

Implementation of a Deep Learning Inference Accelerator on the FPGA

SHENBAGARAMAN RAMAKRISHNAN

MASTER'S THESIS

DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY

FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY



Implementation of a Deep Learning Inference Accelerator on the FPGA

Shenbagaraman Ramakrishnan
sh2053ra-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisor: Liang Liu and Sven Karlsson

Examiner: Erik Larsson

March 23, 2020

© 2020
Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

Today, Artificial Intelligence is one of the most important technologies, ubiquitous in our daily lives. Deep Neural Networks (DNN's) have come up as state of art for various machine intelligence applications such as object detection, image classification, face recognition and performs myriad of activities with exceptional prediction accuracy. AI in this contemporary world is moving towards embedded platforms for inference on the edge. This is essential to avoid latency, enhance data security and realize real-time performance. However, these DNN algorithms are computational and memory intensive. Consequently, exploiting immense energy, compute resources and memory-bandwidth making it difficult to be deployed in embedded devices. To solve this problem and realize an on-device AI acceleration, dedicated energy-efficient hardware accelerators are paramount.

This thesis involves the implementation of such a dedicated deep learning accelerator on the FPGA. The NVIDIA's Deep Learning Accelerator (NVDLA), is encompassed in this research to explore SoC designs for integrated inference acceleration. NVDLA, an open-source architecture, standardizes deep learning inference acceleration on hardware. It optimizes inference acceleration all across the full stack from application through hardware to achieve energy efficiency synergy with the demanding throughput requirements. Therefore, the following thesis probes into the NVDLA framework to perceive the consistent workflow across the whole hardware-software programming hierarchies. Besides, the hardware design parameters, optimization features and system configurations of the NVDLA systems are analyzed for efficient implementations. Also, a comparative study of the diverse NVDLA SoC implementations (`nv_small` and `nv_medium`) with respect to performance metrics such as power, area, and throughput are discussed.

Our approach engages prototyping of Nvidia's Deep Learning Accelerator on a Zynq Ultrascale+ ZCU104 FPGA to examine its system functionality. The Hardware design of the system is carried out using Xilinx's Vivado Design Suite 2018.3 in Verilog. While the on-device software runs Linux kernel 4.14 on Zynq MPSoC. Thus, the software ecosystem is built with PetaLinux tools from Xilinx. The entire system architecture is validated using the pre-built regression tests that verify individual CNN layers. Besides these NVDLA hardware design also runs pre-compiled AlexNet as a benchmark for performance evaluation and comparison.

Popular Science Summary

Today, Artificial Intelligence is at the edge. This edge or endpoint device is becoming more sophisticated with the evolution of Internet of Things (IoT) and 5G. For instance, these devices are employed in different applications such as autonomous cars, drones, and other IoT gadgets. At present, a self-driving car is a data center on wheels, a drone is a data center on wings as well as robots are data centers with arms and legs. All these mechanisms collect vast real-world information that demands to be processed in real-time. Here in these applications, there is no time to send data to the cloud for processing and wait for action. As the decision making needs to be instantaneous. There is a shift in transforming the processing to the edge devices.

The edge acceleration brings computation and data storage closer to the device. With the evolution of specialized hardware's providing increased computational capabilities, the AI models are processed on the edge. As a result, the overall system latency gets reduced, the bandwidth costs for data transfers are lowered and the data processing is done locally enhances privacy concerns. For example, autonomous cars require a spontaneous reaction (in seconds) to avoid potential hazards on the road. Consider the situation where a self-driving car is collecting real word information like images, videos, in this case, assume it's sensing for a stop sign. If the system sends the specific image information to the cloud for processing and waits for a decision to stop. By that response time, the autonomous vehicle could have already blown through the stop sign running over several people. Therefore, it is paramount to process the data in real-time which could be accomplished using dedicated hardware for processing locally.

This thesis primarily explores those hardware architectures for efficient processing of AI algorithms and their corresponding software execution environment setup. The particular thesis was carried out as a joint collaboration between Ericsson and Lund University. Here Nvidia's Deep Learning Accelerator architecture is engaged as a target to comprehend the complete system incorporating a hardware-software co-design. The particular architecture is an essential characteristic of NVIDIA's Xavier Drive chip which is utilized in their autonomous drive platforms.

This thesis is addressed to a variety of audiences who are passionate about Deep Learning, Computer Architecture, and System-on-Chip Design. The thesis illustrates a comprehensive implementation of an AI accelerator to envision AI processing on the edge.

Acknowledgement

Foremost, I would like to take this opportunity to thank my family for supporting me to pursue my dreams. Their unconditional love and encouragement paved the way for my graduate studies in Sweden. Next, I would thank all my professors, supervisors, classmates and colleagues at Lund University as well as at Ericsson, Lund for assisting me with everything during the course of two years.

Besides, I would like to express my sincere gratitude to the following people who have helped me undertake the particular research. My university supervisor Mr. Liang Liu for his enthusiasm on this project, for his support, encouragement, and patience; Mr. Sven Karlsson, my supervisor at Ericsson for his valuable inputs throughout the master thesis; To all the lab supervisors and Ph.D. students at Lund University for their assistance during the challenging times of the thesis. And the Ericsson Research team, Lund for providing adept suggestions.

Shenbagaraman Ramakrishnan
Stockholm, March 2020

Acronyms

AI	Artificial Intelligence.
DL	Deep Learning.
ML	Machine Learning.
DNNs	Deep Neural Networks.
CNNs	Convolutional Neural Networks.
MLP	Multi Layer Perceptron.
ReLU	Rectified Linear Unit.
PReLU	Parametric Rectified Linear Unit.
FC	Fully Connected.
LRN	Local Response Normalization.
BN	Batch Normalization.
NVDLA	NVIDIA Deep Learning Accelerator.
FPGA	Field Programmable Gate Arrays.
PS	Processing System.
PL	Programmable Logic.
ASIC	Application Specific Integrated Circuit.
SoC	System On Chip.

CPU	Central Processing Unit.
GPU	Graphical Processing Unit.
MAC	Multiply and Accumulate.
RTL	Register Transfer Level.
VMOD	Verilog Model.
AMBA	Advanced Micro-controller Bus Architecture.
AXI	Advanced eXtensible Interface.
APB	Advanced Peripheral Bus.
HP	High Performance.
SDK	Software Development Kit.
FFT	Fast Fourier Transforms.
DRAM/SRAM	Dynamic/Static Random Access Memory.
IoT	Internet of Things.
BSP	Board Support Package.
ioctl	Input Output Control.
API	Advanced Peripheral Interface.
ONNC	Open Neural Network Compiler.
SD	Secure Digital.
DFT	Design For Test.
IRQ	Interrupt Request.

Table of Contents

1	Introduction	1
1.1	Context	1
1.2	Project Goals and Challenges	4
1.3	NVDLA Design Flow	5
1.4	Implementation Methodology	6
2	Theoretical Background	7
2.1	Convolutional Neural Networks	7
2.2	Design Methodologies for Efficient Inference	15
2.3	NVDLA Architecture Design and Specification	18
3	NVDLA SoC Design Implementation	25
3.1	Environment Setup	25
3.2	NVDLA SoC design in Vivado	27
4	Configuring the Software Environment	35
4.1	Compilation Build	35
4.2	Run-time Execution	36
4.3	PetaLinux Flow	37
5	Results Analysis	41
5.1	Hardware Performance Parameters	41
5.2	Accelerator Performance comparisons	46
5.3	System Verification	48
6	Conclusion and Future Works	51
	References	53
A	NVDLA Specification Files	57
B	PetaLinux Design Flow	61
C	Miscellaneous	63

C.1	The Multi Layer Perceptron model	63
C.2	AlexNet Architecture	64

List of Figures

1.1	AI Hardware Target Domains	3
1.2	NVDLA Design Flow	5
2.1	Convolutional Neural Network	7
2.2	2D Direct Convolution	8
2.3	Relu and PRelu Activation	9
2.4	Sigmoid Activation	10
2.5	Hyperbolic-Tangent Activation	11
2.6	Max Pooling	11
2.7	Fully Connected Layer	12
2.8	Local Response Normalization	13
2.9	Pruning Neural Networks	15
2.10	Quantization Technique	16
2.11	Winograd Algorithm	17
2.12	Batched Convolution	17
2.13	NVDLA Hardware Primitives	19
2.14	External Interfaces	20
2.15	Parameters Configuration	22
3.1	Tree Build Setup	25
3.2	VMOD Partition	27
3.3	NVDLA Core Packaged IP	29
3.4	APB to CSB Bridge Packaged IP	30
3.5	Packaged NVDLA Wrapper IP	30
3.6	NVDLA Hardware Architecture	31
3.7	Block Diagram of Zynq Ultrascale+ MPSoC	32
3.8	Address Segmentation of peripherals attached to processor	32
3.9	NVDLA System Architecture	34
4.1	NVDLA Software Stack	35
4.2	Linux Execution Environment Framework	37
4.3	PetaLinux Flow	38
4.4	Device Tree	40

5.1	NVDLA Power Consumption graph	43
5.2	NV Small Utilization graph	45
5.3	NV medium utilization graph	46
5.4	Performance Comparison of DNN Hardwares	47
5.5	Sanity Tests Execution on FPGA	48
5.6	AlexNet Execution on FPGA	49
A.1	NVDLA Small Specification file.	57
A.2	NVDLA medium Specification file.	58
A.3	NVDLA large Specification file.	59
B.1	General design flow in PetaLinux.source (From PetaLinux reference guide).	61
C.1	A Multi Layer Perceptron model with two hidden layers.	63
C.2	AlexNet Architecture.	64
C.3	AlexNet Layer-wise Analysis.	64

List of Tables

3.1	Verilog Macros	28
5.1	Post Implementation timing summary.	42
5.2	Post Implementation power consumption.	43
5.3	Post Synthesis Utilization results for NV Small.	44
5.4	Post Implementation utilization results for NV Small.	44
5.5	Post Synthesis utilization results for NV Medium.	45
5.6	Post Implementation utilization results for NV Medium.	46

Introduction

In this introductory section, the context of hardware acceleration of deep learning will be conferred together with the previous state of the arts in this field. This is followed by the general deep learning acceleration design flow along with the research questions and challenges of this project. Furthermore, a methodology to accomplish the objectives are outlined.

Artificial Intelligence is the science of contriving machines to make intelligent decisions, processing several computational learning algorithms. They are broadly classified into Machine Learning and Deep Learning. Machine Learning is the ability to learn from data. While Deep learning is a specialized technique that implements machine learning tasks through artificial neural networks, inspired by the human brain. The advent of Deep Learning started from the influential paper of Geoffrey Hinton in 2012 [1], where these deep neural networks performed image classification exceptionally than the traditional computer vision, machine learning, and feature engineering algorithms. At the moment, Artificial Intelligence is empowering endless capabilities for the present as well as the future. Specifically, with the deep neural network architectures establishing the foundation for its modern applications in image, face and speech recognition's. Apparently, AI is advancing as the driving force in our socio-economic progress at the prominence of technological revolution and industrial transformation. This thesis primarily scrutinizes deep learning in the context of AI and its respective inference acceleration.

1.1 Context



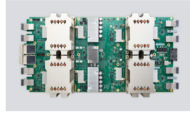

At present, Deep learning is transforming our everyday experience by initiating smart capabilities to industrial and consumer products. Deep Neural Networks revolutionize in the field of computer vision [1] [2] [3], natural language processing [4] [5], autonomous vehicles [6], etc. These DNN's achieve near state of the art accuracy on many AI applications as mentioned above. This exceptional performance is due to its inherent ability of feature extraction on their own from raw data. However, this super-human accuracy comes at the expense of demanding computational complexities. Besides, these deep algorithms consume greater energy consumption taking into account their large memory footprints. As a result,

the design of an efficient deep learning accelerator evolved as a demanding subject of interest for research.

A general AI flow involves two phases training and inference. During training, the deep learning algorithms acquire intelligence from the existing data and use cases. The primary objective here is to minimize the error function of parameters (weights) for a given neural network. This phase exploits enormous resources and time. Also, the process demands high throughput, therefore they are processed on cloud clusters or GPUs. Whilst during Inference, the developed intelligence from training is applied to test on new sets of scenarios, to validate the general system performance. This inference acceleration is carried out on dedicated compute devices as they are latency and resource critical. Thus, the computational approaches differ significantly for both the training and inference process.

There are lots of research in academia and industry focused on designing hardware systems for training and inference operations. It is challenging to design a hardware system specific to certain AI applications. Since the field of AI is constantly advancing with the development of novel algorithms, architectures, and applications. Besides, the costs involved in designing custom AI hardware is also immense. Therefore the AI hardware systems typically are domain-specific rather than application-specific. Hence can possess more degree of freedom in their target architectures to comply with different application requirements.

The AI hardware domain is classified based on their application such as training or inference. As well as, categorized based on deployments either on the cloud or edge as shown in the figure 1.1. The inference is essentially targeted for the edge/embedded platforms. While the training on the edge is still an active area of research where the computational requirements are not yet characterized distinctively. The training is usually accomplished on the cloud. Sometimes high-performance edge devices can also be exploited in training as cloud devices. This thesis primarily delves into hardware accelerators focusing inference on the edge devices.

	Cloud Platforms	Edge Platforms
Training	<ul style="list-style-type: none"> • High Performance • High Precision • High Flexibility • High Throughput • Distributed 	<ul style="list-style-type: none"> • Active area of research • Computational capabilities are not well defined 
Inference	<ul style="list-style-type: none"> • High Throughput • Low Latency • Power Efficiency • Scalable 	<ul style="list-style-type: none"> • Wide applications requirements • Low cost and latency • Low – medium throughput • Power efficient 

CPU/GPU/ASIC/FPGA

Figure 1.1: AI Hardware Target Domains

1.1.1 Deep Neural Networks Processing in Hardware.

A deep learning accelerator generally refers to a physical chip that speeds up computations. These accelerators primarily target convolution and fully connected layers for acceleration as they are computationally intensive. The particular layers primarily involve multiplications and accumulation (MAC) operations which can easily be executed in parallel. Thus, the initial hardware inference accelerators exploited the inherent spatial parallelism in DNNs and performed parallel processing over an array of processing elements (PEs) [7] [8] [9]. These accelerators employed data-flow architectures that efficiently mapped convolution operations in hardware as well as specialized on highly parallel computational paradigm. As a result the custom accelerators accomplished more efficiency than the traditional CPUs and GPUs.

But the most important bottleneck in the specific implementations included memory access and data transfers [10, 13]. The previous accelerator designs inferred that the data transfers utilized immense energy costs than the actual computations. Accordingly, the next evolution of inference accelerators focused on energy efficient designs which maximized data reuse and reduced memory overheads. The MIT Eyeriss [11] exploited a row stationary (RS) data-flow [12] architecture that efficiently reused input feature maps and filter weights. Accelerators such as Dian-Nao [14] and its variant designs [15, 16] reduced memory references by introducing on-chip buffers to store all the weights and avoid DRAM access. Besides, novel architectures started to employ deep compression techniques such as low precision and quantization to curtail memory traffic at the cost of reduced accuracy [17, 18]. These compression techniques exploit the neural networks inherent error resilience ability. In the current accelerator trends, the optimizations were accomplished not only on the hardware level but also on the algorithm level of deep neural networks [19]. As a result, the particular approaches achieved enhanced performance and energy efficiency.

NVIDIA’s Deep Learning Accelerator (NVDLA) is a standardized inference accelerator implemented based on the current accelerator design trends especially sparse compression techniques [20]. The particular accelerator addresses the computing demands of inference along with a flexible and energy efficient hardware implementation. The specific acceleration solution equips a simplified, re-configurable design which supports different performance requirements. Besides, the NVDLA is an open source and free architecture that promotes a standard method of designing inference accelerators on the edge.

1.2 Project Goals and Challenges

The primary objective of this thesis is to implement the NVDLA architecture as an SoC design and prototype on an FPGA platform to run on-device inference. This is accomplished to comprehend the consistent workflow of NVIDIA’s deep learning accelerator standards. The NVDLA architecture supports a complete deep learning inference framework succeeding in a hardware-software co-design. Thus, providing an opportunity to examine the complete system integration from neural networks compilation to its deployment in edge devices. This thesis illustrates the entire NVDLA project development constituting environment setup, RTL design, SoC implementation, and software execution environment.

In this research, the hardware acceleration is focused essentially on FPGA/ASIC platforms for edge IoT applications. Recent trends in this domain incorporate special-purpose hardware for inference acceleration on the edge. The particular inference accelerator demands high versatility, prediction accuracy and compels near real-time solution. This poses a great challenge when the implementations accomplished on precise SoC architectures are resource-constrained in terms of power, memory and computational capabilities.

To engage the computational challenges of the neural networks with the resource constraints in implementations different design methodologies are leveraged. It comprises an array of micro-engines with highly coupled local memories that minimizes data movements, improves data re-usage and enables parallel processing. The techniques such as Weight compression and network pruning are employed to enhance computation performance as well as reduce hardware cost utilizing small network sizes and data-types [19]. Sometimes simplified architectures for fast convolutions such as Winograd transformation [21], FFT’s [22] are modeled for optimizing operational efficiency. These architecture designs for efficient hardware implementation for the edge inference are examined comprehensively on the NVDLA framework. The NVDLA architecture can be configured differently based on the performance levels. Two such architectures (nv_small and nv_medium) are analyzed in-depth concerning power, area, and throughput requirements. Besides these NVDLA hardware architectures also runs a neural network for inference like AlexNet on the FPGA [23]. This operates as a benchmark to evaluate performance and prediction accuracy metrics across different implementations.

This NVDLA core is very attractive among enthusiasts as it is an open-source design. It can be utilized by the AI developers as an initial target model for inference acceleration. However, the subsequent system has its challenges while implementation. This open-source approach is noble and the complete hierarchical system design can be immense. The technological breadth, design tools and set up environments of the NVDLA flow are boundless. The system comprises of engaging in all the levels of programming hierarchies to realize its design implementation. As a result, poses a greater challenge in developing in a shorter time frame. Besides NVDLA executes only specific pre-compiled deep neural network models. As the inherent compiler environment was not available for the user's examination during this implementation.

1.3 NVDLA Design Flow

The inference process is where the AI's substantial power after training meets the real-world application. To envision an AI-driven smart ecosystem, inference acceleration is primarily decisive. The NVIDIA's Deep Learning Accelerator (NVDLA), provides a robust, versatile, configurable architecture to standardize deep learning inference operation. This accelerator is an essential characteristic of NVIDIA's Xavier Drive System-on-a-chip which is employed in autonomous vehicles.

The following NVDLA design flow comprises a hardware accelerator as well as a software ecosystem that co-ordinates to accelerate the inference process effectively. The general flow is represented as shown in the figure 1.2.

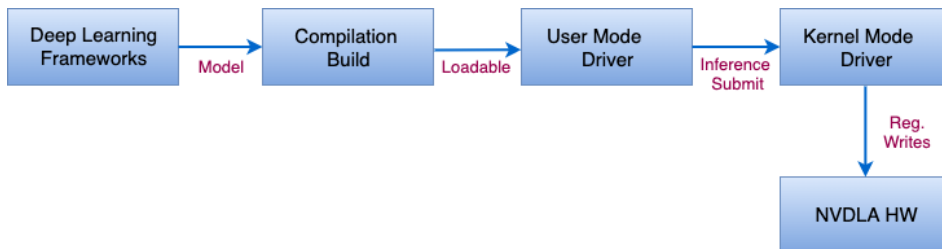


Figure 1.2: NVDLA Design Flow

The neural networks are generally trained in TensorFlow or Caffe software. The hardware acceleration for the training of neural networks on-device is still an active area of research. Next, the trained model is fed to a compilation build, that converts deep neural networks into a sequence of hardware layers based on its underlying accelerator configuration. The compiler generates hardware layers appropriate to NVDLA configuration given as an input. Later, these compiled models are stored as NVDLA loadable image. The specific loadable images are pre-compiled and only supports specific deep learning models.

Thereafter, this loadable is passed to a run-time environment that co-ordinates between software and hardware executions. It constitutes of device drivers to synchronize the hierarchical design, schedule functional blocks and handles interrupt from the NVDLA hardware. The hardware accelerator involves different functional blocks that perform convolution, activations, pooling, normalization, data transformations, and memory transfer operations. Each of these functional blocks can be configured exclusively. Different configurations concerning these scalable parameters culminate distinctive implementations such as `nv_small`, `nv_large` models. The explicit configurations are utilized based on performance specifications.

NVDLA caters to simplified system integration. A host processor is the main control unit that manages the process flow on the hardware core. The hardware core employs standard AXI buses to interface with memory for data transfers. Whereas the control channel utilizes a register file and interrupts interface to the host processor. The primary memory, in general, a DRAM is mapped to both processors and external peripherals. In the case of high memory bandwidth applications, a dedicated high-speed SRAM interface is exploited. The host processor associates with the external drivers and peripherals through memory-mapped addresses.

1.4 Implementation Methodology

The design procedure for the NVDLA implementation involves a hardware-software co-design. As a result, various application tools are realized to accomplish a complete development environment. The embedded hardware design flow is carried out in Xilinx's Vivado Design Suite tools 2018.3. While the software development and embedded drivers setup are accomplished in the PetaLinux platform from Xilinx. The Zynq Ultrascale+ ZCU102 is employed as the FPGA target to map the NVDLA core. While the Zynq MPSoC functions as the primary processing system in the SoC architecture. The project implementation is subdivided as follows:

1. An Environment setup to establish a tree build that generates RTL in Verilog from a SystemC hardware specification file.
2. An Embedded SoC architecture that accomplishes relevant communication between the host processor and NVDLA core.
3. A run-time execution environment to load and process compiled neural networks in the NVDLA system implemented in the Petalinux application.
4. An Inference setup on Zynq Ultrascale+ FPGA for executing Neural Networks especially AlexNet as bench-mark and run regression tests for evaluating the implemented system.

Theoretical Background

In this chapter, the background of Convolutional Neural Networks is presented. Besides, the design methodologies leveraged for efficient hardware implementation of CNN's are discussed. Later, the hardware specifications of NVDLA architecture are illustrated.

2.1 Convolutional Neural Networks

A Convolutional Neural Network is a deep learning algorithm consists of neurons with learnable weights and biases. Each neuron performs a dot product of its input and weights followed by a non-linear activation. The CNN architecture is prominent because of its proficiency to learn feature extractions on their own from raw image pixels, and inherent fault tolerance to inputs. These CNNs perform exceptionally well on image classification, natural language processing, recommender systems and more. Also, the CNN possess a remarkable capability to express output as a single differentiable score emerged just from the raw input image. A CNN incorporates a series of layer operations to accomplish the above characteristics as shown in the figure 2.1 subsequently.

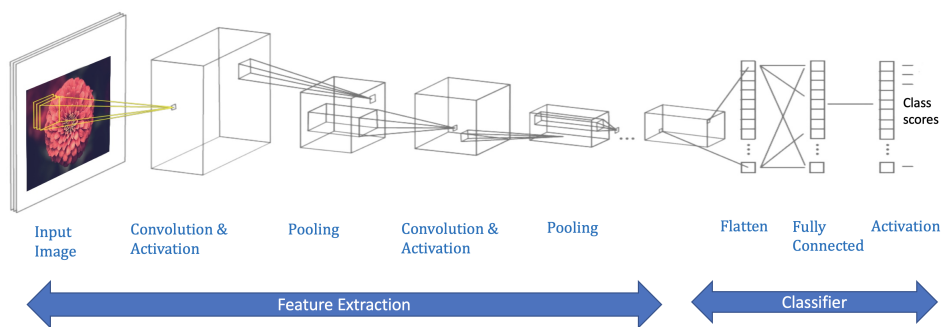


Figure 2.1: Convolutional Neural Network

2.1.1 Convolution Layer

The Convolution layer extracts information from the input image. It is an element-wise multiplication operation between the kernel weights and input feature maps. This operation slides the kernel across the input at every location, shifting one neuron each time (for stride of 1). The partial sums of these convolutions are aggregated into respective output feature maps. For simplicity, the input image is given by $I(i,j)$ considering every pixel to be scalar. The filter is represented as a kernel $K(n,m)$ and the convolved output is given by $h(i,j)$. A figure 2.2 successively represents a convolution operation.

$$\begin{aligned} h(i, j) &= (I * K)(i, j) \\ &= \sum_m \sum_n I(i - m, i - n) K(m, n) \end{aligned}$$

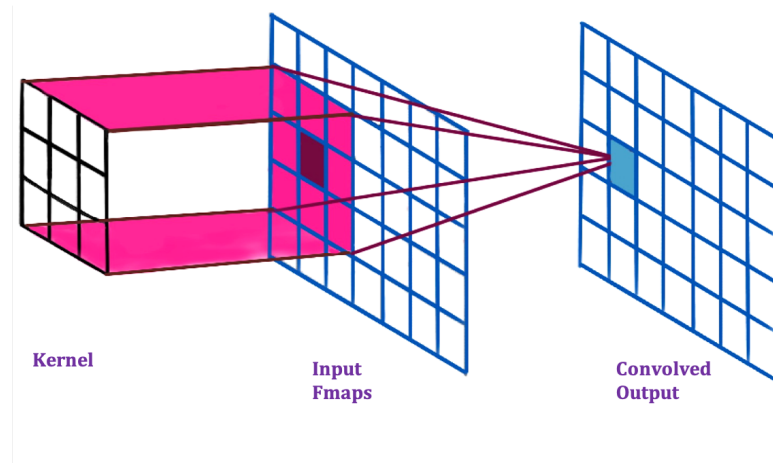


Figure 2.2: 2D Direct Convolution

This sliding window kernel operation also facilitates sparse connectivity and parameter sharing. When considering a convolution layer the receptive field for a given hidden node is small. As the node only maps to the kernel parameters, unlike a Multi-Layer Perceptron (Appendix C.1) that links a hidden node to all of the inputs. Therefore leading to fewer network inter-connections empowering sparse connectivity. In the kernel process, all the parameters stay constant during the sliding operations. As a result, the kernel weights can be shared while performing the weighted sums for respective hidden nodes. Besides the convolution layer, there are more computational blocks when constructing a complete CNN. Those blocks are discussed subsequently.

2.1.2 Activation Layer

The activation layers are the most integral part that adds non-linearity to the system. They enable the network to learn highly complex relationships between the feature maps. An activation function determines the output of the model, their prediction accuracy and computational efficiency during training. These functionalities are realized through dedicated hardware logic and Look-Up-Table in the hardware implementation. Different activation functions are chosen based on the specific problem statement. They are as follows:

- **ReLU.** The most popular activation function for CNN's are the Rectified Linear Unit [24]. Since they result in sparse activation reducing the network parameters. They are defined as.

$$f(y) = \max(0, y)$$

This activation gives an output y if y is positive and 0 otherwise. It is advantageous since there is no problem with vanishing gradients and they are computationally less expensive. The only problem with this type of activation is the decaying nodes. Sometimes the ReLU nodes are pushed into regions of inactivity (i.e zero) for all inputs. This results in dead neurons which can no longer be used.

- **PReLU.** To solve the above problem, a Parametric ReLU is used [25]. Instead of squashing the outputs to 0 when the inputs are less than or equal to zero. PReLU keeps a small linear trainable parameter (a) that learns with other neural network parameters to avoid dead neurons. An image of ReLU and PReLU is presented in the figure 2.3.

$$f(y) = \begin{cases} y & \text{if } y > 0 \\ ay & \text{otherwise} \end{cases}$$

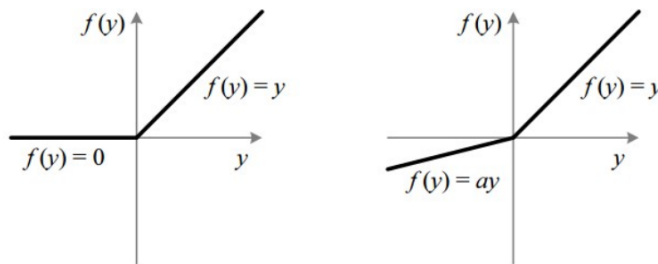


Figure 2.3: Relu and PRelu Activation

- **Sigmoid.** A Sigmoid function activates the input between 0 and 1. This operation is bounded and the results are always positive. Sigmoid is mainly used as output activation for binary classification problems. They have a

complication of diminishing gradients. Also, the curve is not zero centered as shown in the figure 2.4.

$$\frac{1}{1 + e^{-x}}$$

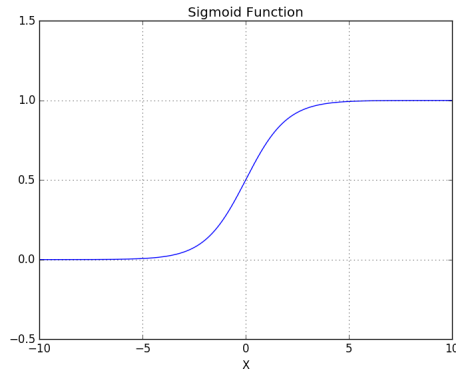


Figure 2.4: Sigmoid Activation

- **Hyperbolic Tangent.** This is a bounded activation function that ranges between -1 and 1. The gradients for this operation are stronger than sigmoid as the derivatives are steeper. Deciding between sigmoid and tanh will depend on the requirement of gradient strength. This activation also faces the problem of diminishing gradients. An image 2.5 of the following activation is presented.

$$f(x) = \tanh(x)$$
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

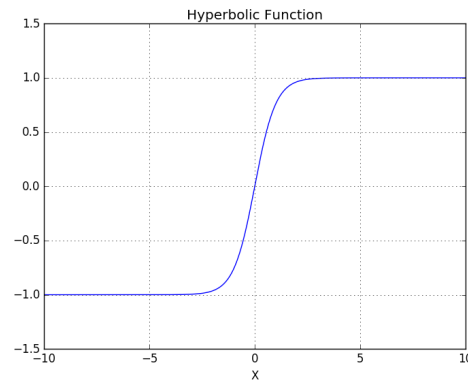


Figure 2.5: Hyperbolic-Tangent Activation

2.1.3 Pooling Layer

A pooling layer summarizes the neighborhood of a respective hidden node. This is achieved by sliding the window over the inputs and extract a single value based on the type of pooling operation. As a result, this layer down samples the filtered image reducing the training and inference time. Besides, this operation makes the features invariant to small changes in the raw input evading the problem of over-fitting. The commonly employed pooling functions are enumerated.

- **Max Pool** : This operation is the most commonly used type of pooling. It returns the maximum value within the receptive field. An example is shown in the successive image 2.6.
- **Min Pool** : In this operation the minimum value within the pooling window is selected.
- **Average Pool** : This function returns the average value within the receptive field.

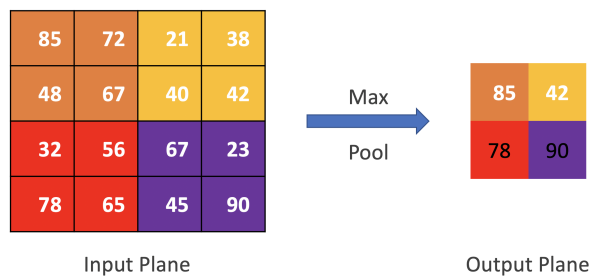


Figure 2.6: Max Pooling

2.1.4 Fully Connected Layer

A Fully Connected (FC) layer resembles a multi-layer perceptron model where each neuron has full connections to the previous layers as shown in the figure 2.7. The input for this layer is a 1D vector of numbers flattened from the previous convolution layers comprising of 3D volumes. While the output of an FC layer is a list of probabilities of different class scores. The particular FC layer functions as a classifier. This operation analyzes the previous layer's high-level features representations (after convolution and activation) and applies weights to predict which features correlate to the specific class. The class score constituting the highest probability is the classifier's decision. For instance, if a model is predicting that the image is a bird, then it holds large value in the activation maps representing high-level features such as wings or beaks. Thus the classifier correlates the extracted high-level feature maps to definite class scores.

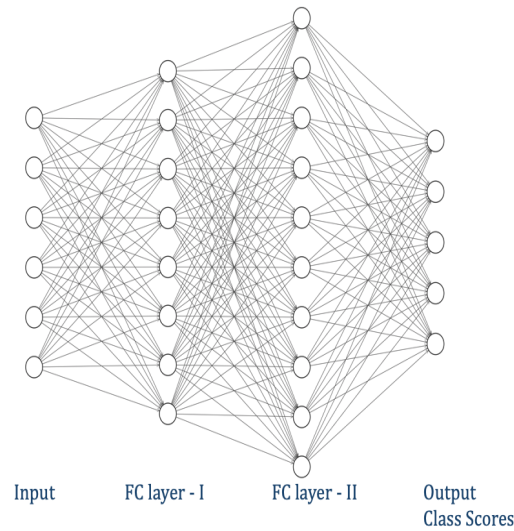


Figure 2.7: Fully Connected Layer

2.1.5 Normalization Layer

A normalization layer is necessary in Convolutional Neural Network architectures, considering the unbounded nature of certain activation functions such as ReLU. This process limits the outputs of the activation's and is applied usually before the activation layers. There are two types of normalization layers commonly employed.

- **Local Response Normalization.** The Alex-Net architectures using ReLU activation introduced this concept [23]. It resembles the neuro-biology concept of 'lateral inhibition' to boost the neurons and subdue its neighbors [26]. This inhibition forms a local maxima of values for excitation in subsequent layers. Here the neurons excitation are enhanced and the surrounding local neighborhood is dampened. It is achieved by square normalizing the image pixels of feature maps over a local neighbourhood that extends across or within the respective channels as shown in the figure 2.8.

The AlexNet utilized a local response normalization across channels (in the dimension of depth) and is given by the subsequent formula. In the formula, i indicates the output of filter i . $a(x,y)$, $b(x,y)$ represents the pixel values before and after normalization. Here (k, α, β) are hyper-parameters. n is the size of normalization neighbourhood and N is the total number of kernels. In the corresponding figure $n=1$ and $N=3$.

$$b_{xy}^i = \frac{a_{xy}^i}{\left(k + \alpha \sum_{j=\max(0, i-n/2)}^{\min(N-1, i+n/2)} (a_{xy}^j)^2\right)^\beta} \quad (2.1)$$

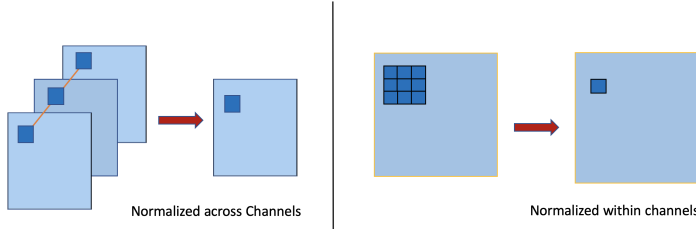


Figure 2.8: Local Response Normalization

- **Batch Normalization.** This technique normalizes the output of previous activation layer to reduce the internal co variance shift [27, 28]. This arises due to changing distributions in hidden neurons/activations. As a result, the batch normalization allows each layer to learn independently of preceding layers. This is accomplished by subtracting the batch mean and dividing by the batch standard deviation. The particular technique improves the stability of a neural network and reduces over-fitting as it contains regularization effects top. The mathematical representation is described. Here the batch normalization transform (BN) is applied to activation x over a mini batch B of size m .

Input = Values of x over a mini – batch (B).

$$B = x_1 \dots x_m;$$

Learning Parameters = γ, β

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i \quad \text{mean}$$

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2 \quad \text{variance}$$

$$\vec{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad \text{normalize}$$

$$\vec{y}_i = \gamma \vec{x}_i + \beta \equiv BN_{\gamma, \beta}(x_i) \quad \text{shifting}$$

2.2 Design Methodologies for Efficient Inference

In this section, the design methodologies leveraged for efficient hardware implementation are discussed in detail. The current trends of hardware accelerators in deep learning employs "Deep compression" techniques for efficient inference. In this method, the deep neural networks are compressed before their inference acceleration in hardware targets. Consequently, it reduces the number of parameters as well as its computational efforts. Deep compression mainly includes network pruning and quantization techniques. Besides, the computational complexity can also be reduced by exploiting Winograd and batching transformations. These optimization methods applied to realize efficient inference are interpreted below.

1. Pruning.

The pruning mechanism was inspired to eliminate the redundant parameters of the neural networks that do not contribute significantly to the outputs. Especially, the zero weights and activations. This procedure can be exercised to all the layers as a whole or specific layer of CNN. This facilitates sparse executions, which enhances computational performance as well as reduce memory overheads.

Pruning can be implemented using ranking methods. The rankings are generally based on L1/L2 norm of weights, mean activation's, etc., considering individual layers. The alternative technique employed is iterative pruning, which applies to the complete network. In this process, CNN is fine-tuned until the pruning objective is attained. For example, the objective for pruning could be in terms of definite model size or execution performance.

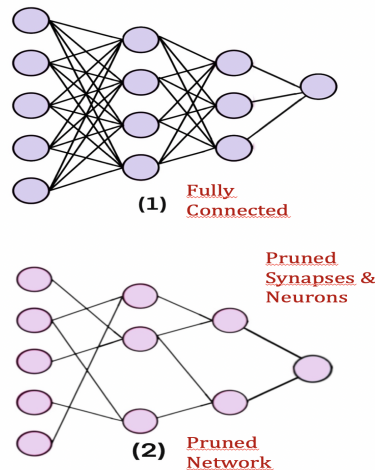


Figure 2.9: Pruning Neural Networks

2. Quantization.

The quantization method adopts reduced precision as well as smaller data representations of weights and activations. This is achieved to primarily reduce the hardware cost, memory accesses and increase parallel executions. The quantization to the desired number of bits solely depends on the requirements of the application and accuracy. Therefore, it is difficult to speculate an optimal precision that suffices all-purpose.

Generally multiple precision such as INT4, INT8, FP16, FP32 are utilized for different applications. Recently INT8 datatype is widely used for inference while FP16, FP32 data-types are exploited for training. The quantization techniques are applied to the granularity of channel level for weights and layer level for activations. Normally the activation quantization has a significant impact on the accuracy than that of the weight's quantization. To improve the accuracy, the quantified parameters are fine-tuned by retraining recurrently with the original data-set. Sometimes to avoid retraining, smaller floating-point precision is exploited.

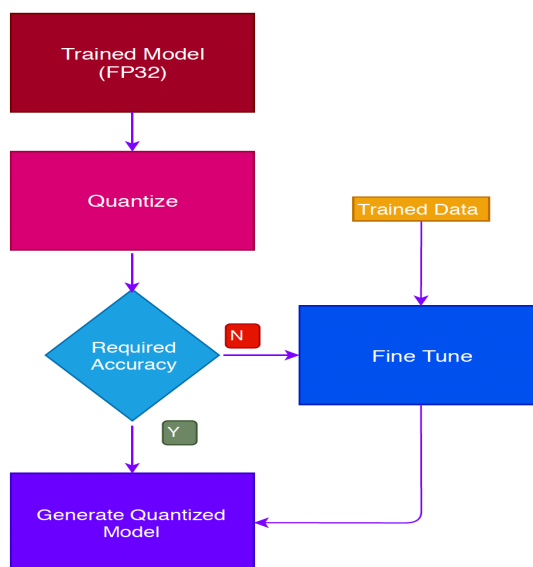


Figure 2.10: Quantization Technique

3. Winograd Transformation.

Winograd transformation is an optimized convolution algorithm that reduces the number of multiplier units and increases functional throughput [21]. These methods work well on 3x3 convolutions. Here the inputs and

kernel parameters are transformed. These transformed values are subjected to an element-wise multiplication then the results are converted back to obtain the convolved output.

Consider an input image of $4 \times 4 \times C$, kernel window of $3 \times 3 \times C$. To compute the output feature map ($2 \times 2 \times 3$) the required functional units (MACs) includes: Direct Convolution demands $(4 \times 3 \times 3 \times C)$ $36 \times C$ functional units. While, the Winograd expects $16 \times C$ functional units. Therefore, it improves performance by a factor of 2.25x.

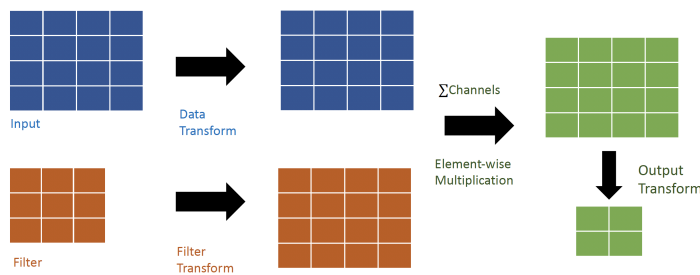


Figure 2.11: Winograd Algorithm

4. Batching.

The Batching mechanism is applied to the fully connected layers. These layers functions as a classifier that encompasses full interconnections with their previous layer. Subsequently, do not perform any sliding window operations, confining weight sharing and sparsity. As a result, this functional block demands high memory bandwidth as well as compute resources. To solve this problem batching is employed for reusing kernel weights. Here multiple feature maps of the input image are processed in parallel to perform dot products with the kernel weights. This facilitates the re-usability of weights across multiple images.

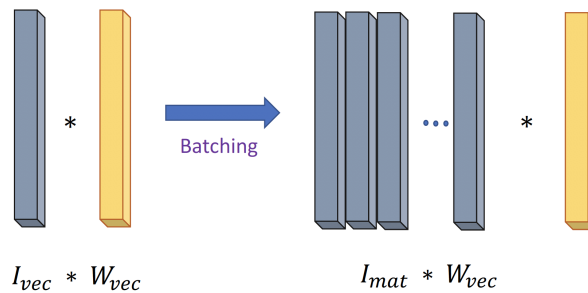


Figure 2.12: Batched Convolution

2.3 NVDLA Architecture Design and Specification

This section describes the design of NVDLA core, its respective interconnects, hardware specifications and parameter configurations.

2.3.1 Hardware Primitives

The NVDLA core comprises of hardware primitives to realize inference acceleration on deep neural networks. Each of the modules performs a specific operation as discussed previously in the background section. Also, they can be configured independently. It is a versatile, highly scalable architecture that standardizes hardware acceleration of AI inference. For instance, if a system requires multiple layers of convolution operations then the particular building block can be scaled up. Suppose the design does not require any pooling function then the planar data processing engine can be removed exclusively. Therefore, the complete system can be sized appropriately based on application requisites. The hardware primitive blocks are:

- Convolution Core - advanced high performance convolution engine.
- Single Data Processor - single point lookup engine for activation operations.
- Planar Data Processor - planar averaging engine for pooling function.
- Channel Data Processor - multi-channel averaging engine for optimized normalizing functions.
- Dedicated Memory Engine - dedicated high performance memory interfaces.
- Data Reshape Engine - data format transformations for tensor reshaping.

The hardware architecture comprises of four computational blocks (CONV, SDP, PDP, CDP) and two data transformation engines (RUBIK, BDMA) as shown in the figure 2.13. The compiler maps the layers of the deep neural networks to these computational blocks based on the dependency graphs. The inference flow starts when an activate command is sent from the host processor along with the configurations of one hardware layer. If there are no data dependencies among multiple hardware layer configurations, then they are assigned and executed simultaneously. When the corresponding layer finishes its process, it issues an interrupt to the co-processor reporting completion. Consequently, the processor engages in processing the next layers. Thus the system follows command-execute-interrupt flow for completion of the whole convolution network.

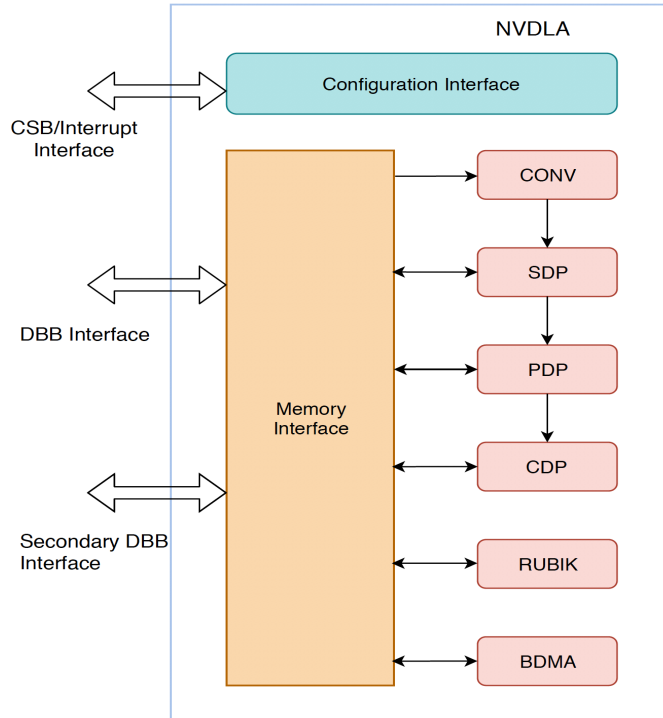


Figure 2.13: NVDLA Hardware Primitives

2.3.2 Interconnects

The NVDLA core connects to the rest of the system through the following interface options as illustrated in fig 2.14.

- Configuration Space Bus Interface (CSB).** The host processor uses this 32-bit synchronous bus to access the NVDLA configuration register sets. NVDLA acts as a slave on the CSB interface. It implements a simple address/data interface to establish communication.
- External Interrupt (IRQ).** This system follows a command-execute-interrupt flow where every layer operation is asserted through interrupts to the host processor. It is a 1-bit level driven interrupt that affirms completion and error conditions.
- Data Backbone (DBBIF).** NVDLA has its own DMA subsystem that connects to each block. This interface is an AMBA AXI-4 compliant that associates the DRAM memory to the DMA engine. It is a high speed, synchronous and extremely configurable data bus.

- **SRAM Connection (SRAMIF).** For high performance-oriented systems which emphasize high throughput and low latency. An SRAM memory is used as cache. This interface provides a connection to the cache memory.

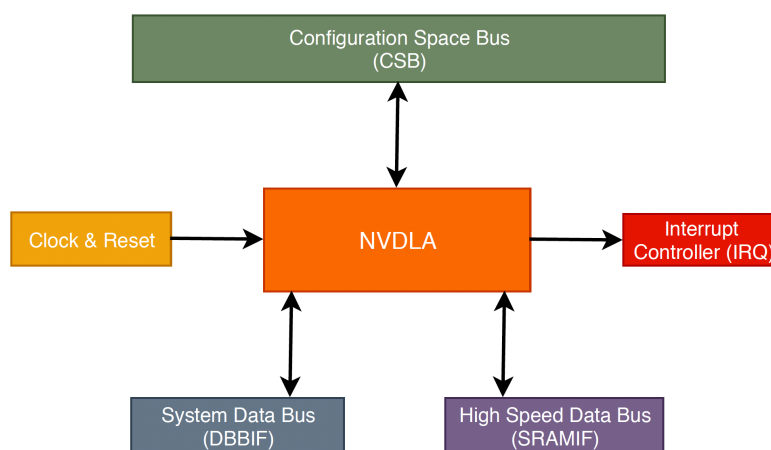


Figure 2.14: External Interfaces

2.3.3 Hardware Specifications

NVDLA hardware provides a scalable architecture that can be configured for different application requisites. These hardware specifications depend on the requirements of the CNN employed for inference acceleration also the recommended performance measures. Consequently, performance measures have a significant impact on the following parameters.

1. Convolution Buffer

The Convolution buffer holds both weights and feature data for individual layers. These buffers reduce the number of memory access and enhance energy efficiency. The size of these buffers is configured within the range of 4KB to 32KB. The model size of the CNN is paramount for the buffer sizing.

NVDLA uses a ping-pong buffer mechanism to improve system efficiency. The reprogramming latency can be reduced by utilizing two register groups. This methodology concurrently programs the second group of buffer when the first group is processing the convolution computations. The hardware

switch between the register groups through an interrupt based control flow.

2. Number of MAC units

The number of MAC units determines the overall system throughput. The MAC operations in a convolution operation can easily be calculated. They depend on the size of input feature maps, kernel size and number of kernels.

$$MAC\ Units = 2 * nK * (Kx * Ky * Kz + 1)(Ix * Iz) \quad (2.2)$$

The hardware architectures leverage parallelism in convolution computations exploiting the input and output feature channels. They are represented as Atomic-C and Atomic-K values. The MAC computations on the input feature channels are assigned between 16 to 128. While the computations on the output feature channels are assigned values between 4 to 16. These sizing values impact the number of MAC arrays. For example, if the NVDLA configurations of Atomic C is 32 and Atomic-K is 16 then,

$$number\ of\ MAC = Atomic - C * Atomic - K = 32 * 16 = 512\ instances \quad (2.3)$$

3. Memory Bandwidth

For high-performance systems that are time-critical, high memory bandwidth can be capitalized using an on-chip SRAM. It functions as a second-level cache memory block. An on-chip SRAM is less expensive in implementation than a large convolution buffer considering its wide ports and demanding timing requirements. But provides less in exchange if the layers are limited by convolution buffer size. So from an implementation perspective, the convolution buffer size is enhanced to reduce memory bandwidth demands and the auxiliary SRAM advances the total available memory bandwidth of the system.

Considering the above hardware specifications the NVDLA architecture can be implemented for distinct operations. Also, the architecture can be configured based on the required performance requirement. While the primary configuration includes `nv_small` and `nv_large` models. The figure 2.15 shown below interprets the different configurable parameters.

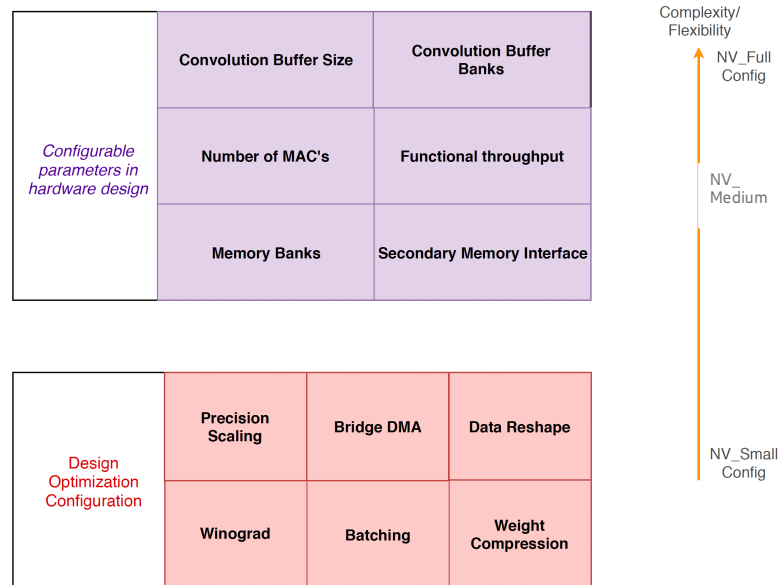


Figure 2.15: Parameters Configuration

- **NVDLA Small Model**

The NVDLA small model is well compliant with AI, IoT systems where performance levels are less of a preference and primarily targets cost and resource efficiency. These implementations provide a basic architecture that supports smaller tasks. Typically, the NVDLA small configurations incorporate reduced precision arithmetic, fewer design optimization features, and limited throughput enhancements. The specified design parameters are enlisted below

Feature	Configured Option
Data Type Precision	INT8.
Winograd Support	Not supported.
Compression Support	Not supported.
Second Memory Bus	Not supported.
Image Input format	Supports 8-bit RGB/YUV.
SDP function	Single Scaling
Bridge DMA	Not supported.
Rubik	Not supported.
Atomic-C	8
Atomic-K	8
Layer throughput	One output feature data gets generated every clock cycle.
Convolution Buffer size	32KB

- **NVDLA Large/Full Model**

The NVDLA large is utilized for high performance and dynamic applications. These models are highly flexible and multiple tasks can be completed concurrently. As a result, the design includes a high bandwidth SRAM interface connected to the design implementation that shares workloads with the NVDLA core. This SRAM is used as cache memory. These systems incorporate an increased number of processing elements, maximum operational throughput and enable all design optimizations. The hardware design sizing of NVDLA Large is described below.

Feature	Configured Option
Data Type Precision	FP16/INT16.
Winograd Support	Yes, supported.
Compression Support	Yes, supported.
Second Memory Bus	Yes, supported.
Image Input format	Supports 8/16-bit RGB/YUV.
SDP function	Single Scaling/LUT
Bridge DMA	Yes, supported.
Rubik	Not supported.
Atomic-C	64
Atomic-K	16
Layer throughput	Four output feature data gets generated every clock cycle.
Convolution Buffer size	32KB

NVDLA SoC Design Implementation

This chapter provides an in-depth understanding of integrating NVDLA into an SoC for FPGA implementation. It provides information on setting up the environment for model creation, project development in Vivado as well as system architecture design of the SoC.

3.1 Environment Setup

To develop the RTL model of NVDLA core, a tree build setup is constructed. The following environment generates multiple configurable hardware designs from a single source. These hardware configurations are defined on a specification file. Based on these specifications, a Verilog RTL code is generated appropriately. Different NVDLA configuration can be characterized in the following specification file to develop diverse hardware cores in regards to application requirements. The spec file definitions are provided in the appendix A.

The environment setup requires different tools to build the configured RTL. The essential software's are chartered below in the 3.1.

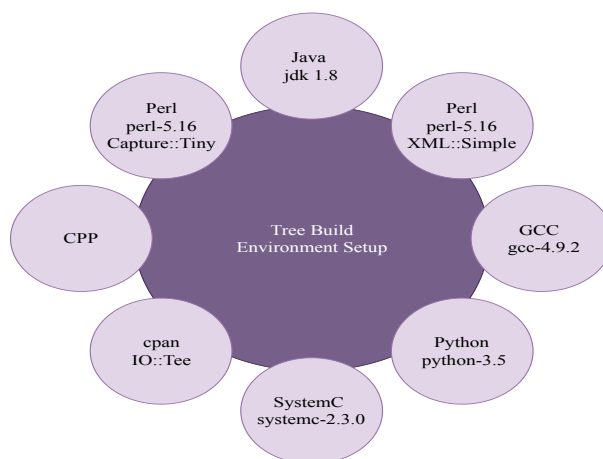


Figure 3.1: Tree Build Setup

The path to above tools and configurations set-up are constituted in a `tree.make` file. Once when all these composed modules are compiled through `build`, an `outdir` directory is established incorporating the Verilog model of the NVDLA hardware. Thus, a successful RTL tree build for a relevant specification file is shown in the subsequent image.

3.1.1 Overview of Generated VMOD

The Verilog model generated from the tree build engages `NV_Small` specification for the following case. Here the organizational structure of the generated NVDLA core is discussed. The top module includes the following partitions as also depicted in the figure 3.2.

1. `Partition_o` : This section controls the communication between processing elements with external controllers and memory units.
 - `CSB` - This module reads and writes configuration registers of each layer in NVDLA core. This transfers data from the external management processor through the APB interface.
 - `CFGROM` - This maintains configurable parameters of the core for its respective specification definition.
 - `MCIF` - This interface communicates with all subunits that access the external DDR. This data bus uses an AXI protocol.
 - `PDP` and `CDP` - These units perform pooling and local response normalization respectively.
 - `GLB` - They control the output interrupt signals of all the sub cores of NVDLA.
2. `Partition_c`: This section manages various convolution kernel operations such as `CDMA`, `CBUF`, and `CSC`.
 - `CDMA` - Convolution DMA fetches data from SRAM/DRAM and stores it in a convolution buffer. It comprises two read ports namely weight read and data read ports that connect to the AXI interface to obtain weight/feature data.
 - `CBUF` - Convolution buffer is the next stage of the pipeline. It is a 512KB of SRAM cache that stores input data and weights.
 - `CSC` - Convolution Sequence Controller loads the stored data from the buffer to its respective MAC units appropriately. Thus, controlling the computation sequence in the convolution pipeline.
3. `Partition_m`: This partition performs multiplication and addition computations. This Convolution MAC module comprises of 16 MAC cells. Each of these cells includes 64 16-bit multipliers and 72 adders.
4. `Partition_a`: This module accumulates the partial sums from the MAC arrays and estimates the results before sending it to the next stage of activations.

5. Partition_p: This section performs various linear and non-linear operations as discussed.

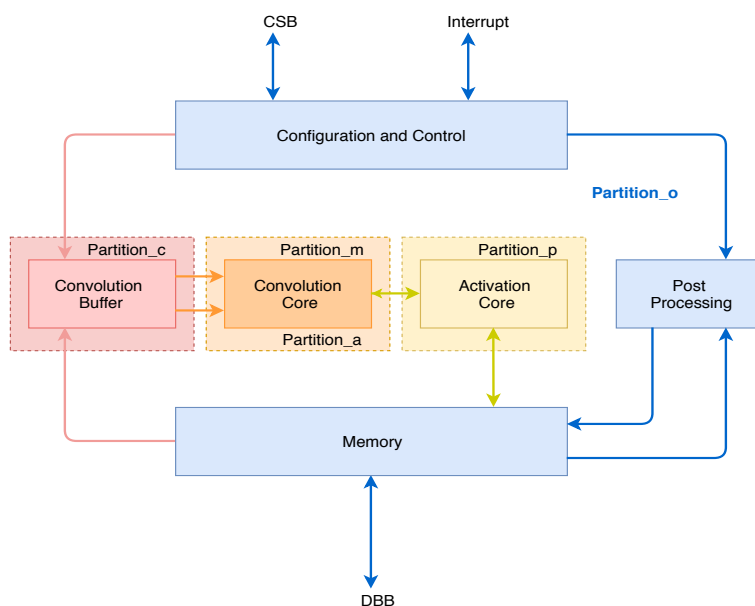


Figure 3.2: VMOD Partition

3.2 NVDLA SoC design in Vivado

The integration of NVDLA core into SoC design is carried out using Vivado Design Suite - 2018.3 provided by Xilinx. This software accelerates the design and verification of NVDLA architecture. Several tools from Xilinx such as Vivado IP integrator, RTL Synthesis, Vivado Simulator, Implementation, Xilinx SDK and Petalinux applications are exercised during the project development .

The design methodology for the implementation of NVDLA SoC is divided into two parts.

- Design of NVDLA Hardware Core and Wrapper
- SoC System Architecture

3.2.1 NVDLA Hardware Core

The NVDLA hardware core is built from the generated Verilog RTL model. Certain modifications are performed in the developed RTL model to integrate the accelerator core into FPGA. As the NVDLA implementation on its own is well defined for custom ASIC design. For instance, the implementation comprises clock

gating and power gating features to limit dynamic power dissipation at the cost of more hardware logic. Contrarily, these features when used in FPGA prototyping results in timing violations. Therefore, definite macros are added in Verilog header files and set as global include to disable the auxiliary hardware architectures. Besides, the inferred RAM resources included in the source files also disable these features for the prior purpose.

```

VLIB_BYPASS_POWER_CG
NV_FPGA_FIFOGEN
FPGA
FIFOGEN_MASTER_CLK_GATING_DISABLED
RAM_DISABLE_POWER_GATING_FPGA

```

Table 3.1: Verilog Macros

Here synthesis is accomplished to verify the substantiation of source code for preceding user alterations. The tool generates top-level instances with all of its references along with appropriate reports that are utilized to validate the developed RTL design. In general, synthesis performs a high-level abstraction to logic gate levels where it maps the generic gate-level netlists. It also incorporates advanced design optimization's for timing and area. The behavioral model sources along with netlists from synthesis are packaged into an IP using Vivado IP integrator. The packaged NVDLA Core IP is shown below in the fig 3.3 includes the following external pins and interfaces:

- Clock Interface: NVDLA core consists of *dla_core_clk* and *dla_csb_clk*. The *dla_core_clk* acts as the functional clock for the complete system while *dla_csb_clk* for the configurable interface. These clock interfaces are connected as a system clock.
- Reset Interface: The *dla_reset_rstn* operates as a primary functional reset for the NVDLA core. While the *direct_reset_* serves for DFT reset. They are connected to the processor system reset.
- Clock Gating Interface: Here *global_clk_ovr_on* is tied low to disable non-inferred clock gating. *tmc2slcg_disable_clock_gating* is also kept low to disable the same during DFT.
- System Data Interface: The external data are accessed through master AXI interfaces *nvdla_core2dbb_axi*. The DBBIF and SRAMIF buses fetch data from off-chip DRAM and on-chip SRAM respectively.
- Configuration Interface: This slave interface addresses the configurations of hardware layers through an APB bus, *APB_S*.
- Interrupt Port: A single bit level driven interrupt port is used *dla_intr* to control the processing of different layers.

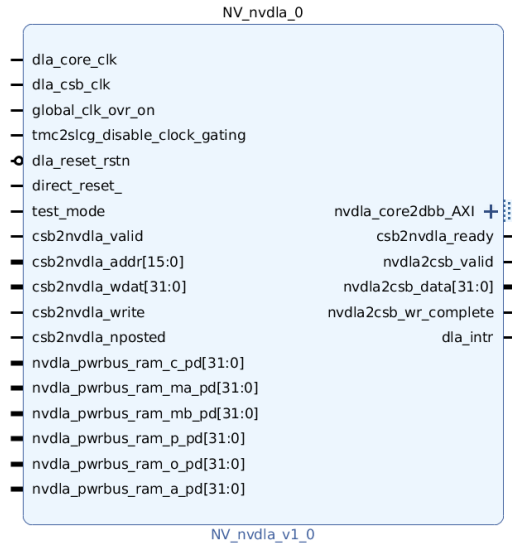


Figure 3.3: NVDLA Core Packaged IP

- Power Gating Interface: Power gating features are established when these power buses *nvdla_pwrbus_ram* are tied to 0.
- DFT Interface : The *test_mode* is preferred low to disable DFT mode.

3.2.2 NVDLA Wrapper

The NVDLA Wrapper combines NVDLA Core functionality with the configuration and control operations to constitute an exclusive hierarchical system design. The configuration definitions from the host processor are delegated to NVDLA core for managing underlying hardware layers for the provided network description. This interconnection is achieved through configuration space bus, exposed as an APB interface that establishes communication with the central management processor. This APB to CSB bridge RTL model is associated with NVDLA core to develop an NVDLA subsystem. The APB2CSB IP is shown below in the fig 3.4.

In the developed wrapper module the clock and reset interfaces of the custom IPs are made as external pins. While the clock-gating and power-gating features are disabled, connecting to zero constants. Data and configuration interfaces are made as external ports. Also, the interrupt port is connected as an external link. Once these relevant interconnections are completed, the NVDLA Wrapper is packaged as a custom IP to be utilized in the hardware system architecture. The corresponding image of the wrapper module IP is displayed in image 3.5.

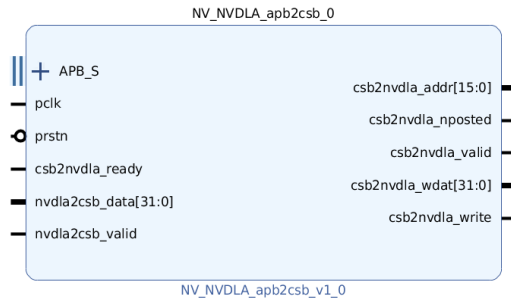


Figure 3.4: APB to CSB Bridge Packaged IP



Figure 3.5: Packaged NVDLA Wrapper IP

3.2.3 Hardware System Architecture

The hardware system architecture establishes different system functionalities. Primarily, constitutes the communication between NVDLA hardware core and management processor of the FPGA. Besides, the system architecture specifies memory mapping of different associated soft IP's (peripherals) as well as resource utilization of hardware modules on the FPGA board.

The hardware architecture includes the following and depicted in the succeeding figure3.6.

- A host processor: Zynq Ultrascale+ MPSoC.
- Associated DMA engine, an AMBA AXI4-compliant acting as memory bus.
- Instructions regulated through Configuration bus and interrupts.
- Peripherals such as AXI-APB bridges, AXI interconnects and constants.
- Clock and reset generators adhered to the whole system.

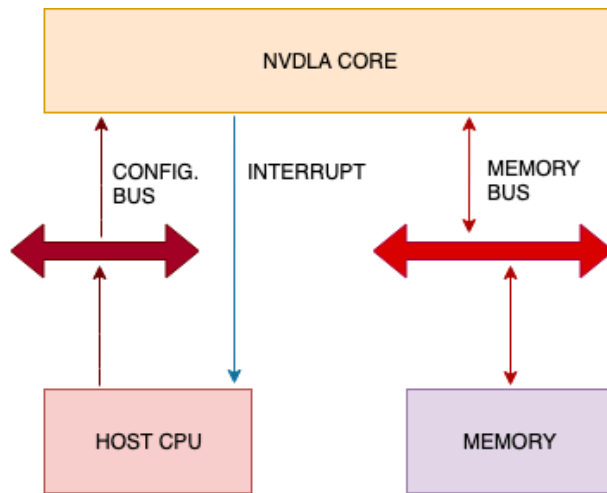


Figure 3.6: NVDLA Hardware Architecture

To accomplish the communication flow of the implemented system, the custom NVDLA wrapper IP communicates with a series of interconnections through PS-PL interfaces of the host processor. The host processor acts as a logical interface between PS and PL while integrating custom IPs in the fabric as a PS+PL configuration. The PS-PL interfaces are configured as follows, the data bus of NVDLA wrapper is connected to High Performance (HP) slave port 0 AXI in full power domain. This S-AXI HP port directly communicates with DDR Controller. While the configuration bus is associated with High-Performance Master (HPM) 0 in full power domain. This HPM-FPD port establishes transmission with the Processing System through the APU engine. The block diagram of Zynq Ultrascale+ MPSoC is shown in the subsequent figure 3.7.

The host processor communicates with its memory and other peripherals at different addresses in the address space. The intercommunication is accomplished through memory mapping at a specific address. The configuration space bus of NVDLA hardware reaches out to the processor through a reserved memory of 64KB. Also, the data bus is interlinked with DDR controllers at a specified address spacing. Thus the memory mapping to different peripherals and memories is done systematically by the tool, utilizing a mapping strategy that minimizes address

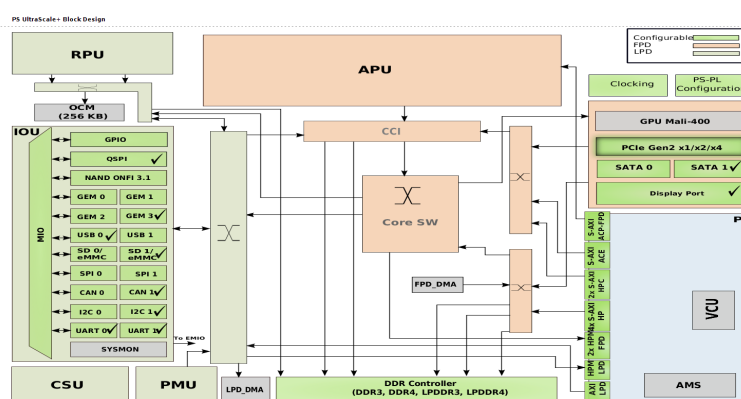


Figure 3.7: Block Diagram of Zynq Ultrascale+ MPSoC

decoding complexity. The address segmentation of different peripherals is given below in the figure 3.8.

```

zynq_ultra_ps_e_0
├── Data (40 address bits : 0x00A0000000 [ 256M ], 0x0400000000 [ 4G ], 0x1000000000 [ 224G ], 0x0B00000000 [ 256M ], 0x0500000000 [ 4G ], 0x4800000000 [ 224G ])
│   ├── design_1_wrapper_0
│   │   └── APB_S reg 64K 0x00_A000_0000 0x00_A000_FFFF
│   └── design_1_wrapper_0
│       ├── mvdla_core2dbb_axi (32 address bits : 4G)
│       │   ├── zynq_ultra_ps_e_0 S_AXI_HP1_FPD HP1_DDR_LOW 2G 0x0000_0000 0x7FFF_FFFF
│       │   ├── zynq_ultra_ps_e_0 S_AXI_HP1_FPD HP1_LPS_OCM 16M 0xFF00_0000 0xFFFF_FFFF
│       │   └── zynq_ultra_ps_e_0 S_AXI_HP1_FPD HP1_QSPI 512M 0xC000_0000 0xDFFF_FFFF

```

Figure 3.8: Address Segmentation of peripherals attached to processor

These memory-mapped master and slave devices are connected through a series of Xilinx's Core IP's to establish appropriate communication. The system comprises two main transmission flows. One for the data and other for the NVDLA configuration space bus (instruction bus). The complete system architecture is discussed and also a corresponding figure 3.9 is presented.

1. Data Transmission flow: The data backbone interface of NVDLA wrapper is connected as Direct Memory Access (DMA) which accelerates data transfer from external DDR4 memory of Zynq FPGA. The data transmission takes place directly to the NVDLA wrapper without passing through the host processor. These interface connections can be instantiated given all the AXI ports are properly matched between the peripherals. This high-performance slave interface is part of programming logic in Zynq MPSoC and designated a data width of 64 bits.

2. Configuration flow: The configurations space bus is APB compliant. It is a simplified design for low bandwidth control access like the register interfaces. Unfortunately, these APB interfaces are not supported in Zynq Ultrascale+ FPGA. Therefore transformed into AXI-4 compatible.
 - In this case, an AXI-APB peripheral is utilized to translate APB transfers into AXI-4 transfers. This soft IP core functions as a slave on the AXI-4 interface and master on the APB3/APB4 bus. They interlink APB slaves to AXI masters, supporting 32-bit data widths.
 - The connection to the memory-mapped master of Zynq MPSoC is established through an AXI interconnect IP. It enables a connection between an AXI master and slave device. The following system employs a data width of 32-bits. This subsequent interface is exploited for AXI memory-mapped transactions.
3. Zynq UltraScale+ MPSoC is the main processing system of the hardware architecture. IP cores are attached as a PS+PL configuration. The following PS-PL configurations can be customized based on different requirements. Here the default settings are employed. And the output clock configuration is set to 100MHz.
4. The one-bit level driven interrupt interface of NVDLA IP is connected to PL to PS interrupts handled by the host processor.
5. The clock and reset interfaces of the system architecture are connected appropriately and sourced from PL clock and resetn pins of the processor core.

Once the system integration is done, the developed block design is validated for errors and warnings. Subsequently, the output products of the design are generated as out-of-context as well as a top-level HDL wrapper is constructed for synthesizing the advanced design. Followed by an implementation process carried out to place and route the design components pertinently. This process generates a design specific to the FPGA prototype. Next, the low-level configuration for the specific FPGA is generated as bitstream. Finally, the hardware is exported to SDK including bitstream, this provides the HDF file which configures hardware platform specification. The following emulates hardware-specific flow but the software to run on it is established using Petalinux tools described in the next section.

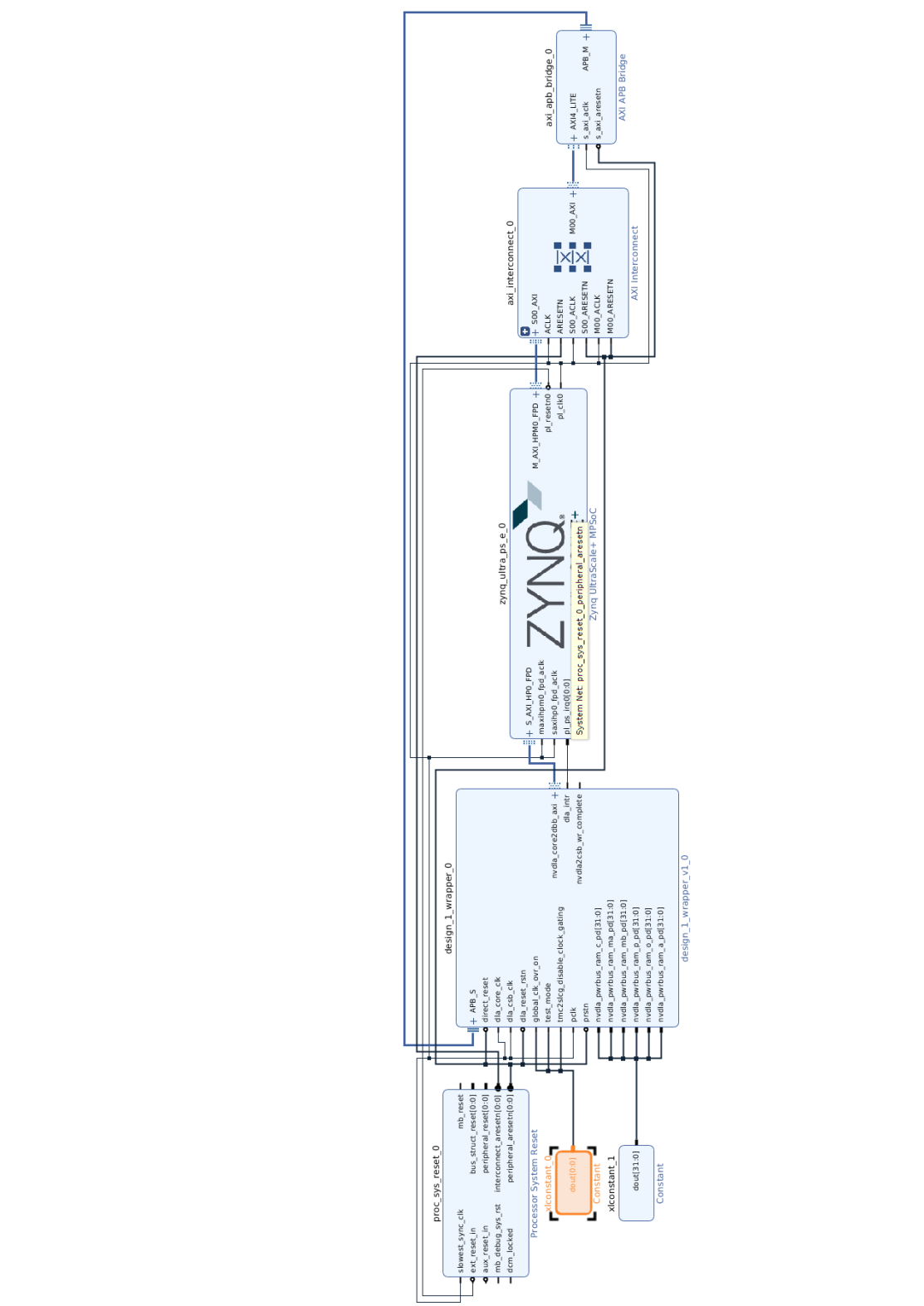


Figure 3.9: NVDLA System Architecture

Configuring the Software Environment

This chapter encompasses the software flow for the NVDLA system which comprises compilation and run-time environments as shown in the subsequent image 4.1. The compiler incorporates different machine learning model transformations and run-time environment executes these compiled architectures. Here an existing model of Alex-Net (a deep convolutional neural network) architecture is used as a standard compilation model. While the run-time environment engages the software to run this standard on the NVDLA hardware. The `PetaLinux` tool is used to realize these functions and deploy embedded Linux solutions on FPGA prototyping systems. A detailed description regarding the organization of the execution environment is explained in the consequent chapter.

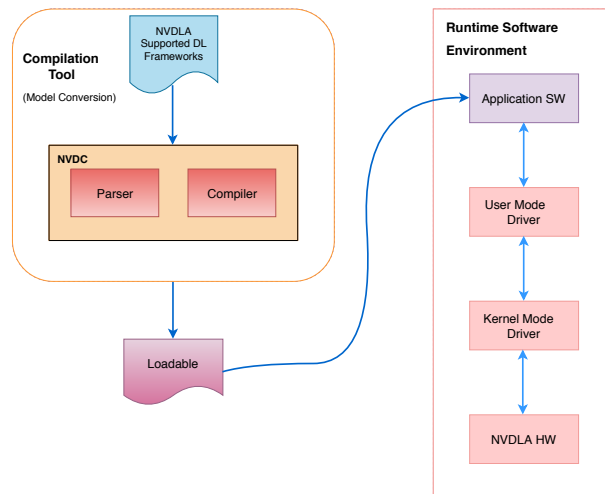


Figure 4.1: NVDLA Software Stack

4.1 Compilation Build

The compilation build converts a deep neural network into a series of hardware layers compatible with the specified NVDLA hardware configuration. The com-

piled model is optimized for the underlying hardware enhances the performance reducing the model size and its execution time. This build consists of two main phases.

- **Parser:** The parser creates a representation of the neural network as a set of different layers translated from the pre-trained Caffe model given as input. It passes this representation to the compiler and acknowledges if these networks are consistent with the given NVDLA flow.
- **Compiler:** The compiler translates the representations from parser into a series of hardware layers following the insights of NVDLA specification defined during implementation. It maps the network operations to the respective functional blocks of NVDLA. The compiler designates these functionalities primarily based on the NVDLA configurations. These operations are done offline and the compiled neural networks are stored in a standard format known as NVDLA Loadable.

Sadly, the NVDLA compiler was not open-sourced and transparent during this project phase. As a result, the compiler wasn't supporting various network models and their hyper-parameter configurations. Therefore the available network standard of Alex-Net, an used case with the NVDLA hardware, was utilized as a benchmark to evaluate the complete system architecture.

4.2 Run-time Execution

This run-time execution layer includes device drivers to the application software namely **User Mode Driver** and **Kernel Mode Driver**. These drivers provide abstraction serving as a translator between the NVDLA hardware and PetaLinux application software. These drivers are incorporated in PetaLinux environments as part of the kernel through loadable modules. Also, they are defined as Application Programming Interfaces wrapped around system portability layers, to facilitate flexible coherence with different hardware platforms. A general framework of the Linux execution environment is described in the figure 4.2.

- **User Mode Driver:** The User Mode Driver is the primary interface to the application software. It loads the NVDLA loadable standard and submits the inference task to Kernel Mode Driver. This driver loads the network tensors into memory engaging as an `ioctl()` function. Besides, this also employs synchronizations with different hierarchies before carrying out with inference task.
- **Kernel Mode Driver:** The Kernel Mode Driver acts as the core module of the software flow. It handles interrupts, associates optimized scheduling of layers and updates dependencies after operations. Further, these drivers program the functional blocks of the underlying NVDLA hardware.

The run time environment of NVDLA architecture is established employing PetaLinux tools. The execution of those device drivers are elucidated in the ensuing sections.

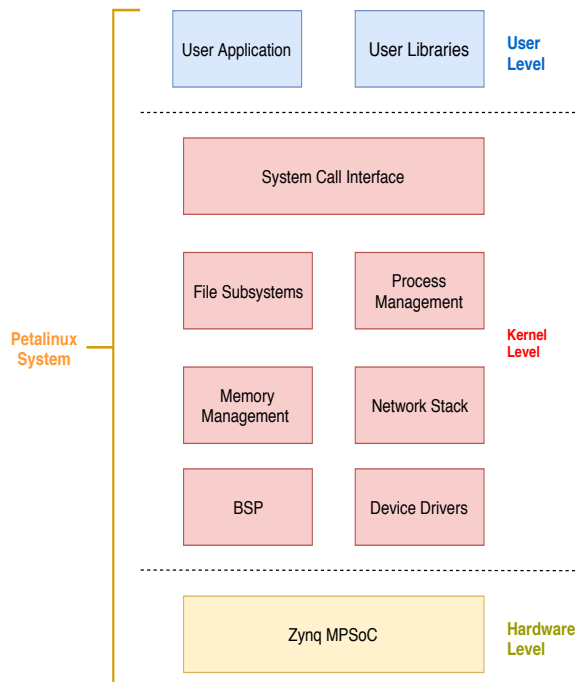


Figure 4.2: Linux Execution Environment Framework

4.3 PetaLinux Flow

The PetaLinux tools are exploited in this project to accelerate the development of NVDLA's run-time environment adopting an embedded Linux based solution. PetaLinux tools facilitate a hardware/software co-design offering integrated Linux configurations and software development tools. Simplifying deployments of hardware designs on FPGA platforms. The complete integration of software stack inclusive of kernels, device drivers, system boot and UMD realizations are described in this section.

The PetaLinux 2018.3 utilized for this purpose operates on a 4.14 Linux Kernel created as an application software available from Xilinx Git. As they do not deliver any commercial Linux distribution. This Linux execution environment is configured on Zynq Ultrascale+ MPSoC, the host processor of NVDLA hardware. The hardware design accomplished in Vivado earlier is exported as a hardware description file to initialize PetaLinux software. The software stack developments of this architecture follows the subsequent workflow as shown in fig 4.3.

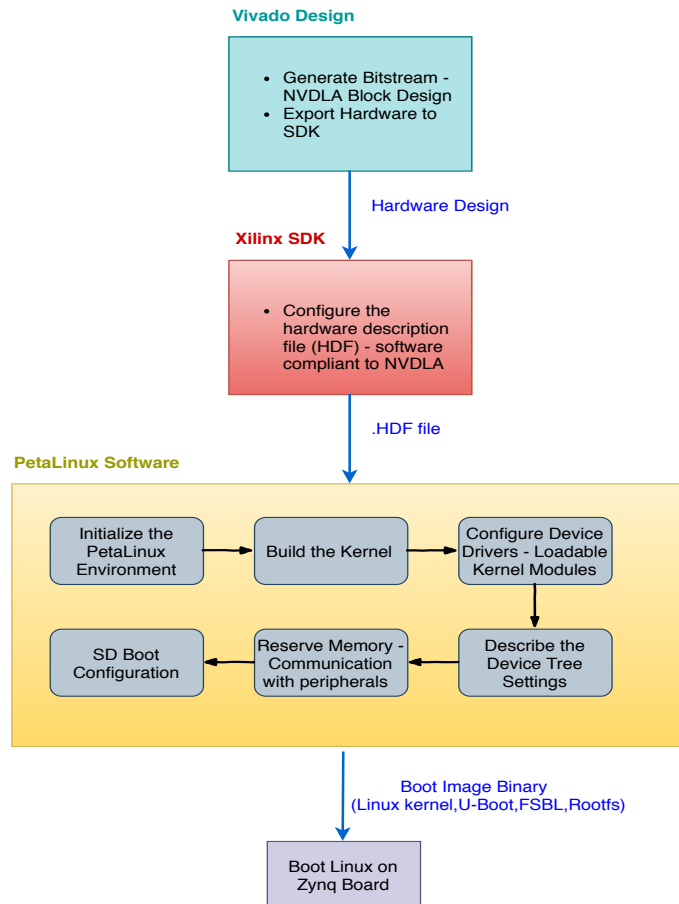


Figure 4.3: PetaLinux Flow

4.3.1 Initializing the PetaLinux environment

The primary step constitutes of initializing the PetaLinux environment. The hardware description file imported from the corresponding Vivado tools provides information to PetaLinux, generating appropriate software settings and Boot-loaders. Essentially boot header files, device tree source files, and Kernel drivers. Here the PetaLinux environment is initialized to configure kernel and u-boot to point to the SD boot. The SD card acts as the only memory storage device in this case, therefore the Root Filesystem is set to the same type.

4.3.2 Building the Kernel

PetaLinux provides the flexibility to build custom Linux distribution establishing diverse system tool-kits and libraries in the root directory. An optimal package group with essential tools are adopted in the Linux kernel to build API sub-routines interface to hardware. With these elementary system organizations complete, the

petalinux-build is executed to build the kernel system image.

4.3.3 Device Drivers as Loadable Kernel Modules

Most of the fundamental device drivers are enabled while building the above kernel image. But explicit NVDLA device drivers are defined as loadable kernel modules. As a result, an external module is created over the pre-built kernel.

Linux Operating Systems typically maintains high compliance providing capabilities to extend kernel features during run-time. The functionalities of these Linux kernels are customized on the fly through programmable modules. These modules are compiled with the Linux kernel and built into Kernel object files (.ko) which are loaded using insmod during boot. These modules comprise programmable sections that establish communication between the kernel and NVDLA hardware.

4.3.4 Device trees and Reserved Memory

Device trees are wielded as the default methodology to describe low-level hardware information from the boot-loader to the kernel. They are constituted in the source include files (.dtsi). It includes a simple tree structure of nodes and properties that determines the peripheral devices available for the kernel in the current environment. The OS uniquely identifies the underlying NVDLA hardware through the compatible property specifying the exact device information.

A reserved memory is delegated for custom device driver usage from system RAM and these address spaces cannot be exploited by the kernel. Here 1GB memory is reserved for memory-mapped devices specifically for DDR memory interface through Processor Subsystems (PS DDR). This PS DDR is assigned a base address and memory size. The address-cells and size-cells are selected 2 for designating 64-bit addressing schemes in the device tree. The reg field defines the address range used by the device. The PetaLinux project is recompiled to effect the encompassed module and device tree settings. A snippet of the device tree is encompassed below in fig 4.4.

```
/include/ "system-conf.dtsi"
/{
    reserved-memory {
        #address-cells = <2>;
        #size-cells = <2>;
        ranges;

        nvdla_reserved: buffer@0 {
            no-map;
            reg = <0x0 0x40000000 0x0 0x40000000>;
        };
    };
};

&design_1_wrapper_0 {
    compatible = "nvidia,nvdla_2";
    memory-region = <&nvdla_reserved>;
};
```

Figure 4.4: Device Tree

4.3.5 SD Boot Configuration

The SD card is partitioned into three divisions such as Boot, Root, and Test respectively. The boot partition refers to embedded Linux image which contains everything inclusive of u-boot binary files, FPGA bitstreams, first stage bootloader as well as kernel image. The secondary part holds the root file system. These segregations are formatted as FAT32 and EXT4 types respectively. The third partition includes the source files for compiling UMD also NVDLA loadable for inferencing AlexNet on the subsequent setup. Lastly, the FPGA board is set up to SD boot mode using the dip switches appropriately. Thus, the run-time execution environment of NVDLA is accomplished in PetaLinux.

Results Analysis

This chapter summarizes the results of NVDLA SoC implementation on the FPGA. The initial section explicitly explains the NVDLA hardware design results from Xilinx Vivado. While the later section highlights sanity tests applied for functional verification of the implemented system on FPGA. Lastly, a neural network such as AlexNet is employed as a benchmark to run on-device inference on the FPGA hardware.

5.1 Hardware Performance Parameters

The NVDLA hardware design results are discussed in terms of device utilization, power consumption, and system frequency parameters. These results are accomplished using Xilinx's Vivado Design Suite tools along with Zynq Ultra-scale+ ZCU104 as an FPGA target. The NVDLA architectures `nv_small` and `nv_medium` are studied to these performance parameters. While the `nv_large` architecture was an immense design that did not fit in this respective FPGA model. These performance parameters for the corresponding NVDLA architectures are explored based on Synthesis and Implementation processes.

5.1.1 System Frequency and Timing Analysis

The NVDLA architecture is executed at a system frequency of 100MHz (10ns). When the frequency is increased beyond this subsequent value, the timing constraints for the design were not met. It is important to meet the timing, to avoid long execution times and low system performances. Initially, this system architecture resulted in a negative slack even for the corresponding frequency. Since the primitive design included clock and power gating features in its native ASIC implementation. Unfortunately, these optimizations resulted in timing violations. Thus, the gating features were disabled by defining appropriate Verilog macros as explained in the hardware implementation section. Once disabled, the timing closure for the design is met at a frequency of 100MHz.

Setup	Hold	Pulse Width
Primitive Design		
Worst Negative : 0.443ns Slack (WNS)	Worst Hold : -0.341ns Slack(WHS)	Worst Pulse Width : 3.498ns Slack(WPWS)
Total Negative : 0.000ns Slack (TNS)	Total Hold : -26.785ns Slack(THS)	Total Pulse Width : 0.000ns Slack(TPWS)
Optimized Design		
Worst Negative : 3.814ns Slack (WNS)	Worst Hold : 0.030ns Slack(WHS)	Worst Pulse Width : 3.498ns Slack(WPWS)
Total Negative : 0.000ns Slack (TNS)	Total Hold : 0.000ns Slack(THS)	Total Pulse Width : 0.000ns Slack(TPWS)

Table 5.1: Post Implementation timing summary.

The design meets timing ideally when the values of Total Negative Slack (TNS), Total Hold Slack (THS) and Total Pulse Width Slack (TPWS) are 0ns. The sum of all these slack values represents the final timing results. From the timing results table 5.1, it can be seen that the Worst Negative Slack for setup time has positive values in both the designs. This indicates that the data arrives earlier, before its required time during the setup time analysis. As a result, the design path meets the setup timing constraints.

But the primitive design retained a negative slack value in the hold timing constraints where the data takes longer time to arrive than its required time. Consequently, failed to satisfy the timing results. Hence the extra hardware logic utilized for gating mechanisms is removed. Then it can be perceived from the optimized design that hold timing constraints results to 0ns. Thus, verifying clean timing sign off for the implemented design.

5.1.2 Device and Power Utilization

The device utilization, as well as power results, are analyzed for `nv_small` and `nv_medium` architectures. The power values are estimated from the implemented netlists in Vivado. These performance metrics specially power and resource consumption are directly dependent on each other. The NVDLA's architecture primarily comprises of MAC units and convolution buffers. Therefore, those parameters are paramount during performance/ efficiency (Area and Power) trade-off analysis. The next critical parameter that affects remarkably the above results is memory bandwidth. The succeeding results table 5.2 compares these parameters for respective NVDLA architectures. The result presents `nv_medium` consuming 0,38W power greater than the `nv_small` design. While the `nv_medium` caters higher operational throughput and performance at the cost of increased power utilization.

Feature	NV_Small	NV_Medium
# MAC	64	512
Buffer Size	128KB	256KB
Memory Bandwidth	1GBs	10GBs
Power Consumption	4.441W	4.820W

Table 5.2: Post Implementation power consumption.

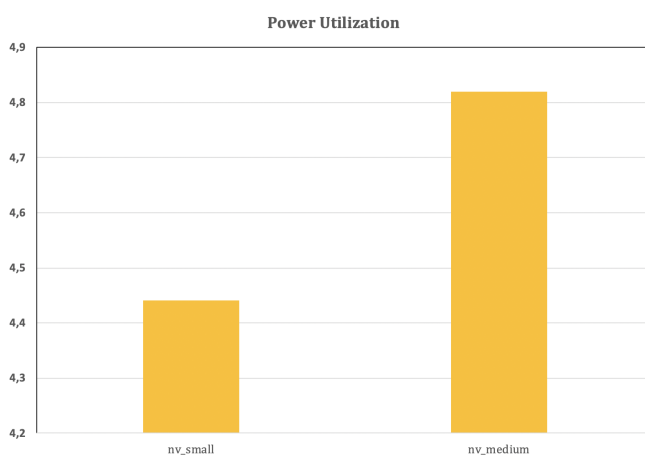


Figure 5.1: NVDLA Power Consumption graph

In terms of device utilization, from the below results table 5.4, 5.6 it can be observed that the usage of Look Up Tables (LUT), Flip Flops (FF), Block memory (BRAM) and Digital Signal Processor (DSP) for the `nv_medium` architecture is doubled than that of `nv_small` design. The results are evident due to enlarged feature values in `nv_medium` implementation. Specifically, the number of MAC units, convolution buffer size, and memory bandwidth features.

Resource	Utilization	Utilization%
LUT	80521	29.38
FF	86309	15.75
BRAM	93	10.20
DSP	32	1.27

Table 5.3: Post Synthesis Utilization results for NV Small.

Resource	Utilization	Utilization%
LUT	78437	28.62
FF	85266	15.55
BRAM	93	10.20
DSP	32	1.27

Table 5.4: Post Implementation utilization results for NV Small.

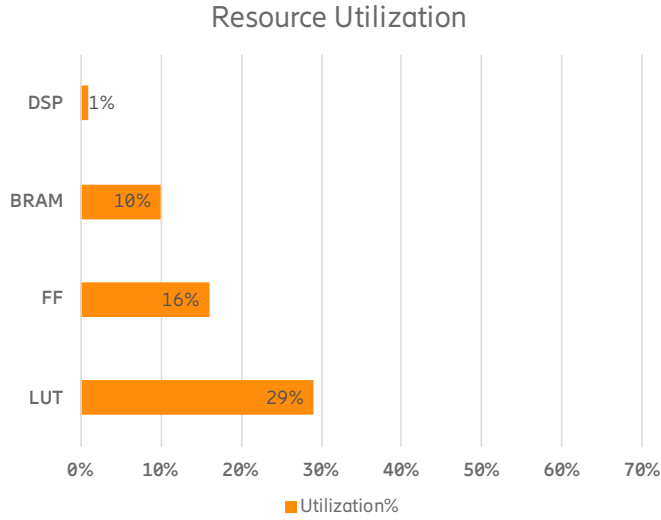


Figure 5.2: NV Small Utilization graph

Besides, it can be observed from the subsequent results, that device utilization values after implementation possessed lower utilization percent than synthesis. This condition exists because different IP's in the NVDLA block design were executed in OOC (Out of Context) mode during synthesis. As a result, accurate utilization values could not be obtained. Whereas, the results after implementation are definite. Since the process is accomplished over the synthesized net-lists. It is also interesting to observe that the DSP usage was significantly low in comparison to other resources considering the substantial usage of multiply-accumulate operations of NVDLA. From the RTL description, it is examined that the multipliers utilized an efficient hardware implementation of Wallace tree multiplication [29]. Correspondingly, they are implemented in the PL fabric of the FPGA and not mapped to DSP slices.

Resource	Utilization	Utilization%
LUT	161324	58.86
FF	152566	27.83
BRAM	185	20.28
DSP	65	2.59

Table 5.5: Post Synthesis utilization results for NV Medium.

Resource	Utilization	Utilization%
LUT	159240	58.10
FF	151950	27.72
BRAM	185	20.28
DSP	65	2.59

Table 5.6: Post Implementation utilization results for NV Medium.

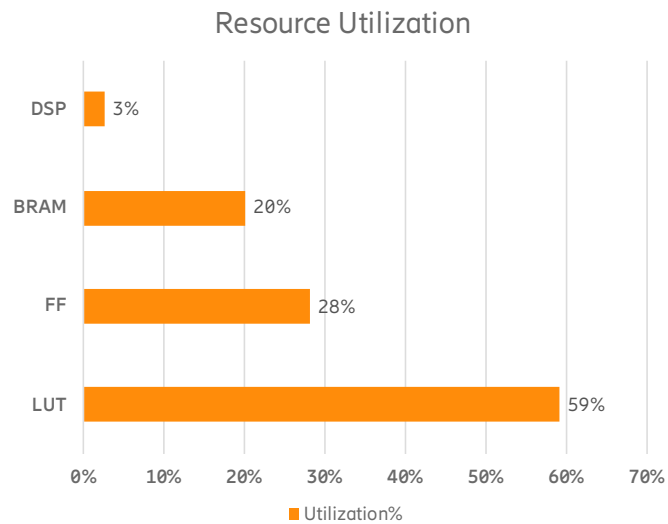


Figure 5.3: NV medium utilization graph

5.2 Accelerator Performance comparisons

To compare and evaluate different DNN hardware architectures, it is necessary to consider various bench-marking metrics. The metrics primarily comprise of throughput/latency, power/energy consumption and cost parameters. The latency/throughput determines if the system can process in real-time. The power/energy defines the hardware design specifications. The cost primarily in terms of area states how much one pays in terms of silicon for the particular solution.

In the image 5.4 different Deep Learning Accelerators (DLA) implemented on the FPGA are compared. The comparison includes recent accelerator designs such as OpenCL based DLA [30], Automated systolic array architectures [31], ALAMO [32], Angel-Eye [33] compared with the architectures realized in this

Specifications	Opencl based DLA	Systolic Array Architecture	Alamo	Angel-Eye	<i>NV_Small</i>	<i>NV_Medium</i>
Platform	Arria 10 GX 1150	Arria 10 GX 1150	Stratix-V GXA7	Zynq XC7Z020	Zynq Ultrascale+ ZCU104	Zynq Ultrascale+ ZCU104
Precision	16 bit Half (binary 16)	FP32	INT8/INT16	INT8	INT8	INT8
Frequency (MHz)	303	240	100	214	100	100
Resources						
LUT	247,776	350,304	122,054	29,792	78437	159240
FF	683,520	N/A	N/A	35,112	85366	151950
BRAM	2496	2360	1562	86	93	185
DSP	1473	1290	256	190	32	65
Power (W)	45	17.36	19.5	3.49	4.441	4.820
DL Benchmark	AlexNet	AlexNet	AlexNet	VGG-16	ResNet-50	ResNet-50
Performance (GOPS)	1382	360.4	114.5	84.3	12.8	102.4
Power Efficiency (GOPS/W)	30.71	20.75	5.87	24.1	2.88	21.24

Figure 5.4: Performance Comparison of DNN Hardwares

thesis (nv_small and nv_medium). It is difficult to compare straightaway the performances of the above architectures. As there are lots of parameters needs to be considered and they are not evaluated consistently. The main objective of this comparison chart is to understand the co-relations of different specifications and their relative impacts on design implementations.

Generally, the hardware accelerators performances are compared when executing popular DNN models such as AlexNet, ResNet and GoogLeNet on ImageNet or MNIST data sets. From the illustration 5.4, it can be observed that the OpenCL based accelerator achieves the highest throughput performance (GOPS). As the specific architecture operates at a higher frequency and utilizes larger processing elements, evident from the values of the resources. Consequently, it consumes immense power. The OpenCL based accelerator and automated systolic array architecture comparatively consume large Block RAMs due to their increased precision for inference. The chart also portrays that the OpenCL based DLA approach achieves the highest power efficiency. But those values are evaluated for AlexNet which possesses relatively smaller model size compared to ResNet and VGG models.

The NVDLA implementations are so robust and versatile that it can be re-configurable based on performance requirements. The figure demonstrates that NVDLA realizations provide optimal performance. In terms of throughput and power efficiency even though the specific implementations include only the default design optimizations. As complete optimizations on hardware and algorithms levels are included only in NV_Large architecture. Besides, these corresponding architectures can be used as an initial hardware target for accelerating DNNs and evaluate the performance metrics.

5.3 System Verification

5.3.1 Regression Tests

When the NVDLA ecosystem is fully systematized, the sanity checks are employed to verify the basic functionality of the underlying hardware accelerator. The sanity checks are performed for every functional block (CONV, SDP, PDP, CDP) in the `nv_small` architecture. This test reads a respective input file, executes its corresponding layer and produces an output. The subsequent output calculates an md5 value that is compared with the appropriate md5 value of the golden data for each CNN layer. If they are equivalent, then the test case gets passed for the successive hardware block. Successively, the convolution, activation, pooling and normalization layers of NVDLA are verified individually using the above pre-compiled sanity tests as shown in the following fig 5.5.

```
root@NVDLA_v2:/mnt/umd/out/runtime/nvdl_runtime# ./nvdl_runtime -loadable
/mnt/kmd/PDP/PDP_L0_0_small_fbuf
creating new runtime context...
Emulator starting
submitting tasks...
Shutdown signal received, exiting
Test pass
root@NVDLA_v2:/mnt/umd/out/runtime/nvdl_runtime# ./nvdl_runtime -loadable
/mnt/kmd/CONV/CONV_L0_0_small_fbuf
creating new runtime context...
Emulator starting
submitting tasks...
Shutdown signal received, exiting
Test pass
root@NVDLA_v2:/mnt/umd/out/runtime/nvdl_runtime# ./nvdl_runtime -loadable
/mnt/kmd/SDP/SDP_X1_L0_0_small_fbuf
creating new runtime context...
Emulator starting
submitting tasks...
Shutdown signal received, exiting
Test pass
root@NVDLA_v2:/mnt/umd/out/runtime/nvdl_runtime# ./nvdl_runtime -loadable
/mnt/kmd/CDP/CDP_L0_0_small_fbuf
creating new runtime context...
Emulator starting
submitting tasks...
Shutdown signal received, exiting
Test pass
```

Figure 5.5: Sanity Tests Execution on FPGA

Generally, these pre-compiled loadable and regression test files are only compatible and verifiable for specific NVDLA architectures. The loadable files comprise of fixed network description, tensor representation, memory partition, and scheduling information. The programmability of these already compiled intermediate representations is limited. Since the NVDLA compiler was not open-sourced during the time of the thesis. Even for the verification, only the built-in tests were utilized. These verification tests were hardcoded specifically to certain architec-

tures as well as no real image inputs were applied. The complete specifications of the input image and layer information were encoded in those above-mentioned regression tests. Therefore, the degree of freedom for user modifications in the test environments is constrained.

5.3.2 AlexNet Execution

The NVDLA's verification environment also included a pre-compiled loadable file, that executes a complete CNN to demonstrate on-device inference acceleration. The pre-compiled file explicitly comprises of AlexNet network for the `nv_small` architecture (Appendix C.2). AlexNet is a powerful architecture that consists of five convolution layers and three fully connected layers. The following CNN architecture popularized ReLu activation and overlapping pooling techniques. As a result, it reduced training time and improved classification accuracy to a greater extent. Also, this network is typically used as a benchmark to examine performance metrics for different hardware inference accelerators.

The regression test with AlexNet is performed for this implemented hardware architecture. Unfortunately, the validation was unsuccessful. The test stalled during the execution of the second convolution layer as depicted in the fig 5.6. The NVDLA web-references estimated a possible run time for the following test to be around 6000s. But the output prevailed at the same convolution layer even after longer execution times on the FPGA. The main reason for the consequent output state was because of a control issue in the ping-pong buffer mechanism employed in convolution buffers. This mechanism as explained in (chapter 2) was exploited to improve system efficiency by reducing reprogramming latency. This buffer methodology concurrently programs the second group of buffers when the first group is processing the convolution computations. The hardware switches between the register groups through an interrupt based control flow. In this case, the second group of buffers after executing the second convolution layer waits for an interrupt to check if the previous group buffer has completed execution. Unfortunately, this interrupt was not issued. As a result, the second group of buffers was always waiting for the consecutive interrupt from the first group.

```
root@NVDLA_v2:/mnt/umd/out/runtime/nvdla_runtime# ./nvdla_runtime -loadable
/mnt/kmd/NN/NN_L0_1_small_fbuf
creating new runtime context...
Emulator starting
submitting tasks...

The process stalls!
```

Figure 5.6: AlexNet Execution on FPGA

To troubleshoot the above problem different methods are examined. Initially, it is important to verify if an appropriate interrupt handler is registered properly in the implemented system. When the interrupt outputs are analyzed, it returned a value specific to NVDLA substantiating that the system can receive and service interrupts. The next step analysed, if the memory mapping of DDR region shared between the host and FPGA was sufficient to handle a complete CNN like AlexNet. Even after allocating 1GB of reserved memory for the validation process, the output results did not show any progress with convolution layers.

NVDLA online community also discussed the same issues when executing AlexNet on NVDLA architecture. Some users highlighted that the updated device driver firm wares of runtime execution environment (UMD/KMD) were not adaptable with the old NVDLA's master branch on GitHub. It was very difficult to troubleshoot these run-time errors without a transparent NVDLA compiler and device driver information, which was not released during the thesis period. Besides, only a few pre-compiled neural networks were exploited on NVDLA to comprehend the inference acceleration of CNN in hardware platforms. Various CNN's like MobileNet, GoogLeNet, SqueezeNet could not be compiled in the Nvidia's Compiler framework (NVDC). As a result, some research groups started scrutinizing retargetable compiler frameworks designs for proprietary deep learning accelerators such as ONNC (Open Neural Network Compiler). ONNC designed the first open-source compiler platform that can be ported to NVDLA specific hardware platforms. Thus furnishing a greater degree of freedom to explore the NVDLA system flow. As an extension to the following thesis, the ONNC frontend can be linked with the underlying NVDLA hardware to research diverse CNN inference acceleration on specialized hardware.

Conclusion and Future Works

In this thesis, the primary objective was to implement the NVDLA architecture as an SoC design and prototype on the FPGA platform to perform on-device inference acceleration of CNN's. The particular implementation was accomplished to comprehend the consistent design flow of NVIDIA's deep learning standard frameworks. As a result, this thesis examined the complete system integration from the runtime execution environment of neural networks to efficient hardware implementation on Zynq Ultrascale+ FPGA. This precise framework standardizes deep learning inference acceleration and could be engaged as an initial target platform for DNN inference on the edge/mobile devices.

NVDLA caters to a versatile, robust architecture that can be configured differently based on the required performance levels and type of CNN employed for inference. In this thesis, two such architectures (small, medium) were explored following the comprehensive workflow along with the respective designs were prototyped on the FPGA. The hardware implementation results of the appropriate architectures illustrated that the key features especially the size of convolution buffers, number of MAC units and memory bandwidth have a significant impact on performance measures concerning execution time and power consumption. The Deep Compression techniques applied to the algorithm levels enhanced the energy efficiency by reducing the network sizes at the cost of optimal prediction accuracy. The NV_full architecture intended for high-performance applications with all the feature optimization was not implemented here. Since the FPGA employed did not fit the respective model.

The complete system was verified using the pre-compiled regression tests and AlexNet alike Intermediate Representation (IR) files. As the NVDLA's compiler was not open-sourced during the time of this thesis. The programmability on the compiler levels was limited. The pre-compiled files employed in this thesis comprises fixed network descriptions and scheduling operations. Besides, only specific CNN architectures such as AlexNet can be utilized for the on-device inference process.

Thus, future work concerning this thesis can include explorations of compilation frameworks that can be ported to the NVDLA hardware for the execution of diverse neural network models. Besides, the compiler architecture could be retargetable to different proprietary deep learning accelerators to leverage hybrid acceleration platforms. Also, some research can be undertaken to use CNN specific NVDLA accelerator for more target domains such as Recurrent Neural Networks as well as signal processing.

Key Takeaways:

- The NVDLA implementation is quite challenging considering the vast technological breadth of its ecosystem. Therefore it's necessary to be definite with the research objectives for examination as the NVDLA environment provides opportunities to explore different system hierarchies (Low-Level Hardware, System architecture, Compilation environment or System verification) exclusively.
- To accomplish the complete system integration start with the hardware accelerator part followed by run-time execution setup and test the developed system with the given pre-compiled models. Then if all the tests are passed carry on with the compilation environment.
- To develop and debug the NVDLA system rapidly, use the pre-built virtual simulators that run on Amazon Web Services (AWS) FPGA platforms.
- The hardware accelerator RTL generated in Verilog presents only a low-level netlist. Thus, the hardware logic becomes difficult to comprehend. To understand the hardware modules of the accelerator it is recommended to set up the NVDLA Verification suite, which is not accomplished in this thesis.
- The PetaLinux tools utilized for establishing embedded Linux applications in this thesis is not the best solution as the tool is not very flexible and compatible with different development environments. It might be preferred to build your custom kernel which isn't tool-specific and fulfills your requirements appropriately.
- The compiler platform during the thesis period was not open-sourced. Therefore, pre-compiled neural network models were utilized. To obtain more programmability to the implemented system, it is necessary to deep dive into AI compilers. To start with, ONNC (Open Neural Network Compiler) could be employed and ported to the underlying NVDLA hardware for executing different deep learning models.

References

- [1] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems*. 2012.
- [2] He, Kaiming, et al. "Deep residual learning for image recognition." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.
- [3] Taigman, Yaniv, et al. "Deepface: Closing the gap to human-level performance in face verification." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014.
- [4] Hannun, Awni, et al. "Deep speech: Scaling up end-to-end speech recognition." *arXiv preprint arXiv:1412.5567* (2014).
- [5] Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. "Neural machine translation by jointly learning to align and translate." *arXiv preprint arXiv:1409.0473* (2014).
- [6] Bahdanau Bojarski, Mariusz, et al. "End to end learning for self-driving cars." *arXiv preprint arXiv:1604.07316* (2016).
- [7] Farabet, Clément, et al. "Neuflow: A runtime reconfigurable dataflow processor for vision." *Cvpr 2011 Workshops*. IEEE, 2011.
- [8] Chakradhar, Srimat, et al. "A dynamically configurable coprocessor for convolutional neural networks." *Proceedings of the 37th annual international symposium on Computer architecture*. 2010.
- [9] Farabet, Clément, et al. "Cnp: An fpga-based processor for convolutional networks." *2009 International Conference on Field Programmable Logic and Applications*. IEEE, 2009.
- [10] Horowitz, Mark. "1.1 computing's energy problem (and what we can do about it)." *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 2014.
- [11] Chen, Yu-Hsin, et al. "Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional

-
- [12] Chen, Yu-Hsin, Joel Emer, and Vivienne Sze. "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks." *ACM SIGARCH Computer Architecture News* 44.3 (2016): 367-379.
- [13] Gao, Mingyu, et al. "Tetris: Scalable and efficient neural network acceleration with 3d memory." *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 2017.
- [14] Chen, Tianshi, et al. "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning." *ACM SIGARCH Computer Architecture News* 42.1 (2014): 269-284.
- [15] Chen, Yunji, et al. "Dadiannao: A machine-learning supercomputer." *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2014.
- [16] Du, Zidong, et al. "ShiDianNao: Shifting vision processing closer to the sensor." *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 2015.
- [17] Jouppi, Norman P., et al. "In-datacenter performance analysis of a tensor processing unit." *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017.
- [18] Parashar, Angshuman, et al. "Scnn: An accelerator for compressed-sparse convolutional neural networks." *ACM SIGARCH Computer Architecture News* 45.2 (2017): 27-40.
- [19] Han, Song, Huizi Mao, and William J. Dally. "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding." *arXiv preprint arXiv:1510.00149* (2015).
- [20] <http://nvdla.org/>
- [21] Lavin, Andrew, and Scott Gray. "Fast algorithms for convolutional neural networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2016.
- [22] Mathieu, Michael, Mikael Henaff, and Yann LeCun. "Fast training of convolutional networks through ffts." *arXiv preprint arXiv:1312.5851* (2013).
- [23] Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems*. 2012.
- [24] Nair, Vinod, and Geoffrey E. Hinton. "Rectified linear units improve restricted boltzmann machines." *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010.
- [25] He, Kaiming, et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification." *Proceedings of the IEEE international conference on computer vision*. 2015.

-
- [26] Shamma, Shihab A. "Speech processing in the auditory system II: Lateral inhibition and the central processing of speech evoked activity in the auditory nerve." *The Journal of the Acoustical Society of America* 78.5 (1985): 1622-1632.
- [27] Ioffe, Sergey, and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift." *arXiv preprint arXiv:1502.03167* (2015).
- [28] <https://www.learnopencv.com/batch-normalization-in-deep-networks/>
- [29] Wallace, Christopher S. "A suggestion for a fast multiplier." *IEEE Transactions on electronic Computers* 1 (1964): 14-17.
- [30] Aydonat, Utku, et al. "An openc1TM deep learning accelerator on arria 10." *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017.
- [31] Wei, Xuechao, et al. "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs." *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017.
- [32] Ma, Yufei, et al. "ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler." *Integration* 62 (2018): 14-23.
- [33] Lu, Liqiang, et al. "Evaluating fast algorithms for convolutional neural networks on FPGAs." *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017.

NVDLA Specification Files

The different configurations of NVDLA implementations especially (nv_small, nv_medium, nv_large) source files are illustrated in the succeeding figures(A.1,A.2,A.3) respectively. The hardware tree build generates a Verilog RTL code respectively based on the given configuration source file.

```
##define NV_SMALL 1
#define FEATURE_DATA_TYPE_INT8
#define WEIGHT_DATA_TYPE_INT8
#define WEIGHT_COMPRESSION_DISABLE
#define WINOGRAD_DISABLE
#define BATCH_DISABLE
#define SECONDARY_MEMIF_DISABLE
#define SDP_LUT_DISABLE
#define SDP_BS_ENABLE
#define SDP_BN_ENABLE
#define SDP_EW_DISABLE
#define BDMA_DISABLE
#define RUBIK_DISABLE
#define RUBIK_CONTRACT_DISABLE
#define RUBIK_RESHAPE_DISABLE
#define PDP_ENABLE
#define CDP_ENABLE
#define RETIMING_DISABLE
#define MAC_ATOMIC_C_SIZE_8
#define MAC_ATOMIC_K_SIZE_8
#define MEMORY_ATOMIC_SIZE_8
#define MAX_BATCH_SIZE_x
#define CBUF_BANK_NUMBER_32
#define CBUF_BANK_WIDTH_8
#define CBUF_BANK_DEPTH_512
#define SDP_BS_THROUGHPUT_1
#define SDP_BN_THROUGHPUT_1
#define SDP_EW_THROUGHPUT_x
#define PDP_THROUGHPUT_1
#define CDP_THROUGHPUT_1
#define PRIMARY_MEMIF_LATENCY_64
#define SECONDARY_MEMIF_LATENCY_x
#define PRIMARY_MEMIF_MAX_BURST_LENGTH_1
#define PRIMARY_MEMIF_WIDTH_64
#define SECONDARY_MEMIF_MAX_BURST_LENGTH_x
#define SECONDARY_MEMIF_WIDTH_x
#define MEM_ADDRESS_WIDTH_32
#define NUM_DMA_READ_CLIENTS_7
#define NUM_DMA_WRITE_CLIENTS_3

#include "projects.spec"
```

Figure A.1: NVDLA Small Specification file.

```
#define FEATURE_DATA_TYPE_INT8
#define WEIGHT_DATA_TYPE_INT8
#define WEIGHT_COMPRESSION_DISABLE
#define WINOGRAD_DISABLE
#define BATCH_DISABLE
#define SECONDARY_MEMIF_DISABLE
#define SDP_LUT_DISABLE
#define SDP_BS_ENABLE
#define SDP_BN_ENABLE
#define SDP_EW_DISABLE
#define BDMA_DISABLE
#define RUBIK_DISABLE
#define RUBIK_CONTRACT_DISABLE
#define RUBIK_RESHAPE_DISABLE
#define PDP_ENABLE
#define CDP_ENABLE
#define RETIMING_DISABLE
#define MAC_ATOMIC_C_SIZE_32
#define MAC_ATOMIC_K_SIZE_16
#define MEMORY_ATOMIC_SIZE_16
#define MAX_BATCH_SIZE_x
#define CBUF_BANK_NUMBER_32
#define CBUF_BANK_WIDTH_32
#define CBUF_BANK_DEPTH_512
#define SDP_BS_THROUGHPUT_4
#define SDP_BN_THROUGHPUT_4
#define SDP_EW_THROUGHPUT_x
#define PDP_THROUGHPUT_2
#define CDP_THROUGHPUT_2
#define PRIMARY_MEMIF_LATENCY_256
#define SECONDARY_MEMIF_LATENCY_x
#define PRIMARY_MEMIF_MAX_BURST_LENGTH_4
#define PRIMARY_MEMIF_WIDTH_128
#define SECONDARY_MEMIF_MAX_BURST_LENGTH_x
#define SECONDARY_MEMIF_WIDTH_x
#define MEM_ADDRESS_WIDTH_64
#define NUM_DMA_READ_CLIENTS_7
#define NUM_DMA_WRITE_CLIENTS_3

#include "projects.spec"
```

Figure A.2: NVDLA medium Specification file.

```
#define FEATURE_DATA_TYPE_INT8
#define WEIGHT_DATA_TYPE_INT8
#define WEIGHT_COMPRESSION_DISABLE
#define WINOGRAD_DISABLE
#define BATCH_DISABLE
#define SECONDARY_MEMIF_ENABLE
#define SDP_LUT_ENABLE
#define SDP_BS_ENABLE
#define SDP_BN_ENABLE
#define SDP_EW_ENABLE
#define BDMA_DISABLE
#define RUBIK_DISABLE
#define RUBIK_CONTRACT_DISABLE
#define RUBIK_RESHAPE_DISABLE
#define PDP_ENABLE
#define CDP_ENABLE
#define RETIMING_DISABLE
#define MAC_ATOMIC_C_SIZE_64
#define MAC_ATOMIC_K_SIZE_32
#define MEMORY_ATOMIC_SIZE_32
#define MAX_BATCH_SIZE_32
#define CBUF_BANK_NUMBER_16
#define CBUF_BANK_WIDTH_64
#define CBUF_BANK_DEPTH_512
#define SDP_BS_THROUGHPUT_16
#define SDP_BN_THROUGHPUT_16
#define SDP_EW_THROUGHPUT_4
#define PDP_THROUGHPUT_8
#define CDP_THROUGHPUT_8
#define PRIMARY_MEMIF_LATENCY_1024
#define SECONDARY_MEMIF_LATENCY_1024
#define PRIMARY_MEMIF_MAX_BURST_LENGTH_1
#define PRIMARY_MEMIF_WIDTH_256
#define SECONDARY_MEMIF_MAX_BURST_LENGTH_1
#define SECONDARY_MEMIF_WIDTH_256
#define MEM_ADDRESS_WIDTH_64
#define NUM_DMA_READ_CLIENTS_8
#define NUM_DMA_WRITE_CLIENTS_3

#include "projects.spec"
```

Figure A.3: NVDLA large Specification file.

PetaLinux Design Flow

A detailed block diagram illustrating the complete integrated design flow in PetaLinux environment is shown below in the figureB.1.

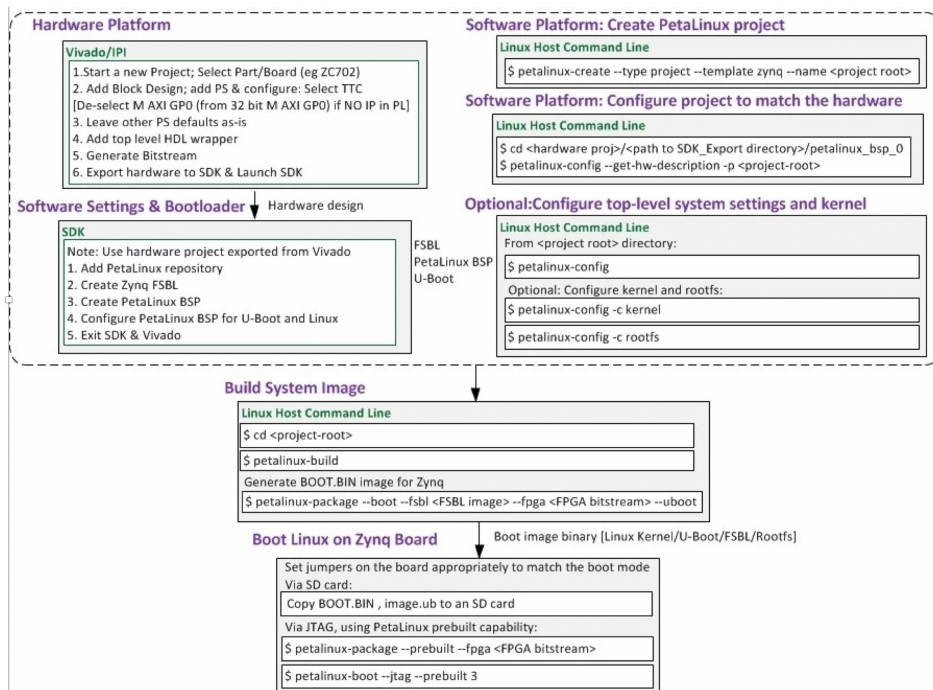


Figure B.1: General design flow in PetaLinux.source (From PetaLinux reference guide).

C.1 The Multi Layer Perceptron model

A Multi Layer Perceptron (MLP) comprises of an arbitrary number of hidden layers. The particular hidden layers commonly employ non-linear activations such as sigmoid and tanh functions. While at the output layer the activations utilized are linear for functional approximation, sigmoid for binary classification and soft-max for multi-class problem respectively.

For a given MLP model, consider the Input nodes x_k , Hidden layers h_j , Weights W_{jk} (Input to hidden layer), W_{ij} (between hidden layers and outputs), and Output nodes y_i . The output calculated through a forward pass is:

$$y_i(x_n) = \varphi_o\left(\sum_j w_{ij}\varphi_h\left(\sum_k w_{jk}x_{nk}\right)\right) = \varphi_o(w_i^T h_n) \quad (\text{C.1})$$

Here φ_o, φ_h are the output and hidden activations respectively.

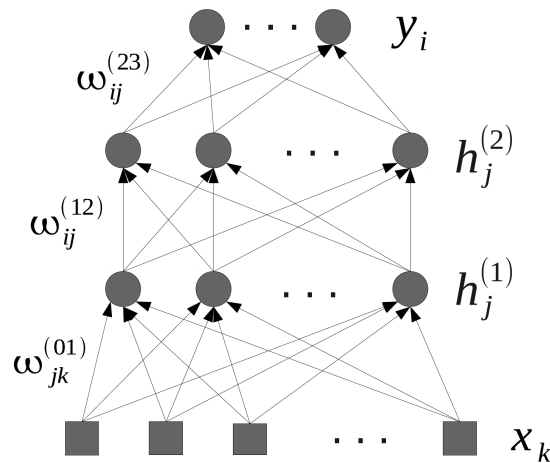


Figure C.1: A Multi Layer Perceptron model with two hidden layers.

C.2 AlexNet Architecture

The following figure C.2 contains a split into two pathways indicating the processing of the network in two GPUs. The input to the specific network is an RGB image of size 256x256. This network is much larger than previous CNNs such as LeNet used for computer vision application. AlexNet includes 60 million parameters and 650,000 neurons. The architecture took five to six days to train on two GTX 580 3GB GPUs. A layer wise summary of the network is illustrated in the fig C.3

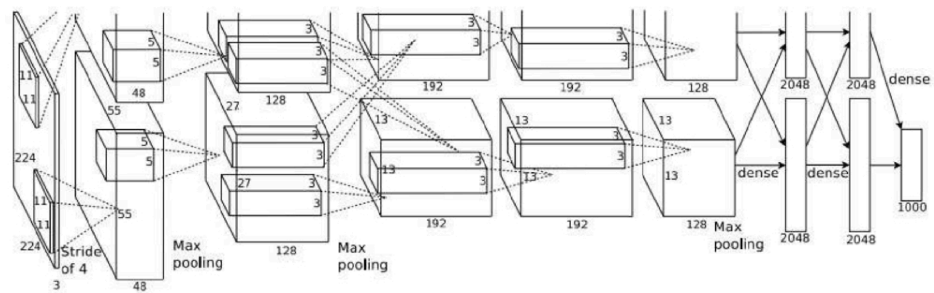


Figure C.2: AlexNet Architecture.

Layer	Units	Weights	Connections
L_1 (Conv)	290,400	34,848	105,415,200
L_2 (Conv)	186,624	307,200	111,974,400
L_3 (Conv)	64,896	884,736	149,520,384
L_4 (Conv)	64,869	663,552	112,140,288
L_5 (Conv)	43,264	442,368	74,760,192
L_6 (Dense)	4096	37,748,736	37,748,736
L_7 (Dense)	4096	16,777,216	16,777,216
L_8 (Dense)	1000	4,096,000	4,096,000
Conv Subtotal	650,080	2,332,704	553,810,464
Dense Subtotal	9192	58,621,952	58,621,952
Total	659,272	60,954,656	612,432,416

Figure C.3: AlexNet Layer-wise Analysis.
(Sourced from cs.toronto.edu/csc321 2018)



LUND
UNIVERSITY

Series of Master's theses
Department of Electrical and Information Technology
LU/LTH-EIT 2020-751
<http://www.eit.lth.se>