SCALABLE HARDWARE ARCHITECTURE FOR REAL-TIME AI ON THE EDGE

by

Kaustubh Manohar Mhatre

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

ABSTRACT

KAUSTUBH MANOHAR MHATRE. Scalable hardware architecture for real-time AI on the edge. (Under the direction of DR. HAMED TABKHI)

Deep Learning has brought a massive and revolutionary impact to the field of machine learning. The research in Neural Network algorithms has made them more efficient and powerful in the recent years. which gave rise to a need to enhance their performance on the edge. The focus in AI processing has attracted the hardware community and has led development in customizable hardware for AI. GPU's have proved to be efficient for processing the AI workloads. Researcher's today are more focused on reducing the computational complexity and memory footprint of the networks. This has led to more sparsity in the Network known as depthwise separable convolutional neural networks (DSCNNs) e.g: MobileNet and EfficientNet. GPU's are not designed to take advantage of the sparsity of such networks. However FPGAs take advantage of the reconfigurability and design a customizable data path for DSCNNs. This thesis focus on the FPGA Hardware design for powerful and efficient implementation of the DSCNNs on the edge FPGAs. It focuses on designing highly optimized convolutional operators like depthwise, pointwise and normal convolution and an architecture to support crucial heterogeneous compute units (CUs). It also focus on scalable development of those compute units for ease of implementation and support for the future networks. The hardware is designed using the Xilinx Vivado HLS 2018.3. HLS accelerates the development in hardware design. The execution results on Xilinx ZCU102 FPGA board demonstrate 47.4 and 233.3 FPS/Watt for MobileNet-V2 and a compact version of EfficientNet, respectively, as two state-of-the-art depthwise separable CNNs. These comparisons showcase how this design improves FPS/Watt by 2.2$\times$ and 1.51$\times$ over Jetson Nano high and low power modes, respectively.

DEDICATION

This work is dedicated to my father Dr. Manohar Mhatre and my mother Dr. Pratibha Mhatre for there constant support and encouragement. I would also like to thank my friends and family who have helped me throughout this journey.

ACKNOWLEDGEMENTS

I would like to thank Dr. Tabkhi for guiding me at every step during my research. I am grateful for all the support provided by him. He has motivated me to pursue research in this field. I thank Dr. Chen, Dr. Fareena and Dr. Ronald Sass for being on my thesis committee.

I would like to specially thank my colleague Mohammad Reza who has helped me in each and every part of my work. He has guided and helped me to better understand hardware desgin for FPGA, Xilinx HLS and helped me write better codes.

I would also like to thank my Mom and Dad for inspiring me all my years to take few important steps in my life that led to this thesis. I thank my friends for their constant love and support which encouraged me to work on my thesis and always go an extra mile. I would also like to acknowledge Google Meet for providing a platform for online conferencing. The final defense was conducted online using google meet due to the uncertain situation created by COVID-19.

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

Over the period of time the field of Deep neural Network has shown tremendous growth. Particularly Convolutional Neural Networks (CNNs) has enabled many exciting applications in visual analytics like image classification Fig:1.2 , object detection/tracking Fig.1.1, semantic segmentation Fig.1.3. Deep Learning networks need extensive compute power to train the network to have considerable accuracy. With the increasing popularity of the CNN algorithms a need for real time execution of those algorithms has been a necessity. The high performance CNNs come with a high computation complexity. Due to the increasing demand in the deep learning paradigm there has been a shift towards the Domain-specific architecture eg: systolic array, CGRAs, Tensor Cores. Deep separable CNNs have emerged as an innovative algorithmic solution to achieve higher accuracy with relatively lower parameters and operations. The most recent state of the art network have more structural sparsity compared to traditional CNNs. These new CNNs have more data dependent layer to layer communication and less data reuse potential. The modular design of the DSCNNs with the structural sparsity allow the designer to trade between algorithmic accuracy and the computational demand. MobileNet has a knob to change the width of the network which they call width multiplier. Changing the width multiplier changes the accuracy as well as the computation complexity.

1.1 Motivation

The current design of the DSA is more suitable for dense computations. These DSA does not perform well for the DSCNNs networks. Fig.1.4 show the structure of the DSCNNs. DSCNNs first has a Deptheise convolution layer followed by the Pointwise

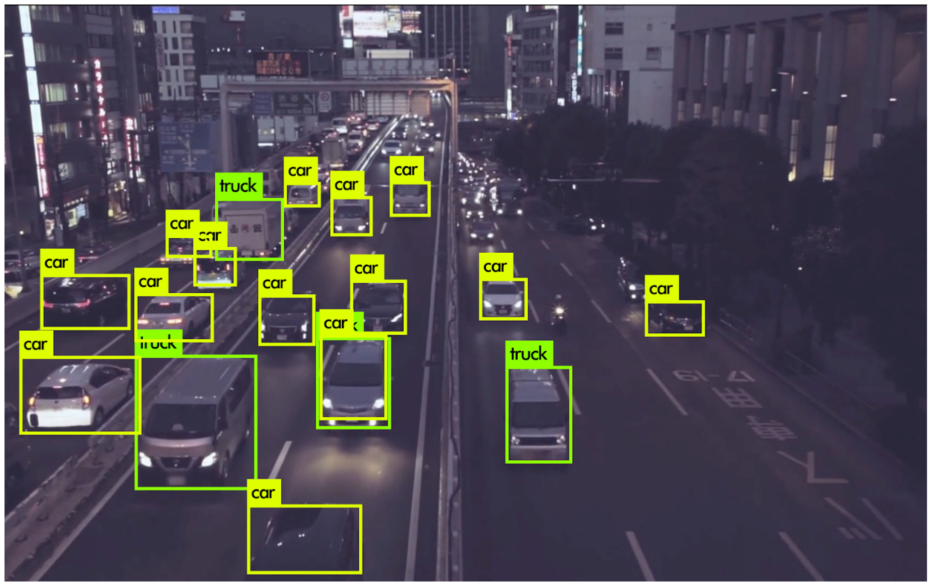


Figure 1.1: Object Tracking

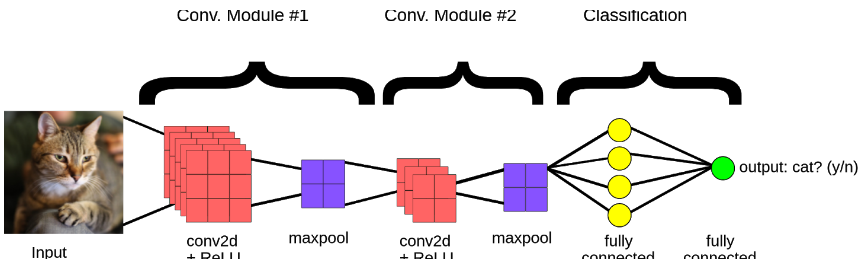


Figure 1.2: Classification

convolution layer. These DSAs are often optimized for single point in isolation. They convert the sparse convolution to dense convolutions by data replication which leads to higher data redundancy and higher computational overhead. VTA has to make a special version of MobileNet which they call mobile net G to remover the depthwise convolution to make it running smoothly on the systolic array implemented on the FPGA.

1.2 Contribution

DeepDive is a unified End - to - End fully vertical framework that can be used to convert any DSCNNs into FPGA realizable hardware that has optimum performance on the target FPGA board. DeepDive is designed to identify the key heterogeneous

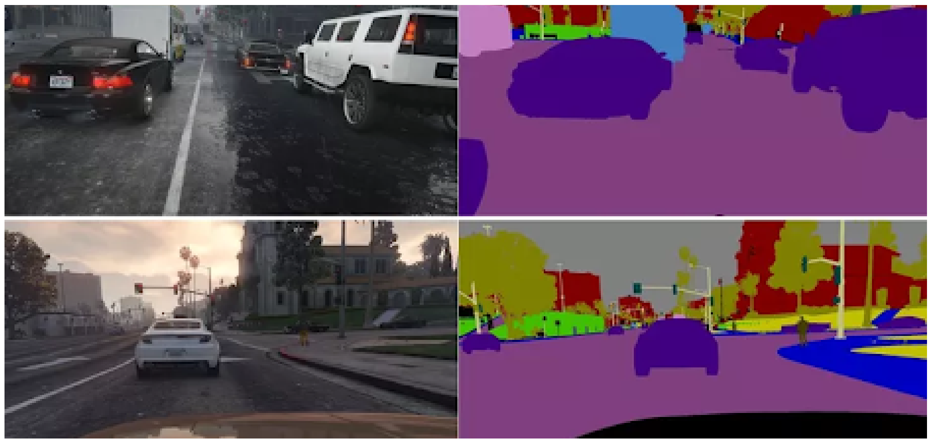


Figure 1.3: Semantic Segmentation

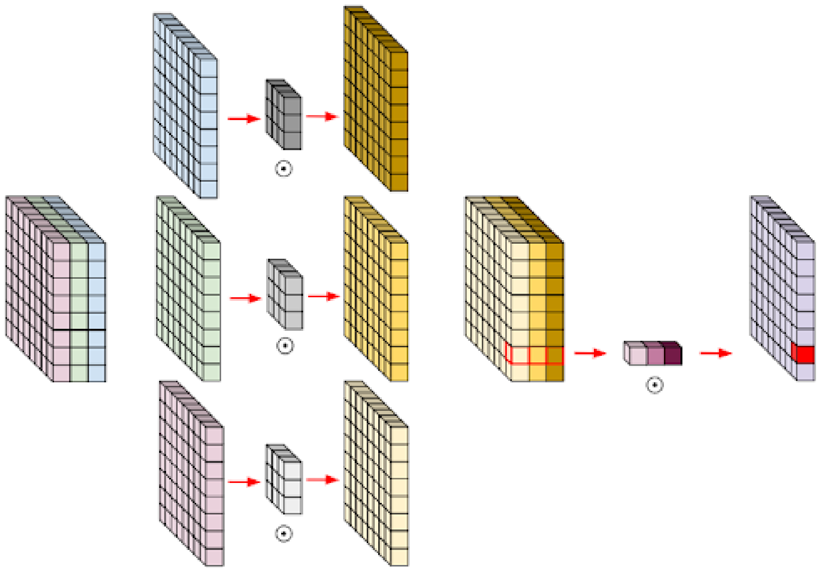


Figure 1.4: Depthwise Separable Convolution

convolutions operations such as group, depthwise, and pointwise convolutions. The frontend uses FPPGA-aware training and online quantization to optimize the model provided by the model description file (eg. PyTorch). Algorithm-specific fusing of batch normalization and convolution operators reduce the computation by 4 percent and extremely low bit per channel quantization across all separable convolutions layers also reduce 75 percent of model size when compared to Float32. The frontend will provide a QNET which contains all the meta data like the model parameters, quantization parameters, and network configuration graph of the network. The network

SOC compiler uses the pre-designed CUs and the provided convolutional operators to create a customize memory path and synthesizable model of the entire hardware accelerator for the programmable logic. It also generated the host CPU code running on the ARM cores. This host CPU code located in the Processing System(PS) side of the SoC is mainly responsible for synchronization and scheduling of the hardware resources. Below are the Deepdive contributions followed by my individual contributions

- A scalable framework that enables optimized execution of the DSCNNs on FPGA
- Highly optimized Convolution operators kernel with flexible computational core for adjusting 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 Individual Contributions

- Developing the Backend of DeepDive on ZCU102
- Developing synthesizable compute units : Head, Body, Tail, Classifier
- Porting Entire design to Vitis
- Profiling kernel to find bottlenecks to improve the performance of the kernel
- Network functional verification to have functionally correct output
- Contribute to the overall design flow of Deepdive

1.3 Thesis Outline

The thesis is further organised in the following manner. Chapter 2 will give a background about the techniques used to reduce the computation complexity with some related work. Chapter 3 talks about the DeepDive frontend where it generates the hardware specification and the quantized network. Chapter 4 dives into the backend which describes the basic building blocks of the accelerator. The working of the Network SoC compiler which is responsible to find the repeated pattern inside the network graph and based on that pattern divide the network into 4 basic compute unites (Head, Body, Tail, Classifier). Chapter 5 discusses the experimental setup and results of the whole framework. The experimental results compare the the execution of similar networks on embedded GPU and FPGA. Also compare it with the state of the art solutions. Chapter 6 concludes the thesis with some prospect for future work.

CHAPTER 2: BACKGROUND AND RELATED WORK

This Chapter focuses on the overall background. Right from some basic model optimization techniques like Quantization, Batch normalization which helps in reducing the model size and model operations. Then we take a overview of the tools that helped us develop such a framework without worrying about the complexities involved in the Domain of hardware design.

2.1 Background

2.1.1 Network Model Optimization

Floating point computation is the popular data type used in deep neural networks as it is best to handle the overflow involved in the computation. More research in the field suggest that lower bit fixed point arithmetic can drastically reduce the model size of the network with very less loss in the accuracy. Thus the process to move from floating point to fixed point is known as quantization. Batchnorm fusing and activation fusing are some of the optimizations that optimized the model.

2.1.2 Quantization

Quantization is a process that reduces the number of bits that represents a number. The desire for reduced bandwidth and compute requirement has driven research to focus on low bit resolution format. 8Bit integer has shown extensive progress in reducing the model size with minimum accuracy loss. The use of even lower bit width such as 4 2 1 is an active field of research which has promised to show great progress. The method that we used requires quantization aware training. we use Range-Based Linear Quantization. Here linear means the floating point value is quantized with a numeric constant also called as the scaling factor. The scale factor is calculated by looking at the actual range of the tensor's values. A naive approach which is generally called clipping based only takes the min max of the tensor. An advance technique uses some derivation based on the tensor's range which removes the possible outliers in the network. Further moving there are again two types of quantization modes:

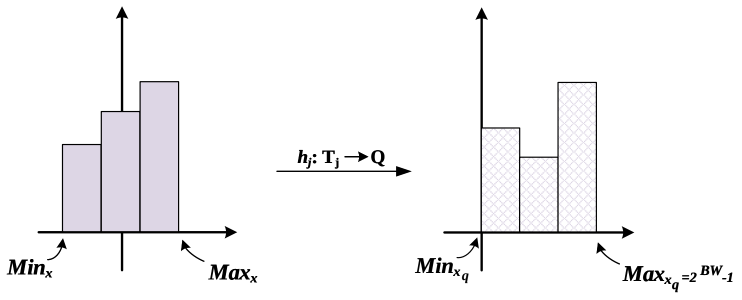


Figure 2.1: Symmetric and Asymmetric Qauntization

- Asymmetric In Asymmetric mode the min/max value of the float are mapped to the min max value of the integer. This mapping requires a zero point also known as quantization bias. The final range is always in between 0 and 2 raised to n bit width. The quantized values are always positive

- Symmetric In Symmetric mode the maximum absolute value between min and max is cheesed to map the min/max of float with the min/max of integer value. The the mapped values are between a range of -ve to +ve. Thus here we do not need any zero point.

2.1.3 Xilinx Vivado HLS

Recent advances in the field of High Level Synthesis has enabled the software community to take advantage of the Hardwware by hiding the complexities related to Digital Hardware Design. HLS is a automated design process that interprets an algorithm description of a desired behaviour and creates a digital hardware which can be implemented on a FPGA. Xilinx HLS has expedited the hardware design process. It allows functions written in C, C++, SystemC, and OpenCL kernels to be synthesized into RTL design. It also provides specialized libraries for math functions. HLS also provide support for any arbitrary bit width data type. It also provides us with streaming interface for data structures which are designed to obtain best performance and area. Xilnx has provided us with compiler directives or optimization pragmas. These pragmas help us to determine the required amount of parallelism in the application. Below are some example of the prgamas that are used in the current design

2.1.3.1 pragma HLS DATAFLOW

The Dataflow pragma increases the concurrency fo the RTL implementation and increased the overall throughput of the design by allowing functions and loop to overlap in their operations by enabling the task level pipeline. The functions inside a data flow region start concurrently. Thus if there is a data dependency between 2 loops the dataflow can result into inefficient design. Sharing data from same array inside the dataflow region is not permitted in HLS.

2.1.3.2 pragma HLS PIPELINE II=1

The pipeline pragma reduces the initiation interval by allowing the concurrent execution of the operations. In default prgama is tries to make the computation to run in single cycle. Thus any thing that is inside the pipeline pragma is required to run concurrently.

2.1.3.3 pragma HLS ARRAY PARTITION

By default every BRAM inside the ZCU012 has 2 read and write ports. Using this prgma we can increases the read and write ports for the storage. When we are trying to perform several computations at the same time the data demanded by the computation needs to be supplied simultaneously. This is only possible by array partitioning the data array that is accessed by the communication buffer.

2.1.4 Pytorch

For training the DNNs we used pytorch. Pytorch framework provides us with awesome debugging capabilities. The dataset used for training in ImageNet. For quantization and batch normalization we use intel distiller. This distiller can convert pytorh models to support quantization.

2.2 Related work

Modern CNN accelerators can be divided into two main categories: single compute engine [1, 2, 3, 4, 5, 6], and multiple streaming compute engines [7, 8, 9, 10, 11, 3]. Single compute-engine accelerators are typically a systolic array of processing elements (PEs). These kind on 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 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 [12, 13].

Many recent frameworks have proposed a vertical design flow from algorithm to the hardware [1, 7, 3, 9, 14]. 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 [1]. 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. [5] 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 [1], requires user interaction to define the nested convolutional loop using custom pragmas in C++. The custom front-end 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.2.2 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.

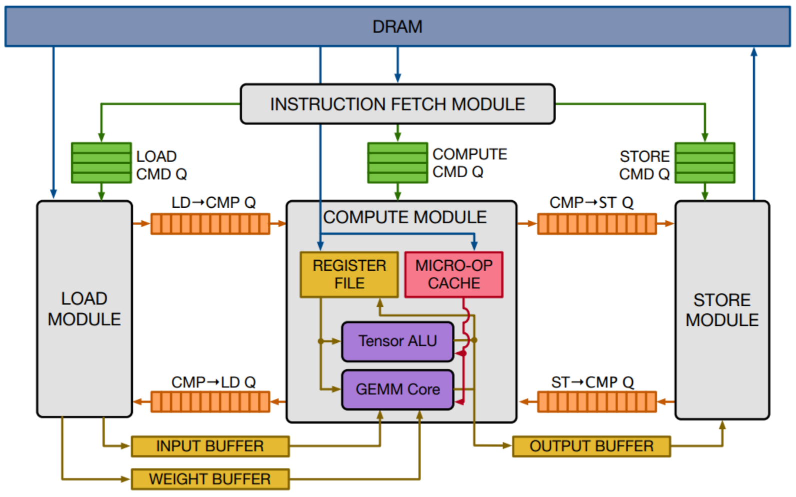


Figure 2.2: VTA (Vresatile Tensor Processing Unit)

The design proposed in [15] presents a framework to minimize the complexity and the model size of dense CNN by mapping normal convolution to depthwise separable

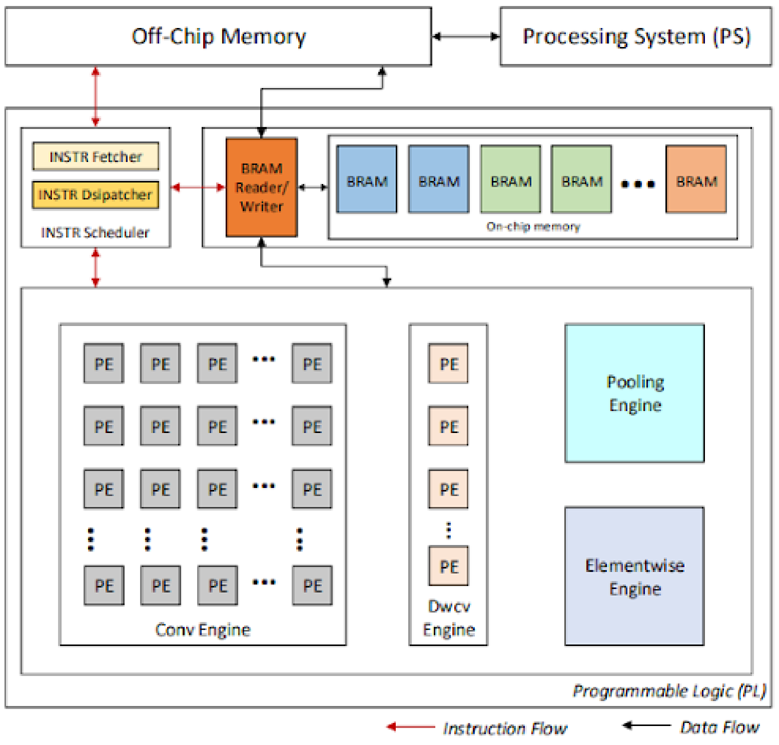


Figure 2.3: DPU (Deep Neural Processing Unit)

convolution. Similarly, TuRF [16] replaces standard convolution layers with depthwise separable convolution and applies layer fusion to enhance the performance of dense networks. The design presented by [17] is another hardware accelerator based on matrix multiplication and customized adder-tree to support MobileNet. However, their fixed design platform is not scalable to support fast-growing and forthcoming DSCNNs. A parallel acceleration scheme proposed in [18], 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 based hardware accelerator on FP32 computation is presented in [19]. DPU [20] is another solution to support MobileNet 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:2.3 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 present a fully vertical framework to implement the-state-of-the-art DSCNN architectures, e.g., EfficientNet family.

CHAPTER 3: DEEPDIVE FRONTEND

This section illustrates the Front End of the DeepDive. This is mainly responsible for bringing hardware-awareness into training DSCNNs. Fig 4 gives a brif understanding about the frontend and its curresoponfin output. A pre-trained floating point network is the 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. Lets discuss this 2 approaches further.

3.1 Batch Normalization Fusing

Batch normalization increases the stability of the network by normalizing the output activation layer by subtracting the batch mean and divining the batch standard deviation. It also increases the training speed of the network. The BN function is defined below

3.2 Online Channel-wise Low-bit Quantization

Quantization is a well-known approach to compress the network model size, and speed up the computation, by mapping number representations from floating-point single precision (FP32) to integer representation. Due to the malleability of FPGA fabrics, designers can greatly reduce the integer bit-width, while minimizing the introduced quantization error, by training the network for the new representation. Deep-

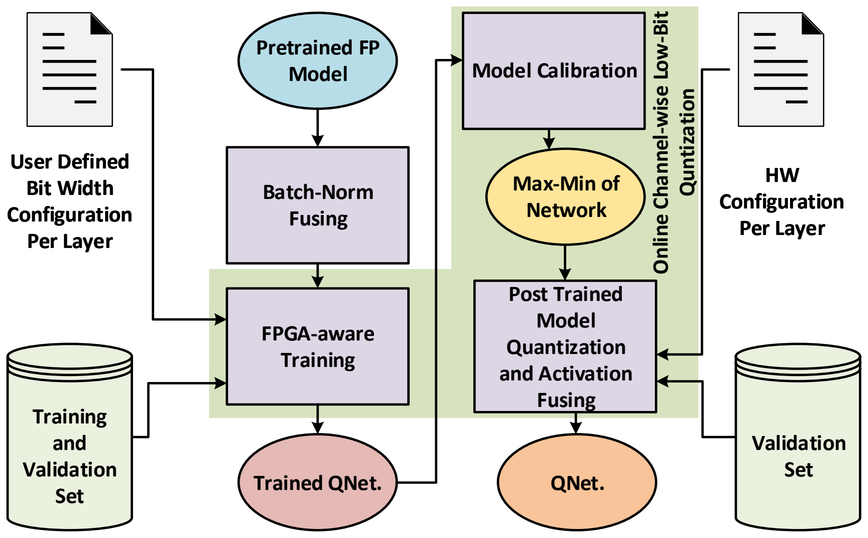

Pretrained FP Model
Batch-Norm Fusing
User Defined Bit Width Configuration Per Layer
FPGA-aware Training
Training and Validation Set
Trained QNet.
Model Calibration
Max-Min of Network
Post Trained Model Quantization and Activation Fusing
Online Channel-wise Low-Bit Quntization
HW Configuration Per Layer
Validation Set
QNet.

Figure 3.1: DeepDive: Front-end.

Dive applies the Range-Based Linear quantization to compress the network weights and biases. Let's define $T = \{x \mid x \in R\}$, such that T is the floating-point pre-trained network model. Function $h : T \rightarrow Q$ will map and scale T to Q, where Q is quantized integer representation set. Eq. 3.1 defines function h:

$$x = S(x_q + m_{zp}) \mid x_q, m_{zp} \in Q, \tag{3.1}$$

where $S \in R$, is the scaling factor, x_q is the quantized value, and m_{zp} is the zero-point defined to make the right-hand side of Eq. 3.1 equal zero when $x_{fp} = 0$. Based on the range of x_q, two methods of Asymmetric Representation and Symmetric Representation are defined. In asymmetric mode the $min_x = min(x)$ is mapped to 0, while $max_x = max(x)$ is $2^{BW} - 1$, while BW is the bit-width. In contrast, symmetric maps both $[min_x, max_x]$ to $[-(2^{BW-1}), \; 2^{BW-1} - 1]$. MobileNet-V2 uses ReLU6 as its non-linearity function — its output is always positive and less than 6. Therefore, we opted for the asymmetric method, since the negative range of the symmetric representation is not useful, and we are not able to benefit from the full range of representation; thus, it will have an impact on the output accuracy of each activation layer.

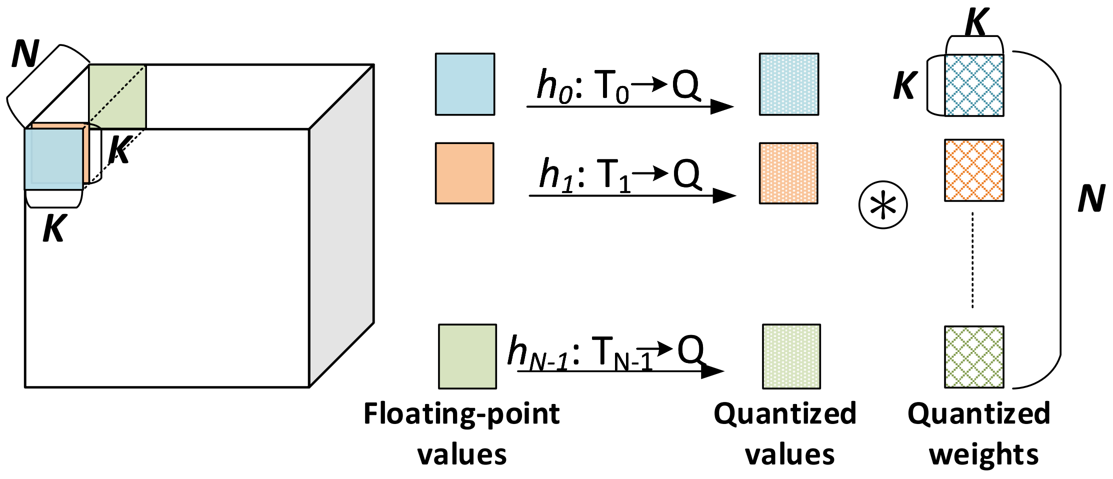


Figure 3.2: Per-channel range-based linear quantization. In this depthwise convolution example, per each N output channel, a separate mapping function is created.

DeepDive can quantize a network model per output channel, or per convolution layer. Per layer approach defines h function per whole convolution layer, while per-channel quantization defines $h_j \mid j = 0, \cdots, M-1$ per each output channel for a convolution operator. For instance, Fig. 3.2 shows the per-channel quantization approach for a depthwise convolution.

After the network is trained and quantized based on the user-provided configuration, the validation set is used again for the network model calibration. The calibration data will be used to make the trained network ready for post-training quantization. In this step, based on the acquired min-max, and the type of quantization, the scaling S and m_{zp} will be recalculated again to re-evaluate h_j, which results in $h_j^{pq} : [0,\ 6] \rightarrow [0,\ 2^{BW}-1]$. By applying this approach, DeepDive fuses the ReLU6 activation to the convolution operator.

CHAPTER 4: DEEPDIVE BACKEND

DeepDive's back-end offers a novel micro-architectural approach, and design flow, customized for efficient execution of DSCNNs on edge FPGAs. Fig. 4.1 presents the DeepDive back-end design flow. The heart of DeepDive's back-end 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 the network layout and data dependencies. It then creates key heterogeneous CUs, called *QNet Accelerators*, with respect to DeepDive's system architecture. In the following, at first, we describe micro-architectural details of convolutional operators, and then we discuss the details of the Network SoC compiler and system architecture.

4.1 Convolutional Operators

Since DeepDive is specially designed for DSCNNs, it naturally supports all convolutional operations, namely, normal convolution , depthwise convolution , and pointwise convolution . Each convolution operator buffers minimum job data size, which is necessary to start the computation, with the assumption that the network parameters necessary for computing are transferred to internal memory, and that the intermediate feature maps are streamed in and out. These operators are pipelined and parallelized in a way that is ideal for both memory-bound and compute-bound

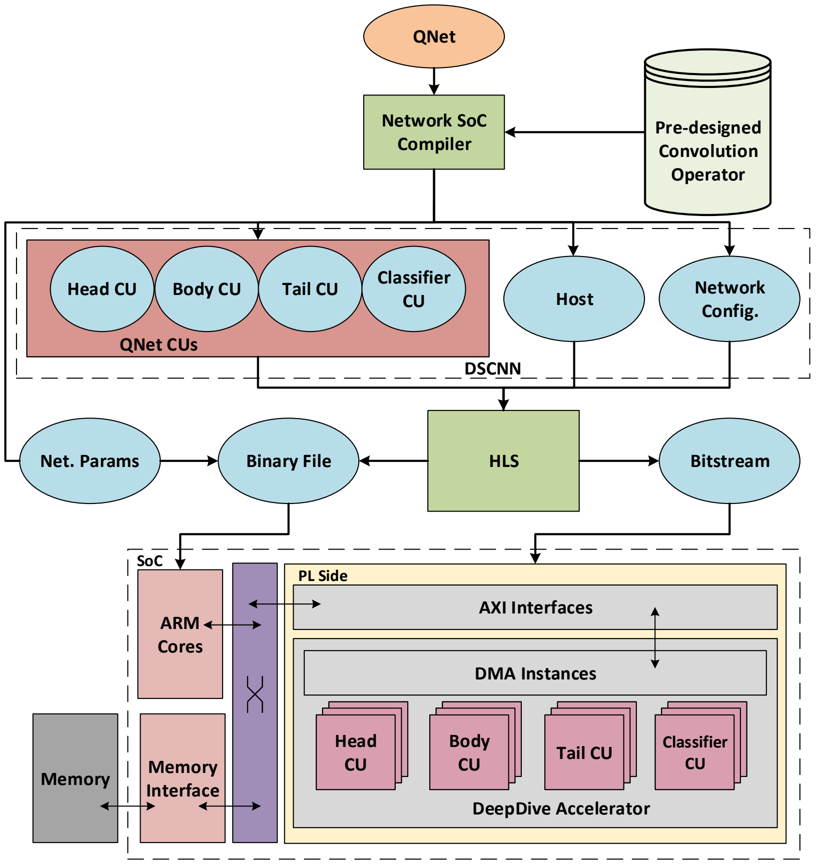


Figure 4.1: DeepDive: Back-end.

operations. The heart of a convolutional operator is a reconfigurable *Direct Convolution* core with different degrees of parallelism. The amount of parallelism defines the utilization, and parallel read/write ports required by the scratchpad or local buffers. This flexibility allows the Network SoC Compiler to manage the resources efficiently by tweaking the parallelism knobs to achieve the best performance (will be further discussed in section 4.2). Next, we elaborate on each operator from the design standpoint. In addition, we formulate the amount of parallelism per each convolutional operator.

4.1.1 Normal Convolution

The DSCNN has one normal convolution , and it is the first operator to embed patterns from both spatial and channel dimensions from the given input image. Since the next layer after normal convolution is depthwise , it is essential to generate output pixels column-wise (spatial dimension) so the depthwise can start the job immediately. Therefore, we improve the parallelism level by having a dedicated adder tree located after the direct convolution kernel for the input channel reduction. The block diagram of normal convolution is, also shown in Fig. 4.2. The parallelism in normal convolution is across kernel size and input channels — described in the following: Parallel Ops = $\mathrm{K}^{nc}_{max} \times K^{nc}_{max} \times N^{nc}_{max}$, 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 movements compared to the depthwise convolution due to the pipelined adder tree implemented at the end of direct convolution core.

4.1.2 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. 4.3. Once the computation is finished, the data in the computation core will be flushed and reloaded with the new 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. 4.2 presents the micro-architecture of depthwise and normal convolution operators. As depicted in Fig. 4.2, 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 scratch pad. 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,

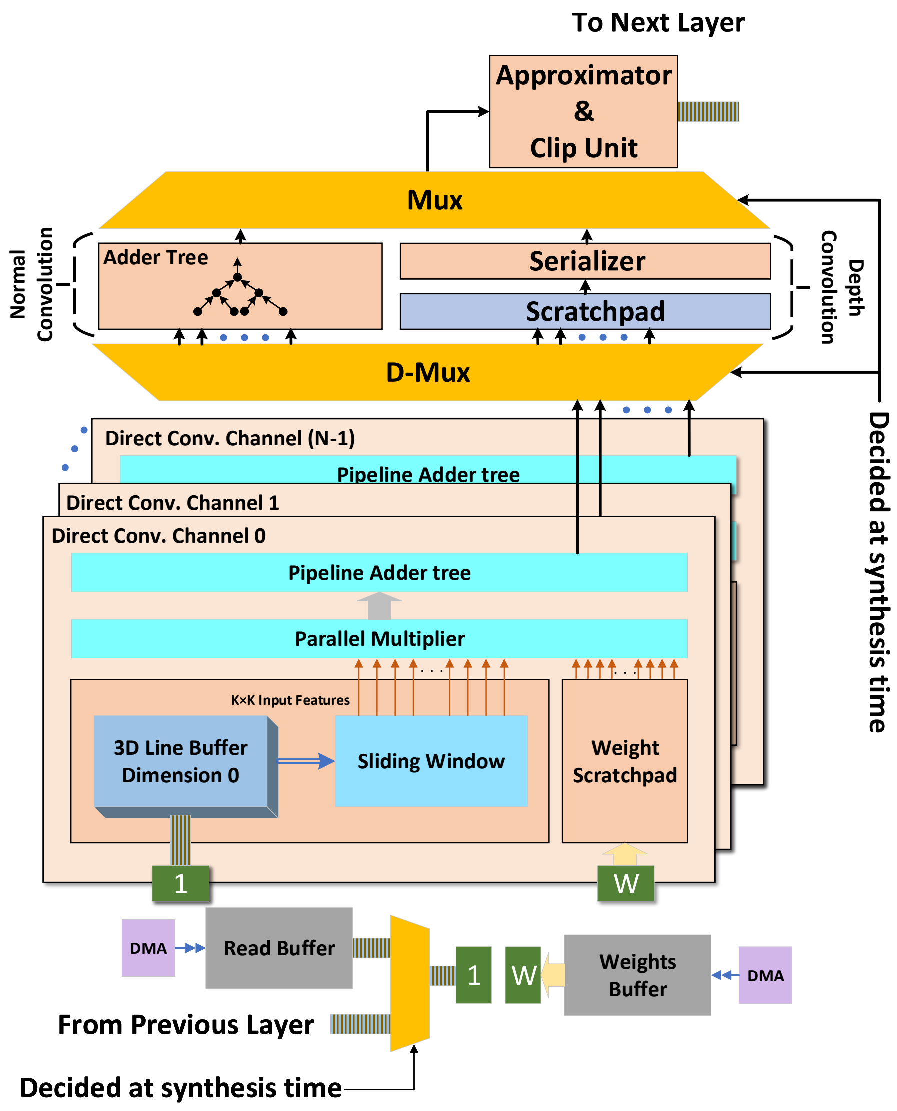


Figure 4.2: Schematic block diagram of depthwise and normal convolution

and a pipelined adder tree, together which carryout the MAC operation, followed by the 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 in MobileNet V2 or EfficientNet. The depthwise convolution is more sparse, and has the least amount of data reuse. The maximum parallel operations are calculated as the following: Parallel Ops $= \mathrm{K}^{dw}_{max} \times K^{dw}_{max} \times N^{dw}_{max}$, In Eq. 4.1.2, K^{dw}_{max}, and N^{dw}_{max} are the maximum kernel size and maximum input-channel across all

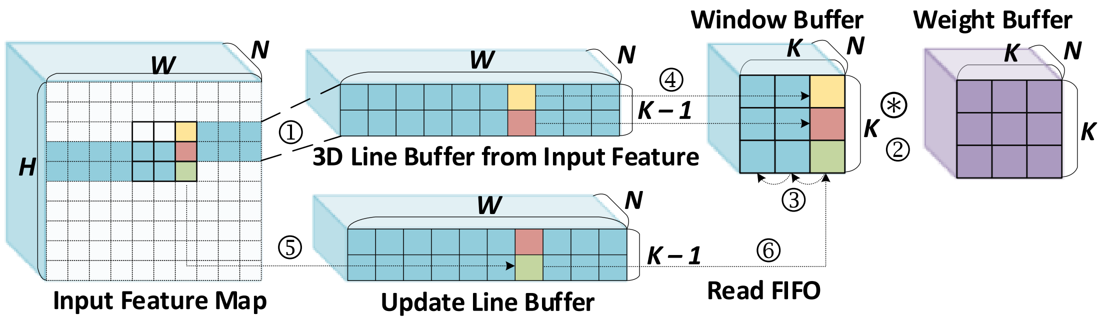


Figure 4.3: Shift and updateThe data movement and update mechanism of Window and Line Buffer. ① Line Buffer is filled with input feature data. ② Window Buffer is convoluted with weights. ③ The data in window is left shifted. ④ New data from the line buffer is copied in to the window. ⑤ & ⑥ Data from the FIFO is then copied into the line buffer and window buffer. All the Data Movements are pipelined.

the depthwise convolutions in the network, respectively.

4.1.3 Pointwise Convolution

Due to the dense operation of pointwise , the design of this operator can be similar to the design of a general matrix multiplication, which is well suited for the systolic array. With maximum data reuse, this operator can leverage maximum parallelism. It has both fewer algorithmic, and fewer data movement complexity, which makes it best fit for a high amount of parallelism. Fig. 4.4 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 parallel fashion. The amount of parallelism for our design is across the input channels

Parallel Ops $= \mathrm{N}_{max}^{PW_{type}}$,

where $N_{max}^{PW_{type}}$ is the maximum input channel size across all the specific *type* (eg. projection or expansion pointwise in the MobileNet V2) of pointwise convolutions mapped to specific compute unit.

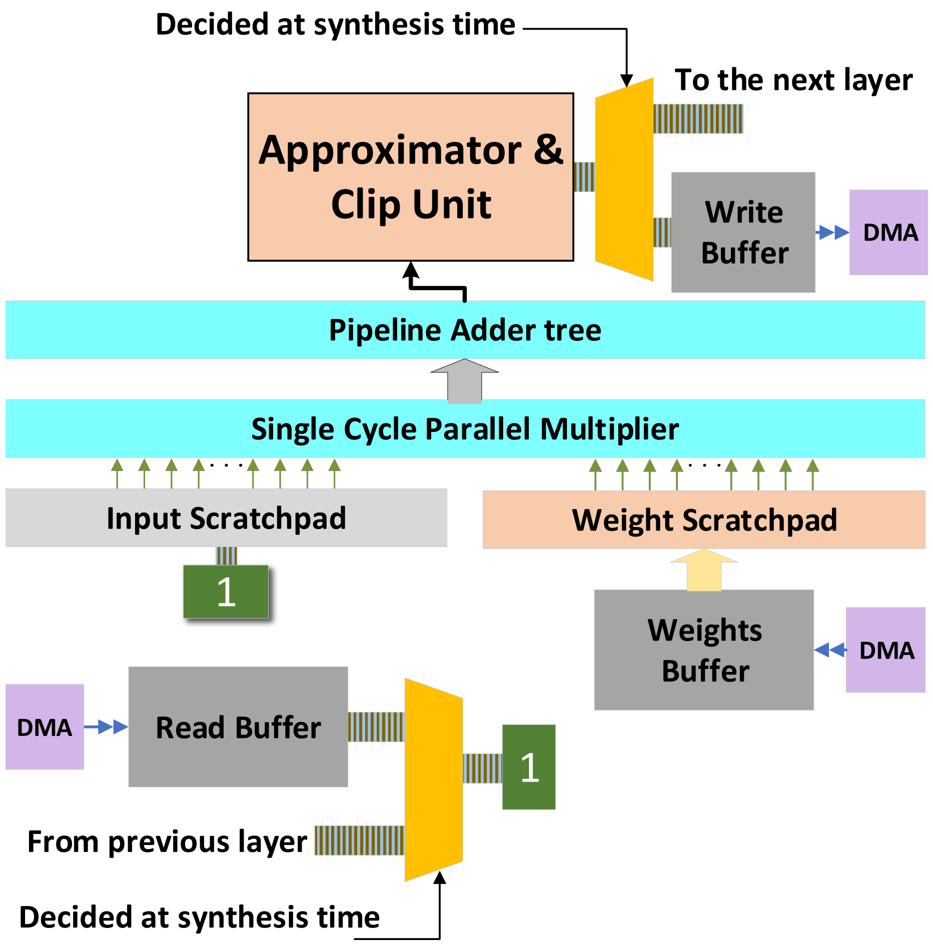


Figure 4.4: Schematic block diagram of Pointwise Convolution.

4.2 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*. Below, we describe the details of Network

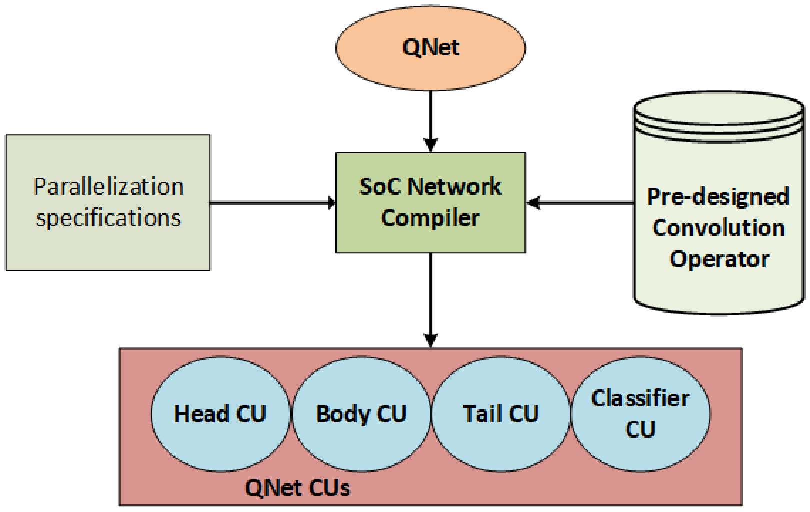


Figure 4.5: Network SoC Compiler

SoC Synthesizer including, system architecture, memory organization, Heterogeneous *QNet* CUs, host code scheduling and CUs management.

4.2.1 DepDive System Architecture

As emphasized before, the convolutional operators of DSCNNs demonstrate a repetitive structural behavior wherein some either appear once, or they are repeated across the entire network. Depending on the recurrence of the convolutional operators, they are mapped to the Head, Body, Tail, and Classifier CU. Fig. 4.6 shows the system architecture of DeepDive Hardware Accelerator. Each CU has its own dedicated Direct Memory Access (DMA), and its parameters, such as array pointers, N, M, and H, can be configured at runtime via the control bus (e.g., AXI Lite Bus). After configuration, each CU can transfer the input/output features map and weights tensors via streaming channels (e.g., AXI HP Interface) through System Memory Management Unit (SMMU). The composition of CU is parameterized by the buffer shapes, data type widths, and the computation core, which are a few of the architectural knobs provided while designing the hardware accelerator. This makes our design scalable and reconfigurable for DSCNNs. We will discuss our hardware knobs and each CU's internal composition in detail after we explain the memory transactions and management. The CUs are scheduled and pipelined to increase the concurrency.

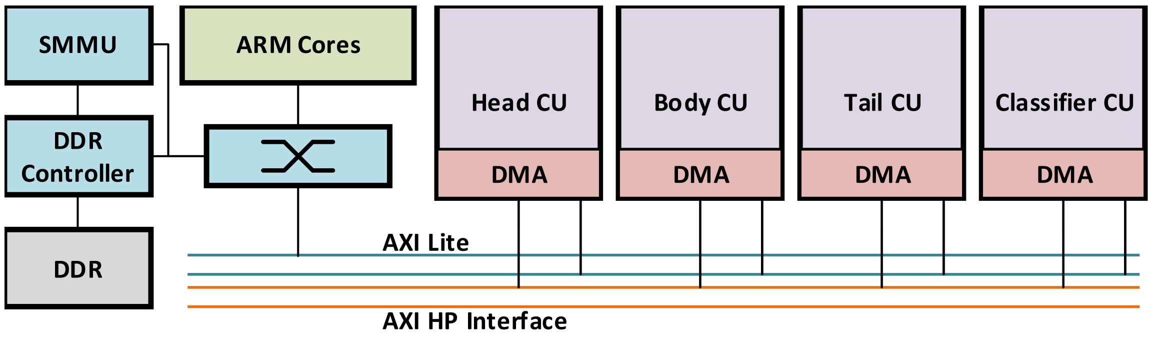


Figure 4.6: System level architecture of DeepDive.

4.2.2 Memory Organization

Each CU has its own dedicated buffer and scratchpad to handle its memory requirements. The memory layout of the on-chip buffers are designed to satisfy the data access pattern required by the convolutional operators, in order to minimize the pipeline depth implemented in the computation core. The memory transactions in the CUs can be categorized into two groups: ① memory to memory transaction, where data is burst read from DDR memory to PL memory, and ② memory to stream transaction, where data is streamed via DMA to or from PL memory. As an example, Fig. 4.7 demonstrates the memory transactions for Head CU targeted for MobileNet V2 . Convolutional network parameters like weights, quantization parameters, and biases are burst read from DDR to PL buffers. The input/output feature maps are streamed from DDR to PL. Apart from memory transactions of input/output features between DDR and PL, the inter-CU data transfers within its operators also occurs in streaming fashion, where intermediate feature map data is streamed in-between different convolutional layers. Stream FIFO offers two main advantages, memory and computation latency overlap and data movement reduction between DDR and PL.

4.2.3 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 DSC-NNs. The CUs are completely parameterizable, and customizable, for scalability and flexibility. Following section describes each CU in detail. We also provide illustrative figures for the example of MobileNet V2 .

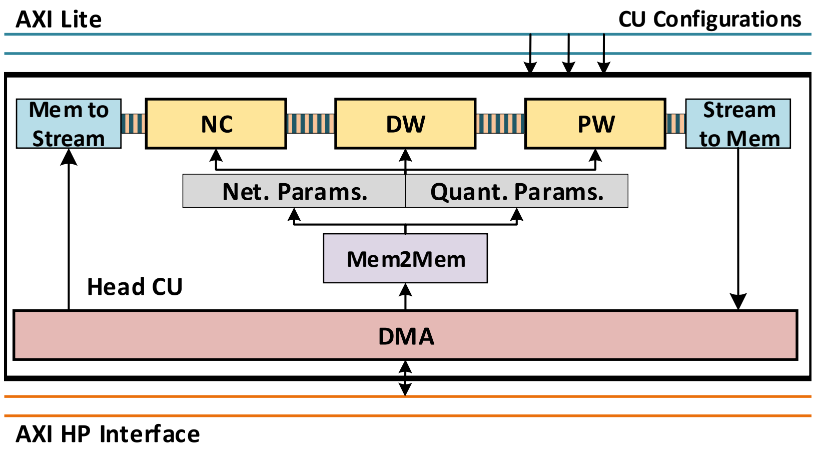


Figure 4.7: MobilenNet V2 Head Computing Unit

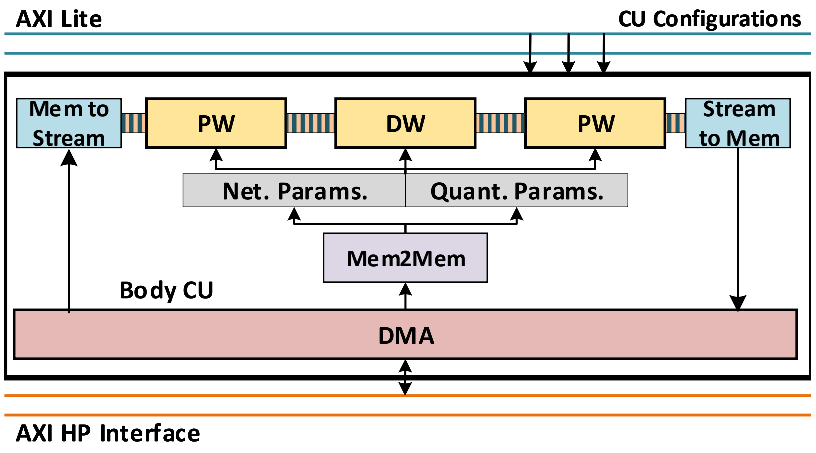


Figure 4.8: MobilenNet V2 Body Computing Unit

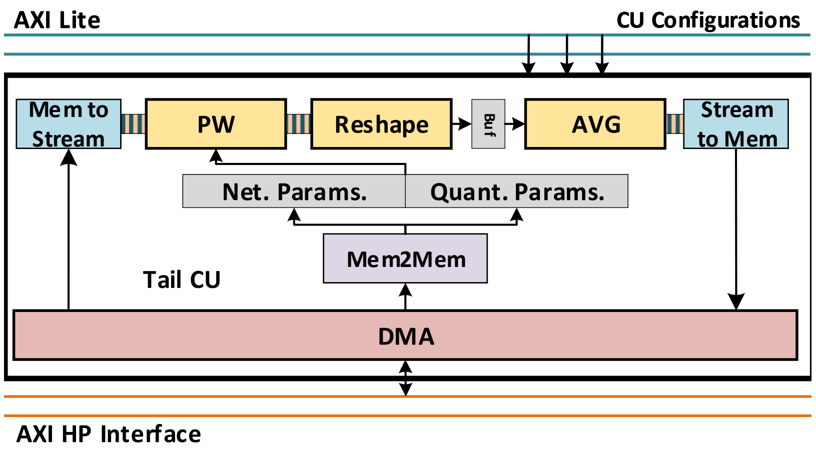


Figure 4.9: MobilenNet V2 Tail Computing Unit

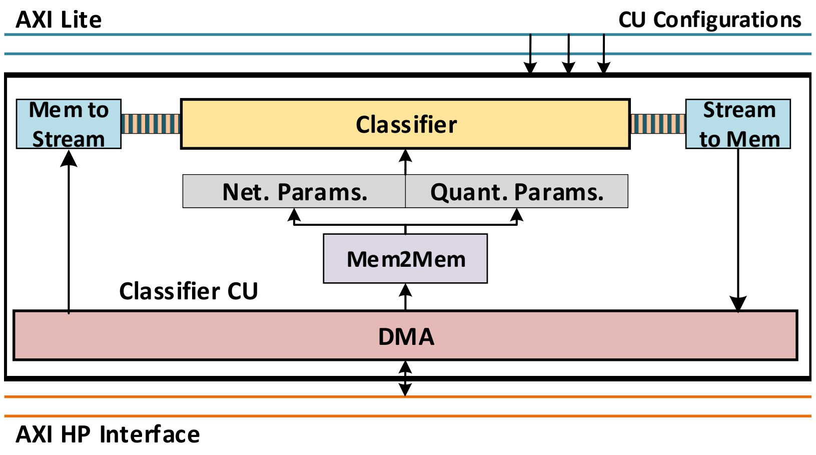


Figure 4.10: MobilenNet V2 Classifier Computing Unit

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. As explained in the section **??**, the Head CU has its own dedicated internal memory for buffers. The data transactions occur in memory-to-memory mode and the intermediate data streams between convolutional layers within the head CU. As an example, Fig. 4.7 demonstrates the Head CU for MobileNet V2 model, which is composed of normal convolution followed by depthwise and pointwise convolution , 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.

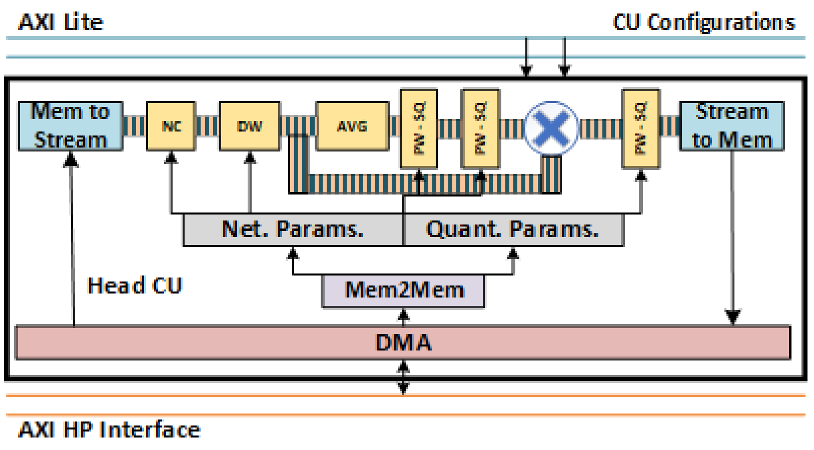


Figure 4.11: EfficientNet Head Computing Unit

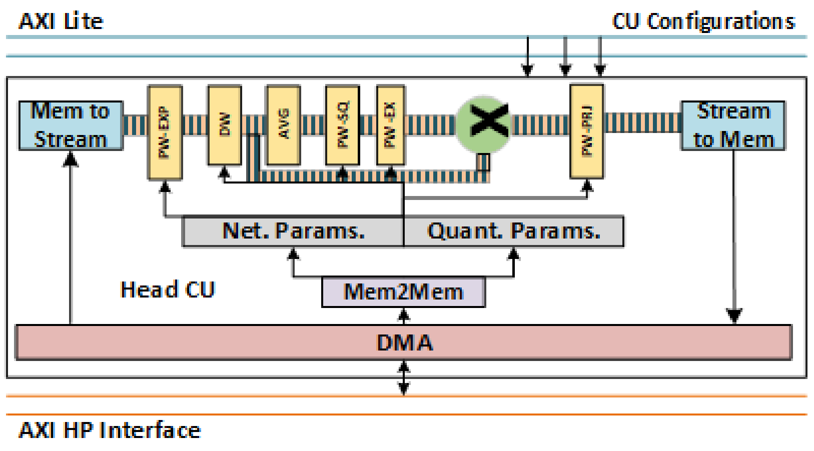


Figure 4.12: EfficientNet Body Computing Unit

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 MobileNet V2 , is entirely mapped to the Body CU. The IRB consists of pointwise (expansion), depthwise , and pointwise (projection) layers, all running concurrently in a fused fashion within the Body CU. Fig. 4.8 shows the structure of this CU for MobileNet V2 . 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. 4.9 represents the structure of Tail CU in MobileNet V2 . 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. 4.10 represents the MobileNet V2 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.

4.2.4 Host Code Scheduling and CUs Management

Finally, the Network Soc Compiler also manages the host-level scheduling of CUs. Fig. 4.13 visualizes the CUs scheduling and their memory footprints on shared memory. The host or PS initializes the DDR with network models and quantization parameters. The DeepDive back-end generates the memory layout so that the network data region is shared between PL and PS. Therefore at each CU invocation, the PS only passes the data pointer, and the PL fetches the data based on the provided pointer rather than copying the data to its region. This memory layout will remove the necessity of copying data between the PL and PS memory region. The host starts scheduling procedure by configuring the Head CU with appropriate memory pointer addresses, offsets, network parameters, and network configuration, i.e., M, N, H, which are compiled into network configuration header files. When Head CU completes execution, it writes back the data in feature tensors and interrupts the host CPU. Following the same trend, the host will schedule the Body CUs for j times, where j is the number of Body CU invocations calculated based on CU's mapping. Host CPU then schedules the Tail CU, which executes the compute-bound operations quickly. And finally, the last call is to the Classifier CU, which will update the content of feature tensor needed by the softmax layer to calculate the confidence. Host CPU creates a sequential yet fused scheduling and management of CUs for DSCNNs.

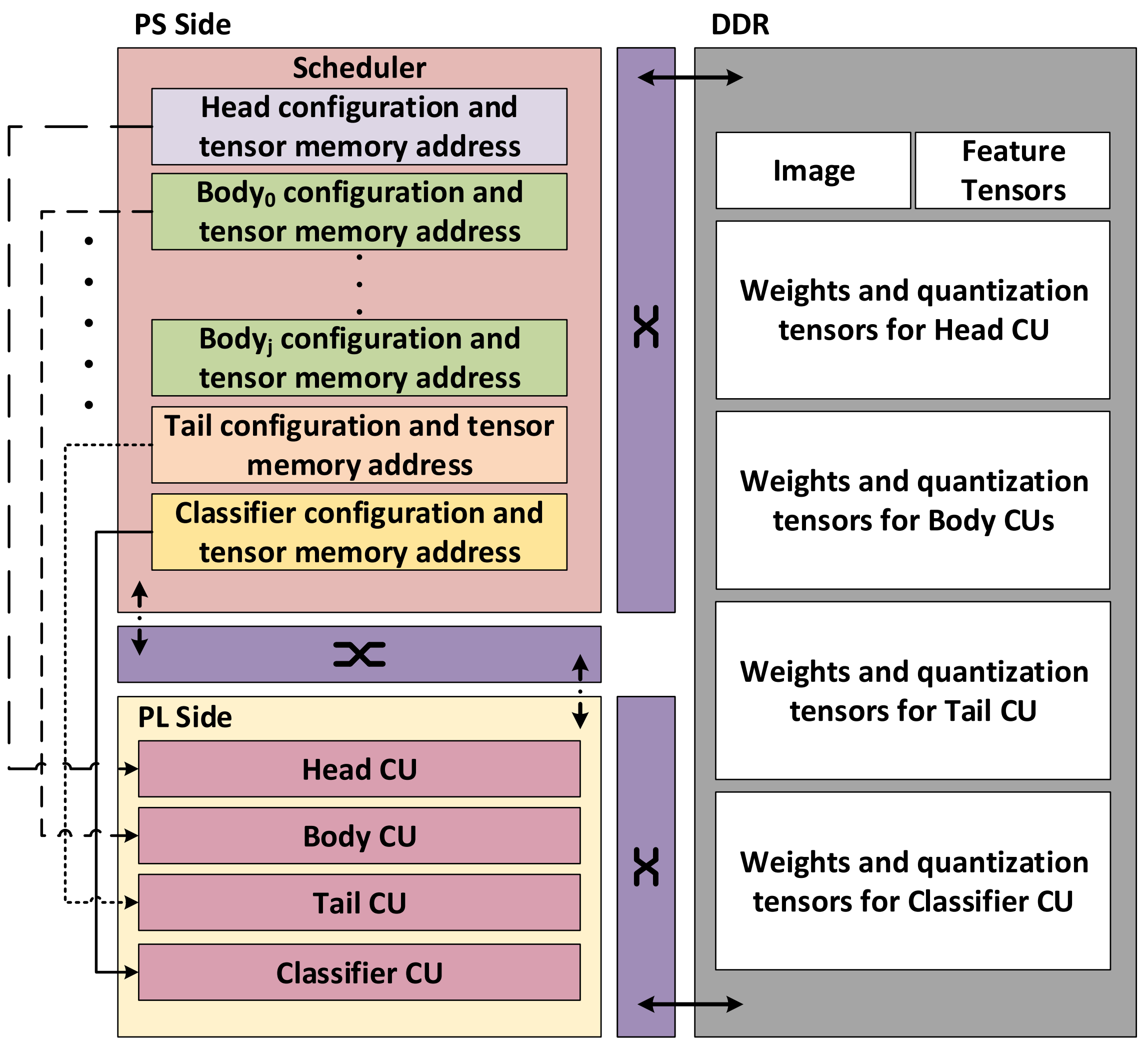

PS Side
DDR
Scheduler
Head configuration and tensor memory address
Body_0 configuration and tensor memory address
Body_j configuration and tensor memory address
Tail configuration and tensor memory address
Classifier configuration and tensor memory address
PL Side
Head CU
Body CU
Tail CU
Classifier CU
Image
Feature Tensors
Weights and quantization tensors for Head CU
Weights and quantization tensors for Body CUs
Weights and quantization tensors for Tail CU
Weights and quantization tensors for Classifier CU

Figure 4.13: Host level scheduling and memory footprint of CUs.

CHAPTER 5: EXPERIMENTAL RESULTS

5.1 Case Study: MobileNet V2

The procedure starts from a PyTorch model of MobileNet-V2, 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. Fig. **??** shows the Top-1 accuracy for MobileNet-V2 when its $\alpha = 0.75$ and $H = 160$. As can be seen, DeepDive maintains accuracy with respect to FP32 by reducing the BW to 8 for first Normal Convolution, and 4 for the rest of the layers, respectively. The per layer-specific quantization compresses the model size with a ratio of 8, with 4.4% degradation in Top1 accuracy. The results demonstrate a dramatic drop in accuracy for $BW = 3$. For the rest of this case study, $BW = 4$, as it achieves competitive accuracy with considerably smaller model size.

5.1.1 Design Exploration

The front-end is configured to re-train, quantize, and calibrate the network for different α and H values. Table 5.1 summarizes the model size, operation numbers and Top1 accuracy per each design point. Based on Table 5.1, we observe that model size is only effected by α, while the number of operation number is a function of both α and H. Top1 accuracy is also a function of both H and α; however, it is not a linear relationship. For instance, design point $(H = 224, \alpha = 0.75)$ has better Top1 accuracy compared to design point $(H = 160, \alpha = 1)$ while its model size is 33% less than the latter one. Therefore, we introduce the network complexity as the product of the network model size and network operation number to consider both of them.

Fig. **??** depicts the Top1-Network Complexity Pareto front. The network complex-

Table 5.1: Effect of altering α and H for fixed $BW = 4$

α	1					0.75					0.5					0.35				
H	224	192	160	128	96	224	192	160	128	96	224	192	160	128	96	224	192	160	128	96
Params(Mb)	13.31	13.31	13.31	13.31	13.31	10.01	10.01	10.01	10.01	10.01	7.48	7.48	7.48	7.48	7.48	6.37	6.37	6.37	6.37	6.37
#Ops(M)	313.621	230.755	160.638	103.269	58.649	220.326	162.212	113.038	72.805	41.513	104.164	76.868	53.772	34.875	20.177	64.835	47.973	33.706	22.033	12.953
Top1(%)	69.07	67.256	65.78	62.3	56.036	66.404	64.364	59.928	53.112	43.002	59.502	57.452	52.608	45.316	34.88	54.43	51.214	46.59	39.328	27.2

Table 5.2: Effect of altering α and H for fixed $BW = 4$ at 200Mhz on FPS and FPGA Resource Utilization

α	0.75					0.5					0.35				
H	224	192	160	128	96	224	192	160	128	96	224	192	160	128	96
FPS	11	14	18	22	28	16	19	25	30	37	20	25	31	40	51
Power(mW)	460	450	440	370	350	400	320	310	300	290	270	270	260	250	250
DSP(%)	57	57	58	57	57	37	37	37	37	37	24	24	24	24	24
LUTs(%)	75	74	76	74	74	71	70	70	70	70	68	67	67	67	67
BRAM(%)	96	96	97	92	90	92	91	89	88	87	84	84	82	81	80

ity helps the front-end to measure the final hardware complexity at a higher level of abstraction. We annotate the starting point of each α in this figure and one non-Pareto point for the sake of comparison. Here, we observed that the design point ($H = 96, \alpha = 1$) has approximately the same network complexity with respect to ($H = 224, \alpha = 0.5$), while its Top1 accuracy is almost 4% less than top achievable accuracy.

5.1.2 Execution Results and Comparison

This subsection evaluates DeepDive's execution performance for MobileNet-V2 on the Hardware Accelerator, different energy-efficient design points implementations, and finally provides a comparison against two other FPGA accelerators [14, 18]. Since there are no other solutions that support both MobileNet-V2 and EfficientNet, we also compare it against Nvidia's Jetson Nano as existing state-of-the-art system.

Mapping: As discussed in section **??**, based on the network graph generated by Network Compiler, DeepDive's back-end identifies the mapping between the convolutional operators and heterogeneous CUs. Fig. 5.1 reveals the mapping of MobileNet-V2 to heterogeneous CUs. The Head, Tail, and Classifier CU are scheduled only once, but the Body is scheduled 16 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. For $\alpha = 1.0$, DeepDive was not able to fit the design in XCZU9EG SoC chip. If we configure the DeepDive to select different values, less than N_{max} per operator, we observed a significant degradation in the final accelerator performance. Therefore, for the rest of this section, we did not consider these design points.

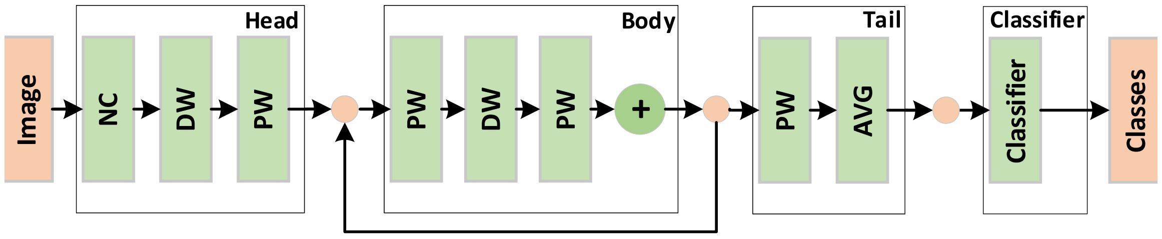


Figure 5.1: MobileNet V2 mapped to CUs.

Energy efficiency: Here we configure the back-end to compile different network architecture by altering α and H. Multiple fully functional execution instances have been created for all configurations in Table 5.1, except when $\alpha = 1.0$.

Table 5.2 summarizes the power consummation, FPS, and hardware utilization. The power is measured using a power monitoring device, as shown in Fig. **??**. The measured power is the difference of the idle power dissipation of the board and the power consumed by the DeepDive accelerator while running inference. This power is consumed by MPSoC (ARM cores + FPGA fabric), memory hierarchies, and shared DDR memory during the inference. Resource utilization is directly proportional to α, while the power is a function of both α and the input resolution H. Design point $(H = 96, \alpha = 0.35)$ has the lowest power consumption at 250mW, as compared to $(H = 224, \alpha = 0.75)$ with the highest power consumption at 460mW. Fig. 5.2 depicts Top1-Energy Efficiency (FPS/Watt) Pareto front. We only annotate the design points that have a higher than 50% accuracy. DeepDive enables us to understand the relationship between energy efficiency and accuracy. As we can see, design point $(H = 160, \alpha = 0.75)$ has almost same FSP/Watt and Top1 accuracy with $(H = 224, \alpha = 0.5)$.

Similarly, the next design point, $(H = 192, \alpha = 0.5)$, can improve energy efficiency by 45.14%, while the accuracy is dropped by only 2.48%. Based on the design points provided by DeepDive, it can be observed that by decreasing α and increasing the H, we can improve FPS/Watt without sacrificing the Top1-accuracy dramatically.

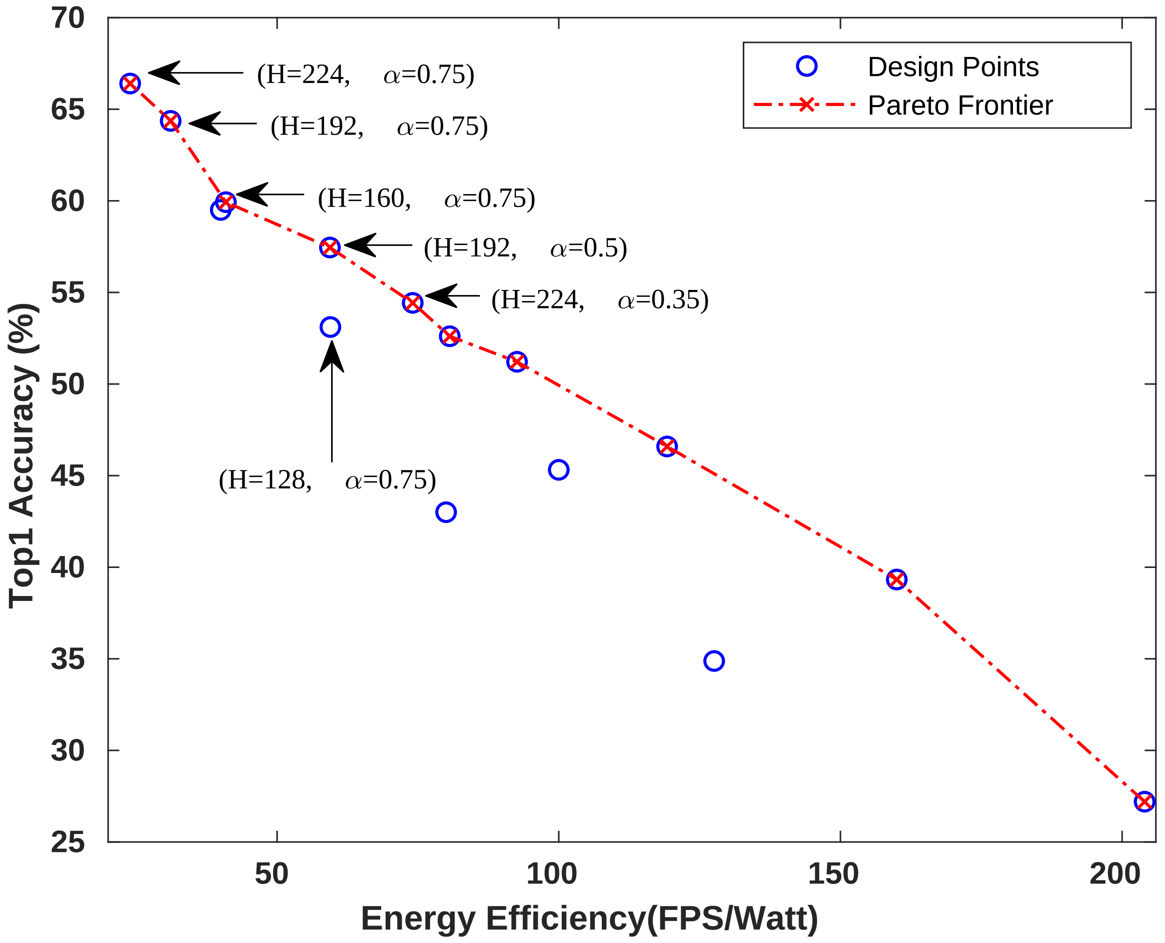


Figure 5.2: Top1-Energy Efficiency Pareto front. Design point $(H = 192, \alpha = 0.5)$ and $(H = 128, \alpha = 0.75)$ has similar energy efficiency while Top1 accuracy for $(H = 192, \alpha = 0.5)$ is more.

Comparison: To showcase the energy efficiency of DeepDive, we compare its FPS/Watt against off-the-shelf Nvidia Jetson Nano IoT Edge Device. We mapped the design points of Table 5.2 to TensorRT and obtained the metrics after its graph optimization and quantization. Similar to the DeepDive, we calculate the power consumption only for inference time. We compared the delay and power consumption

Table 5.3: Power Consumption and delay for MobileNet

H	Power(W)			Delay(ms)		
	Nano(H)	Nano(L)	DeepDive	Nano(H)	Nano(L)	DeepDive
224	5.49	2.64	0.46	14.91	20.73	88.49
192	5.22	2.51	0.45	13.61	19.96	70.32
160	4.78	1.88	0.44	13.07	19.6	54.45
128	3.35	1.56	0.37	11.24	17.19	45.51
64	3.25	1.32	0.35	7.89	13.91	35.71

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. Fig. 5.3 shows the comparison of the Jetson Nano energy efficiency against DeepDive for different input sizes while $\alpha = 0.75$. DeepDive, on average, can improve the FPS/Watt 2.2$\times$ and 1.51$\times$ 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.

Table 5.4 provides a comparison between DeepDive configured with ($H = 224, \alpha = 0.75$) design and other similar accelerators. Since VTA's [14] architecture does not support depthwise convolution, they modify the MobileNet to have group convolutions instead of depthwise convolutions, coined MobileNetG. Their MobileNetG was not accessible; hence, there was no chance to present a straightforward comparison. However, we realized that ResNet-18 has almost same inference latency when com-

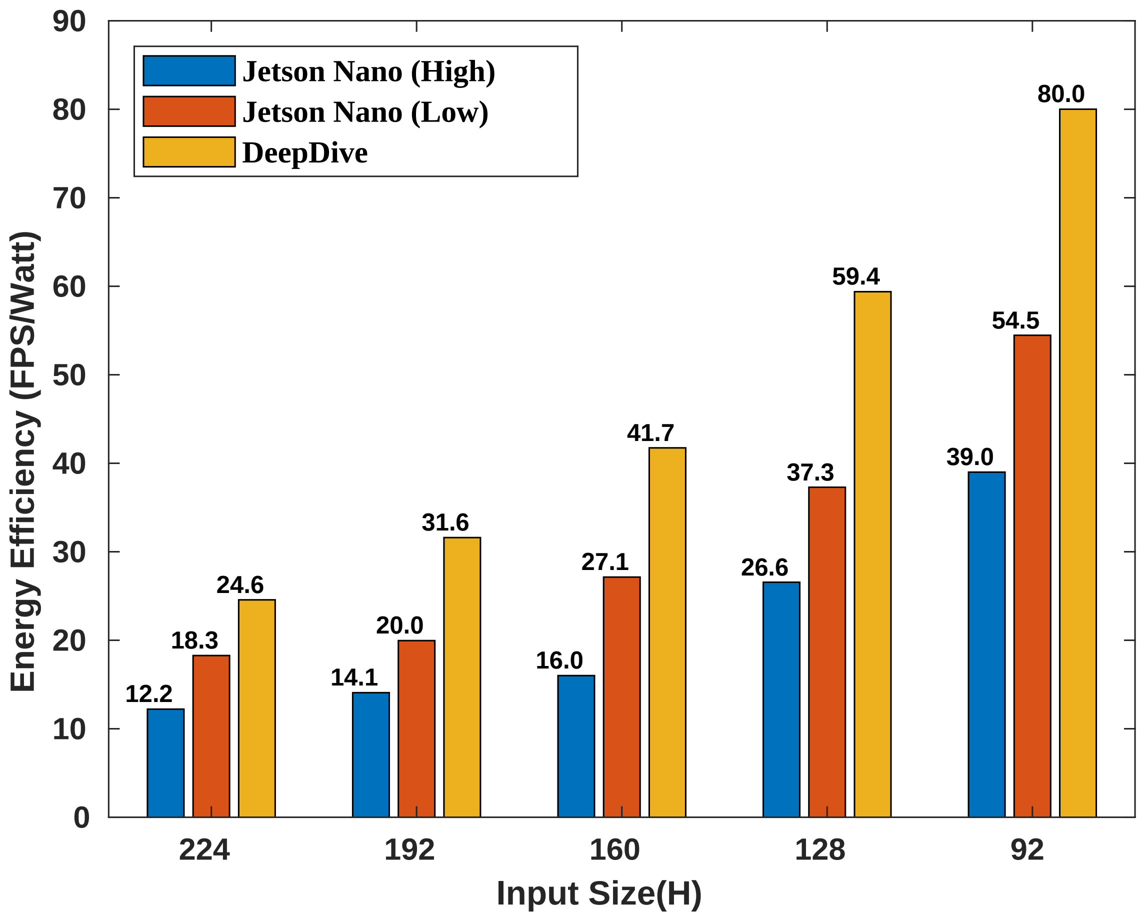


Figure 5.3: The energy efficiency (FPS/Energy) comparison of DeepDive against Jetson Nano for both high and low power mode.

pared to MobileNetG based on their results, so we decided to compare the energy efficiency of VTA running ResNet-18. As we can see, DeepDive can improve energy efficiency 2.27×. The instruction-based scheduling approach, and versatile systolic array adopted by VTA, both need to consume more power to decode instructions and map layers to the ALU sequentially, which leads to more shared memory transactions and higher power dissipation. Similarly, we compare DeepDive with the hardware accelerator presented by [18]. DeepDive outperforms [18] by 37.25× in energy efficiency. This improvement is because of two main reasons: ① Extreme bit-quantization, BN, and ReLU activation fusion accomplished by front-end which increases the efficiency of the hardware accelerator. ② DeepDive groups the convolutional operators in the CUs at higher granularity to overlap the memory transactions and computations.

Table 5.4: Performance Comparison in Classification

Design	Network	Platform	Freq. (MHz)	Speed (FPS)	Power (W)	Energy Efficiency (FPS/W)
VTA [14]	ResNet-18	ZCU102	200	15.44	1.47	10.51
[18]	0.5 MobileNet	ZYNQ 7Z045	100	1.38	2.15	0.6418
Ours	MobileNet-V2	ZCU102	200	11	0.46	23.91

5.1.2.1 6-bit Data-path

To showcase the ability to create random bit data-paths, We reconfigure DeepDive to generate and synthesize the MobileNet-V2 for $BW = 6$ to understand the effect of different bit resolution on the final Top1 accuracy and the hardware efficiency. We observe that $BW = 6$ can improve the Top1 accuracy by 1.49%, while the effectiveness of the hardware (FPS/W) drops by 4.88% on the average.

Overall, DeepDive improves hardware efficiency by adopting customized functional blocks for depthwise and pointwise convolutions. Heterogeneous CUs also remove unnecessary memory transactions between the PL and shared memory by fused pipeline execution across layers within a block which decreases the power consumption, while improving the overall system performance.

5.2 Case Study: EfficientNet

The baseline EfficientNet model was intentionally designed to be larger than MobileNet-V2. While this might be ideal for state-of-the-art accuracy, it was not suitable for low-power embedded devices. Taking advantage of the compound model scaling factors introduced in [21], we were able to compress the model using smaller α, network depth, and H, to achieve a model size capable of running on edge devices. The algorithmic details and hardware resource utilization of this model can be seen in Table 5.5.

Table 5.5: 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

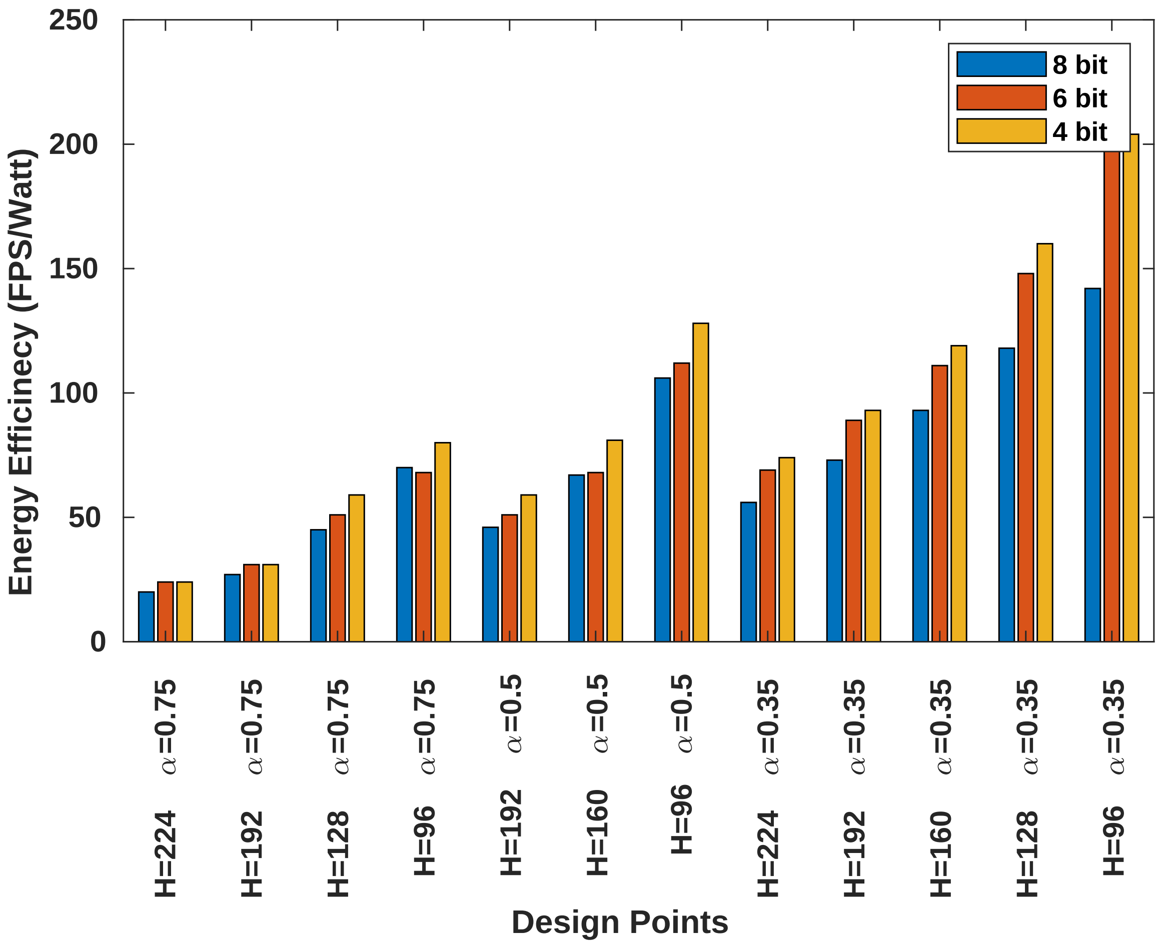


Figure 5.4: Effect of Quantization on Energy Efficiency

Mapping: EfficientNet is structurally different as compared to MobileNet-V2. Fig. 5.5 shows the mapping of EfficientNet to the CUs. The squeeze and excitation convolutional operators are represented as PW-SQ and PW-EX, respectively. DeepDive takes advantage of EfficientNet architecture by fusing more convolutional operators together. EfficientNet comparatively has a larger body than the MobileNet-V2, with six layers fused. This mapping helps in achieving better performance by reducing more memory transactions by invoking the Body CU only nine times. For the case of EfficientNet, we excluded the classifier from mapping and also comparison.

Energy Efficiency: As we can see in Table 5.5, the number of body CU invocation is 1.78× less than MobileNet-V2, which leads to less power consumption and higher FPS due to fewer memory transactions. Table 5.5 shows DeepDive reaches to 35 FPS

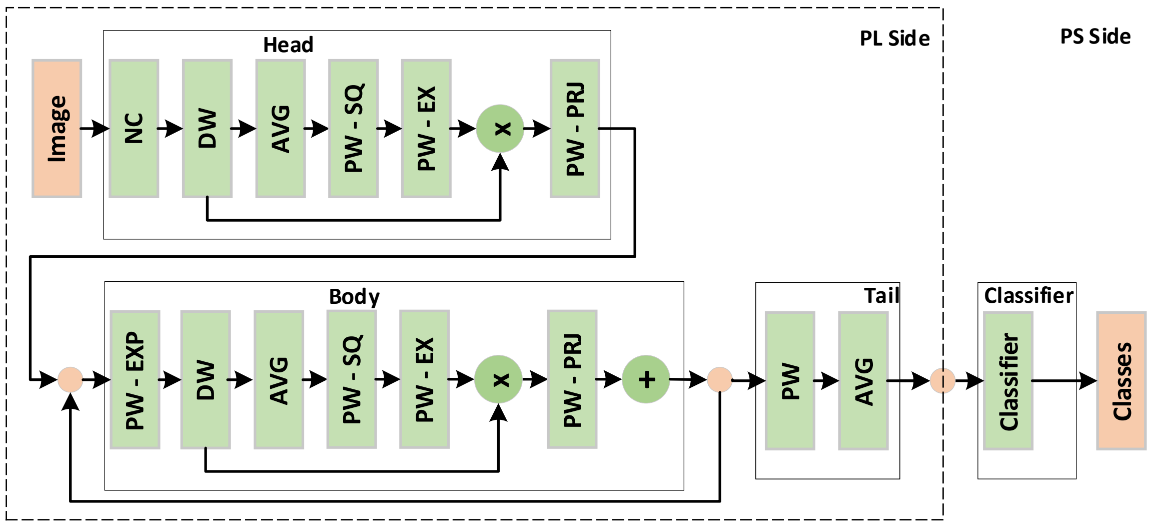


Figure 5.5: EfficientNet mapped to CUs.

Table 5.6: 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

for a power consummation of 150mW. This model gives us the Energy Efficiency of 233.3 FPS/Watt.

Comparison: Table 5.6 compares the FPS/Watt against Nvidia Jetson Nano. For EfficientNet, DeepDive can improve the FPS/Watt 8.6× and 6.7× against high and low power mode, respectively. Based on the massively fused layers in Body CU, fewer memory transactions translates to more energy-efficient hardware.

5.3 Moving to Vitis

Vitis is a unified programming model that supports both the edge and cloud computing applications. Vitis provides us with the flexibility to design and develop one single application which can be run on both edge and the cloud. Vitis AI and Vitis libraries allow end-to-end application acceleration with just software defined flow with minimum hardware expertise. Vitis is a successor of SDAceel and hence it follows OpenCL semantic. Vitis AI has a well vertically integrated software stack. It has algorithmic optimization built in into it. Right from compressing the model, quantization and pruning Vitis has build in support for all these operations. While conducting experiments to understand the difference of DeepDive on SDSoC and Vitis. We found that Vitis HLS is highly optimized for resources rather then performance. Which means you can fit even bigger design compared to SDSoC at the cost of performance. Vitis even solves the inherent timing issue that SDSoC was not able to synthesize. This sometimes comes at the cost of performance. Vitis tries to add resistors to relax the timing constrains and sacrifice more clock cycles for the smae. Vitis is more suitable for throughput oriented design rather then the latency oriented design. Overall Vitis unified flow makes it more user friendly when it comes to developing AI application. As Vitis uses OpenCL semantic style of execution. It also supports OpenCL profiling tools. Kernel profiling is done on the hardware level. Vitis uses openCL runtime which they call (XRT Runtime). The XRT consumes some resources inside the FPGA to do scheduling. Enabling Hardware Profiling puts additional content in the XRT to profile the hardware running on the system. It supports both Data Transfer and Compute profiling.

5.3.1 Effect of Vitis on Resources utilization of EfficientNet BC4 compared to SDSoc

As discussed Vitis is more optimized for resources rather the performance. We observed comparatively low resource utilization then compared to the SDSoC. Table: 5.7 shows the resource utilization of EfficientNet BC4 on SDSoC and Vitis. DSP and LUTs has similar amount of utilization for the same network whreas there is a massive reduction in the BRAM utilization when compared to SDSoC. Vitis uses 36% less BRAM compared to SDSoC. The FPS on vitis is low then the SDSoC as vitis design principles are different then the SDSoC Design principles. Vitis is more throughput oriented then SDSoC. on the other hand SDSoC is more performance oriented with less consideration for the resource. Hence Vitis has little poor performance when compared to SDSoC. Architectural modification which are more suitable to the Vitis design flow can boost the performance of the Vitis Design.

Table 5.7: Resourse and Performace comprison for SDSoC and Vitis

Software	DSP	BRAM	LUTs	Speed (FPS)	Frequncy
SDSoC	90%	68%	80%	35	200
Vitis	86.75%	43.49%	81.22%	24	150

5.3.2 Profiling DeepDive on Vitis

Due to the advances in profiling the application on the vitis. We are able to find the performacne bottlenecks compared to the SDSoC. Fig: 5.6 shows the execution timeline of the DeepDive Compute units on the ZCU102 platform. As we can see the green blocks are the actual total execution time required by the Compute units. Which includes the data transfer as well as the computation time utilized by the compute unit. They body Compute unit is invoked multiple times as the body is invoked multiple times during the execution. It also gives us an insite about the data transfer prallalization. As we can see the blue block are the data read into the FPGA and the red block are the Data write back to the DDR.

The timeline gives us a brief idea about the data transfers and their initiation. Figure: 5.7 Show the zoomed in view of the profiler. This enabled us to find the bottleneck in the deepdive design or any design. The Head and Body figure shows that the data commute kernel consumes a lot of time

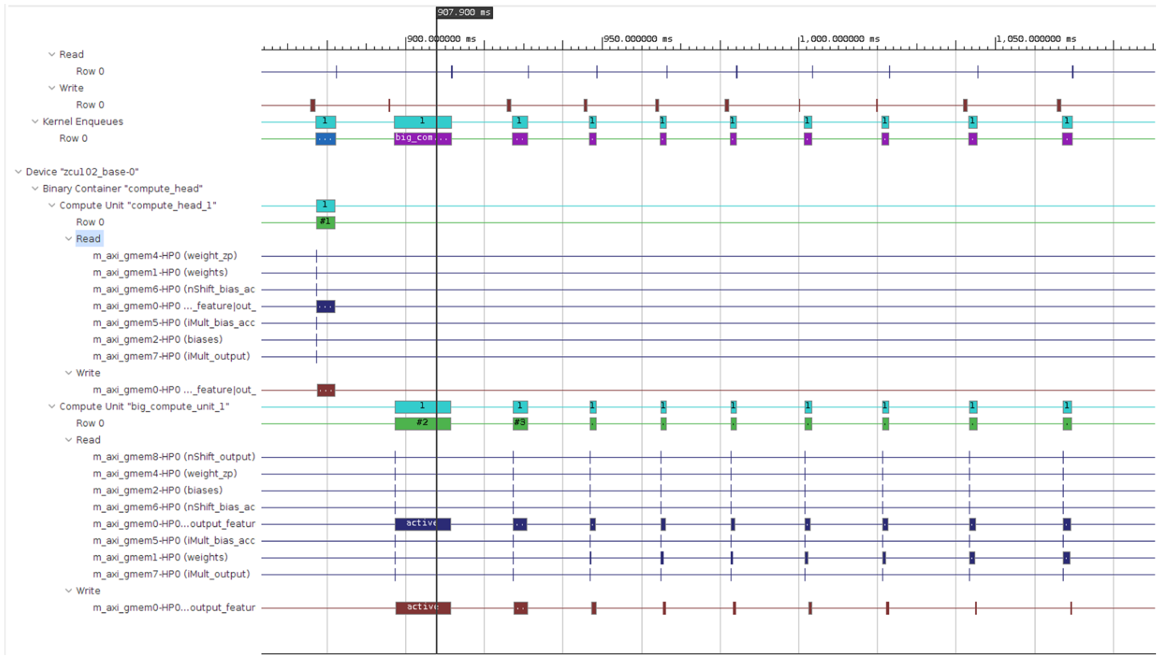
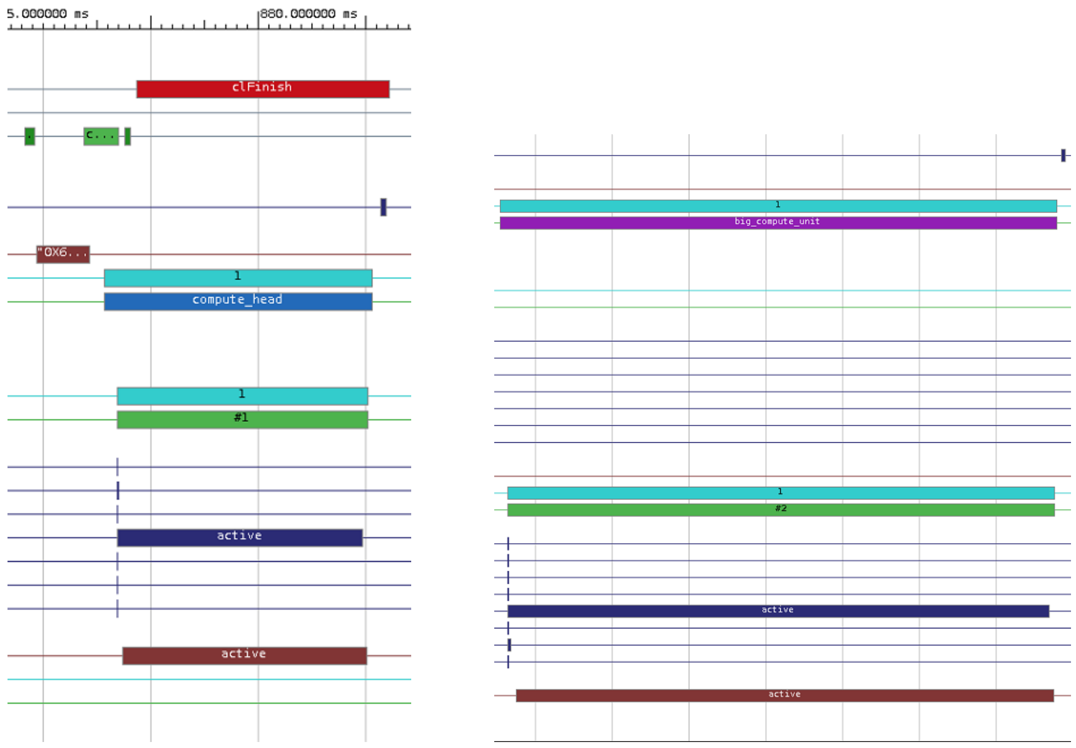

907.900 ms
900.000000 ms
950.000000 ms
1,000.000000 ms
1,050.000000 ms
Read
Row 0
Write
Row 0
Kernel Enqueues
Row 0
Device "zcu102_base-0"
Binary Container "compute_head"
Compute Unit "compute_head_1"
Row 0
Read
m_axi_gmem4-HP0 (weight_zp)
m_axi_gmem1-HP0 (weights)
m_axi_gmem6-HP0 (nShift_bias_ac
m_axi_gmem0-HP0 ..._feature|out_
m_axi_gmem5-HP0 (iMult_bias_acc
m_axi_gmem2-HP0 (biases)
m_axi_gmem7-HP0 (iMult_output)
Write
m_axi_gmem0-HP0 ..._feature|out_
Compute Unit "big_compute_unit_1"
Row 0
Read
m_axi_gmem8-HP0 (nShift_output)
m_axi_gmem4-HP0 (weight_zp)
m_axi_gmem2-HP0 (biases)
m_axi_gmem6-HP0 (nShift_bias_ac
m_axi_gmem0-HP0...output_featur
m_axi_gmem5-HP0 (iMult_bias_acc
m_axi_gmem1-HP0 (weights)
m_axi_gmem7-HP0 (iMult_output)
Write
m_axi_gmem0-HP0...output_featur
big_com...
#1
#2
#3
active

Figure 5.6: Profiling DeepDive on Vitis



(a) head (b) Body

Figure 5.7: Head and Body Compute Unit Profiling

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

6.1 Conclusion

This paper introduced DeepDive, as a fully functional framework for an agile, power-efficient execution of DSCNNs on 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 back-end, Network SoC Compiler 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 DSCNNs, MobileNet-V2 and EfficientNet on Xilinx's ZCU102 FPGA board. The execution results demonstrated 47.4 and 233.3 FPS/Watt for MobileNet-V2 and a compact version of EfficientNet, respectively. These comparisons showcased how DeepDive improved FPS/Watt by 2.2$\times$ and 1.51$\times$ over Jetson Nano high and low power modes, respectively. It also enhances FPS/Watt about 2.27$\times$ and 37.25$\times$ over two other FPGA implementations.

6.2 Future Work

As future work, we plan to improve the back-end of DeepDive to support cloud-based FPGAs such as Alveo family. We plan to extend support for multiple instances of Body CU to improve both latency and throughput. Each body could have a different level of parallelization based on the knobs introduced in Section **??**. The host would also map the IRB layers to the body CUs based on the required computation power. Various Body CUs with varying degrees of parallelization could improve DeepDive without power and hardware resource compromises.

REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] M. Tan and Q. V. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, pp. 6105–6114, 2019.