* (2016), 779–788.

[34] Ananda Samajdar, Yuhao Zhu, Paul N. Whatmough, Matthew Mattina, and Tushar Krishna. 2018. SCALE-Sim: Systolic CNN accelerator. arXiv:1811.02883. Retrieved from http://arxiv.org/abs/1811.02883.

[35] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE, 1–12.

[36] Yongming Shen, Michael Ferdman, and Peter A. Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. In *Proceedings of the ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA'17)*, 535–547.

[37] K. Simonyan and A. Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *CoRR* abs/1409.1556 (2014).

[38] Salita Sombatsiri, Seiya Shibata, Yuki Kobayashi, Hiroaki Inoue, Takashi Takenaka, Takeo Hosomi, Jaehoon Yu, and Yoshinori Takeuchi. 2019. Parallelism-flexible convolution core for sparse convolutional neural networks on FPGA. *IPSJ Trans. Syst. LSI Des. Methodol.* 12 (01 2019), 22–37. https://doi.org/10.2197/ipsjtsldm.12.22

[39] Greg Stitt, Abhay Gupta, Madison N. Emas, David Wilson, and Austin Baylis. 2018. Scalable window generation for the intel Broadwell+Arria 10 and High-Bandwidth FPGA systems. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*. Association for Computing Machinery, New York, NY, 173–182. https://doi.org/10.1145/3174243.3174262

[40] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Heng Wai Leong, Magnus Jahre, and Kees A. Vissers. 2016. FINN: A framework for fast, scalable binarized neural network inference. arXiv:1612.07119. Retrieved from http://arxiv.org/abs/1612.07119.

[41] S. I. Venieris and C. Bouganis. 2019. fpgaConvNet: Mapping regular and irregular convolutional neural networks on FPGAs. *IEEE Trans. Neural Netw. Learn. Syst.* 30, 2 (2019), 326–342. https://doi.org/10.1109/TNNLS.2018.2844093

[42] Stylianos I. Venieris and Christos-Savvas Bouganis. 2017. fpgaConvNet: Automated mapping of convolutional neural networks on FPGAs (abstract only). In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. ACM, New York, NY, 291–292. https://doi.org/10.1145/3020078.3021791

[43] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. 2018. Toolflows for mapping convolutional neural networks on FPGAs: A survey and future directions. *ACM Comput. Surv.* 51, 3, Article 56 (June 2018), 39 pages. https://doi.org/10.1145/3186332

[44] T. Wang, T. Geng, A. Li, X. Jin, and M. Herbordt. 2020. FPDeep: Scalable acceleration of CNN training on deeply-pipelined FPGA clusters. *IEEE Trans. Comput.* 69, 08 (August 2020), 1143–1158. https://doi.org/10.1109/TC.2020.3000118

[45] Ying Wang, Jie Xu, Yinhe Han, Huawei Li, and Xiaowei Li. DeepBurning: Automatic generation of FPGA-based learning accelerators for the neural network family. In *Proceedings of the Design Automation Conference (DAC'16)*.

[46] Xuechao Wei, Yun Liang, Xiuhong Li, Cody Hao Yu, Peng Zhang, and Jason Cong. 2018. TGPA: Tile-grained pipeline architecture for low latency CNN inference. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD'18)*. Association for Computing Machinery, New York, NY, Article 58, 8 pages. https://doi.org/10.1145/3240765.3240856

[47] X. Wei, Y. Liang, X. Li, C. H. Yu, P. Zhang, and J. Cong. 2018. TGPA: Tile-grained pipeline architecture for low latency CNN inference. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*.

[48] Chen Wu, Mingyu Wang, Xinyuan Chu, Kun Wang, and Lei He. 2020. Low precision floating point arithmetic for high performance FPGA-based CNN acceleration. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*. Association for Computing Machinery, New York, NY, 318. https://doi.org/10.1145/3373087.3375361

[49] D. Wu, Y. Zhang, X. Jia, L. Tian, T. Li, L. Sui, D. Xie, and Y. Shan. 2019. A high-performance CNN processor based on FPGA for MobileNets. In *Proceedings of the 29th International Conference on Field Programmable Logic and Applications (FPL'19)*. 136–143. https://doi.org/10.1109/FPL.2019.00030

[50] Y. Yu, C. Wu, T. Zhao, K. Wang, and L. He. 2020. OPU: An FPGA-based overlay processor for convolutional neural networks. *IEEE Trans. VLSI Syst.* (2020).

[51] Yunxuan Yu, Tiandong Zhao, Kun Wang, and Lei He. 2020. Light-OPU: An FPGA-based overlay processor for lightweight convolutional neural networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*. 122–132.

[52] Chen Zhang, Zhenman Fang, Peipei Zhou, Peichen Pan, and Jason Cong. 2016. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD'16)*. 1–8.

[53] Chen Zhang, Di Wu, Jiayu Sun, Guangyu Sun, Guojie Luo, and Jason Cong. 2016. Energy-efficient CNN implementation on a deeply pipelined FPGA cluster. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'16)*.
[54] Jialiang Zhang and Jing Li. 2017. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. Association for Computing Machinery.
[55] S. Zhang, J. Cao, Q. Zhang, Q. Zhang, Y. Zhang, and Y. Wang. 2020. An FPGA-based reconfigurable CNN accelerator for YOLO. In *Proceedings of the IEEE 3rd International Conference on Electronics Technology (ICET'20)*.
[56] Ruizhe Zhao, Ho-Cheung Ng, Wayne Luk, and Xinyu Niu. 2018. Towards efficient convolutional neural network for domain-specific applications on FPGA. 147–1477. https://doi.org/10.1109/FPL.2018.00033

Received June 2021; revised December 2021; accepted February 2022