SCALABLE HARDWARE ACCELERATOR DESIGN FOR FPGA PLATFORMS

by

Ushma Sunil Bharucha

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in
Electrical Engineering

Charlotte

2020

Approved by:

Dr. Hamed Tabkhivayghan

Dr. Fareena Saqib

Dr. Andrew Willis

ABSTRACT

USHMA SUNIL BHARUCHA. Scalable Hardware Accelerator Design for FPGA Platforms. (Under the direction of DR. HAMED TABKHIVAYGHAN)

Deep Learning revolutionized the field of computer vision when convolutional neural networks (CNNs) solved complex computer vision problems with promising developments in the areas of research in artificial intelligence (AI). The progress in AI has attracted the hardware community to accommodate the growing demand of computationally expensive state-of-the-art Deep CNNs, coupled with diminishing performance gains of general-purpose architectures, which has fueled the need for specialized and scalable hardware accelerator designs and architectures for Deep CNNs. Moreover, Deep Separable Convolutional Neural Networks (DSCNNs) has become an emerging paradigm in the field of computer vision by offering modular networks with structural sparsity to achieve higher accuracy with relatively lower operations and parameters. However, there is a lack of customized architectures that can provide flexible solutions that fit the sparsity of the DSCNNs. The purpose of the domain-specific accelerators is to satisfy two requirements: (1) execution of DSCNN models with low latency, high throughput, and high efficiency, and (2) flexibility to accommodate evolving state-of-the-art models like EfficientNet families without costly silicon updates. On this note, the state-of-the-art GPUs tend to be too power-hungry and ASICs are too inflexible. This is where FPGA shines due to its architectural reconfigurability, ability to accommodate custom datatypes, and process irregular parallelism, power efficiency, and low latency which extends its usability in real-time. This work proposes DeepDive, which is a fully-functional, vertical hardware-software co-design architecture for the power-efficient implementation of DSCNNs on both edge and cloud FPGA platforms. With two different architectural principles applied for DeepDive's implementation on edge and cloud, the architecture for the former demonstrates latency-orient design

whereas the architecture for the later demonstrates a throughput-orient design each of which is designed to fully support DSCNNs with various convolutional operators interconnected with structural sparsity. The accelerator design for both introduces parameterized, configurable, and scalable compute units that can be configured based on the user-specific requirement depending on the hardware it is implemented on, the degree of parallelism required, and the family of DSCNN chosen for inference. This accelerator design was implemented using Xilinx Vitis HLS 2019.2 tool. The execution results for DeepDive - Edge accelerator on Xilinx ZCU102 Edge FPGA demonstrates 233.3 FPS/Watt for a compact version of EfficientNet the state-of-the-art DSCNN. These comparisons showcase how this edge design improves FPS/Watt by 2.2× and 1.51× over Jetson Nano high and low power modes, respectively. Whereas, DeepDive - Cloud accelerator achieves 87 FPS on Xilinx Alveo U50 with a power efficiency of 7.25 FPS/Watt for the baseline version of EfficientNet.

DEDICATION

To my mimo and daddy for all their support and belief that made me what I am today.

ACKNOWLEDGEMENTS

First and foremost I would like to thank my advisor Dr. Hamed Tabkhi for his constant support and trust in me. It was because of his immense guidance and directions I am at this position to present my work. He motivated me to pursue thesis due to his passion for Computer Architecture and Deep Learning. He ignited this thirst to become a researcher within me and demonstrate the best of my abilities in whatever path I pursue. He threw numerous challenges at me so that I can hone my technical and interpersonal skills. I thank Dr. Saqib and Dr. Willis for being in my thesis committee and dedicating their time and efforts to critique my work.

I would like to extend special thanks to my colleague Mohammad Reza Baharani who understood the way I work and helped me become a better hardware developer. When I started my research I was pretty naive in various aspects and he would always correct me and guide me when I made mistakes. He is not only technically sound and an all-rounder but he tried to impart those qualities to me as well. He valued my opinions and always kept an open mind for my suggestions during brainstorming sessions.

I wouldn't be writing this thesis if I didn't have two strong pillars in my life, my mom and dad. It is because of the trust they had in me that I would do something remarkable once is what has brought me here today. It was because of their constant love and support that I was able to come to UNCC and pursue my master's degree. I would also like to specially thank my partner Deepak, who provided me strength during the times when I felt down and boosted my moral when I felt like quitting. I would like to thank my roommates to whom I would rant about whenever I felt like thing weren't going the way I wanted. Finally, I would like to thank all my friends for their constant motivation and support for encouraging me to work to the best of my abilities and go an extra mile for whichever task I took up.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

CNN Convolutional Neural Network

DNN Deep Neural Networks

DPU Deep Learning Processing Unit

DSCNN Deep Separable Convolutional Neural Network

FPGA Field Programmable Gate Array

HLS High Level Synthesis

VTA Versatile Tensor Accelerator

CHAPTER 1: INTRODUCTION

With the recent advances in the study of deep convolutional neural networks, deep learning has boosted the already rapidly developing field of computer vision. This has brought about the introduction of non-traditional and efficient solutions to several problems that had long remained unsolved and are now becoming parts of our everyday lives. These include image classification Fig:1.1, object detection Fig.1.2, semantic segmentation Fig:1.3, machine vision in self-driving cars and many more. Deep learning methods are achieving state-of-the-art results on these applications. This impressive performance comes at the cost of increased workload and bulkier networks with a compute-intensive structure. Today's age of artificial intelligence (AI) has attracted the hardware community to develop customized hardware of numerous deep learning applications. Out of the currently available accelerator engines, GPUs tend to be too power-intensive, fail in the test of real-time processing and architectural flexibility. On the other hand, ASICs are faster than GPUs because they run on bare-metal but due to poor architectural flexibility are ranked lower. This is where FPGAs wins the price with their architectural flexibility in terms of reconfigurability, which has attracted hardware developers to consider FPGA as one of the top choices for developing deep learning applications. Moreover, to achieve better accuracy and accommodate the growing computer vision demand, the CNNs have become computationally intensive and bulkier. Deep Separable CNNs (DSCNNs) has emerged as an innovative algorithmic solution to achieve higher accuracy with relatively lower parameters and operations. State-of-the-art deep separable CNNs, e.g., MobileNet family and EfficientNet family, offer modular networks with structural sparsity over various convolutional operators like group, depthwise, and, pointwise convolution.

Apart from this, FPGA platforms scale from edge to cloud, however, recently introduced hardware accelerators are designed to be generic, one-size-fits-all architectures. But due to platform-specific requirements, there was a need for separate design for edge and cloud platforms. This thesis presents DeepDive a fully functional framework for an agile, power-efficient execution of DSCNNs for both edge and cloud platforms.

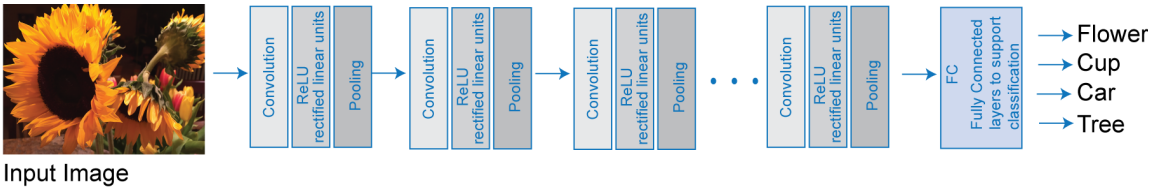

Figure 1.1: Classification

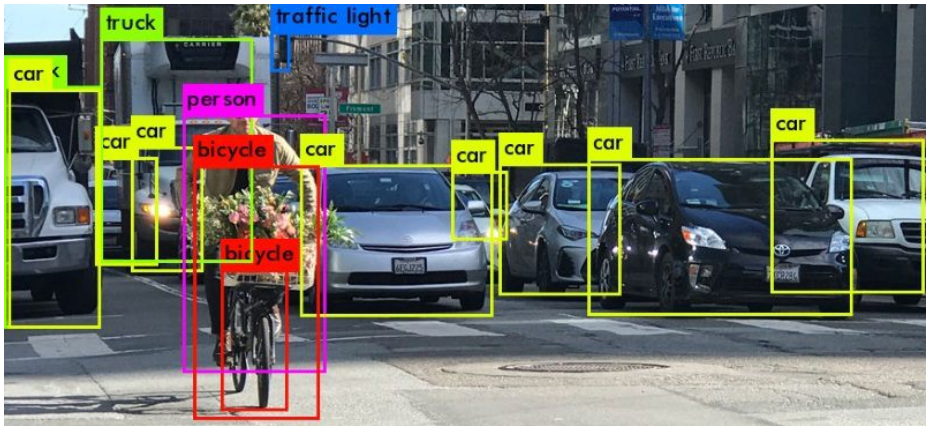

Figure 1.2: Object Detection

1.1 Motivation

If we take edge devices into consideration, we find some distinguishing characteristics which include data proximity, real-time interaction, and energy-efficient design. By pushing the data processing to the edge devices, a large amount of raw senor or camera data can be processed locally in order to produce compact and rich information. This reduces the data transfer time and thus allows a faster response time. Edge devices promote real-time inference and supplements users with data privacy which is one of the major concerns as AI grows. The power consumption dramatically reduces as we move to edge devices and thus, making it possible to achieve

Figure 1.3: Semantic Segmentation

the power-efficient design. Keeping in mind these characteristics of edge devices, it is necessary to have a latency-oriented accelerator design for edge devices. However, when we talk about cloud devices, we see that cloud platforms are blessed with an abundance of resources which makes them capable enough to process any quantity of workload. Hardware scalability and reconfigurability make it possible to design an accelerator with demonstrates high performance when compared to other available hardware accelerators. Cloud devices are expected to process larger workloads for single run supplementing higher throughput per query. Hence, it is necessary to have a throughput-oriented design for cloud devices that executes in SIMD fashion, meaning, higher data-level parallelism to achieve higher throughput.

1.2 Contribution

DeepDive is a fully functional, end-to-end vertical framework that is agile and demonstrates power-efficient execution of DSCNNs on both edge and cloud FPGA platforms. DeepDive offers a novel architecture for efficient execution of DSCNNs, combined with a vertical algorithm/architecture optimization and synthesis on different FPGA platforms. At the frontend, DeepDive receives a network description model (e.g., Pytorch, Onyx, etc.) and optimizes the model based on the FPGA-aware

training and online quantization. This includes algorithm-specific fusing of batch normalization and convolutional operators along with extreme low-bit per-channel-quantization across all separable convolution layers. The output of the front-end will be *QNet*, which contains all of the meta-data regarding the FPGA-aware trained as well as the quantized network model. Frontend is unified implementation and independent of different FPGA platforms be it edge or cloud. At the backend for the edge, the Network SoC compiler creates a customized memory path and synthesizable model of the entire hardware accelerator based on the pre-designed CUs and provided convolutional operators such as group, depthwise, and pointwise convolutions. The network SoC compiler is also responsible for generating the host CPU code running on ARM cores for synchronization and scheduling. As opposed to the edge, the backend for cloud platform consists of multiple instances of a homogeneous compute unit comprising of heterogeneous processing elements, each of which has convolutional operators with either direct convolution or GEMM-based algorithms. All the compute units are instantiated in the Super Logic Region (SLR) of cloud FPGA with each compute unit interacting with the processor via its individual High Memory Bandwidth (HBM) Channel. Below are the DeepDive specifications followed by my contributions. DeepDive Specifications:

- Scalable framework enabling optimized execution of the DSCNNs on different FPGA platforms
- Parameterized, configurable, and highly optimized convolutional operators with flexible compute core for accommodating user required parallelism
- First scalable solution with the support of recently introduced EfficientNet DSCNN families
- The vertical integration and library-based operation mapping enables true comprehensive design space exploration on FPGAs

My Contributions:

- DeepDive - Cloud:
  - Designed and developed cloud specific homogeneous compute unit design
  - Introduced GEMM based convolutional operators for processing elements
  - Separated architecture source code and configuration via cmake integration
  - Integrated the backend of DeepDive with hlslib extensions
- DeepDive - Edge:
  - As a part of collaborative effort helped in developing synthesizable compute units : Head, Body, Tail, Classifier
  - Extended support for EfficientNet in the backend for DeepDive on edge.
  - Contribute to the overall design flow of DeepDive for edge

1.3 Thesis Outline

The thesis is further organized in the following manner. Chapter 2 will discuss related works that have hardware accelerator implementations along with background about hlslib and newly introduced DSCNN family EfficientNet with some of the techniques used to reduce computation complexity in CNNs. Chapter 3 talks about the DeepDive cloud implementation and the execution dataflow of the accelerator and convolutional operators implemented in processing elements. Chapter 4 highlights DeepDive frontend and backend implementation on the edge which describes the working of Network SoC Compiler and elaborates on the four basic compute units. Chapter 5 discusses the experimental setup and results on both edge and cloud. The experimental results compare single vs multiple compute unit implementation on cloud FPGA and later compare the execution of DSCNNs on embedded GPU and FPGA. Chapter 6 concludes the thesis along with some prospective future work.

CHAPTER 2: RELATED WORKS AND BACKGROUND

2.1 Related work

Developing for FPGA is an arduous and time-consuming task because of its complexity and the infinite number of solutions. Although the advances of HLS reduces the design complexity and improve the productivity, DNN hardware realization for FPGA still needs a large amount of engineering effort. Hao et al. [1] proposed a framework, consists of *Auto-DNN*, and *Auto-HLS*, for design exploration of DNN. They introduced predefined CNN blocks, called CNN bundle, to shrink the space exploration. This decision helps them to analyze the impact of final hardware solutions on overall DNN performance and vice-versa; however, it costs them the inability of the state-of-the-art deep CNN implementation.

Modern CNN accelerators can be divided into two main categories: single compute engine [2, 3, 4, 5, 6, 7], and multiple streaming compute engines [8, 9, 10, 11, 12, 4]. Single compute-engine accelerators are typically a systolic array of processing elements (PEs). These kinds of accelerators execute the target CNN layer-by-layer sequentially. They have a versatile solution to support different CNNs with the cost of some execution deficiencies. This architecture design has a high amount of memory transactions. In contrast, streaming architectures consist of multiple dedicated hardware blocks, customized for the target CNN's layers running in producer/consumer fashion. While achieving relatively higher efficiency, they have less scalability to support different networks [13, 14].

Many recent frameworks have proposed a vertical design flow from algorithm to the hardware [2, 8, 4, 10, 15]. However, the primary focus is on optimizing classical CNNs with dense operation with regular memory access, such as YOLO and ResNet

network family. One notable example of single-engine architecture is DNNWeaver [2]. It offers customizable, hand-optimized RTL templates capable of shrinking or expanding the architecture based on the target CNN workload and target device hardware constraints. The templates support common CNN layer operations such as standard convolution, pooling, and batch normalization. However, the design-flow is not autonomous as it requires the user to define the network topology and layer structure. Wei et al. [6] designed a novel 2D systolic array that localizes data shifting to between neighboring PEs. This removes the need for multiplexers and simplifies the routing complexity, allowing for higher throughput. They also employ a custom C-based front-end, which, similar to [2], requires user interaction to define the nested convolutional loop using custom pragmas in C++. The custom frontend makes it more challenging to integrate with existing high-level DNN libraries (PyTorch, TensorFlow, Caffe, etc).

VTA is another recently introduced approach, which presents a versatile hardware solution to support different dense CNNs. Fig.2.1 shows the block diagram of the VTA. VTA enjoys the generality by adapting instruction-based scheduling and flexible systolic array. However, this generality leads to more power dissipation. Another aspect that should be considered is that solutions based on versatile systolic arrays intrinsically do not support depthwise convolutions due to introduced sparsity in these types of convolutions; thus, users need to convert the depthwise convolutions to group convolution to execute a DSCNN on designs similar to VTA. All these succumb to more power dissipation and memory transactions, which lead to having an inefficient hardware solution for DSCNNs.

The design proposed in [16] presents a framework to minimize the complexity and the model size of dense CNN by mapping normal convolution to depthwise separable convolution. Similarly, TuRF [17] replaces standard convolution layers with depthwise separable convolution and applies layer fusion to enhance the performance of dense

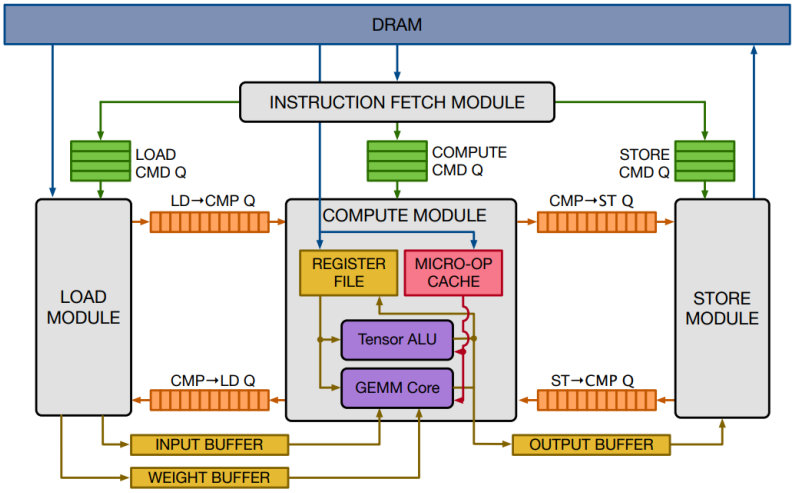

Figure 2.1: VTA (Vresatile Tensor Processing Unit)

networks. The design presented by [18] is another hardware accelerator based on matrix multiplication and customized adder-tree to support MobileNet-V2. However, their fixed design platform is not scalable to support fast-growing and forthcoming DSCNNs. A parallel acceleration scheme proposed in [19], demonstrates computing reusability with design reconfigurability. However, the accelerator suffers from massive data movements due to frequent reads and writebacks to the DDR because of the lack of fused layer execution. Moreover, the design-flow is not autonomous and requires the user to define the layer structure. A MobileNet-V2 based hardware accelerator on FP32 computation is presented in [20].

DPU [21] is another solution to support MobileNet-V2 based on an optimized RTL hardware model with a dedicated operator for depthwise; however, it cannot be considered as a versatile solution to support DSCNNs due to lake of support for swish activation function and pointwise multiplication. Fig:2.2 show the basic block diagram of the DPU. It also has a instruction based scheduler with a core engine for normal and pointwise convolution and a separate engine for depthwise convolution. DPU does not support elementwise multiplication, which makes it incompatible with the EfficientNet model. To the best of our knowledge, none of the above approaches

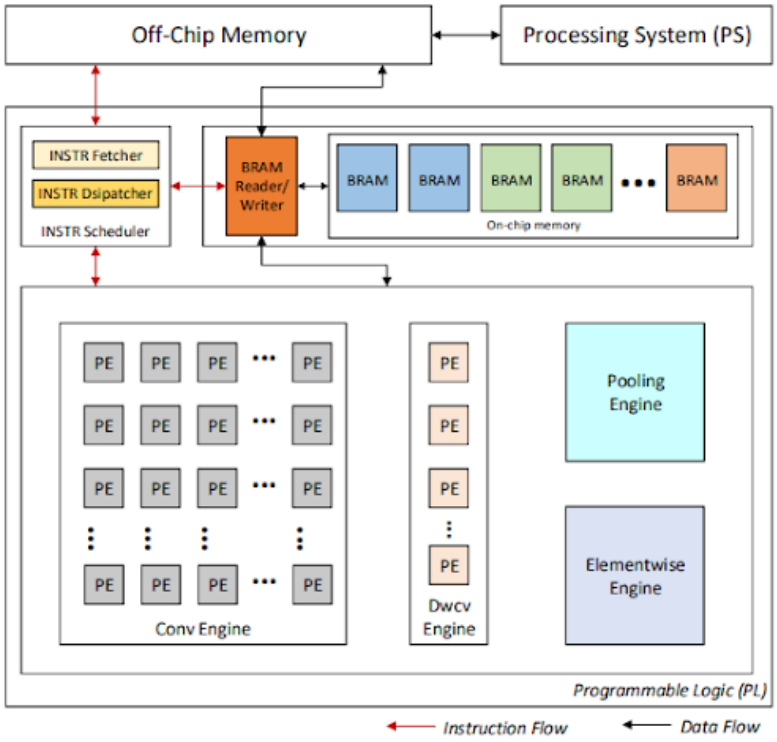

Figure 2.2: DPU (Deep Neural Processing Unit)

present a fully vertical framework to implement the-state-of-the-art DSCNN architectures, e.g., EfficientNet family.

2.2 Background

Here we go through some background information regarding some libraries, basic building block of convolution. Then we go present an overview of the HLS tools and pragmas which enabled us to develop DeepDive framework with translated source source code to desired hardware design.

2.2.1 Hlslib

HLS tools have encouraged FPGA development by allowing programmers to design architectures using familiar languages such as C++ and OpenCL. However, the additional layer of abstraction added by HLS, makes it difficult to track problems in the final architecture which is quite frustrating since it is nearly impossible to debug

HLS code which makes debugging and optimization degenerating into a trial-and-error process. Hence, we used hlslib [22] which is an open-source collection of tools, modules, and scripts that helps in improving the quality of life of HLS developers. Hlslib improves the HLS workflow by introducing the following functionalities.

2.2.1.1 CMake Integration

CMake tool is generally used to configure and build C/C++ projects. We can set project parameters during configuration, as compilation is performed out-of-source, and dependencies are automatically located on the host system in a portable fashion without the need to explicitly setting the host processor environment. hlslib provides supports for FPGA projects in CMake which allows separation of source code and configuration through CMake scripts. Users can explicitly design their run-time based on the compiler flags, headers and libraries included with CMake configuration. Also, with constant updates in the workflow development and run-time platforms by vendors, using CMake offloads the responsibility of setting up the HLS environment to hlslib making project robust to setup changes by vendors.

2.2.1.2 Software Simulation of Hardware

HLS provides numerous pragmas to pipeline and parallelizes the processing elements (PEs) execution in hardware. Accurately emulating the semantics of such multiple concurrent PEs executing in hardware is really important in the testing process such that the design is correctly interpreted in hardware by HLS in order to achieve high performance. PEs typically communicate via blocking channels, implying synchronization points between them. Emulating concurrent PEs thus requires a multi-threaded environment with thread-safe constructs. Hlslib makes it possible to debug in HLS by providing a set of thin wrapper macros that interprets these concurrent PEs as concurrent threads at run-time while running software emulation which is inherently a thread-safe design.

2.2.1.3 Thread-safe Streams

Streams are used as communication primitives between PEs or as buffers with FIFO semantics. Hlslib interprets these HLS streams as thread-safe streams which mimic the inter-PE communication between multiple PE during the simulation. Furthermore, streams are bounded by default, like the hardware implement they represent. Additionally, hlslib will implement streams in the resource suggested by the user if explicitly mentioned in an optional template argument.

2.2.1.4 Wide Data Buses and Vectorization

Instantiating wide data paths in HLS is necessary to exploit memory bandwidth, and to achieve parallel architectures through vectorization. hlslib provides the templated DataPack class for Vitis HLS, which exposes a versatile interface for implementing wide buses, registers, memory interfaces, and computations that consist of multiple data elements.

2.2.2 Gemm Hls

Data movement is the dominating factor affecting performance and energy in modern computing systems. GEMM being a data redundant approach it is necessary to design a matrix multiplication approach that has minimized the number of I/O operations and maximized performance for the off-chip necessary data movement. GEMM_HLS presents an optimized matrix multiplication for FPGA platforms, simultaneously targeting maximum performance and minimum off-chip data movement, within constraints set by the hardware. The GEMM is configurable based on the targeted platform, degree of parallelism, and is designed with hlslib extensions. The proposed method presents optimum memory bandwidth utilization for data transfer from the processor to FPGA.

2.2.3 EfficientNet

As mentioned earlier, depthwise convolution minimizes computation by removing reduction along the input channels; thus, it is not able to capture the channel-wise information. In the same fashion, pointwise convolution reduces the computation complexity by removing spatial filtering, while it has a full reduction in channel depth. Depthwise separable convolution, used in EfficientNet, is an integrated operator composed of a depthwise convolution, followed by pointwise convolution, in order to capture information in both spatial and channel domains, respectively. However, there is still information loss as features move along the network depth and are embedded into lower-dimensional space. EfficientNet has inverted residual connections, further reducing both multiply-add operations, and model size, without sacrificing the network accuracy. The idea of residual connections was inspired by the ResNet [23] architecture to minimize information loss and speed up the training phase. Fig. 2.3 shows the structure of the Inverted Residual Block (IRB) for EfficientNet. IRB consists of a pointwise (expansion) convolution, followed by a depthwise convolution, followed by squeeze and excitation (SE) block which is followed by another pointwise (projection) convolution, to embed the features in a lower dimension. The SE block consists of a squeeze operation that captures the global spatial features, followed by an excitation operation that uses a gating function to allow important features to be captured while ignoring the rest. Traditionally, the normal sigmoid is used as the gating function for the SE block but is replaced with the hard sigmoid to further reduce computation complexity. The hard sigmoid is a non-smooth approximation of the sigmoid function.

EfficientNet is known for its compound scaling method. In order to achieve better accuracy and efficiency, it is critical to balance all dimensions of network width, depth, and resolution 2.4 which can be achieved by scaling each of these knobs with a constant ratio. Intuitively, the compound scaling method makes sense because if the input

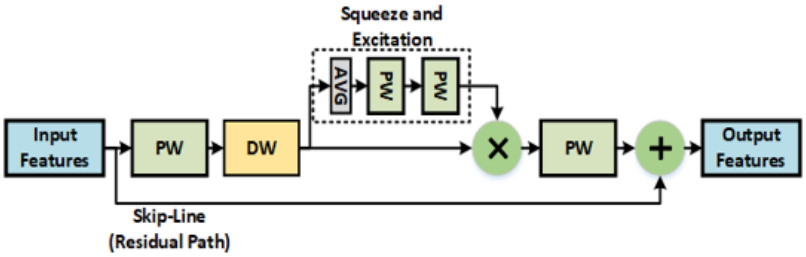

Figure 2.3: Inverted Residual Block: EfficientNet. The illustration of Batch Normalization and Activation layers repeated after each convolution are ignored.

image is bigger, then the network needs more layers to increase the receptive field and more channels to capture more fine-grained patterns on the bigger image. While bigger and bulkier networks are ideal for cloud devices it was necessary to identify networks that would take into consideration the resource constraints introduced while using edge devices. In separate research in our lab, we identified that it was possible to scale down EfficientNet so as to achieve compressed models of EfficientNet without sacrificing the accuracy by a huge margin which would be an idea for edge devices. Table 2.1 shows a comparison of two different EfficientNet model configurations, out of which one is BC4 which is the compressed version of EfficientNet vs the baseline version of EfficientNet. Looking at the model size and the number of Ops required for the EfficientNet BC4 model makes it a suitable candidate for inference on edge devices and EfficientNet B0 which is the baseline model is inferred on cloud FPGA.

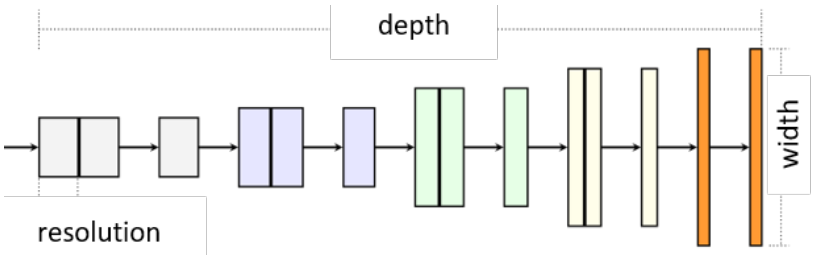

Figure 2.4: Compound Scaling of depth, width and resolution.

Table 2.1: EfficientNet Configurations Comparison

EfficientNet Configuration	BC4	B0
Input Size	128	224
Parameter (Mb)	7.81	31.74
#Ops (M)	4.91	390
Top1 (%)	55.02	77.1

2.2.4 Xilinx Vitis HLS

HLS is an automated design process that interprets an algorithmic description of the desired behavior and creates digital hardware that implements that behavior on FPGA. By targeting an FPGA as the execution fabric, Vitis HLS enables a software engineer to optimize code for throughout, power, and latency without the need to address the performance bottleneck of single memory space and limited computational resources. It is developed to simplify the use of C/C++ functions for implementation as hardware kernels in Vitis application acceleration development flow for developing RTL IP for FPGA designs. HLS also provides support for any arbitrary bit-width data type. It also provides us with a streaming interface for data structures that are designed to obtain the best performance and area. Xilinx has provided us with compiler directives or optimization pragmas. These pragmas can be used to optimize the design, reduce latency, improve throughput performance, and reduce area and device resource usage of the resulting RTL code. Below are some example of the pragmas that are used in the current design

2.2.4.1 pragma HLS DATAFLOW

The Dataflow pragma enables task-level pipelining allowing functions and loops to overlap their execution which increases the concurrency of the RTL design. In order to use dataflow pragma, the functions must be executing in producer-consumer fashion communicating via HLS streams. Dataflow forbids concurrent access on the

same array, either the array needs to have multiple ports such that each function in dataflow can have independent access to the given variable or allocate separate resources altogether.

2.2.4.2 pragma HLS PIPELINE

The pipeline pragma reduces the initiation interval by allowing the concurrent execution of the operations. In default, pragma is trying to make the computation run in a single cycle. However, loop carry dependencies prevent pipelining. Hence, we need to make sure that HLS understands that there is no loop carried dependencies.

2.2.4.3 pragma HLS ARRAY PARTITION

An array when declared in hardware is allocated in BRAM depending on the size of the array and the size of the BRAM of targeted hardware. By default, each BRAM has 2 ports. If we want to access the same resource multiple times simultaneously, it is necessary to partition the array such that HLS would allocate separate hardware for each access such that all the resources can be accessed simultaneously due to the limited BRAM ports.

CHAPTER 3: DEEPDIVE CLOUD

3.1 Backend

DeepDive's backend for cloud platform offers a novel micro-architectural approach, and design flow, customized for efficient execution of DSCNNs. Fig. 3.1 presents the DeepDive Cloud backend design flow. The processor and the FPGA communicate via Peripheral Component Interconnect Express (PCIe) bus. The image data, network weight, biases, and quantization parameters are loaded by the host CPU to device memory (HBM stack) via PCIe interface, and AXI interfaces are used by the compute unit to read the master interface to fetch these data for the processing elements (PEs). The compute units are part of the SLR region in FPGA fabric. The number of compute units can be configured based on the resource availability of the targeted FPGA and user required parallelism. The backend of cloud FPGA demonstrates a throughput orient design flow wherein there are many homogeneous compute units instantiated simultaneously processing on different data which is similar to the SIMD style approach we see in GPUs but each compute unit has a customized datapath designed to optimize performance from the processing elements within each compute unit. Because of this design, DeepDive for cloud is better able to address the irregularities in the DSCNN structure introduced by different convolutional layers.

Each compute unit has is own dedicated HBM channel to interact with the HBM stack and the network parameters for each compute unit are read and written via individual AXI interface to the HBM stack. Each compute unit has on-chip buffers which are used to store intermediate activation features, network weights, biases, and quantization parameters from the device memory. Each compute unit needs concurrent access to these buffers in order to execute concurrently as a result each

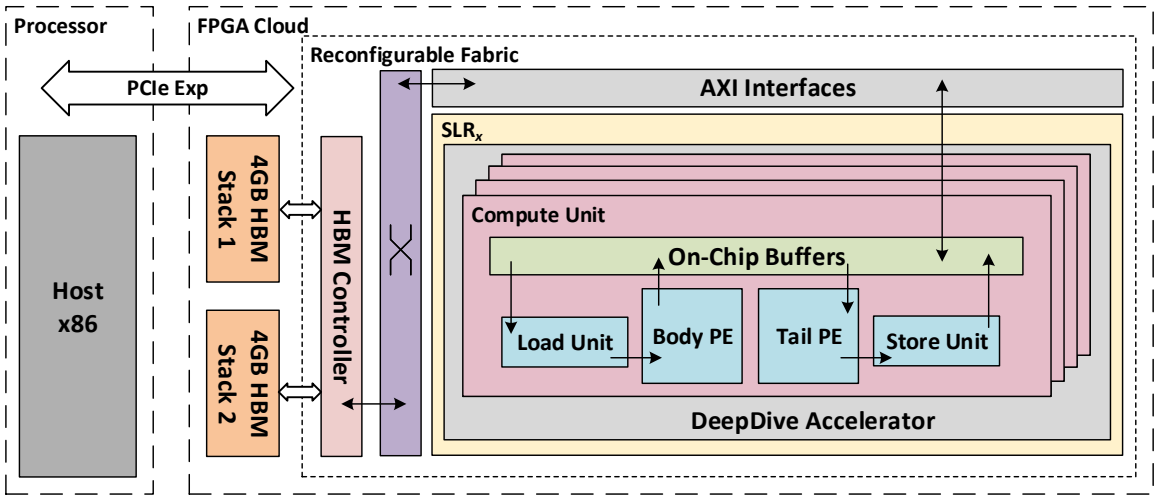

Figure 3.1: DeepDive - Cloud: Backend.

compute unit maintains a copy of these buffers.

3.2 DeepDive Execution Dataflow

The host-level scheduling of compute units is handled by the processor. The host is designed in OpenCL fashion. Since the hardware is composed of multiple compute units, the host needs to instantiate multiple concurrent commands to the compute units. It is only possible if the OpenCL command queues are set in out_of_order_execution mode. If this mode is not set, then each compute unit will wait for its previous instance for completion and would wait for unnecessary cycles even after the completion of its task. In the out_of_order_execution mode, the scheduler can dispatch commands from the command queue in any order and the user must explicitly set up event dependencies for synchronization if necessary. On the first instance of scheduling each of the compute units, the image data, weights, biases, quantization parameters along with compute unit configuration parameters are supplemented to the compute units. Upon each subsequent call, just the network configuration parameters are passed to the compute units while scheduling the compute units. This is because the entire memory footprint of the network is contained in the compute units during processing time. Upon completing the classification of

the given image, the output of the classifier is copied back to the scheduler just once. Each compute unit works on a different image which is why we call this single call multiple data execution.

3.3 Convolutional Operators

DSCNNs comprise of two types of operations namely ① Matrix-Matrix Multiplication (MMM) or GEMM and ② Vector-Matrix Multiplication (VMM) or Direct Convolution (DC). Different layers in DSCNNs mimic either of the above behaviors. In order to best understand which layers should be mapped to MMM and VMM, we performed an experiment wherein we implemented the 1st layer of DSCNNs which ideally tends to be Normal Convolution which comprises of MMM operations with GEMM-based algorithm and DC-based algorithm. Table 3.1 shows the execution time and ops performed by both the algorithms. We could clearly see that layers that mimic MMM type of operations would give better results when implemented using a GEMM-based algorithm. This experiment leads to the foundation of the processing elements inside each CU for DeepDive Cloud. As a result, the PEs comprises of heterogeneous convolution operator design where each layer is mapped to the algorithm which obtained the best parallelism and least execution time.

Table 3.1: GEMM vs DC for input size 128 x 128 x 3

	GEMM	DC
Execution Time (ms)	1.557	3.33
GOPs (Mb)	9.83	4.29

3.3.1 General Matrix Multiplication

As discussed above, the MMM operations are mapped to GEMM. Among the different layers of DSCNNs the layers mapped to GEMM for DeepDive Cloud backend include: ① Normal Convolution, ② Pointwise Expansion Convolution and ③

Pointwise Projection Convolution. Each of these layers demonstrates MMM type of operations. The GEMM presented here is an extension of GEMM_HLS which apart from matrix multiplication performs convolution operation and it is integrated into the processing element of the compute unit so as to work in fused fashion with other DSCNN layers. Fig. 3.2 represents the micro-architecture of GEMM convolutional operator. GEMM is capable of processing two types of inputs, either from on-chip buffers or streams, depending upon the position at which it is executed in the processing element. If it is the first layer to be running in the PE then on-chip buffers act as input to the GEMM and if the layer is executing as a part of the fused operation in the PE then the input is taken from the stream written by the previous layer in the fused pipeline. This decision is made at synthesis time by the HLS. Weights and network parameters are always read from the on-chip buffers into the streams. GEMM demonstrates such a boost in performance is because of the fact that we sort of implement data prefetching in this pipelined architecture of GEMM wherein, the data fetching granularity is different as compared to the compute granularity. This sort of double buffering approach helps us to concurrently execute the load and compute operations in GEMM.

The heart of GEMM is the compute core. The number of compute cores and the degree of parallelism which user can have in their design are configurable at compile time. All the compute cores run concurrently and demonstrate pipelined architecture internally. Finally, the output is written back to the output stream or the output on-chip buffers depending on the position at which GEMM is being executed.

3.3.2 Depthwise Convolution

The Depthwise convolution uses a 3D line buffer and 3D window to perform direct convolution. The input feature is streamed into a line buffer and then copied into a window buffer with parallel read access, as shown in Fig. 3.3. Once the computation is finished, the data in the computation core will be flushed and reloaded with the new

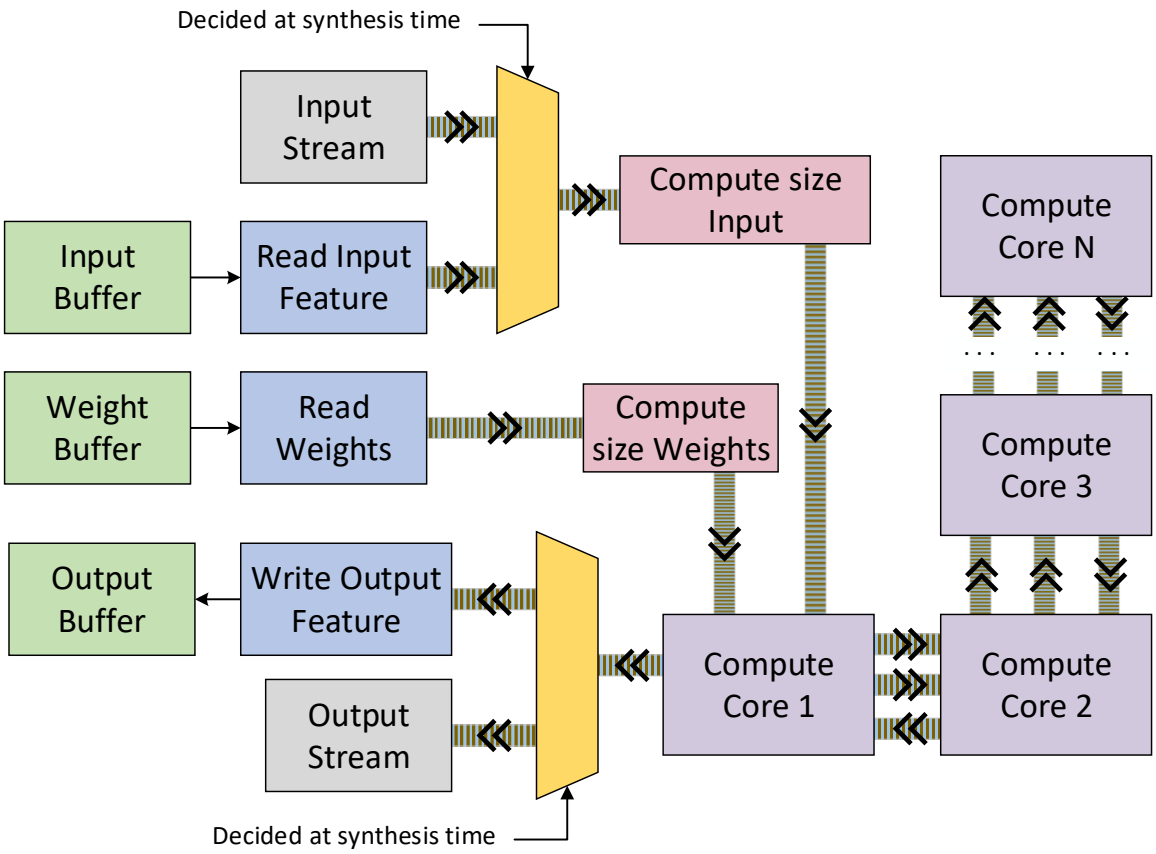

Figure 3.2: GEMM Convolutional operator.

one from the line buffer. The hardware design ensures the data movement involved in this process is fully pipelined, and the initiation interval is limited to a single cycle. Computation starts as soon as the required amount of data is streamed from the main memory. For the current design, the max achievable parallelism is limited to the K and N.

Fig. 3.4 presents the micro-architecture of depthwise convolution operator. As depicted in Fig. 3.4, the selected input is read in streaming fashion into the 3D line buffer and then copied into the sliding window. The weights are burst read into the weight scratchpad. The Sliding Window and the Weight scratchpad have multiple read ports. Every channel of the input is processed by the direct convolution compute core. The direct convolution compute core has a parallel multiplier and a pipelined adder tree, together which carryout the MAC operation, followed by the

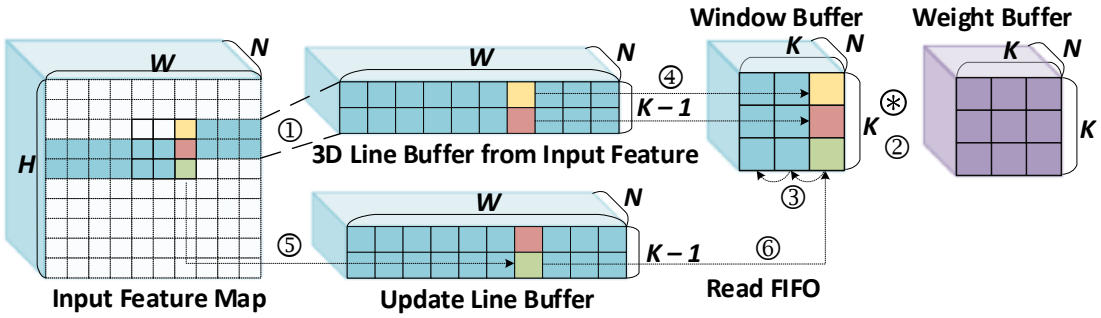

Figure 3.3: Shift and update mechanism of Window and Line Buffer. ① Line Buffer is filled with input feature data. ② Window Buffer is convoluted with weights. ③ The data in window is left shifted. ④ New data from the line buffer is copied in to the window. ⑤ & ⑥ Data from the FIFO is then copied into the line buffer and window buffer. All the Data Movements are pipelined.

Approximator and Clip unit. This unit truncates, or rounds, the results and then clips them to $[0,\ 2^{BW} - 1]$ based on the quantization parameters extracted at the front-end for this operator. Therefore, this unit also acts as the ReLU6 activation layer defined EfficientNet. The depthwise convolution is more sparse and has the least amount of data reuse. The maximum parallel operations are calculated as the following:

$$ParallelOps = K_{max}^{dw} \times K_{max}^{dw} \times N_{max}^{dw}, \tag{3.1}$$

In Eq. 3.1, K_{max}^{dw}, and N_{max}^{dw} are the maximum kernel size and maximum input-channel across all the depthwise convolutions in the network, respectively.

3.3.3 Pointwise Convolution

The Pointwise convolution for Squeeze and Excite layers for EfficientNet demonstrates VMM operations hence are mapped to direct convolution. We see this behavior because the Average pool layer vectorizes output which is then processed by the pointwise squeeze and excite layers. Due to the dense operation of pointwise, the design of this operator can be similar to the design of a systolic array implementa-

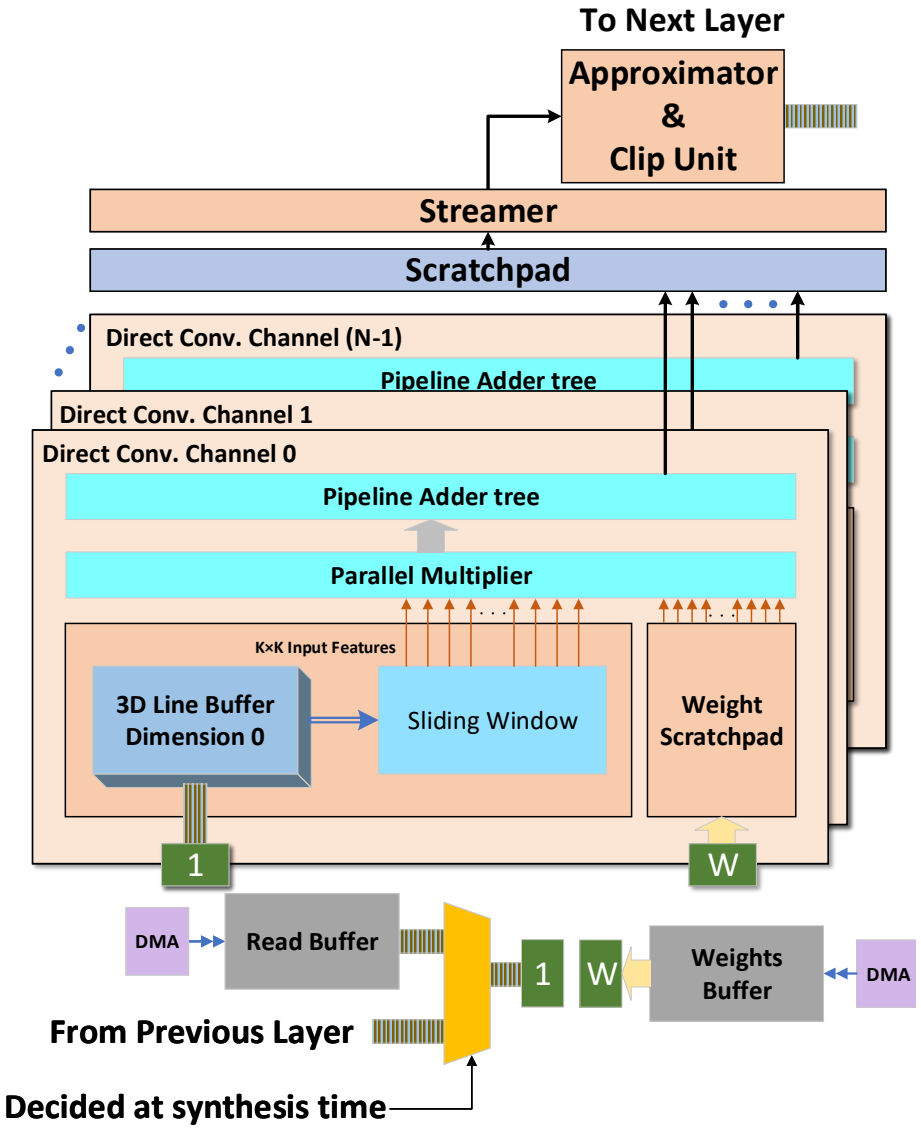

Figure 3.4: Schematic block diagram of Depthwise Convolution.

tion. With maximum data reuse, this operator can leverage maximum parallelism. It has both fewer algorithmic and fewer data movement complexity, which makes it the best fit for a high amount of parallelism. Fig. 3.5 shows the structure of pointwise convolution operator. The required input is directly read into the input scratchpad from the read buffer. The weights are burst read into the weight scratchpad. The input buffer and the weight scratchpad have multiple read ports for parallel data access. The single-cycle parallel multiplier and the adder tree take advantage of the

multiple ports to perform the MAC operations in a parallel fashion. The amount of parallelism for our design is across the input channels

$$ParallelOps = N_{max}^{PW_{type}}, \tag{3.2}$$

where $N_{max}^{PW_{type}}$ is the maximum input channel size across all the specific *type* of pointwise convolutions mapped to specific compute unit.

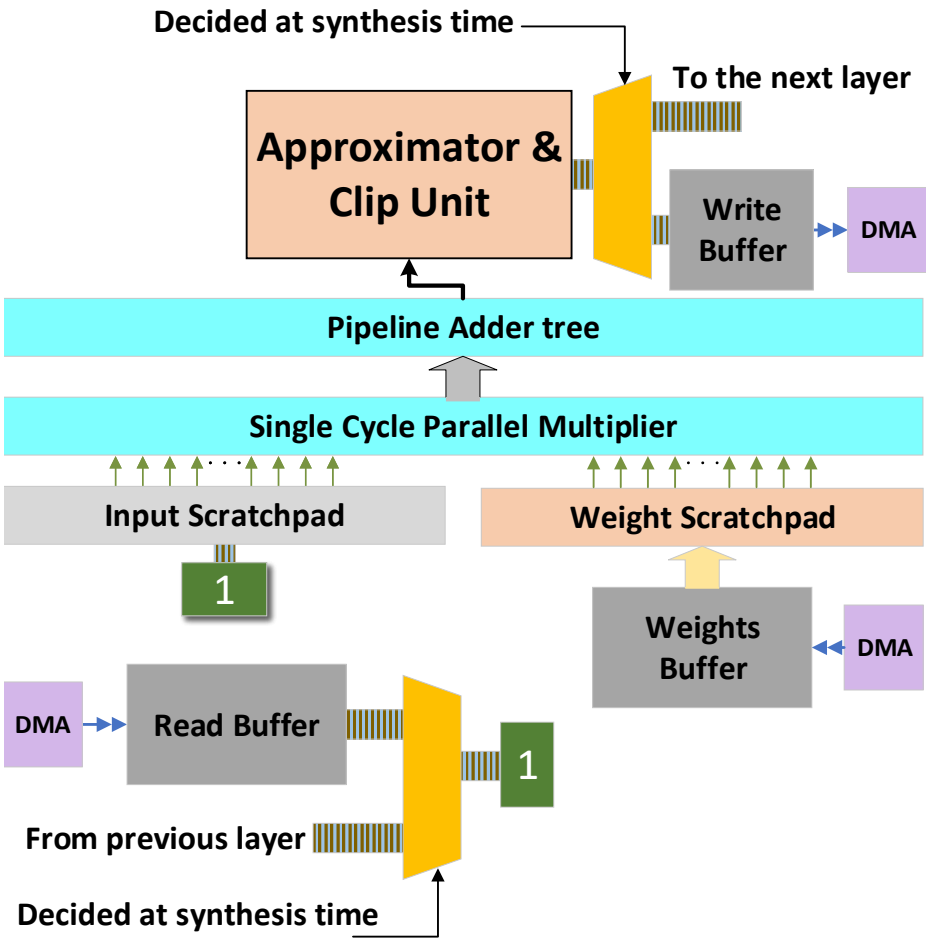

Figure 3.5: Schematic block diagram of Pointwise Convolution.

3.4 Cloud Compute Unit

Fig. 3.6 shows the schematic for DeepDive - Cloud compute unit. Each compute unit comprise of four modules: ① Load Unit, ② Body Processing Element, ③ Tail Processing Element, ④ Store Unit. These modules are scheduled at run-time by the scheduler depending upon the structure of the chosen DSCNN. The load unit and store unit are scheduled once during the entire execution of DSCNN. During the initial scheduling of the compute unit the load unit is instantiated for all the compute units. It is responsible to copy the data from the device memory to the on-chip buffers. These on-chip buffers are then used by the PEs for their fused execution of multiple layers. The body PE is composed of: ① Normal or Pointwise Expansion, ② Depthwise, ③ Average pool, ④ Pointwise Squeeze, ⑤ Pointwise Excite and ⑥ Pointwise Projection layers, all running concurrently in fused fashion within the body PE. This pattern is chosen for the body PE because it is responsible for executing the majority of DSCNNs blocks iteratively. This is because the IRB block which is the most repetitive block of EfficientNet is entirely mapped to body PE. Fig. 3.6 also highlights which of the layers are running the GEMM-based algorithm vs which layers are executing a DC-based algorithm. Thus, we describe this DeepDive - Cloud backend design as having homogeneous compute units with heterogeneous processing elements.

Since the compute unit is a homogeneous block, it also provides support for tail PE which is repeated only once per DSCNN implementation and is scheduled at run-time. Tail PE is composed of the last pointwise convolution followed by the average pool and finally the classifier. Similar to the load unit, the store unit is scheduled once per the DSCNN execution cycle. The output of the classifier is written by the store unit to the device memory from the on-chip buffers which are then read by the processor to depict the inference results to the user. Each compute unit is scheduled similarly by the processor and we get inference for N inputs simultaneously due to

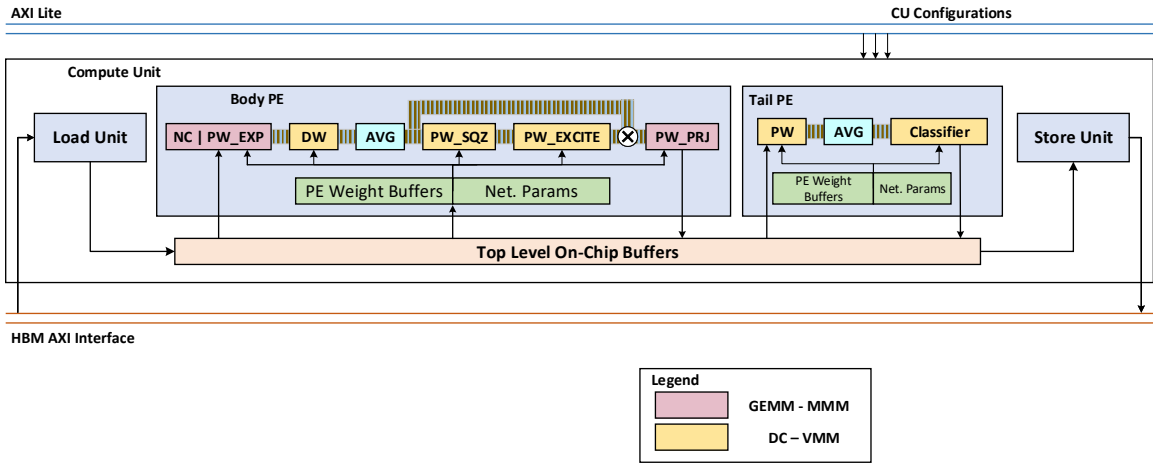

Figure 3.6: DeepDive - Cloud: Compute Unit Schematic.

the out_of_order_execution of multiple compute units on the cloud.

CHAPTER 4: DEEPDIVE EDGE

The DeepDive - Edge architecture presents a framework which is composed of Frontend and Backend.

4.1 DeepDive - Edge: Frontend

This section illustrates the Front End of the DeepDive. This is mainly responsible for bringing hardware-awareness into training DSCNNs. Fig 4.1 gives a brief understanding of the frontend and its corresponding output. A pre-trained floating-point network is an input to the deep dive system. To reduce the computation complexity we try to fuse the batch normalization into the convolution. So the final network will not have any computation related to the batch normalization. This reduces the operation by a small amount. The other feature of the front end is to perform the online channel-wise low bit quantization. The quantization can be performed for arbitrary bit precision (3, 6, 8 bit). This post-training linear quantization also fuses the ReLU6 into the convolution operators.

4.2 DeepDive - Edge: Backend

Fig. 4.2 presents the DeepDive - Edge backend design flow. The heart of DeepDive's backend is the *Network SoC Compiler*. It receives the design properties from DeepDive's front-end and generates a full design of the system for both hardware (as synthesizable C++ models mapped to FPGAs fabric), software codes, and system configurations. To generate the optimized hardware for DSCNNs, the Network SoC Compiler uses pre-designed highly-optimized RTL micro-architectural blocks or synthesizable C++ model for depthwise, pointwise, and normal convolution operators. In simple words, the Network SoC Compiler generates a network graph containing

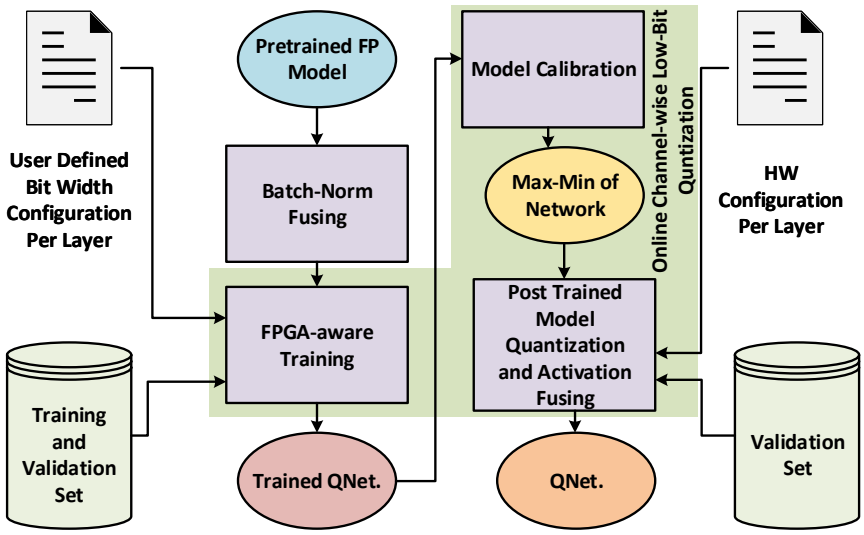

Figure 4.1: DeepDive: Frontend.

the network layout and data dependencies. It then creates key heterogeneous CUs, called *QNet Accelerators*, with respect to DeepDive's system architecture.

4.2.1 Convolutional Operators

The convolutional operators for DeepDive edge are similar to that of the once we discussed for DeepDive - Cloud with the major difference being that edge doesn't support GEMM. The convolutional operators purely perform direct convolution. When comparing cloud and edge convolutional operator implementation we see that for the edge, the normal and depthwise convolutions are implemented using the 3D line buffer and 3D windowing approach, whereas the pointwise convolutions are implemented in the similar systolic style direct convolution like we observed for the cloud. The parallelism in normal convolution is across kernel size and input channels — described as:

$$ParallelOps = K^{nc}_{max} \times K^{nc}_{max} \times N^{nc}_{max}, \tag{4.1}$$

where $N^{nc}_{MaxSize}$ is the maximum input channel size, and K^{nc}_{max} is the maximum kernel size, assigned from all normal convolution. Normal convolution has slightly more data

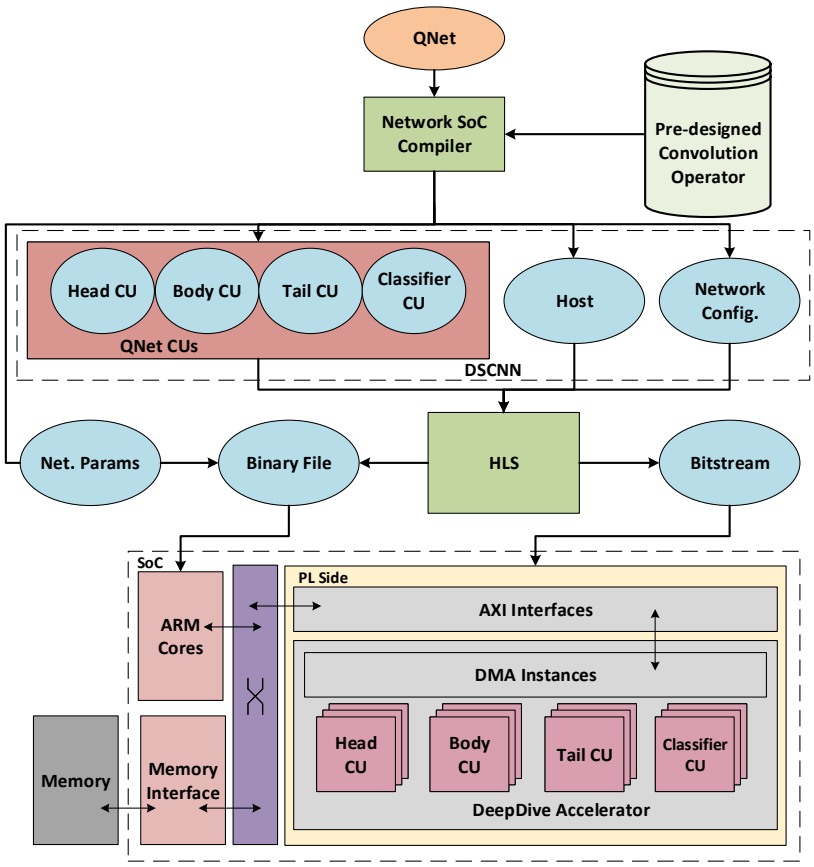

Figure 4.2: DeepDive - Edge: Backend.

movements compared to the depthwise convolution due to the pipelined adder tree implemented at the end of direct convolution core.

4.3 Network SoC compiler

The Network SoC Compiler observes the network graph, the targeted hardware device, and existing pre-designed synthesizable C++ IPs for convolution, and then translates the network graph by grouping the convolutional operators into customized *QNet* CUs with respect to system architecture. It tweaks the hardware architectural knobs to maximize parallelism, fusing as many convolutional operators as possible to reduce the number of shared memory transactions, and increase the overlap between computation and memory latency. Based on the repetitive pattern, it wraps the convolution operators in four different heterogeneous CUs: ① The *Head CU* generally

consists of normal convolution followed by a special case of IRB which is only called once; ② The *Body CU* invokes IRB since it has maximum repetitions based on the DSCNNs architectures; ③ The *Tail CU* usually consists of pointwise convolution followed by Average Pooling to embed the features and make them ready in respect of size and shape for the classifier; ④ Finally, the mapping of Tail CU output to $k-$classes is accomplished by *Classifier CU*.

4.3.1 QNet Heterogeneous CUs

In this subsection, we will explain the heterogeneous CUs, and the available architecture knobs that can be tweaked based on hardware and performance constraints. As mentioned earlier, Network SoC Compiler creates four unique CUs for each DSCNNs. The CUs are completely parameterizable, and customizable, for scalability and flexibility. The following section describes each CU in detail. We also provide illustrative figures for the example of heterogeneous compute units for EfficientNet.

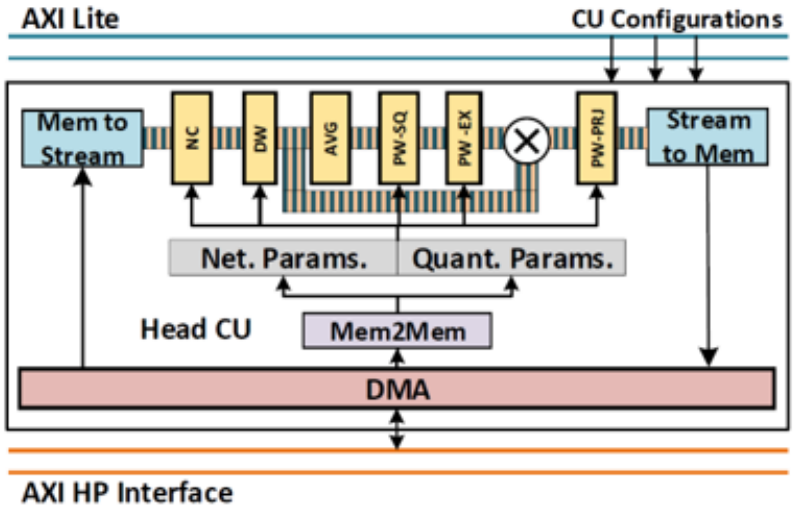

Figure 4.3: EfficientNet Head Computing Unit.

Head CU: DSCNNs tend to start with a particular pattern, which comprises of a fixed set of layers that are not recurrent in any other part of the network. The Head CU has its dedicated internal memory for buffers. The data transactions occur in memory-to-memory mode and the intermediate data streams between convolutional

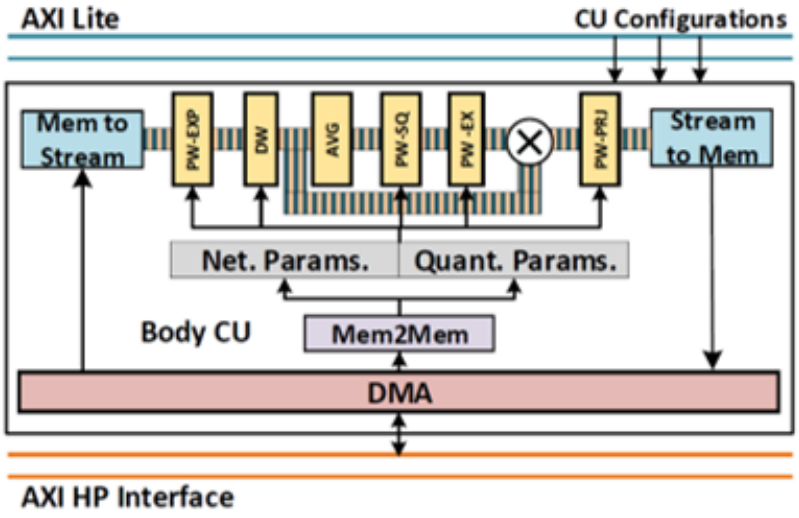

Figure 4.4: EfficientNet Body Computing Unit.

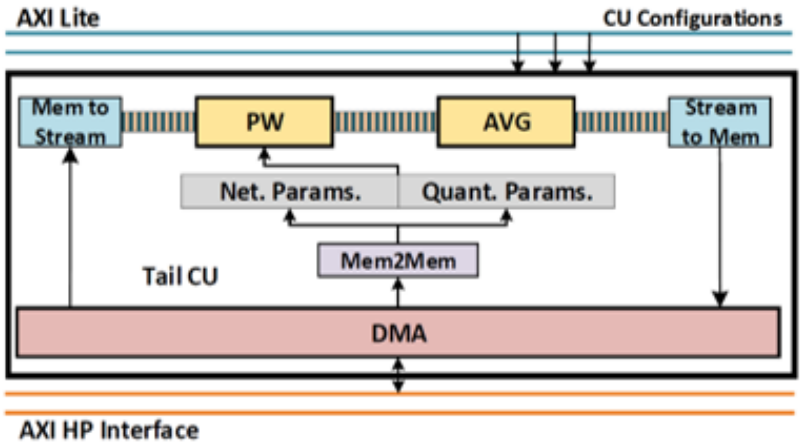

Figure 4.5: EfficientNet Tail Computing Unit.

layers within the head CU. As an example, Fig. 4.3 demonstrates the Head CU for EfficientNet model, which is composed of normal convolution followed by depthwise, average pool, pointwise squeeze and excite convolution and lastly, pointwise projection, all fused by FIFO stream. This CU is scheduled once during the course of any DSCNN implementation. After running the head of CU, the repeatable pattern will be merged and mapped to the Body CU explained in the next part.

Body CU: The Body CU is the most important CU within DeepDive's system architecture. It is responsible for executing majority of DSCNNs blocks iteratively. As an example, the IRB, which is the most repetitive block of EfficientNet, is en-

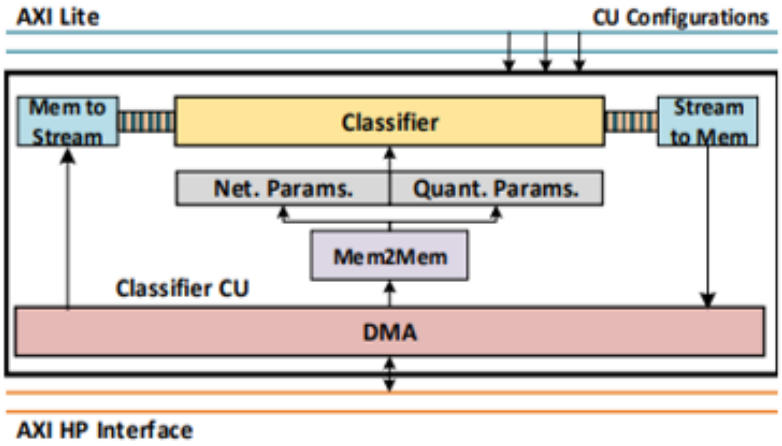

Figure 4.6: EfficientNet Classifier Computing Unit.

tirely mapped to the Body CU. The IRB consists of pointwise (expansion), depthwise followed by average pool, pointwise squeeze and excite layers and finally pointwise (projection) layers, all running concurrently in a fused fashion within the Body CU. Fig. 4.4 shows the structure of this CU for EfficientNet. Upon examining the network graph of DSCNNs, we see that occasionally, the IRB needs to perform residual connections. Depending upon the network graph, DeepDive facilitates residual connections implementation within or outside the PL targeted device resources. The Body CU is parameterized so as to support both memory-bound IRBs, which ideally are earlier blocks of DSCNNs, and compute-bound IRBs, which tend to be later blocks of DSCNNs. Therefore, the network SoC compiler configures the Body CU with maximum buffer size needed by memory-bound IRBs, and maximum level of parallelism to meet the demand imposed by compute-bound IRBs. At the same time, the Body CU supports convolution operations with variable stride over different IRBs. These features increase the framework inclusiveness by supporting multiple IRB scenarios within the same DSCNN.

Tail CU: The Tail CU consists of the last layers of DSCNNs. The task of this CU is to make the embedded feature size ready for the dense layer implemented in the Classifier CU. Fig. 4.5 represents the structure of Tail CU in EfficientNet. This

CU is comprised of a single pointwise convolution operator, followed by an average pool. As intermediate feature maps are streamed from layer to layer in a channel-wise fashion, the reshape block reorders the memory layout of the feature map in a column-wise mode. Therefore, the average pooling can accumulate the input on-the-fly and stream out.

Classifier CU: The last Compute Unit is the Classifier CU, which concludes the DSCNN implementation. Fig. 4.6 represents the EfficientNet Classifier CU. Similar to others, this CU is parameterized such that the parallelism across the computing core can be adjusted based on the available hardware resources. Classifier CU comprises compute-bound operations and has a similar configuration to the pointwise convolutional operators.

CHAPTER 5: EXPERIMENTAL RESULTS

5.1 DeepDive Cloud

The procedure starts from a PyTorch model of EfficientNet, pre-trained on ImageNet. At DeepDive's front-end, we configured the FPGA-aware training for different BW based on the channel-wise asymmetric ranged linear quantization if the inference is to be performed for an arbitrary bit width. The frontend also gives us 8bit quantized inference parameters which we can use if needed.

Moving to the DeepDive cloud backend, we have chosen Xilinx Alveo U50 cloud FPGA to demonstrate the capabilities of DeepDive cloud implementation. The host x86 processor is running at 3.9 GHz. We also have Vitis HLS 2019.2 tool to synthesize the network models compiled by DeepDive. The FPS and power consumption reported for DeepDive cloud backend are based on the QNet accelerator running at 500MHz. We targeted Efficient baseline implementation for Alveo U50. The Top-1 accuracy reported in this section is based on training and evaluating the network on the ImageNet dataset.

5.1.1 DeepDive - Cloud Backend: Mapping

DeepDive's backend for cloud identifies the mapping between the convolutional operators and CUs. Fig. 5.1 reveals the mapping of EfficientNet to multiple homogeneous CUs. We can see that each compute unit is scheduled multiple times depending upon the structure of DSCNNs with all N compute units running in parallel.

5.1.2 Design Exploration of EfficientNet B0 on Alveo U50

To show the boost in the performance for the throughput oriented design for cloud FPGAs, we compare the implementation of single compute unit vs multiple compute

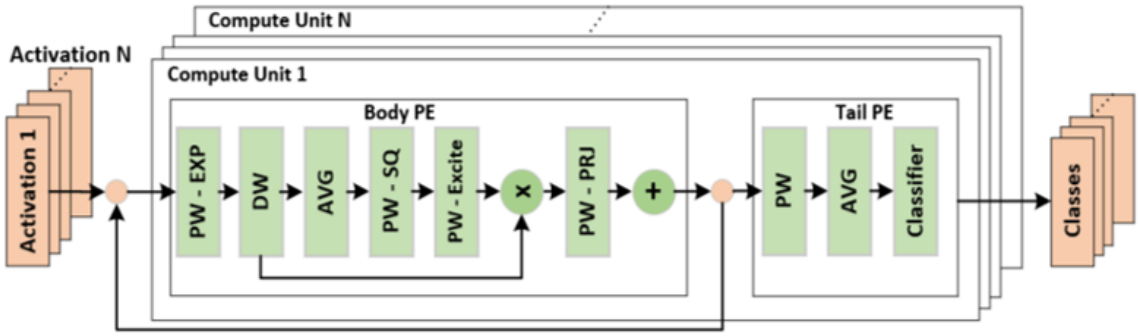

Figure 5.1: EfficientNet mapped to DeepDive - Cloud CUs.

units on Alveo U50 for the baseline EfficientNet configuration. In the multiple compute unit design, there are 17 instances of compute units processing multiple data in parallel. As discussed before, the design is synthesized to run at 500MHz on Alveo U50. We also compare the run-time power consumption by both the designs along with the comparison of their resource utilization. The table 5.1, demonstrates the comparison for single CU implementation vs multiple CUs implementation of the baseline EfficientNet model on Alveo U50.

We see that for single compute unit implementation on Alveo U50 we get a speed of 9 FPS. This speed is nothing for the cloud FPGA also the resources consumed by this design are way insignificant for the cloud and there is a lot of scope for increasing the throughput. Hence when we instantiate multiple instances of a single compute unit on the FPGA we see a dramatic boost in the performance with the same inherent design but scaled enough to get higher throughput. For the design with multiple CU configuration, we see a dramatic increase in resource utilization which is expected. However, run-time power consumption is justifiably increased. Overall, with multiple CU configuration, we observe 8.38× improvement in FPS. The power reported in the below table is the inference time power consumption, which is obtained using xbutil APIs provided by Xilinx XRT runtime by querying Alveo U50 stats while it is running the inference.

Table 5.1: EfficientNet B0 on Alveo U50

Single CU					**Multiple CUs**				
FPS	Power(W)	DSP(%)	LUTs(%)	BRAM(%)	FPS	Power(W)	DSP(%)	LUTs(%)	BRAM(%)
9.27	5	4.03	2.09	0.25	87	12	68.51	35.53	4.25

5.2 DeepDive Edge

The experimental setup for the edge, we have chosen the Xilinx Zynq UltraScale+ MPSoC ZCU102 evaluation board, which has XCZU9EG chip, to demonstrate the capabilities of DeepDive on edge. The ARM processors host Ubuntu 16.04, running at 1.2GHz; the OS can program the FPGA fabric at run-time. We also use Vivado HLS 2018.3 to synthesize the network models compiled by DeepDive. The FPS and power consumption reported for DeepDive are based on *QNet* accelerator running at 200MHz. We targeted EfficientNet DSCNNs as a case study to demonstrate the performance of DeepDive edge accelerator design. The Top-1 accuracy reported in this section is based on training and evaluating the network on the ImageNet dataset.

5.2.1 EfficientNet mapped to DeepDive - Edge CUs

Based on the network graph generated by Network Compiler, DeepDive's backend identifies the mapping between the convolutional operators and heterogeneous CUs. Fig. 5.2 reveals the mapping of EfficientNet to heterogeneous CUs. The Head, Tail, and Classifier CU are scheduled only once, but the Body is scheduled 9 times. Because of this, DeepDive allocates maximum resources to the Body CU to gain maximum performance. It makes the body CU support both memory-bound and compute-bound operations.

5.2.2 Energy Efficiency

Table 5.2 summarizes the power consummation, FPS, and hardware utilization. Power is measured using a power monitoring device. Measured power is the difference between the idle power dissipation of the board and the power consumed by

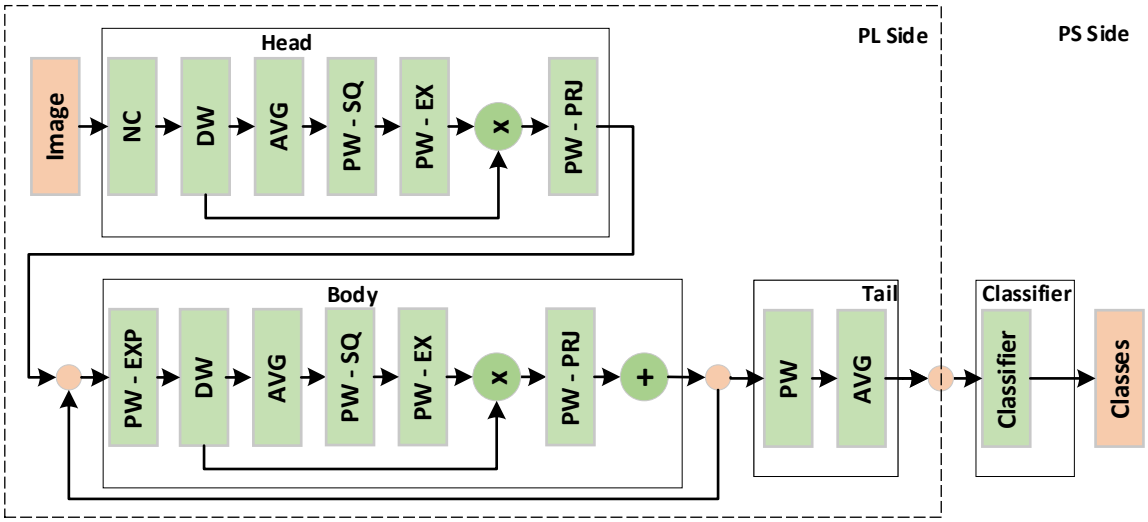

Figure 5.2: EfficientNet mapped to CUs.

the DeepDive edge accelerator while running inference. This power is consumed by MPSoC (ARM cores + FPGA fabric), memory hierarchies, and shared DDR memory during inference. The below table shows DeepDive reaches to 35 FPS for a power consummation of 150mW. This model gives us the energy efficiency of 233.3 FPS/Watt.

Table 5.2: Compressed EfficientNet Algorithmic Specs and FPGA Resource Utilization with fixed BW = 4, Frequency = 200 MHz

Algorithmic Parameters				Hardware Parameters				
H	Parameters (Mb)	#Ops (M)	Top1 (%)	FPS	Power (mW)	DSP (%)	LUTs (%)	BRAM (%)
128	7.81	4.914	55.02	35	150	90	80	68

5.2.3 DeepDive - Edge vs Jetson Nano

To showcase the energy efficiency of DeepDive edge accelerator design, we compare its FPS/Watt against off-the-shelf Nvidia Jetson Nano IoT Edge Device. Similar to the DeepDive, we calculate the power consumption only for inference time. We compared the delay and power consumption between DeepDive and Jetson Nano in two different power consumption modes: high power, and low power. It can be seen that DeepDive consumes a lot less power when compared to Jetson Nano, as depicted in Table 5.3. DeepDive can improve the FPS/Watt 8.6× and 6.7× against high and low

power mode, respectively. DeepDive outperforms Nano because ① DeepDive performs extreme bit quantization as opposed to nano which uses FP16; ② Although, TensorRT optimized the network model to fuse convolutional operators, DeepDive groups the convolutional operators in heterogeneous CUs at higher granularity. This heterogeneity effectively reduces the shared memory transactions and overlaps both computing and memory latency; ③ DeepDive provides a customized dataflow for depthwise separable convolution as opposed to Jetson Nano which performs general matrix multiplication for depthwise convolution due to fixed systolic array implementation. Also, for EfficientNet, based on the massively fused layers in Body CU, fewer memory transactions translates to more energy-efficient hardware.

Table 5.3: Power Consumption and delay for Compressed EfficientNet

H	Power(W)			Delay(mS)		
	Nano(H)	Nano(L)	DeepDive	Nano(H)	Nano(L)	DeepDive
128	5.61	2.22	0.15	6.581	12.6	28.57

5.3 DeepDive Edge vs DeepDive Cloud

Finally, in this section, we summarize the above results by providing a holistic comparison between two different architectures of DeepDive proposed in this thesis for edge and cloud FPGA platforms. The main purpose of providing this comparison is to highlight the massive shift from latency-oriented design paradigm to throughput-oriented design and how we shift the computational complexity of the network from edge to cloud devices. Table 5.4 compares two different EfficientNet Configurations implemented on edge and cloud platforms with a compressed version of EfficientNet BC4 running on DeepDive - edge accelerator and the baseline model of EfficientNet B0 running on DeepDive - cloud accelerator. The table accurately shows the massive increase in the model size and the number of flops as we go from the compressed EfficientNet model to its baseline model and we demonstrate that how well DeepDive

- cloud handles the massive workload while giving a tremendous performance of 87 FPS on Alveo U50. The table also provides absolute resource utilization for both the hardware accelerators. We see that in the DeepDive - cloud accelerator implementation there are still a lot of resources available and we can further increase the parallelism within each compute unit to further maximize the performance on the cloud. Even-though the accuracy of the compressed version of EfficientNet is less when compared to the baseline, if the task at hand for the edge device is to perform image classification for just a couple of images, we don't need a bulkier network like the baseline, the compressed version of EfficientNet would easily suffice for such a task.

Table 5.4: DeepDive Edge vs DeepDive Cloud

		ZCU102	Alveo U50
Algorithmic Parameters	EfficientNet Config	BC4	B0
	Input Size	128	224
	Parameter (Mb)	7.81	31.74
	#Ops (M)	4.91	390
	Top1 (%)	55.02	77.1
Hardware Parameters	FPS	35	87
	Runtime Dynamic Power (W)	0.15	12
	DSP	2265	4080
	LUTs	220211	309599
	BRAM	622	51

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

6.1 Conclusion

This thesis introduced DeepDive, as a fully functional framework for an agile, power-efficient execution of DSCNNs on both cloud and edge FPGAs. DeepDive offers a vertical algorithm/architecture optimization, starting from the network description model down to full system synthesis and implementation. At the front-end, DeepDive performs high-level optimization such as BN fusing, and Online channel-wise low-Bit quantization at extremely low-bit resolutions to bring FPGA-awareness when training DSCNNs. At the backend, DeepDive provides support for both cloud and edge platforms. DeepDive - Cloud backend presents a throughput oriented design which comprises of multiple homogeneous compute units performing tasks in SIMD fashion, wherein each compute unit comprise of heterogeneous processing elements. On the other hand, DeepDive - Edge backend introduces Network SoC Compiler. Which receives the design properties from DeepDive's front-end and generates a full design of the system for both hardware model and software host codes. To generate the optimized hardware for DSCNNs, the Network SoC Compiler uses pre-designed micro-architectural blocks for depthwise, pointwise, and normal convolution operators. For the results, we have synthesized, executed, and validated two state-of-the-art DSCNN, EfficientNet on Xilinx's Cloud FPGA Alveo U50, and Xilinx's Edge ZCU102 FPGA board. The execution results demonstrated how DeepDive - cloud implementation can achieve the performance of 87 FPS for the baseline model of EfficientNet and 233.3 FPS/Watt for a compact version of EfficientNeton edge FPGA. These comparisons showcased how DeepDive - edge implementation can improve the FPS/Watt 8.6× and 6.7× against high and low power mode, respectively for Jetson Nano.

6.2 Future Work

As future work, we plan to improve the back-end of DeepDive - cloud implementation by increasing the parallelism within each compute unit. We plan on making DeepDive more modularized such that it can support any of the upcoming DSCNN families without much change in the accelerator architecture. We also plan to devise a run-time scheduler for the accelerator so that the hardware doesn't need to pass the control to the processor at all.

REFERENCES

[1] C. Hao, X. Zhang, Y. Li, S. Huang, J. Xiong, K. Rupnow, W.-m. Hwu, and D. Chen, "Fpga/dnn co-design: An efficient design methodology for iot intelligence on the edge," in *Proceedings of the 56th Annual Design Automation Conference 2019*, DAC â19, (New York, NY, USA), Association for Computing Machinery, 2019.

[2] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From high-level deep neural models to fpgas," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pp. 1–12, IEEE, 2016.

[3] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang, "Angel-eye: A complete design flow for mapping cnn onto embedded fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. PP, pp. 1–1, 05 2017.

[4] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "Dnnbuilder: An automated tool for building high-performance dnn hardware accelerators for fpgas," in *Proceedings of the International Conference on Computer-Aided Design*, ICCAD '18, (New York, NY, USA), pp. 56:1–56:8, ACM, 2018.

[5] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner, "The snowflake elastic data warehouse," in *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, (New York, NY, USA), pp. 215–226, ACM, 2016.

[6] X. Wei, C. H. Yu, P. Zhang, Y. Chen, Y. Wang, H. Hu, Y. Liang, and J. Cong, "Automated systolic array architecture synthesis for high throughput cnn inference on fpgas," in *Proceedings of the 54th Annual Design Automation Conference 2017*, DAC '17, (New York, NY, USA), pp. 29:1–29:6, ACM, 2017.

[7] C. Zhang, Z. Fang, P. Zhou, P. Pan, and J. Cong, "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks," in *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD '16, (New York, NY, USA), pp. 12:1–12:8, ACM, 2016.

[8] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li, "Deepburning: Automatic generation of fpga-based learning accelerators for the neural network family," in *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, (New York, NY, USA), pp. 110:1–110:6, ACM, 2016.

[9] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, (New York, NY, USA), pp. 65–74, ACM, 2017.

[10] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: Automated mapping of convolutional neural networks on fpgas (abstract only)," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '17, (New York, NY, USA), pp. 291–292, ACM, 2017.

[11] K. Abdelouahab, M. Pelcat, J. Serot, C. Bourrasset, and F. Berry, "Tactics to directly map cnn graphs on embedded fpgas," *IEEE Embedded Systems Letters*, pp. 1–4, 2017.

[12] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. Oâbrien, Y. Umuroglu, M. Leeser, and K. Vissers, "Finn-r: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 11, Dec. 2018.

[13] C. Baskin, N. Liss, A. Mendelson, and E. Zheltonozhskii, "Streaming architecture for large-scale quantized neural networks on an fpga-based dataflow platform," *CoRR*, vol. abs/1708.00052, 2017.

[14] M. Samragh, M. Javaheripi, and F. Koushanfar, "Codex: Bit-flexible encoding for streaming-based FPGA acceleration of dnns," *CoRR*, vol. abs/1901.05582, 2019.

[15] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "A hardwareâsoftware blueprint for flexible deep learning specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, 2019.

[16] R. Zhao, X. Niu, and W. Luk, "Automatic optimising cnn with depthwise separable convolution on fpga: (abstact only)," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA â18, (New York, NY, USA), p. 285, Association for Computing Machinery, 2018.

[17] R. Zhao, H.-C. Ng, W. Luk, and X. Niu, "Towards efficient convolutional neural network for domain-specific applications on fpga," pp. 147–1477, 08 2018.

[18] L. Bai, Y. Zhao, and X. Huang, "A cnn accelerator on fpga using depthwise separable convolution," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 65, no. 10, pp. 1415–1419, 2018.

[19] J. Liao, L. Cai, Y. Xu, and M. He, "Design of accelerator for mobilenet convolutional neural network based on fpga," in *2019 IEEE 4th Advanced Information Technology, Electronic and Automation Control Conference (IAEAC)*, vol. 1, pp. 1392–1396, 2019.

[20] B. Liu, D. Zou, L. Feng, S. Feng, P. Fu, and J. Li, "An fpga-based cnn accelerator integrating depthwise separable convolution," *Electronics*, vol. 8, p. 281, 03 2019.

[21] D. Wu, Y. Zhang, X. Jia, L. Tian, T. Li, L. Sui, D. Xie, and Y. Shan, "A high-performance cnn processor based on fpga for mobilenets," pp. 136–143, 09 2019.

[22] J. D. F. Licht and T. Hoefler, "hlslib: Software engineering for hardware design," *ArXiv*, vol. abs/1910.04436, 2019.

[23] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, June 2016.