FeatherNet: An Accelerated Convolutional Neural Network Design for Resource-constrained FPGAs

RAGHID MORCEL, HAZEM HAJJ, MAZEN A. R. SAGHIR, HAITHAM AKKARY, and HASSAN ARTAIL, American University of Beirut, Lebanon
RAHUL KHANNA and ANIL KESHAVAMURTHY, Intel Corporation, USA

Convolutional Neural Network (ConvNet or CNN) algorithms are characterized by a large number of model parameters and high computational complexity. These two requirements have made it challenging for implementations on resource-limited FPGAs. The challenges are magnified when considering designs for low-end FPGAs. While previous work has demonstrated successful ConvNet implementations with high-end FPGAs, this article presents a ConvNet accelerator design that enables the implementation of complex deep ConvNet architectures on resource-constrained FPGA platforms aimed at the IoT market. We call the design "FeatherNet" for its light resource utilization. The implementations are VHDL-based providing flexibility in design optimizations. As part of the design process, new methods are introduced to address several design challenges. The first method is a novel stride-aware graph-based method targeted at ConvNets that aims at achieving efficient signal processing with reduced resource utilization. The second method addresses the challenge of determining the minimal precision arithmetic needed while preserving high accuracy. For this challenge, we propose variable-width dynamic fixed-point representations combined with a layer-by-layer design-space pruning heuristic across the different layers of the deep ConvNet model. The third method aims at achieving a modular design that can support different types of ConvNet layers while ensuring low resource utilization. For this challenge, we propose the modules to be relatively small and composed of computational filters that can be interconnected to build an entire accelerator design. These model elements can be easily configured through HDL parameters (e.g., layer type, mask size, stride, etc.) to meet the needs of specific ConvNet implementations and thus they can be reused to implement a wide variety of ConvNet architectures. The fourth method addresses the challenge of design portability between two different FPGA vendor platforms, namely, Intel/Altera and Xilinx. For this challenge, we propose to instantiate the device-specific hardware blocks needed in each computational filter, rather than relying on the synthesis tools to infer these blocks, while keeping track of the similarities and differences between the two platforms. We believe that the solutions to these design challenges further advance knowledge as they can benefit designers and other researchers using similar devices or facing similar challenges. Our results demonstrated the success of addressing the design challenges and achieving low (30%) resource utilization for the low-end FPGA platforms: Zedboard and Cyclone V. The design overcame the limitation of designs targeted for high-end platforms and that cannot fit on low-end IoT platforms. Furthermore, our design showed superior performance results (measured in terms of [Frame/s/W] per Dollar) compared to high-end optimized designs.

Authors' addresses: R. Morcel, H. Hajj, M. A. R. Saghir, H. Akkary, and H. Artail, American University of Beirut, P.O. Box 11-0236, Beirut, 1107-2020, Lebanon; emails: rhm20@mail.aub.edu, {hh63, mazen, ha95, ha27}@aub.edu.lb; R. Khanna and A. Keshavamurthy, Intel Corporation, Hillsboro, Oregon, USA; emails: {rahul.khanna, anil.s.keshavamurthy}@intel.com.

1936-7406/2019/03-ART6 $15.00
https://doi.org/10.1145/3306202

CCS Concepts: • **Computer systems organization → Reconfigurable computing**; • **Hardware → Hardware accelerators**; **Reconfigurable logic applications**;

Additional Key Words and Phrases: Convolutional neural networks, embedded-vision, IoT applications, resource-constrained FPGAs

ACM Reference format:
Raghid Morcel, Hazem Hajj, Mazen A. R. Saghir, Haitham Akkary, Hassan Artail, Rahul Khanna, and Anil Keshavamurthy. 2019. FeatherNet: An Accelerated Convolutional Neural Network Design for Resource-constrained FPGAs. *ACM Trans. Reconfigurable Technol. Syst.* 12, 2, Article 6 (March 2019), 27 pages.
https://doi.org/10.1145/3306202

1 INTRODUCTION

With the proliferation of IoT-based applications, a surge of interest in embedded vision systems has emerged in both the industrial and research communities. Applications of embedded-based vision systems include, among many others, Unmanned Aerial Vehicles (UAVs) with object/face detection/localization capabilities, pedestrian detection systems in autonomous vehicles, and smart domestic robots. However, recent advancements in machine learning have led to a widespread rise of what is commonly known as deep-learning applications. Deep Convolutional Neural Networks, also known as ConvNets or CNNs, are a special kind of deep machine-learning models inspired by the structure of the mammalian visual cortex [25]. Nowadays, state-of-the-art performance in image classification/detection is being achieved by employing deep ConvNets [24, 35, 37]. However, these networks are very compute intensive in both the training and inference phases, and their success in recent years has been mainly due to two factors: (1) the unprecedented computing power made available by general-purpose computing platforms and (2) the availability of large collections of labeled training datasets.

Typically, IoT and embedded vision systems are limited in both computational processing and power capacity [5, 22]; moreover, these platforms should be available at the lowest possible cost to economically justify their use [36]. Consequently, deploying pre-trained ConvNets on embedded vision platforms to harness their inference ability must rely on energy efficient custom-designed accelerators that are adapted for convolutional neural computations. Among acceleration technologies, Application Specific Integrated Circuits (ASICs) provide the best performance per energy figures as they can be strictly tailored for a certain application. However, they lack the flexibility of general-purpose computing platforms such as CPUs and GPUs, and thus they fail at adapting to fast changing deep-learning algorithms. Field Programmable Gate Arrays (FPGAs), however, provide a very good balance between hardware acceleration, customizability, and flexibility, through reconfigurability. A lot of work in the research literature targets high-end FPGA platforms and can achieve very high computational throughput (e.g., 1020img/s [4]).

In this work, we present an efficient FPGA hardware template architecture for deep neural inference tailored for low-end, resource-constrained FPGA platforms, which are commonly aimed at edge computing and IoT applications [3, 19, 20, 40], but that are increasingly being used for high-end, computationally and memory intensive ConvNet applications [24, 35, 37, 42]. Our article advances knowledge by providing a new design methodology that achieves the least amount of resources and that targets low-end FPGA platforms for IoT deployment. Additionally, as part of the design process, new methods and techniques were introduced to address several design challenges. We believe that the solutions to these design challenges can benefit designers and other researchers using similar devices or facing similar challenges:

(1) *For efficient signal processing with reduced resource utilization*, we present a novel stride-aware graph-based method targeted at ConvNets. The method is inspired by previous literature, in particular the first Noble identity from the multi-rate DSP literature [28, 29]. In our work, we differ from the previous DSP literature in that we modified the standard Noble identity to a new form (shown in Figure 9(b)) that can be employed to obtain resource-efficient implementations of 2D-convolutions with strides. This modified form of the Noble identity can be repeatedly employed in a series of transformations on signal-flow-graphs for 2D-convolutions to derive compact and minimal convolution filter architectures that can be directly implemented in hardware. In addition to the convolution filters with strides, we also modeled the other computations involved in ConvNets (e.g., 2D-convolutions with stride 1, pooling layers, and normalization layers) and derived corresponding resource-efficient implementations.

(2) *For determining the minimal precision arithmetic needed while preserving high accuracy*, we propose variable-width dynamic fixed-point representations combined with a layer-by-layer design-space pruning heuristic across the different layers of the deep ConvNet model. This is different from the literature where either a single fixed-point representation is chosen at all layers, or the search space is exhaustive and slow. The selected word lengths are also constrained to being multiples of the model parameter word lengths to simplify data storage and movement across the model.

(3) *For achieving a modular design that can support different types of ConvNet layers while ensuring efficient resource utilization*, we propose the computational modules to be relatively small. In the FeatherNet design, the modules are composed of computational filters that can be interconnected to build an entire accelerator design. These model elements can be easily configured through HDL parameters (e.g., layer type, mask size, stride, etc.) to meet the needs of specific ConvNet implementations and thus they can be reused to implement a wide variety of ConvNet architectures. Although we used these model elements to implement AlexNet [24], the modular nature of the design allows the easy implementation of other ConvNets as well. In the context of targeting resource-restricted FPGAs, the modular nature of our design allows us to easily remove elements form an overall design without affecting the flow of data within it, and thus save FPGA resources. In this aspect, our approach is different from previous modular ConvNet designs, where the modules are composed of relatively large blocks of logic resources in contrast to smaller blocks of filters, as in our case.

(4) *For ease of portability between two different FPGA vendor platforms, namely Intel/Altera and Xilinx*, we make sure that our HDL implementation is not specific to one FPGA product, and that the design is portable across two competing FPGA vendor platforms. For proof of concept, we proved this principle on Intel/Altera and on Xilinx. Previous approaches [1, 8] achieve similar portability by implementing their designs in behavioral VHDL/Verilog, and rely on the synthesis tools to resolve the micro-architectural differences and infer the device-specific components for each target device (BRAMs, DSPs, etc.). The limitation of these approaches is that the synthesis tools do not have information about the purpose or intent of the design, and thus may try to optimize for area and performance by inferring device primitives. As a result, the tools may misplace the individual hardware blocks (e.g., BRAMs and LUTs) in certain sub-components of the design. To address this limitation, we rely on instantiating the device-specific hardware blocks needed in each computational filter rather than simply relying on the tools to infer these blocks while keeping track of the similarities and differences between the platforms. For example, due to the dimensions of the ConvNet feature maps involved in the computations in AlexNet, normalization filters

are best implemented using BRAMs, whereas max pooling filters are best implemented using LUTRAMs in Xilinx FPGAs [40] and MLABs in Cyclone V [19]. To achieve the desired portability, we develop a hierarchical design that decouples device-specific features needed to meet the low resource utilization target from the rest of the design. We use VHDL generics to seamlessly instantiate and integrate the device-specific components in our design. Using this decoupling, we are able to successfully implement our ConvNet on two architecturally different FPGA devices; the Xilinx Zynq and Intel Cyclone V, using two different logic synthesis tools. Our design can be easily ported to other devices by editing a small number of VHDL source files.

Our methodology is applied to AlexNet [24], a popular ConvNet for image classification that requires 700 million multiplications with 61 million parameters per image. Although the implementation was customized for AlexNet, the architecture can be adapted for other convolutional network architectures. Our results demonstrated the success of addressing the design challenges and achieving low (30%) resource utilization for the low-end FPGA platforms: Zedboard [3] and Cyclone V [20]. The design overcame the limitation of designs targeted for high-end platforms that cannot fit on low-end IoT devices [5, 22, 36]. Furthermore, our design showed superior performance results (measured in terms of performance/W/Dollar) compared to high-end optimized designs (9.31×10^3 Frame/J/\$ compared to 5.17×10^3 for the state-of-the-art [4]). The designs also showed accurate results consistent with a non-accelerated software implementation, where images from the ILSVRC-2012 dataset were classified with a top-5 accuracy of 83.58% (compared to 79% in Reference [4]) while achieving the low-end FPGA benefits of improved energy efficiency per cost at 9.31×10^3 Frame/J/\$ and a frame rate of up to 9 frames/s.

The rest of this article is structured as follows: Section 2 provides a detailed background on Convolutional Neural Networks. We also shed some light on a selection of the latest related research efforts on accelerating ConvNets on FPGA platforms. In Section 3, we describe the architecture of our proposed accelerator design. Section 4 illustrates the design methodology used to realize efficient implementations of the different computational sub-tasks in the accelerator described in Section 3. Finally, we evaluate the performance and energy efficiency of our implementations in Section 5.

2 BACKGROUND AND RELATED WORK

2.1 Deep Convolutional Neural Networks

Deep Convolutional Neural Networks (ConvNets) are a special kind of feed-forward multi-layer neural networks (NN) that typically consist of a series of convolutional and pooling layers, followed by one or more fully connected layers. In a convolutional layer, the neurons are arranged into several separate rectangular patches commonly known as Feature maps [25]. This arrangement makes them suitable for processing multi-dimensional data such as 2D color images and 3D video signals. In typical multi-dimensional signals such as images and videos, local groups of pixels are highly correlated [25]. Convolutional layers are designed to exploit this spatial correlation in an input neural layer by employing a local connectivity pattern between neurons in consecutive layers; in other words, in a convolutional layer, every neuron is connected to only small patches of neurons, commonly referred to as local receptive fields, residing on different feature maps from the previous convolutional layer. Another key aspect that distinguishes convolutional layers from fully connected NN layers is the parameter sharing scheme in which all neurons that belong to the same convolutional layer share the same set of weights. These arrangements, i.e., local connectivity and parameter sharing, are mathematically described by a multi-dimensional discrete convolution operation followed by a non-linear activation function [25, 43]. Figure 1

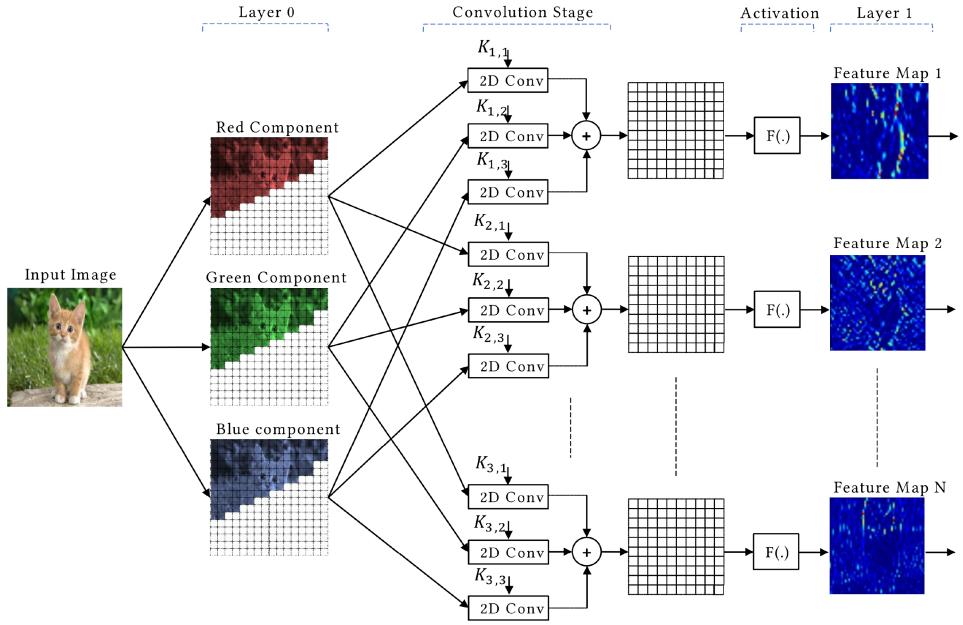


Fig. 1. An illustration of a typical Convolutional Layer.

illustrates the structure of a typical convolutional layer. A convolutional layer can assume multiple hyper-parameters. Among these is the size of the local receptive field, which is the common size of the convolution masks in the 2D convolutions shown in Figure 1. In practical ConvNets, the convolution mask can stride multiple pixels at a time, and thus the stride configuration is another hyper-parameter of a convolutional layer. Another important hyper-parameter is the zero-padding configuration, which determines the number of 0 samples concatenated at the beginning and at the end of every feature map.

The last hyper-parameter for a convolutional layer is the group parameter, which determines the number of groups the input and output feature maps are divided into. Every group of output feature maps is then connected to its corresponding group of input feature maps. This arrangement reduces the complexity of the ConvNet, but it is rarely used nowadays, since the current powerful computing platforms obviate the need for it.

A convolutional layer is often followed by a pooling layer, which computes a statistical summary of local neural activities in a convolutional layer. Feature maps at the input of the pooling layer are first divided into rectangular patches of neurons, also referred to as receptive fields. The neurons in every patch or receptive field are then summarized into a single neuron for every patch, typically using the maximum or the average of the neural activities. The purpose of pooling is to make the detection invariant to minor shifts and distortions [25].

Currently, deep ConvNets employ more processing stages between the convolutional and pooling layers. For example, AlexNet [24], a ConvNet used for image classification, employs a Local Response Normalization (LRN) layer after the first and second convolutional layers. The role of the LRN layer is to drain the responses of local clusters of neurons that have uniformly large activations and boost those neurons that have relatively large activations when compared to their neighbors. In the past few years, many other ConvNet architectures were inspired by AlexNet and were able to achieve better performance. Those include GoogleNet [37], Zeiler and Fergus [42], and VGG [35]. AlexNet, however, is still widely used as a benchmark for evaluating ConvNet

accelerators [4, 33] due to its high computational complexity. Reference [24] provides a detailed description of the hyper-parameters of AlexNet.

2.2 Related Work

The work of Farabet et al. [10–12] was among the first pioneering efforts to accelerate Convolutional Neural inference on low-end DSP-oriented FPGA platforms. In Reference [12], a programmable FPGA-based ConvNet Processor (CNP) was designed and deployed on two different FPGA platforms: a Xilinx Spartan-3A DSP 3400 FPGA and a Xilinx Virtex 4 SX35 FPGA. The CNP consists of a Parallel Vector Arithmetic and Logic Unit (VALU) along with a control unit that sequences the operations of the VALU. The CNP could process 10 frames per second using a ConvNet with 2 million synaptic connections. Current generations of ConvNets, however, are much more complex; AlexNet, for instance, has around 724 million synaptic connections. Moreover, the ConvNets that were deployed in References [10–12] do not contain many of the more intricate structures and layers involved in current state-of-art ConvNets. This means that even though the works in References [10–12] target low-end platforms, they do not support recent ConvNet architectures such as AlexNet [24], GoogleNet [37], and VGG [35].

In recent years, interest in accelerating neural network inference on FPGA platforms has grown significantly and different design approaches were investigated and employed. In the work of Zhang et al. [43], a design space exploration methodology was proposed. Various computation optimizations and transformations such as loop unrolling, pipelining, and tiling were investigated along with memory access optimization schemes. With the help of the roofline performance model, proposed in Reference [39], optimized solutions were identified, which were then implemented using High Level Synthesis tools [18, 41]. The work of Zhang et al. [43] inspired a series of other research efforts such as that of Alwani et al. [2], which observed the existence of a previously unexplored design dimension that focuses on fusing the processing of multiple convolutional layers, and that of Shen et al. [34], which proposed an automated design methodology that divides the FPGA resources into several processors instead of a single large processor achieving more throughput.

The automatic generation of FPGA accelerators from a high-level description of a Convolutional Neural Network was also investigated and implemented by Sharma et al. [33] who proposed a framework, called DNN-Weaver, that automatically generates a synthesizable hardware design for Deep ConvNets. DNN-Weaver was used to generate designs for three high-end FPGAs: Intel Arria 10, Stratix V, and Xilinx Zynq.

The current state-of-the-art appeared in the work of Aydonat et al. [4], who proposed a novel architecture written in OpenCL [18] called Deep-learning Accelerator (DLA). The DLA employs several design techniques such as maximizing data reuse and external memory bandwidth and the use of the Winograd transform to boost the performance of the implementation. The DLA could process 1,020 images per second using AlexNet deployed on a high-end Arria 10 FPGA device.

HDL-based implementations of ConvNet architectures were also explored in the literature [1, 8]. In Reference [8], an open source VHDL-based ConvNet library and toolbox is presented. The toolbox provides an easy method to investigate the implication of employing low-bit fixed-point arithmetic on each individual layer, while the VHDL library provides reference designs for different ConvNet layers (e.g., Hardware synthesizable neurons and a PCIe-based ZF-net [42] layer implementation). The final purpose of this library is to deploy ConvNet inference algorithms on high-end Xilinx Ultrascale FPGA platforms tailored for data center deployment. In the work of Abdelouahab et al. [1], a framework called Haddoc2 was proposed. The Haddoc2 framework can take a Caffe model for a very small ConvNet architecture and automatically generate an equivalent VHDL-based hardware description of the network. The framework also employs other

optimizations such as using short fixed-point arithmetic and implementing multiplications using Logic Elements (LE) rather than hardware DSP blocks.

Implementing deep ConvNet inference in reduced precision fixed-point arithmetic has also been investigated in the literature and the existing approaches generally fall into two categories. The first category, also known as the post-training quantization approach, relies on training deep ConvNets in floating-point and then performing a floating-to-fixed-point conversion where for a given ConvNet layer the output neural activations and the neuron parameters are both represented in fixed-point [14, 27, 30, 38]. The second category relies on trained quantization, where the network is trained with the constraint of having quantized weights [7, 15, 17, 21, 26, 31, 44]. Notable efforts that employ this approach include training Binarized Neural Networks (BNN) whose weights and neural activations are constrained to +1 or −1 [7, 31], or training Ternary Weight Neural Networks (TWN) whose weights are constrained to +1, 0, and −1 [26], or training ConvNets that have reduced precision weights and activations [15, 17, 21, 44]. In the work of Jacob et al. [21] a quantized training framework was proposed to minimize the loss of accuracy from quantization, and the experimental results have shown significant improvements in the tradeoff between accuracy and on-device latency.

In this work, we focus on the development of accelerator designs that enable the implementation of complex ConvNet architectures on resource-constrained FPGA platforms aimed at the IoT market, and thus we seek implementations that consume the least resources possible to target low-end FPGA devices while still achieving acceptable performance. Our work contrasts with the majority of FPGA-based ConvNet accelerator designs reported in the literature, which either leverage the abundant resources of mid- to high-end devices to achieve the highest performance possible when implementing complex ConvNets [2, 4, 8, 33, 34, 43], or resort to implementing relatively small ConvNet architectures that have limited storage and computational requirements on embedded systems [1, 21, 30]. To tackle the challenge of designing a resource-efficient ConvNet accelerator, we devise a twofold methodology. First, we propose a set of data-flow-based design techniques that allow a designer to extract DSP-based graphical representations of deep ConvNet sub-computations and map these representations to efficient hardware implementations. In this context, we also propose a novel stride-aware graph-based method targeted at convolutional layers that employ 2D-convolutions with non-unit strides; this method can be systematically employed to derive compact and minimal-resource 2D-convolution filter architectures that can be directly implemented in hardware. Second, we investigate employing dynamic fixed-point arithmetic, i.e., different layers of the ConvNet use different bit-widths. Our approach is similar to the approaches used in References [14, 30]; However, we differ from these approaches in that we use a layer-by-layer design-space pruning heuristic to infer the minimal required bit-widths and precision-levels in each ConvNet layer.

3 MINIMALIST ACCELERATED CONVNET SYSTEM ARCHITECTURE

In this section, we describe the general functional architecture of our proposed template accelerator (shown in Figure 2). The template accelerator consists of three component types: (1) data-flow computational engines, (2) data movement circuitry, and (3) on-chip cache memory. The Data-flow computational engines are responsible for carrying out all the computational sub-tasks in a certain implementation of a ConvNet. We define a computational stage or sub-task as a set of transformations that starts with a convolutional or fully connected layer, followed by an optional normalization layer and then by an optional pooling layer. This allows ConvNet inference to be modeled as a chain of computational stages. Consequently, ConvNets have two types of computational stages: convolutional and fully connected, and, therefore, in this architecture, there are only two types of Data-flow computational engines: (1) convolutional and (2) fully connected engines.

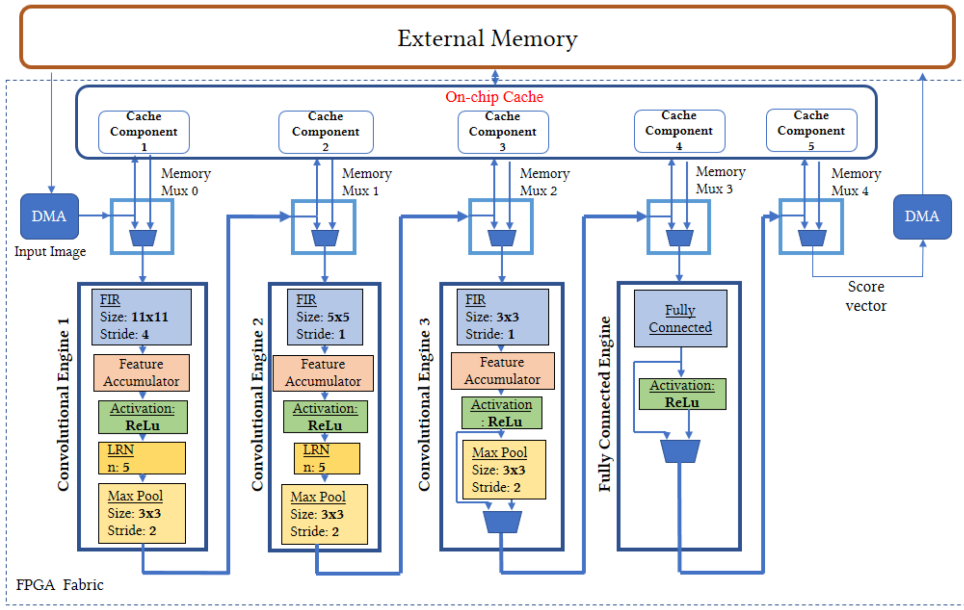


Fig. 2. System architecture.

A typical convolutional engine is a chain of computational filters that begins with a two-dimensional convolution filter and is followed by an activation filter, which computes an element-wise non-linear activation function on an input feature map; a Local Response Normalization (LRN) filter, and a pooling filter. In Section 4, we devise a methodology for designing minimal hardware realizations for each type of those filters. A Fully connected engine, however, consists of a simple dot-product filter. This architecture enables the designer to further boost the performance of a convolutional engine by deploying multiple two-dimensional convolution filters in one convolutional engine, and consequently multiple normalization and pooling filters. However, since we are aiming for a minimalist implementation, we limit the number of filters to one per engine.

The on-chip cache memory, shown in Figure 2, is responsible for locally buffering the input and intermediate feature maps. It is organized into multiple, independently accessible, simple dual-port RAM memories, called cache components. Each cache component acts as a ping-pong buffer, and thus they allow the data-flow computational engines to access different cache components in parallel, allowing the computational engines to operate concurrently while the cache components act as buffers. This choice of cache organization allows a form of temporal parallelism or pipelining to occur, i.e., input features can be received by a computational stage just as they are released by the previous one.

The two data movement circuits are referred to as DMAs in the figure. One DMA unit feeds input images to the pipeline while another DMA writes the resulting score vectors back to memory. Other DMA units (not shown in the figure) are also used to load convolution weights from memory to the convolution filters. Special components called Memory Muxes (cf. Figure 2) are used to control the flow of data between the computational engines, the on-chip cache, and the data movement circuitry. A memory mux can operate in two different modes (cf. Figure 3). In the first mode of operation, referred to as Mode A and depicted in Figure 3(a), the mux alternates between two states: in the first state, the data stream flowing into port 1 is forwarded to both ports 2 and 4 (Flow 1). In the second state, the data stream flowing into port 3 is forwarded to port 4

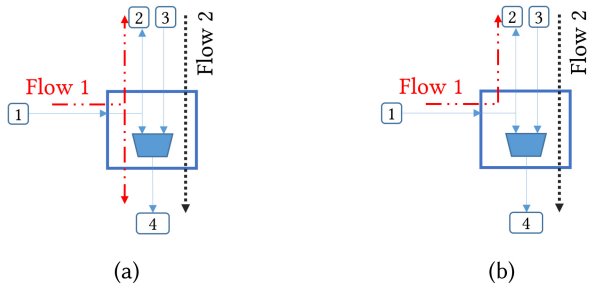


Fig. 3. Memory mux operation.

(Flow 2). This mode of operation is useful when the on-chip cache sub-component is not included in the design and the data-flow computational engines are supposed to buffer their output data on external memory. In Figure 3(a), the data stream received from a previous computational stage through port 1, is simultaneously dispatched to the current computational stage, and to external memory. The memory mux then switches to the next state, where flow 2 is now active, allowing the current computational stage to receive the data that was generated by the previous stage, from external memory. In the second mode of operation, referred to as Mode B and depicted in Figure 3(b), the data stream flowing into port 1 is dispatched to port 2, while the incoming data at port 3 is forwarded to port 4. Note that, in this mode, flows 1 and 2 are both active at the same time. This mode of operation, is useful when the on-chip cache is included in the design, since it allows the previous stage or computational engine to write to a cache component, while the current computational engine is reading from it.

Communications between different computational stages are scheduled by special control units. In fact, every memory mux has its own independent control unit, which controls its behavior and mode of operations. Equally, each cache component has its own control unit, which mediates read and write operations. The data movement circuits or DMAs, however, are controlled by an on-chip processor core. And since the DMAs are configured to operate in scatter-gather mode, the on-chip processor can chain together multiple simple DMA requests to offload multiple interrupts and consequently hide the latency of the interrupts. By controlling communications between different stages, the control units along with the on-chip processor core can handle the scheduling of computations on the computational engines. Upon scheduling a convolution operation, a DMA unit loads the weights that correspond to the scheduled convolution into the corresponding convolutional filter from external memory. Subsequently, the corresponding memory mux is scheduled to read the corresponding feature maps from a cache component and feed them to the convolutional engine.

The rapid evolution of Convolutional Neural Network architectures, and the wide variety of FPGA devices in the market, promote designs with two important properties: (1) Modularity and (2) Ease of portability between different FPGA vendor platforms. To achieve a modular design that can support different types of ConvNet layers while ensuring efficient resource utilization, we propose the modules to be relatively small. In our design, the modules are composed of computational filters that can be interconnected to build an entire accelerator design. In Section 4, we provide a methodology for designing computational filters. These model elements can be easily configured through HDL parameters (e.g., layer type, mask size, stride, etc.) to meet the needs of specific ConvNet implementations and thus they can be reused to implement a wide variety of ConvNet architectures. Although we used these model elements to implement AlexNet, the modular nature of the design allows the easy implementation of other ConvNets as well. In the context of targeting

resource-restricted FPGAs, the modular nature of our design strategy allows us to easily remove model elements from an overall design without affecting the flow of data within it, and thus when targeting the implementation of a specific ConvNet that does not contain a particular layer type, the model element that corresponds to this particular layer can be removed and thus further saving FPGA resources. It is worth noting that our approach is different from previous modular ConvNet designs, where the modules are composed of relatively large blocks of logic resources and aim at achieving specific functionalities instead of smaller blocks of filters.

To ease the portability between two different FPGA vendor platforms, we wanted to make sure that our HDL implementation is not specific to one FPGA product and that the design is portable across two competing FPGA vendor platforms, namely, Intel/Altera and Xilinx. Previous approaches [1, 8] achieve similar portability by implementing their designs in behavioral VHDL/Verilog, and they rely on the design/synthesis tools to resolve the micro-architectural differences and infer the device-specific components for each target device (BRAMs, DSPs, ...). The problem with this approach is that synthesis tools do not have information about the purpose or intent of the design and thus may try to optimize for area and performance by inferring device primitives. As a result, the tools may misplace the individual hardware blocks (e.g., BRAMs and LUTs) in certain sub-components of the design. To address this limitation of proper placement, we relied on instantiating the device-specific hardware blocks needed in each computational filter rather than simply relying on the tools to infer these blocks while keeping track of the similarities and differences between the two platforms. For example, due to the dimensions of the ConvNet feature maps involved in the computations in AlexNet, normalization filters are best implemented using BRAMs, whereas max pooling filters are best implemented using LUTRAMs in Xilinx FPGAs and MLABs in Cyclone V. To achieve the desired portability, we developed a hierarchical design that decouples the device-specific features needed to meet the low resource utilization target from the rest of the design, and we used VHDL generics to seamlessly instantiate and integrate the device-specific components for the targe device. Using this decoupling in our hierarchical design, we were able to successfully implement our ConvNet accelerator design on two architecturally different FPGA devices, the Xilinx Zynq 7020 [40] and Intel Cyclone V [19], using two different logic synthesis tools. Our design can be easily ported to other devices by editing a small number of VHDL source files.

4 DESIGN METHODOLOGY

4.1 Design Constraints

For the design to meet the performance, power, and cost requirements of IoT applications, we identified the following constraints for the design challenges:

(1) The limited resources available on low-end FPGA platforms. These platforms are well suited for low-cost and power sensitive applications and are typically characterized by a limited number of available DSP units, Block RAMs, and programmable logic blocks. They are usually promoted by their manufacturers as capable of delivering the highest DSP performance-per-watt.
(2) The relatively high computational complexity of deep convolutional neural networks. The number of multipliers in an AlexNet [24] inference is on the order of 700 million multiplication per frame. Given the real-time requirement of IoT applications, which is several frames per second and requires billions of operations per second, this places a significant strain on the resource-limited FPGA.
(3) The memory or space requirements of deep convolutional neural networks, where each convolutional layer can produce a multitude of output feature maps. The following

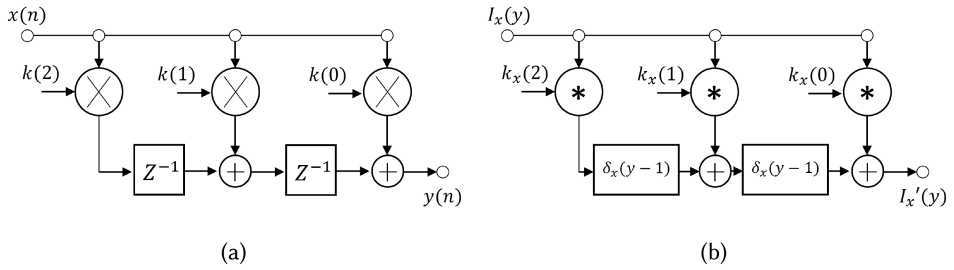


Fig. 4. Block diagram representation for the 1D FIR filter (a) and for the 2D FIR (b).

convolutional layer may use these feature maps multiple times, which requires caching the intermediate feature maps.

(4) The limited on-board memory bandwidth in low-end platforms. The memory bandwidth on such devices is typically limited to around 100Gbps. The Cyclone V dev-kit, for instance, has a theoretical memory bandwidth of 70Gbps and an achievable rate of 58.9Gbps. The Zedboard, another low-end platform, can achieve around 34Gbps. These numbers can be contrasted with those of high-end platforms such as the Intel Arria 10 and Stratix V that have 273 and 372Gbps, respectively.

4.2 Graphical Representation and Modeling of Neural Inference Computations

The computations involved in convolutional neural network inference are best described in terms of Digital Signal Processing (DSP) computations or signal processing chains rather than general-purpose workloads. DSP computations differ from their general-purpose counterparts in two major ways. The first difference is in the real-time requirement where data is received periodically from a source and should be processed within a bounded time frame, as in a real-time vision system with a camera generating frames at a certain rate. The second difference is in the data-flow property where a signal processing sub-task starts executing as soon as the data needed for that sub-task is available at the input. This modeling of computations is usually referred to as the streaming data-flow compute model, in which the computations can be modeled as a set of transformations on an input data-sequence rather than a sequence of operations that needs to be performed on a batch of data. The study of modeling DSP computations and data-flow algorithms is very well established in the literature, especially in the work of Keshab [29] and later Hauck and Dehon [16]. In this work, we reason about the implementation of various sub-tasks involved in convolutional neural networks using the data-flow compute model, in which computations are described using a Graphical representation, that is, a Data-Flow Graph or a Block Diagram. We, then, propose a set of high-level transformations on these graphical representations to derive efficient hardware implementations for many computations involved in deep convolutional neural networks.

4.2.1 Efficient Modeling of 2D Convolutions. The most important and computationally intensive sub-task in a convolutional neural network is the multi-layer convolution operation, which consists of a sequence of two-dimensional discrete-time convolution operations. Every convolution operation involved can be modeled as a discrete linear shift-invariant system with a finite impulse response; consequently, every convolution is, in principle, a Finite Impulse Response (FIR) filter. In this section, we establish a design methodology for deriving hardware implementations for 2D FIR convolution filters by extending the previously known methods in the digital signal processing literature. Figure 4(a), shows a one-dimensional three-tap FIR filter in block diagram representation. For the sake of simplicity and presentation, we will use the three-tap FIR filter as

a typical example to illustrate the proposed design techniques. Note that the block diagram representation of the filter clearly exposes the data-flow properties of the 1D convolution including its data-driven properties, and it unmasks the inherent fine-grain parallelism among different operations in the convolution. The representation shown in Figure 4(a) can be mapped directly to a hardware implementation that consists of three independent hardware multipliers, two adders, and two delay elements, which can be realized using two hardware registers. The filter can convolve an input sequence with an impulse response sequence of size three. At every rising or falling edge of the clock cycle, the filter reads 1 input sample from the input sequence, performs three multiplications and two additions in parallel, and generates 1 output sample. Another architecture can be derived from the same block diagram representation; one that can perform the same convolution with less hardware multipliers and adders, but such an implementation is more serial in nature and cannot process a sample on every iteration. In this work, we seek implementations that can at least process 1 input sample on every iteration.

Although designing a 2D convolution filter is slightly more complex, it can be derived from its 1D counterpart by exploiting the fact that a 2D convolution is essentially made of several concurrent 1D convolution operations. A high-level mathematical formulation of the 2D convolution operation is illustrated in Equation (1):

$$I'(x,y) = \sum_{q} \sum_{p} K(p,q) \times I(x-p, y-q). \tag{1}$$

$I(x,y)$, here, is the input 2D sequence or image. $K(x,y)$ is a 2D sequence that represents the convolution kernel or the filter mask. $I'(x,y)$ is the 2D sequence that results from convolving $I(x,y)$ with $K(x,y)$. It may be shown that Equation (1) can be reformulated in terms of simple 1D convolutions. The reformulation is shown in Equation (2):

$$I'_x(y) = \sum_{q} K_x(q) * I_x(y-q), \tag{2}$$

where $k_x(y)$ denotes the yth row of the filter mask; $I_x(y)$ also denotes the yth line or row of the input image, and the "*" binary operator denotes the 1D linear convolution. Note that although Equation (2) describes a 2D convolution operation, it exhibits a great deal of similarity to that of a 1D convolution, but with the multiplication operator being replaced by a 1D convolution operation denoted by "*". This similarity allows us to draw a comparable block diagram representation for the 2D-FIR filter as shown in Figure 4(b). Although the representations shown in Figures 4(a) and 4(b) share a high degree of similarity in their structures, they employ different operations.

The block-diagram representations shown in Figure 4 are different in two respects:

(1) In the first diagram, the filter consumes 1 sample from the input sequence at every iteration; similarly, every operation in Figure 4(a) processes 1 sample per iteration. The diagram shown in Figure 4(b) receives an entire row of samples from the input 2D sequence at every iteration and every sub-task in this diagram operates on an entire row of samples rather than on 1 sample.
(2) In Figure 4(b) the delay elements denoted by Z^{-1} are replaced by multi-dimensional (2D) linear systems [9] with an impulse response of $\delta_x(y-1)$ each, which is a fancy way to refer to delays along the y-axis. In other words, if we apply a 2D sequence $S(x,y)$ to the input of this system, the output will be the sequence $R(x,y) = S(x,y-1)$.

Many hardware realizations for the 2D FIR filter can be derived from Figure 4(b). However, we can set for a minimal-resource realization that takes a minimal number of multipliers and adders and can process at least 1 input sample on every iteration. We show this realization in Figure 5.

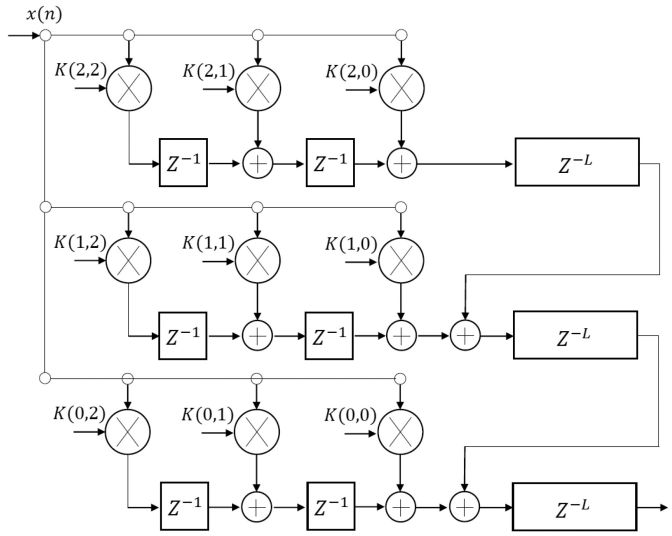


Fig. 5. A hardware realization for the 2D convolution filter derive from Figure 4(b).

Table 1. Hardware Cost of Different 2D FIR Filters

FIR window size	Multipliers	Adders	Registers
3×3	9	8	6
5×5	25	24	20
7×7	49	48	42
11×11	121	120	110

With this realization, the 2D input sequence is processed one row at a time. Equally, every row is processed 1 sample at a time. This means the 2D input sequence should be received by the filter in row major order; and the resulting output sequence is also generated in row-major order. As a result, in this work, all input feature maps are stored and transferred in this particular order.

Different convolution filters with different filter mask sizes can be designed using the same methodology we used to design the 2D FIR filter with a mask size of 3×3. In general, a hardware realization of a 2D convolution filter with a filter mask of size $N \times M$ and designed according to our methodology would result in $N \times M$ hardware multipliers, $N \times M - 1$ adders, $(N - 1) \times M$ registers, and $M - 1$ delay lines of depth L each, where L is the width of the resulting output feature map. Although the 3×3 2D convolution was portrayed in this section as a toy example to illustrate the methodology, this specific convolution kernel is heavily used, along with other convolution mask sizes, in current deep convolutional neural networks. Table 1 summarizes the hardware cost figures of different convolution filter implementations.

As shown in Table 1, the complexity of filters with large FIR windows such as the 7×7 or 11×11 is relatively high, especially when working with resource-limited FPGA platforms. The Zedboard for instance has around 240 DSP48E units, implementing a filter with a convolution kernel of size 11×11 would not fit into the Zedboard's FPGA fabric, if every multiplier is implemented using two DSP48E units. Fortunately, however, in modern convolutional neural networks, some convolutional layers have a stride configuration that differs from 1; i.e., the convolution mask can jump multiple pixels at a time as it slides around the image. For instance, the convolution kernel

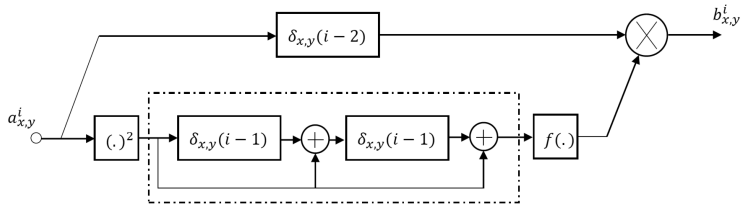


Fig. 6. A block diagram representation for a Local response normalization filter with $n = 3$.

of size 11×11 usually has a stride configuration of at least 4 pixels and hence an 11×11 FIR with a stride configuration of 4 may be implemented with only 9 multipliers instead of 121, greatly reducing its cost. In Section 4.3, we propose a set of high-level transformations and reduction schemes to design hardware implementations for FIR filters with arbitrary stride configurations.

4.2.2 Modeling the Local Response Normalization Layer. The concept of Local Response Normalization (LRN) is borrowed from the phenomenon of "Lateral inhibition" in neurobiology [25]. In biological neural networks, a sturdily excited neuron has the capacity to quell the neighboring neurons improving the perception ability of the entire network. The first effort to introduce the concept of normalization to improve the accuracy of artificial deep convolutional neural networks was in the work of Alex Krizhevsky et al. [24].

The operation for the Local Response Normalization as it appears in Reference [24] is described in Equations (3) and (4):

$$b^i_{x,y} = a^i_{x,y} \times c^i_{x,y}, \tag{3}$$

$$c^i_{x,y} = \left(1 + \frac{\alpha}{k} \sum_{j=i-n/2}^{j=i+n/2} \left(a^i_{x,y}\right)^2\right)^{-\beta}, \tag{4}$$

where $a^i_{x,y}$ denotes the activity of a neuron that belongs to the ith feature map at position (x, y) and $b^i_{x,y}$ represents the normalized neural activities. Note that the sum in $c^i_{x,y}$ runs over multiple adjacent feature maps at the same position. Hence, the 2D arrays $c^i_{x,y}$ depend on n adjacent feature maps. We treat the normalization operation as a non-linear DSP filter and thus it can be represented in a Data-Flow Graph representation or in a block diagram representation as shown in Figure 6. Figure 6 illustrates a normalization filter with $n = 3$ feature maps.

$f(x) = (1 + \frac{\alpha}{k}x)^{-\beta}$ is a non-linear function; and the $\delta_{x,y}(i-1)$ block is a delay unit along the i-axis. The above block diagram can be directly mapped to a hardware implementation, with two multipliers, two adders and three delay lines. The non-linear function can be implemented using lookup tables. The lengths of the delay lines depend on the size of the feature maps involved in the normalization layer.

4.2.3 Modeling the Pooling Layer. In traditional convolutional neural networks, a pooling layer operates on non-overlapping rectangular clusters of neurons within the same feature map. In each cluster, the pooling operation condenses the excitations of neighboring neurons into a single neuron in the next layer. In some cases, such as in Reference [24], the clusters upon which the pooling layer acts, can be overlapped, thus improving the generalization capacity and accuracy of the trained model. By overlapping these clusters, the pooling layer starts resembling the convolutional layer in that the rectangular clusters of neurons are similar to convolution filter masks. Hence, just like convolutions, the pooling can have a stride configuration that determines how far the pooling clusters are overlapping. In this section, we use a design methodology similar to the one we used for 2D convolution filters.

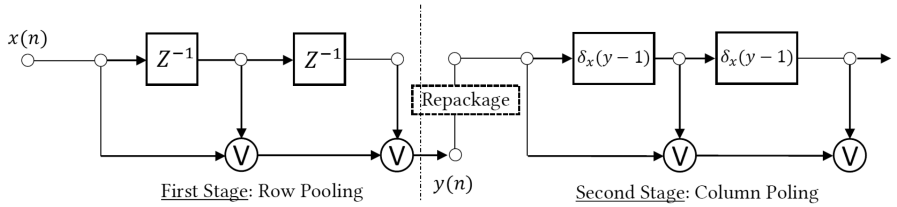


Fig. 7. A block diagram representation for a 2D Pooling filter with kernel size 3×3 and a stride of 1 pixel.

Although there are many pooling methods, the most common pooling schemes are the max pooling and the average pooling. In max pooling, the maximum excitation value in each cluster is used, whereas, in average pooling the average of the excitation values is used. A mathematical formulation of a max pooling operation with a stride of 1 pixel (maximum overlap) in both x and y directions, and a mask of size $N \times M$ is shown in Equation (5):

$$I' = \bigvee_{i=0}^{N-1} \bigvee_{j=0}^{M-1} I(x-i, y-j). \tag{5}$$

The $\vee$ operation denotes the maximum operation; mathematically, $\vee(a, b) = max(a, b)$. Finally, $I(x, y)$ denotes the input feature map and $I'(x, y)$ is the pooled output feature map. Equation (5) can be re-written as

$$I'(x, y) = \bigvee_{j=0}^{M-1} l_x(j). \tag{6}$$

In this reformulation, $l_x(j)$ denotes the result of pooling the jth row or line of the input feature map using a 1D pooling operation. If the feature map arrives at the input of the pooling filter in row-major order, then, according to Equation (6), the 2D pooling computation can proceed in two stages. In the first stage, every row is pooled individually. In the second stage, the pooled rows obtained from the first stage are combined to obtain the resulting pooled feature map. Figure 7 shows a block diagram representation for a 2D pooling filter with a kernel size of 3×3 and a stride of 1 pixel.

Note that the maximum operation, shown in Figure 7, is implemented using FPGA carry-chain and lookup table resources, which are abundantly available on most FPGAs, as opposed to DSP units, which are available in limited numbers. The unit delay elements are implemented using FPGA slice registers, while the delays along the y-axis are implemented using Block RAMs configured as delay lines. Pooling filters with larger kernel sizes and different stride configurations can be designed in a similar fashion. However, when the stride configuration is different from one, we employ down-samplers at the output of both first and second stages. In the next section, we derive a method to reduce the resource utilization for convolution filters with stride configurations.

4.3 Optimization and Reduction Schemes for 2D Convolutions with Strides

As mentioned earlier, in Section 4.2.1, some convolutional layers have convolution masks that can jump multiple pixels at a time both vertically and horizontally. We referred to such convolution operations as convolution filters with a stride configuration. We also laid down a systematic technique to derive a hardware implementation for the 2D convolution filter with a stride of 1 pixel. In this section, we extend our methodology to include implementing convolution filters with arbitrary stride configurations. Designing convolution filters with stride configurations requires a certain understanding of multi-rate signal processing; consequently, we will review some

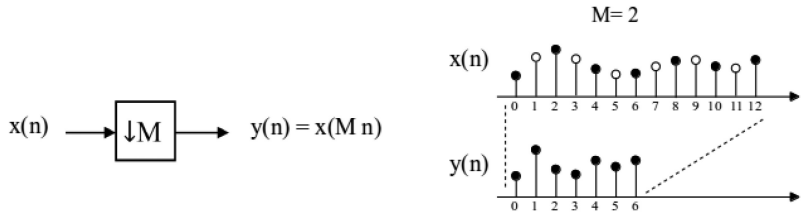
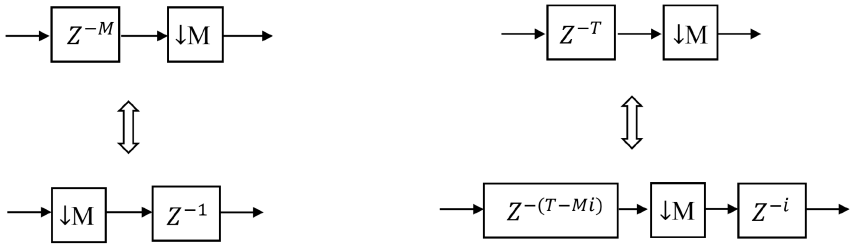


Fig. 8. A symbolic notation for the compressor and a typical example with $M = 2$.



(a) The first Noble Identity (b) Our proposed transformation

Fig. 9. Multi-rate signal processing transformations.

of the notions of multi-rate processing, and then propose a multi-rate-based transformation technique that helps in mapping a 2D convolution with an arbitrary stride configuration to a hardware realization.

4.3.1 Multi-rate Processing Perspective for Strides in ConvNets. In DSP systems, it is possible to reduce the sampling rate of a sequence by an integer factor M by using a sampling rate compressor. The compressor performs periodic sampling of the original sequence at time instances that are multiples of M. To increase the rate, however, a sampling rate expander is used along with a low-pass filter. In this work, we limit our scope to studying the sampling rate compressor as it can serve our interest in deriving hardware architectures for convolution filters with multiple strides. Figure 8 shows the notation of the sample rate compressor and illustrates a typical example of how the compressor reduces the sampling rate of a 1D sequence. We are going to use this notation to represent our proposed high-level transformation and reduction scheme.

Multi-rate signal processing generally refers to a set of techniques that utilizes sample rate compressors and expanders to reduce the computational cost of some multi-rate signal processing systems and consequently improving their efficiency. Two important results from multi-rate processing are the Noble Identities, which allow interchanging of filtering and compressing or expanding operations and the polyphase decomposition of a filter's impulse response. Reference [28] provides a complete treaty on Noble identities and polyphase decomposition. Figure 9(a) illustrates the first Noble identity.

4.3.2 Efficient Multi-rate-based Transformation for Strides. The Noble identity shown in Figure 9(a), is used for the analysis of multi-rate systems. M delay elements can be transferred from the input of an M-fold rate compressor to its output as 1 delay element, as illustrated in Figure 9(a). We propose a generalized form of this identity in Figure 9(b). Given T delay elements at the input of an M-fold rate compressor, we can transfer $M \times i$ of these elements from the input of the compressor to its output as i delay elements. In this proposed identity, T, M and i can be any combination of natural numbers. We will show in Section 4.3.3, that the identity illustrated in

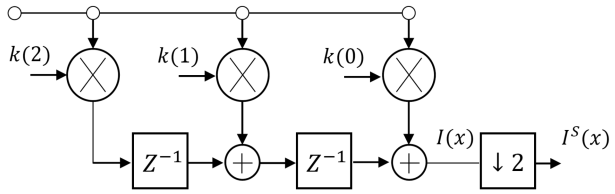


Fig. 10. A block diagram representation for a 1D convolution with a stride of 2 samples.

Figure 9(b) can be very useful in deriving efficient hardware architectures for convolution filters with strides.

4.3.3 Designing 2D Convolution Kernels with Strides. In this section, we tackle the general case of s_h stride, where the convolution mask can jump s_h samples at a time, where s_h is an arbitrary positive integer greater than or equal to 1. A 1D convolution with horizontal strides s_h is defined as

$$I^S(x) = \sum_p K(p) \times I(s_h x - p). \tag{7}$$

Note that we can break down Equation (7) into two steps; the first step consists of performing a normal 1D convolution with a stride of 1 sample as shown in Equation (8):

$$I(x) = \sum_p K(p) \times I(x - p). \tag{8}$$

The next step consists of down-sampling $I(x)$ to obtain $I^S(x)$:

$$I^S(x) = I(s_h \times x). \tag{9}$$

Equation (9) is an s_h-fold sample rate compressor. By employing an s_h-fold compressor at the output of the normal 2D convolution, we obtain the same output as that of a convolution with a horizontal stride s_h. Knowing this fact, a block diagram representation for a convolution operation with a stride configuration s_h can be drawn. Figure 10 illustrates a block diagram representation for a 1D convolution filter with a convolution mask of size 3 and a stride of 2 samples.

A direct implementation of the arrangement shown in Figure 10 is clearly inefficient, since half of the samples computed by the FIR filter are dropped by the compressor. We employ the transformation shown the Figure 9(b) to convert the above diagram to a more efficient data-flow representation in which unneeded computations are avoided. Then, we fold the data-flow representation to derive an efficient hardware implementation for the 1D convolution with a stride of 2 samples.

The final adder at the output of the FIR filter (refer to Figure 10) and the compressor can be interchanged. The result of interchanging the order of the compressor and the adder is shown in Figure 11. Note that the single delay element followed by the twofold compressor, designated using a dotted rectangle in Figure 11, is a special case of the proposed transformation shown in Figure 9(b) with $T = 1, M = 2$, and $i = 1$. Hence, this arrangement can be replaced by a unit advance element, followed by a twofold compressor, and a unit delay element as shown in Figure 9(b).

Similarly, the unit advance element can be moved to the inputs of the second adder. The unit advance element cancels the effect of the second unit delay element in the FIR filter. And the compressor can be moved to the input of the second adder in a similar fashion. After the conversion, we end up with the diagram shown in Figure 12.

The diagram in Figure 12 consists of two components: a polyphase decomposition unit and a computation unit. The polyphase decomposition unit, designated with a dotted rectangle in Figure 12, performs serial to parallel conversion, meaning that it receives 2 samples from the input

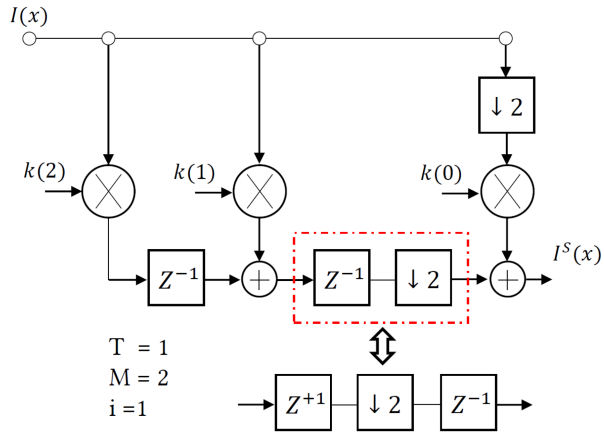


Fig. 11. Partially transformed 1D convolution with a stride of 2.

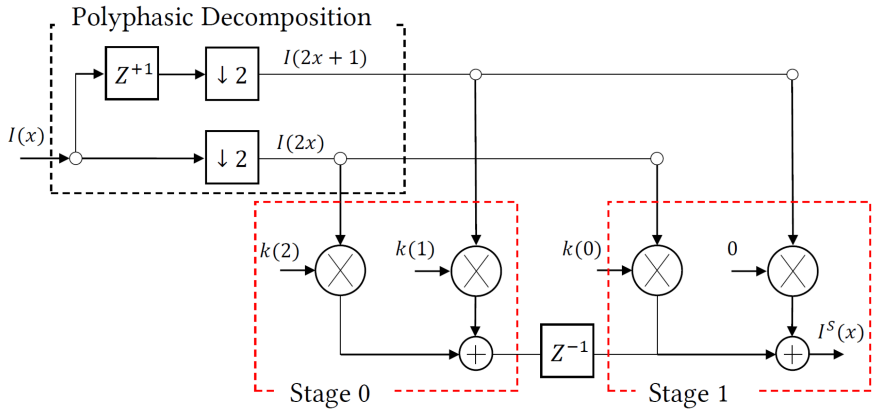


Fig. 12. Fully transformed 1D convolution.

sequence in a serial fashion and delivers them in parallel to the computation unit. The computation unit is made of two pipeline stages of computations and each stage consists of two multiplications and two additions. We designated each stage with a dotted rectangle in Figure 12. It is worth noting here, that the multipliers in each stage cannot operate at every rising or falling edge of the clock, due to the serial-to-parallel converter, which delivers 2 samples every other clock cycle. Hence, in the final transformation, we remove the serial to parallel converter and collapse the multipliers that are within the same stage into a single multiplier. The resulting block diagram is shown in Figure 13, and it can be directly mapped into a hardware implementation. Note that the convolution weights shown in Figure 13 are delivered to the multipliers in a pre-defined pattern that depends on the convolution's mask size, stride configuration, and padding parameters. We employed shift registers (shown in Figure 13) to store the parameters on the FPGA. The shift registers exhibit a control unit that ensures its proper and synchronized operation. Moreover, a weight-loading mechanism is implemented to enable an external circuit to load a new set of weights whenever a new convolution operation is scheduled.

The hardware implementation shown in Figure 13, consists of two hardware multipliers, two adders, two multiplexers, and two registers. Compared to the direct implementation, shown in

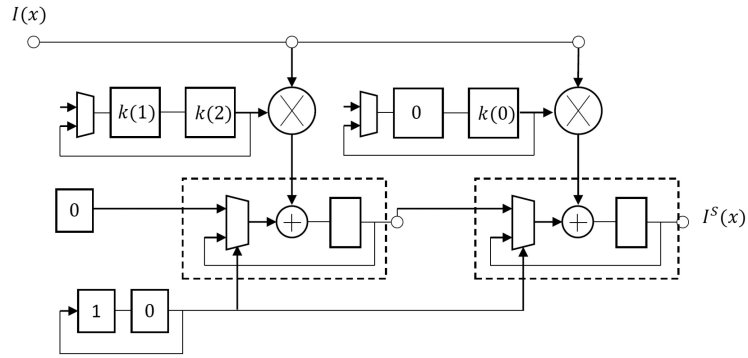


Fig. 13. A minimal hardware implementation for a 1D convolution filter with a size of 3 and a stride of 2.

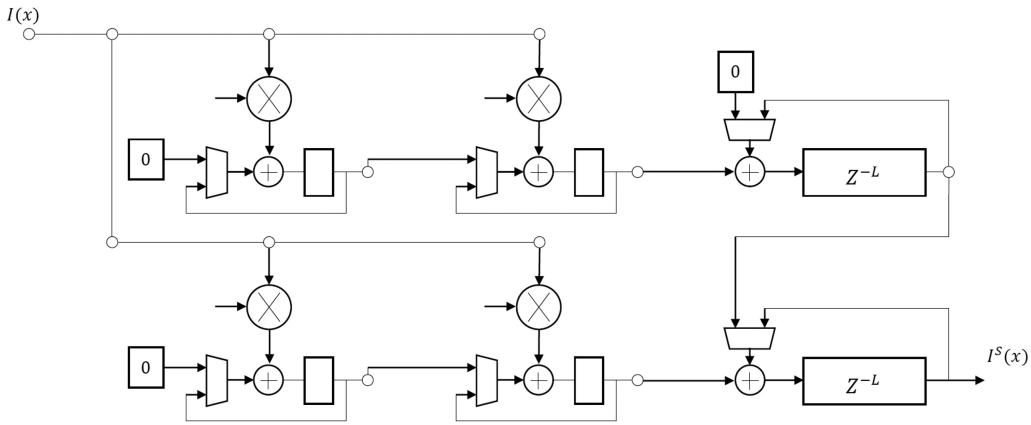


Fig. 14. A minimal hardware implementation for a 2D convolution filter with a size of 3×3 and a stride of 2.

Table 2. Hardware Costs of Different 2D FIR Filters with Different Strides

FIR window size and stride	Latency (clock cycle)	Area without the reduction scheme	Area with the reduction scheme	Reduction in area × latency
3×3, stride = 2	$N \times N$	9	4	2.25×
5×55, stride = 2	$N \times N$	25	9	2.78×
7×7, stride = 2	$N \times N$	49	16	3.1×
11×11, stride = 4	$N \times N$	121	9	13.4×

Figure 10, this reduced implementation saves one hardware multiplier. As with the case in Section 4.2.1, we can extend the 1D convolution with arbitrary strides to 2D. Figure 14, shows a 2D convolution filter with a mask of size 3×3 and a stride of 2 in each direction.

4.3.4 Analyzing the Effect of the Reduction Scheme. A single 2D convolution filter designed according to the reduction scheme proposed in Section 4.3.3 provides a certain level of performance or latency, while occupying a predictable amount of FPGA resources. As mentioned in Section 4.2.1, the filters designed here can process at least 1 input sample on every clock cycle. Consequently, a filter can process an input feature map of size $N \times N$ in around $N \times N$ clock cycles. Table 2 shows

the amount of resources or silicon area occupied by different convolution filter designs with and without employing the reduction scheme. The table also shows the latency for computing the 2D convolution of an input feature map of size $N \times N$. Note here that we only report multiplier count as a rough estimate of the silicon area. Table 2, shows that up to 13× reduction in the area-latency product can be achieved for filters with a mask of size 11×11 and a stride of 4. In general, the reduction in area-latency becomes more pronounced with convolution filter designs with larger stride configurations.

Although the proposed reduction scheme resembles the Winograd technique for the fast computation of the convolution operation in that both schemes can be used to reduce the number of multipliers in an implementation, there are subtle differences between the two. While the Winograd technique can be exclusively used when the convolution stride configuration is 1 [4], the proposed reduction scheme is tailored only for convolutions with stride configurations larger than 1.

4.4 Optimizing for Finite Word-length Representations and Computations

Deep Convolutional Neural Networks are well known for their resilience and their ability to cope with reduced precision arithmetic, especially during inference [1, 13, 38]. Leveraging this property offers the advantage of reducing FPGA logic resources, which inherently leads to a reduction in energy and power consumption. Moreover, quantizing the ConvNet model parameters reduces the storage and memory bandwidth requirements of the deployed ConvNet model [21]. To implement a resource and energy efficient hardware design, we employ limited-precision fixed-point arithmetic. In this section, we describe the methodology we used to implement our design in fixed-point, and we evaluate a variable-precision fixed-point implementation for AlexNet.

4.4.1 Effect of Finite Word-length Representation of Model Parameters. Before we investigate the effect of fixed-point arithmetic, we begin by analyzing the effect of the finite precision fixed-point representation of network parameters on the accuracy of the classifier. We use AlexNet to illustrate the methodology, but the same approach can be employed with any other neural network classifier. To investigate this effect, we use software-based fixed-point simulations, in which we run a fixed-point software implementation of the algorithm on a pre-determined dataset. We use a pre-trained Caffe-compatible AlexNet model obtained from Reference [6]. The images used in the simulation are randomly selected from the ILSVRC-2012 dataset. We evaluate two metrics: the top-1 and top-5 accuracies [32]. Note that in this work we use the 2's complement format and we adopt the notation employed in Reference [13] to characterize the structure of a fixed-point representation. That is, an (n, m) fixed-point format would denote a representation that uses n bits to represent integers and m bits to represent fractions. Note that in the standard notation, m can only be a non-negative number; in this work, however, we introduce the negative m notation. If m is negative, then the fractional part of the representation is first rounded and eliminated completely, and then the least m significant bits of the rounded integer number are removed from the representation as they are considered implied 0 bits. We also observe that AlexNet parameters are always between -1 and 1, hence a suitable fixed-point representation for them is $(1, m)$.

Figure 15 shows the top-1 and top-5 accuracies for different values of m. The top-5 and top-1 accuracies of the floating-point implementation for the same dataset used with the fixed-point models are indicated by dashed horizontal lines on Figure 15. Note that the top-5 accuracy of AlexNet with fixed-point parameters declines only when the number of fractional bits is less than 6. Given that all parameters are between -1 and 1, we can use a global fixed-point representation $(1, 7)$ for all the parameters of AlexNet without impacting the top-5 and top-1 accuracies.

4.4.2 Effect of Fixed-point Arithmetic. One of our aims in this work is to reduce hardware complexity by minimizing $n + m$ without impacting the accuracy of the classifier. This trade-off

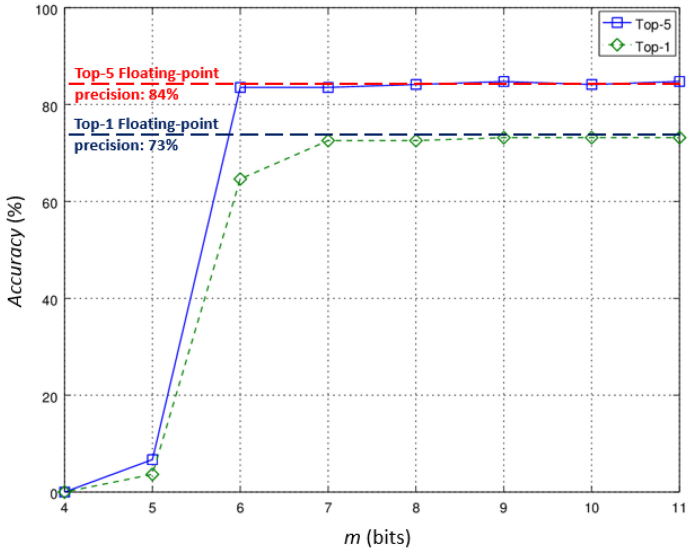


Fig. 15. The effect of fixed-point representation of network parameters on the accuracy of AlexNet.

Table 3. Possible Fixed-point Representations for Every Layer in AlexNet

Layer	Minimum	Peak	n	m	$n+m$
Data	−123	151	9	−1	8
Conv1	−2524	2587	13	{3, 11, 19}	{16, 24, 32}
Norm1	0	139	9	{7, 15, 23}	{16, 24, 32}
Conv2	−923	645	11	{5, 13, 21}	{16, 24, 32}
Norm2	0	139	9	{7, 15, 23}	{16, 24, 32}
Conv3	−489	431	10	{6, 14, 22}	{16, 24, 32}
Conv4	−252	284	10	{6, 24, 22}	{16, 24, 32}
Conv5	−189	207	9	{7, 15, 23}	{16, 24, 32}
FC6	−104	69	9	{0, 8, 16, 24}	{8, 16, 24, 32}
FC7	−31	18	7	{1, 9, 17, 25}	{8, 16, 24, 32}
FC8	−10	35	7	{1, 9, 17, 25}	{8, 16, 24, 32}

between accuracy and hardware complexity is best tackled using the multiple word-length paradigm [16]. In this paradigm, every computational stage in AlexNet inference is assigned a different fixed-point representation instead of a system-wide representation. The reasoning behind this paradigm is twofold: (1) signals in different computational stages have different dynamic ranges and thus require different representations, and (2) every computational stage or layer in AlexNet inference contributes differently to the output stage of the classifier. Table 3 shows the dynamic ranges of different layers in AlexNet and n.

Note that data movement circuits, i.e., circuits that move data between external memory and the FPGA, are much simpler when the word-length representations of data samples are multiples of 8-bits, since typical on-board memories are byte addressable. Hence, the number of fractional bits, m, is chosen in such a way as to keep the total word-length in each layer a multiple of 8-bits.

Table 4. Evaluation Platforms

Platform	ZedBoard™	Cyclone® V Dev-kit
Manufacturer	Avnet™	Intel®/Altera®
On-board RAM	512MB	384MB + 512MB
FPGA	Zynq 7020	Cyclone V GT D9
Block RAMs	140 (5,160Kb)	1220 (12,200Kb)
DSP Units	240 DSP48E	342 DSP
Lookup tables	53,200	113,560
Process Technology	28nm HPL	28nm LP

To find the best combination of m values, we resort to fixed-point simulations. A brute force approach might consist of simulating all possible combinations of values for m and select the combination that gives the best classification accuracy. However, this approach is costly, as there are 139,968 possible combinations. We use a heuristic method to find a minimal solution in terms of hardware cost while keeping the top-5 accuracy at an acceptable range. The reasoning behind this method is that the first few layers are the most critical, and hence we may start by gradually reducing the fractional part of the first layer until the accuracy starts dropping. Then, we shift to the next layer and start reducing its fractional part until the accuracy starts dropping again. This procedure is repeated until we reach the last layer. Following this procedure, we found that the following combination of values of m (−1, 3, 7, 5, 7, 6, 6, 7, 8, 9, 9) does not degrade the accuracy of the classifier. We have used this combination of fixed-point representations for the different layers in our minimal implementation in the Evaluation and results section.

5 EVALUATION AND RESULTS

5.1 Evaluation and Experimental Setup

5.1.1 Evaluation Platforms. As mentioned earlier, in this work, we aim at deploying ConvNets on low-end resource-limited FPGA platforms. For this purpose, we decided to deploy our accelerator design described in Section 3 on two low-end FPGA platforms: The Xilinx Zynq 7020 SoC device [40], and the Intel Altera Cyclone V GT device [19]. We used the "Zynq Evaluation and Development" Board (ZedBoard) from Avnet [3] to target the Zynq 7020 FPGA. For the Cyclone V FPGA, we used the Cyclone V Development Kit [20]. Table 4 summarizes the available resources and the key differences between the two platforms.

5.1.2 Evaluation Network and Dataset. We deployed a pre-trained Caffe-compatible AlexNet model from the Caffe model Zoo [6]. Note that Caffe [23] is a deep-learning software framework developed by the UC Berkeley Artificial Intelligence Lab. This framework is typically used to train and deploy deep neural networks. The pre-trained AlexNet model is trained on the ILSVRC-2012 [32] dataset. We sample a random selection of images from the ILSVRC-2012 validation dataset to test our accelerator.

5.1.3 Evaluation Metrics. To evaluate the performance of our design, we measured seven different performance metrics: FPGA resource utilization, end-to-end latency measured in terms of milliseconds (ms), throughput measured in terms of frame per second (frame/s), performance measured in terms of Giga Operations per Second (GOPs), Energy efficiency measured in terms of Joules per frame (J/frame), Energy efficiency per cost measured in terms of Frame per Joule per Dollar ([frame/J]/Dollar), and finally the total board power consumption measured in terms of Watts.

Table 5. Resource Utilization for AlexNet on ZedBoard and Cyclone V

Platform	ZedBoard™	Cyclone® V
DSP	55/220 (25%)	53/342 (15%)
BRAM	49/140 (35%)	322/1220 (26%)
Lookup Tables	16,536/53,200 (31%)	20,898/113,560 (18%)
Flip Flops	31,976/106,400 (30%)	83,592/454,240 (18%)

Table 6. Performance and Energy Results of AlexNet Implementation Without the On-chip Cache

Platform	ZedBoard™	Cyclone® V Dev-kit
Operating Frequency	100MHz	150MHz
Latency (ms)	1,332.4	1,666
Throughput	0.75 Frame/s	0.6 Frame/s
Energy (J/Frame)	1.918	1.899
Performance (GOPs)	0.543	0.434
Performance Density	3.28×10^{-5} GOPs/LUT	2.08×10^{-5} GOPs/LUT

5.2 Results

5.2.1 Performance of Basic Design. First, we implemented the architecture shown in Figure 2, but without including the on-chip cache memory, which buffers the intermediate feature maps on FPGA Block-RAMs. The buffering scheme was presented in Section 3. The implementation employs the fixed-point representation scheme derived in Section 4.4.2. We collected resource utilization, performance, and energy results on the two low-end FPGA platforms described in 5.1.1. Table 5 shows the resource utilization results on both platforms.

Note that the resource utilization percentages are on average around 30% of all the available resources on the FPGA fabric in both platforms. The performance and energy per frame results are shown in Table 6. Without enabling the on-chip cache memory the execution time of AlexNet is around 1.3s on the ZedBoard and around 1.6s on the Cyclone V dev-kit. This is mainly due to the fact that each individual data-flow computational engine needs to reuse the input feature maps multiple times while computing the output feature maps. Moreover, without the on-chip cache component, input features are buffered on the External Memory (refer to Figure 2), which is much slower than the on-chip cache component.

5.2.2 Performance of the Accelerator when the Caching Components are Enabled. Table 8 shows the measured latency of each computational stage in AlexNet, when the on-chip cache memory, described in Section 3, is included in the design. Note that the end-to-end latency is reduced from 1.3 to 0.6s on the ZedBoard, and from 1.6 to 0.4s on the Cyclone V development kit. The reason for this reduction in the individual latencies of each stage is that enabling the on-chip cache memory allows the intermediate input feature maps to be buffered in the on-chip cache sub-components (refer to Section 3), and thus every convolutional engine can now consistently read 1 sample from the cache sub-components on every clock cycle (refer to Figure 2 and to Section 3); consequently, the latency of convolutional stages is equal to $N \times N \times I_f \times I_o/G$ clock cycles, where $N \times N$ is the size of the input feature, I_f the number of input feature maps, I_o the number of output feature maps, and G the group hyper-parameter; and the latency of fully connected stages is equal to

Table 7. Latency of the Individual Compute Stages of AlexNet with the On-chip Cache Memory Included

Platform	ZedBoard™	Cyclone® V Dev-kit
First Stage	148ms	99ms
Second Stage	89ms	60ms
Third Stage	166ms	110ms
Fourth Stage	124ms	83ms
Fifth Stage	83ms	55ms
FC6 Stage	8.8ms	7.5ms
FC7 Stage	3.95ms	3.35ms
FC8 Stage	0.96ms	0.81ms
Total	623.71ms	418.66ms

Table 8. Comparison with Other Implementations

Reference	Zhang et al. [43]	Aydonat et al. [4]	FeatherNet on ZedBoard™	FeatherNet on Cyclone® V
FPGA Chip	Virtex7 VX485T	Arria 10-1150	Zynq ZC7020	Cyclone V GTD9
Frequency	100MHz	303MHz	100MHz	150MHz
DSP Capacity	2800 DSP	1,518 DSP	240 DSP	342 DSP
Performance	61.62 GOPS	1,382 GOPS	4.358 GOPS	6.516 GOPS
Frame/s	46 (Conv. Layers)	1,020	6	9
Latency (ms)	21.61	0.98	621.23	418.66
Top-5 Accuracy	Not Reported	79%	83.58%	83.58%
Energy/Frame	0.4 J/Frame	0.043 J/Frame	0.239 J/Frame	0.126 J/Frame
Frame/J	2.5 Frame/J	23.2 Frame/J	4.18 Frame/J	7.9 Frame/J
Frame/J/$	7.15×10^{-4}	5.17×10^{-3}	9.31×10^{-3}	4.42×10^{-3}
Board Power	18.61W	45W	2.95W	8.5W

$N_I \times N_O$ memory cycles, where N_I is the number of input neurons, and N_O is the number of output neurons. Moreover, as mentioned Section 3, the on-chip cache sub-components are configured as independently accessible simple dual-port RAM memories. this choice of cache organization allows the different data-flow computational engines to operate in parallel. This temporal parallelism can be exploited to establish an execution pipeline made of the different computational stages of AlexNet. Consequently, the accelerator can run at the rate of the slowest computational stage, i.e., at 6 frames/s on the ZedBoard and at 9 frames/s on the Cyclone V.

5.2.3 Comparison to Other Implementations. In this section, we make a quantitative comparison between our design and two other FPGA-based accelerator implementations from the literature that are based on AlexNet. We reported performance, accuracy, board power, energy, and energy efficiency per cost measurements in Table 8. The first design we are comparing against is from the work of Zhang et al. [43], which deploys their accelerator on a high-end Xilinx Virtex7 FPGA. This implementation employs a MicroBlaze soft processor core to assist with accelerator startup, communication with the host computer, and with time measurements. The second implementation is from the work of Aydonat et al. [4], which is the current state-of-the-art. The platform used in Reference [4] is a high-end Arria 10-1150 platform that has a PCIe interface connecting it to a

host computer, which controls all memory transfers and kernel executions using the Intel SDK for OpenCL.

5.2.4 Discussion and Analysis of Results. The minimalist accelerated ConvNet system architecture shown in Figure 2 was implemented on two different resource-constrained FPGA platforms: The ZedBoard and the Cyclone V Development kit. In terms of suitability for deployment in IoT device, the designs in References [4, 43] use a relatively large number of computational resources and cannot fit on low-end resource-constrained FPGA devices needed for IoT deployment, but rather they fit for data-center deployment. Furthermore, deploying the FPGA devices presented in References [4, 43] in IoT devices is not an option, since they consume a lot of power (18.61W in Reference [43] and 45W in Reference [4]), which can quickly drain the small battery that powers the IoT device. However, both of our implementations, i.e., our implementation on the ZedBoard and on the Cyclone V platform, consumes significantly less power (2.95 and 8.5W, respectively) making them better suited for IoT applications with limited power sources than References [4, 43]. Furthermore, to reflect the tradeoff between performance and platforms' costs, we compare our implementations to References [4, 43] using the "Frame per unit Energy per Cost (measured in frames/joule/dollar)" as our performance metric. Our ZedBoard implementation achieves significantly better than both References [4, 43] in terms of energy efficiency per cost, which is measured in terms of Frames per Joule per Dollar and which reflects the good tradeoff between performance and the costs of ZedBoard. Our Cyclone V implementation also achieves better performance than Reference [43] and comparable performance to Reference [4]. Finally, it is worth mentioning that, as indicated in Table 6, our implementations on both resource-constrained platforms uses only 25% of the DSP resource on the ZedBoard and 15% of the DSP resources on the Cyclone V. This provides an opportunity to further improve performance by leveraging additional resources to improve frame rates or reduce power and energy consumption by migrating to smaller devices. However, we leave the exploration of these two options for a future work.

6 CONCLUSION

This article presents an efficient methodology for mapping large convolutional neural networks to resource-constrained FPGA platforms targeted at IoT deployment; we demonstrated this framework by building a minimalist accelerator design for AlexNet. Our results demonstrated the success of addressing the design challenges and achieving low (30%) resource utilization for the lower-end FPGA platforms: Zedboard [3] and Cyclone V [20]. The design overcame the limitation of designs targeted for high-end platforms that cannot fit on low-end IoT platforms [5, 22, 36]. Furthermore, our design showed superior performance results (measured in terms of performance/W/dollar) compared to high-end optimized designs (9.31/10/3 Frame/J/$ compared to 5.17/10/3 for the state-of-the-art [4]). The designs also showed accurate results consistent with non-accelerated designs, where images from the ILSVRC-2012 dataset were classified with a top-5 accuracy of 83.58% (compared to 79% in Reference [4]) while achieving the low-end FPGA benefits of improved energy efficiency per cost at 9.31/10/3 Frame/J/$ and a frame rate of up to 9 frames/s.

REFERENCES

[1] Kamel Abdelouahab, Maxime Pelcat, Jocelyn Serot, Cedric Bourrasset, and Francois Berry. 2017. Tactics to directly map CNN graphs on embedded FPGAs. *IEEE Embed. Syst. Lett.* 9, 4 (Dec. 2017), 113–116. DOI: https://doi.org/10.1109/LES.2017.2743247

[2] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE Press, Piscataway, NJ. Retrieved from http://dl.acm.org/citation.cfm?id=3195638.3195664.

[3] Avnet. 2017. ZedBoard. Retrieved from http://zedboard.org/product/zedboard.

[4] Utku Aydonat, Shane O'Connell, Davor Capalija, Andrew C. Ling, and Gordon R. Chiu. 2017. An OpenCL deep-learning accelerator on Arria 10. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*. ACM, New York, NY, 55–64. DOI: https://doi.org/10.1145/3020078.3021738

[5] David Blaauw, Dennis Sylvester, Prabal Dutta, Yoonmyung Lee, Inhee Lee, Suyoung Bang, Yejoong Kim, Gyouho Kim, Pat Pannuto, Ye sheng Kuo, Dongmin Yoon, Wanyeong Jung, ZhiYoong Foo, Yen-Po Chen, Sechang Oh, Seokhyeon Jeong, and Mun Ho Choi. 2014. IoT design space challenges: Circuits and systems. In *Proceedings of the Symposium on VLSI Technology (VLSITechnology'14)*. IEEE, 1–2. DOI: https://doi.org/10.1109/VLSIT.2014.6894411

[6] BVLC. 2001. Model Zoo. Retrieved from http://ccrma.stanford.edu/~jos/bayes/bayes.html.

[7] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2016. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or −1. Retrieved from http://arxiv.org/abs/1602.02830.

[8] Alpha Data. 2017. An Open Source FPGA CNN Library. Retrieved from ftp://ftp.alpha-data.com/pub/appnotes/cnn/ad-an-0055_v1_0.pdf.

[9] Dan E. Dudgeon and Russell M. Mersereau. 1983. *Multidimensional Digital Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ.

[10] Clément Farabet, Berin Martini, Polina Akselrod, Selçuk Talay, Yann LeCun, and Eugenio Culurciello. 2010. Hardware accelerated convolutional neural networks for synthetic vision systems. In *Proceedings of the IEEE International Symposium on Circuits and Systems*. IEEE, 257–260. DOI: https://doi.org/10.1109/ISCAS.2010.5537908

[11] Clément Farabet, Berin Martini, Benoit Corda, Polina Akselrod, Eugenio Culurciello, and Yann LeCun. 2011. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops (CVPRW'11)*. IEEE, 109–116. DOI: https://doi.org/10.1109/CVPRW.2011.5981829

[12] Clément Farabet, Cyril Poulet, Jefferson Y. Han, and Yann LeCun. 2009. CNP: An FPGA-based processor for convolutional networks. In *Proceedings of the International Conference on Field Programmable Logic and Applications*. IEEE, 32–37. DOI: https://doi.org/10.1109/FPL.2009.5272559

[13] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. 2015. Deep learning with limited numerical precision. In *Proceedings of the 32nd International Conference on International Conference on Machine Learning (ICML'15)*. JMLR.org, 1737–1746. Retrieved from http://dl.acm.org/citation.cfm?id=3045118.3045303.

[14] Philipp M. Gysel. 2016. *Ristretto: Hardware-oriented approximation of convolutional neural networks*. Master's thesis. University of California, Davis, Davis, CA.

[15] Song Han, Huizi Mao, and William J. Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding. Retrieved from http://arxiv.org/abs/1510.00149.

[16] Scott Hauck and Andre DeHon. 2007. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA.

[17] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. 2017. Quantized neural networks: Training neural networks with low precision weights and activations. *J. Mach. Learn. Res.* 18, 1 (Jan. 2017), 6869–6898. Retrieved from http://dl.acm.org/citation.cfm?id=3122009.3242044.

[18] Intel. 2014. Intel SDK for OpenCL Applications. Retrieved from https://software.intel.com/en-us/intel-opencl.

[19] Intel/Altera. 2017. Cyclone V. Retrieved from https://www.altera.com/products/fpga/cyclone-series/cyclone-v/overview.html.

[20] Intel/Altera. 2017. Cyclone V-GX FPGA Development Kit. Retrieved from https://www.altera.com/products/boards_and_kits/dev-kits/altera/kit-cyclone-v-gx.html.

[21] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew G. Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. IEEE, 2704–2713. DOI: https://doi.org/10.1109/CVPR.2018.00286

[22] Hrishikesh Jayakumar, Kangwoo Lee, Woo Suk Lee, Arnab Raha, Younghyun Kim, and Vijay Raghunathan. 2014. Powering the internet of things. In *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED'14)*. ACM, New York, NY, 375–380. DOI: https://doi.org/10.1145/2627369.2631644

[23] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM International Conference on Multimedia (MM'14)*. ACM, New York, NY, 675–678. DOI: https://doi.org/10.1145/2647868.2654889

[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 1097–1105. Retrieved from http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (May 2015), 436. DOI: https://doi.org/10.1038/nature14539

[26] Fengfu Li and Bin Liu. 2016. Ternary weight networks. Retrieved from http://arxiv.org/abs/1605.04711.

[27] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed point quantization of deep convolutional networks. In *Proceedings of the 33rd International Conference on Machine Learning (ICML'16)*. JMLR.org, 2849–2858. Retrieved from http://dl.acm.org/citation.cfm?id=3045390.3045690.

[28] Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. 1999. *Discrete-time Signal Processing* (2nd Ed.). Prentice-Hall, Inc., Upper Saddle River, NJ.

[29] Keshab K. Parhi. 1999. *VLSI Digital Signal Processing Systems Design and Implementation*. Wiley & Songs, Inc., New York, NY.

[30] Peng Peng, You Mingyu, and Xu Weisheng. 2017. Running 8-bit dynamic fixed-point convolutional neural network on low-cost ARM platforms. In *Proceedings of the Chinese Automation Congress (CAC'17)*. IEEE, 4564–4568. DOI: https://doi.org/10.1109/CAC.2017.8243585

[31] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. 2016. XNOR-Net: ImageNet classification using binary convolutional neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV'16)*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer International Publishing, Cham, 525–542. DOI: https://doi.org/10.1007/978-3-319-46493-0_32

[32] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. 2015. ImageNet large scale visual recognition challenge. *Int. J. Comput. Vision* 115, 3 (Dec. 2015), 211–252. DOI: https://doi.org/10.1007/s11263-015-0816-y

[33] 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 *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*. IEEE Press, Piscataway, NJ. Retrieved from http://dl.acm.org/citation.cfm?id=3195638.3195659.

[34] Yongming Shen, Michael Ferdman, and Peter Milder. 2017. Maximizing CNN accelerator efficiency through resource partitioning. *SIGARCH Comput. Archit. News* 45, 2 (June 2017), 535–547. DOI: https://doi.org/10.1145/3140659.3080221

[35] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. Retrieved from http://arxiv.org/abs/1409.1556.

[36] Hemendra Singh. 2018. How Much Does it Cost to Develop an IoT Application? Retrieved from http://customerthink.com/how-much-does-it-cost-to-develop-an-iot-application/.

[37] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. IEEE, 1–9. DOI: https://doi.org/10.1109/CVPR.2015.7298594

[38] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. 2011. Improving the speed of neural networks on CPUs. In *Proceedings of the Deep Learning and Unsupervised Feature Learning NIPS Workshop*, Vol. 1. Citeseer, 4.

[39] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (Apr. 2009), 65–76. DOI: https://doi.org/10.1145/1498765.1498785

[40] Xilinx. 2017. Zynq-7000: All Programmable SoC with Hardware and Software Programmability. Retrieved from https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html.

[41] Xilinx. 2018. Vivado High-Level Synthesis. Retrieved from https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html.

[42] Matthew D. Zeiler and Rob Fergus. 2014. Visualizing and understanding convolutional networks. In *Proceedings of the European Conference on Computer Vision (ECCV'14)*. Springer, Springer International Publishing, 818–833. DOI: https://doi.org/10.1007/978-3-319-10590-1_53

[43] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. 2015. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'15)*. ACM, New York, NY, 161–170. DOI: https://doi.org/10.1145/2684746.2689060

[44] Shuchang Zhou, Yuxin Wu, Zekun Ni, Xinyu Zhou, He Wen, and Yuheng Zou. 2016. Dorefa-net: Training low bitwidth convolutional neural networks with low bitwidth gradients. Retrieved from http://arxiv.org/abs/1606.06160.

Received November 2017; revised November 2018; accepted January 2019