PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERIA

VALIDATION OF THE NVDLA ARCHITECTURE USING ITS AWS VIRTUAL PROTOTYPE-FPGA CO-SIMULATION PLATFORM

Tesis para obtener el título profesional de Ingeniero Electrónico

AUTOR:

David Steven Freidenson Bejar

ASESOR:

Ernesto Cristopher Villegas Castillo

Lima, April, 2023

Informe de Similitud

Yo, Ernesto Cristopher Villegas Castillo,

docente de la Facultad de Ciencias e Ingeniería de la Pontificia

Universidad Católica del Perú, asesor(a) de la tesis/el trabajo de investigación titulado

Validation of the NVDLA Architecture using its AWS Virtual Prototype-FPGA Co-simulation Platform,

del autor David Steven Freidenson Bejar,

dejo constancia de lo siguiente:

- El mencionado documento tiene un índice de puntuación de similitud de 12%. Así lo consigna el reporte de similitud emitido por el software *Turnitin* el 03/05/2023.
- He revisado con detalle dicho reporte y la Tesis o Trabajo de Suficiencia Profesional, y no se advierte indicios de plagio.
- Las citas a otros autores y sus respectivas referencias cumplen con las pautas académicas.

Lugar y fecha: Lima, 03 de mayo del 2023

Apellidos y nombres del asesor / de la asesora: Villegas Castillo, Ernesto Cristopher	
DNI: 45484048	[illegible]
ORCID: 0009-0005-8586-512X	

Resumen

La inferencia de Redes Neuronales Profundas (o DNNs, por sus siglas en inglés, *Deep Neural Networks*) se ha vuelto cada vez más demandante en términos de almacenamiento de memoria, complejidad computacional y consumo de energía. Desarrollar hardware especializado en DNNs puede ser un proceso tedioso, que se alarga aún más si se considera el tiempo requerido en escribir software para ello. Así, esta tesis consiste en la validación del acelerador de hardware de redes neuronales NVDLA (por sus siglas en inglés, *Nvidia Deep Learning Accelerator*) utilizando un ambiente de co-simulación basado en su plataforma híbrida: un CPU implementado como Prototipo Virtual (PV), basado en el *Quick Emulator* (QEMU), y el modelo de hardware en RTL del NVDLA dentro de un FPGA. Para ello, la arquitectura más portátil del NVDLA *nv_small* es configurada en el FPGA de una instancia F1 del servicio E2C AWS. Para complementar el sistema, el PV del NVDLA es usado, consistiendo de un CPU Arm emulado con QEMU, ejecutando el sistema operativo Linux y el software *runtime* del NVDLA, dentro de una capa de SystemC/TLM conectada al FPGA de la instancia F1 a través de un puerto PCIe. Una vez que la plataforma híbrida de co-simulación está configurada, se ejecutan regresiones de pruebas de hardware en la implementación en el FPGA para revisar la propia funcionalidad e integridad de los bloques que componen al NVDLA. Luego, se ejecutan pruebas de sanidad de software en el PV para confirmar la configuración correcta de todo el sistema integrado. Finalmente, la DNN AlexNet es ejecutada. Los resultados muestran la propia funcionalidad del hardware y del PV, y que la red AlexNet se ejecutó exitosamente en el ambiente de co-simulación, tomando aproximadamente 112 minutos.

Abstract

Deep neural network (DNN) inference has become increasingly demanding over the years in terms of memory storage, computational complexity, and energy consumption. Developing hardware targeting DNNs can be a lengthy process, which only grows if considered the time of writing software for it. Therefore, this thesis consists of the validation of the NVDLA deep learning hardware accelerator (NVDLA) using a co-simulation environment based on its hybrid platform: a CPU implemented as a Virtual Prototype (VP) based on Quick Emulator (QEMU) and the NVDLA RTL hardware model on a FPGA. For this, the more portable nv_small architecture of the NVDLA is configured into the FPGA of a F1 instance from the EC2 AWS service. To complement the system, the VP of the NVDLA is used, consisting of an Arm CPU emulated with QEMU running a Linux OS and the NVDLA runtime software, inside a SystemC/TLM wrapper connected to the F1 instance FPGA through a PCI express port. Once the hybrid co-simulation platform is set up, hardware regression tests are run on the FPGA implementation in order to check proper functionality and integrity of the NVDLA component blocks, sanity software tests are run on the VP to check the correct setup of the whole stack, and finally the AlexNet DNN is executed. The results showed proper hardware and VP functionality, and the AlexNet execution in the co-simulation environment was successful, taking approximately 112 minutes.

To my family for their unconditional love and support.

To my supervisor, Cristopher, for his valuable advice and encouragement.

Table of Contents

List of Figures

List of Tables

Introduction

Deep neural networks (DNNs) are just one of many algorithms that can learn a task when presented with big amounts of data. However, they have become the cornerstone of deep learning by achieving state-of-the art accuracy in many complex tasks such as financial forecasting, computer vision and natural language processing. That astounding precision has also come with very high computational complexity, energy consumption and memory bandwidth requirements. Consequently, deep learning accelerators (DLAs) have been developed. Nonetheless, their development takes a long time and writing software that works on them can lengthen the process even more. This has prompted hardware developers to try to parallelize the hardware and software development processes with high-level modelling and hardware virtualization technologies.

This thesis focuses on the testing and validation of the open-source NVDLA deep learning accelerator (NVDLA), using a hybrid co-simulation platform. The hybrid platform consists of the NVDLA's hardware configured on an FPGA as well as a virtual prototype (VP), comprised of a QEMU-emulated CPU running Linux OS with the NVDLA's runtime software, and SystemC/TLM wrappers for communication with the FPGA through Peripheral Component Interconnect express (PCIe). This hybrid simulation is performed using the F1 instances of the EC2 AWS service [1]. Tests were carried out to verify the proper configuration of the hardware in the instance's FPGA, as well as to check the VP's functionality. Finally, inference over the NVDLA running in the hybrid co-simulation environment is tested with the AlexNet DNN model.

This final coursework is organized in five main parts. Chapter 1 gives an overview of neural networks evolution over the years, the implications in hardware devices and the hardware/software development process. It also outlines some state-of-the art DLAs with their contributions, and finally presents the goals of this work. Chapter 2 establishes the theoretical background of neural network operations, the NVDLA hardware accelerator, high-level hardware modelling technologies and the AlexNet DNN, which was used to validate the system. Chapter 3 details the methodology followed throughout this work to set up, test and validate the NVDLA in the co-simulation environment. Chapter 4 presents the execution results of the testing and DNN inference on the NVDLA. Finally, sections for conclusions, recommendations and future works are added in the end.

Chapter 1: An Introduction To Neural Network Acceleration

In order to understand the challenge neural networks pose for traditional hardware, it is necessary to get familiar with the main trends that drive their development and success. Furthermore, it is important to get familiar with state-of-the-art hardware solutions already proposed and how they are developed. This then facilitates giving an accurate assessment of how co-simulation with a virtual platform can speed up and improve the development process of DNN hardware accelerators, with the aim of outlining the goals of this work.

1.1 Motivation

Deep neural networks are among the most successful algorithms in the field of Artificial Intelligence (AI). These have achieved much greater performance than any other predecessor in multiple applications, such as: computer vision, natural language processing, speech recognition, among others [2].

In recent years, the precision of neural networks has increased at an accelerated rate. For example, the accuracy of winning object recognition neural networks of the ImageNet competition has increased from 84.7% in 2012 to 96.5% in 2015 [3].

This performance increase has made it possible to implement DNNs in multiple commercial applications. A few of them are: Siri, Apple's personal assistant, semiautonomous cars from Tesla, which use computer vision technologies to navigate through traffic, and video editing tools from social networks like Instagram and TikTok. Nevertheless, these new capabilities imply an increase in the complexity of the implemented models.

One particular example is ResNet-52, the neural network which won the ImageNet competition in 2015 [4]. This DNN requires more than one additional order of magnitude of operations to be executed as compared to the winner in 2012, AlexNet [5][3]. Furthermore, AlexNet contains around 61 million parameters [6] occupying about 60 MB of memory, while VGG-16 [7], the DNN that won ImageNet two years after that, has 130 million parameters, requiring 500 MB to be stored [6].

Therefore, the implementation of a given neural network is limited by its inherent computational complexity. For mobile applications of the Internet of Things (IoT), these requirements are a challenge.

In the auto industry, autonomous cars require reliable, real-time processing of the images their cameras capture to navigate safely. According to [8], in order to have greater road safety than the one provided by a human driver's reaction time, a processing speed of a least 10 fps must be achieved. That includes detecting new objects, tracking the positions of the already detected objects and performing localization of the vehicle itself. Object detections constituted 89.5% of the latency of that process with a YOLOv2 network [9] in a CPU. This DNN requires 62.94 GFLOPs per inference [10]. So, based on this data, this application requires a minimum speed of 0.7 GFLOPs/s.

Consequently, the need for creating new specialized hardware architectures, also called domain-specific architectures [11], arises to such extent that it is estimated that by 2025 the demand for specialized HW for neural network processing will be a fifth of the total global demand of the semiconductors industry [12].

Deep neural networks are mostly comprised of mathematical operations with high granularity, a property which can be taken advantage of to design HW architectures with high parallelism [13]. Specifically, Convolutional Neural Networks (CNNs), perform the convolution operation with a set of filters, typically of size 3x3 or 5x5, to the input of the model or the output of one of its layers. The convolution operation is particularly granular and therefore suitable for HW acceleration.

Most of the energy consumption during DNN inference is done during convolution operations and memory accesses to the system RAM when loading the weights and the data to operate upon. This is especially true when the RAM is an off-chip memory since the data is further away from the processing elements (PEs) [3]. Consequently, neural network HW accelerators must prioritize lowering the computational cost of convolution operations and moving data between the PEs and memory.

The optimization of a HW architecture implies a trade-off between processing capacity, silicon area and energy consumption, which are common relevant aspects of chip design. That leads to the concept of energy efficiency, with measures of the number of operations per second per Watt. Moreover, many applications have constraints related to the area that the DNN accelerator can occupy on a chip.

Since neural networks are an emerging technology rather than an established one, their architectures can vary greatly and also in an unpredictable fashion. Therefore, hardware accelerators must be flexible enough as to adapt to the new advances in Deep Learning or they risk becoming obsolete in the near future [14].

Finally, based on the aspects of hardware DNN accelerator design outlined above and the volatility in neural network architectures, it is desirable that these accelerators are configurable, so as to be able adapt to the requirements of each application in which they are to be implemented.

1.2 Related work

Multiple architectures have been proposed to address deep learning hardware acceleration. Most of them are based in the "dataflow" architecture, shown in Figure 1, thus optimizing neural network processing focusing on the movement of data [6].

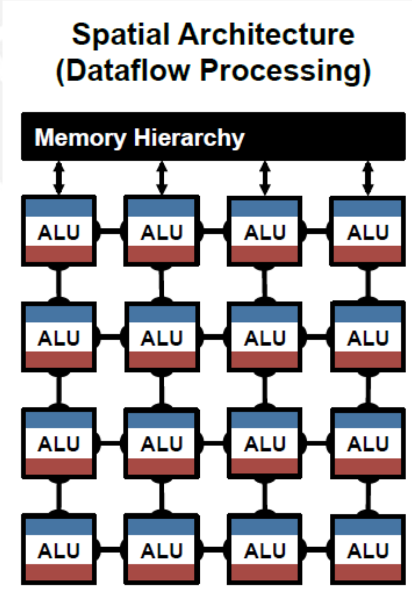


Figure 1. Dataflow paradigm of hardware accelerator architecture. In the upper part, the black block is the memory. Besides, there is an array of PEs, each one with its control registers (in red) and local storage registers (in blue) to minimize data movement of the intermediate results of the operations [6].

With this design paradigm, the deterministic memory access patterns and high parallelism of operations in DNNs can be exploited. Data that is already loaded from memory can be reused and computational efficiency is thus maximized. The ALU blocks in this case is the same as the processing elements, PEs. If these execute multiply-accumulate operations, they are also called MACs.

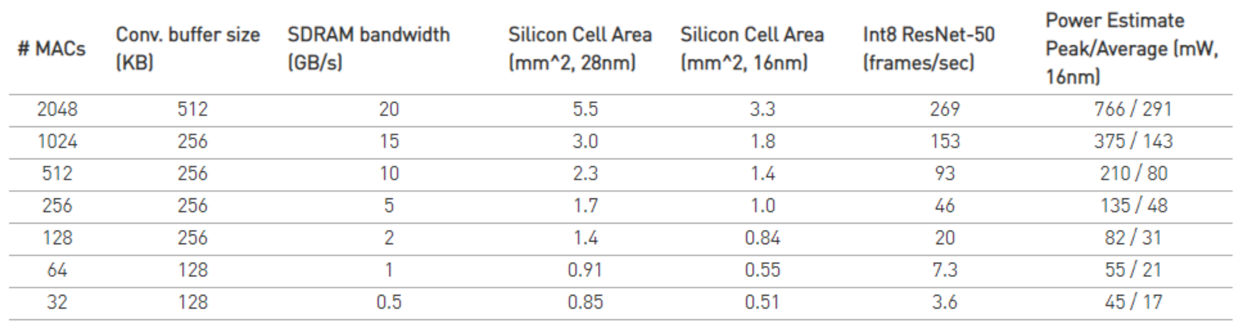
As a first example, the Ethos-78 accelerator from Arm [15] has more than 90 possible configurations, where it is possible to choose between having 512 and 4096 MAC units, INT8 or IN16 precision, and between 1 and 10 TOPs of performance, thus personalizing the accelerator to the application's needs. Additionally, the Ethos-78 can be configured to support the Winograd convolution algorithm [16], which can be up to 2.25 faster than direct convolution. Furthermore, this accelerator can support both convolutional and recurrent networks, covering two of the most common types of neural networks. It also supports weight compression, which lowers the memory bandwidth requirements and reduces the size of the stored model.

Secondly, Efficient Inference Engine (EIE) [3] is an accelerator specifically designed for sparse neural networks, that is, DNNs with a high number of weights equal to 0, and compressed networks. Sparse and compressed network inference implies that there are less weights to be considered during computation but creates the problem that the zero-valued weights are distributed randomly in the networks, making it hard to take advantage of this property. Seeking to address this problem, EIE achieves a performance of 102 GOPs in sparse neural networks, equivalent to 3 TOPs in dense DNNs, consuming only 600 mW. Therefore, it achieves an energy efficiency 24000 times greater than a CPU, and 3400 times greater than a GPU.

Finally, there is the NVDLA, an open-source accelerator published by NVIDIA [17]. It implements a series of blocks, each one dedicated to executing a particular set of common operations in neural networks. The NVDLA offers a wide set of configurable parameters, some of which are shown in Table 1.

Table 1.

Configurable parameters of the NVDLA.



# MACs	Conv. buffer size (KB)	SDRAM bandwidth (GB/s)	Silicon Cell Area (mm^2, 28nm)	Silicon Cell Area (mm^2, 16nm)	Int8 ResNet-50 (frames/sec)	Power Estimate Peak/Average (mW, 16nm)
2048	512	20	5.5	3.3	269	766 / 291
1024	256	15	3.0	1.8	153	375 / 143
512	256	10	2.3	1.4	93	210 / 80
256	256	5	1.7	1.0	46	135 / 48
128	256	2	1.4	0.84	20	82 / 31
64	128	1	0.91	0.55	7.3	55 / 21
32	128	0.5	0.85	0.51	3.6	45 / 17

Note: Taken from "NVDLA Primer — NVDLA Documentation", from Nvidia Corporation, 2018 [18].

The table shows the values of the possible number of MACs, local memory buffer of the convolution block and SRAM bandwidth, if included. Besides, for each configuration the area occupied by the NVDLA is given for 28 and 16 nm technologies, as well as the power consumption and the frames per second achieved when running ResNet-50 with INT8 precision.

Additionally, the NVDLA supports INT8, INT16, INT32, FP16 and FP32 precisions, as well as four types of convolutions, with the Winograd algorithm among them. It is designed to be integrated into a System on a Chip (SoC) in two different ways: one of them includes its own SRAM to store weights and activations, and the other one stores these at the system DRAM, which may be off-chip memory. Finally, the NVDLA supports weight compression as well [18].

The development process of hardware accelerators needs to consider hardware-software co-design, so the resulting hardware meets efficiency and performance targets for the intended workloads, as well as time-to-market objectives. Furthermore, standalone hardware accelerators require a collection of IP modules in order to operate, such as external memory, at least one CPU, bus interconnects, protocol converters, and domain-specific IP such as codecs for video processing, all bundled together in a SoC.

To that end, a series of development platforms are used in order to design quality DLAs. One of them is Cadence Tensilica [19]. It provides a collection of IP in the form of configurable RTL for specific applications such as audio and video processing, processing units such as CPUs that work with a RISC-style ISA that is both highly configurable and extensible and integration tools to include standalone DLAs or use Cadence's AI engines [20]. It also includes a toolchain to develop the software for the whole SoC, and a set of models that include Instruction Set Simulators (ISS) and cycle-accurate models in order to functionally run software, as well as model hardware, in early stages of the development process [21]. This platform enables quick development of a highly customizable SoC solution containing a DLA, since the whole toolchain facilitates design exploration from ISA to RTL, and both SW and HW can be developed in parallel, in a co-designed fashion.

Another development platform for the design of DLA solutions is the Synopsys Virtual Prototype models [22]. These models include a wide range of technologies, such as a TLM Library for Arm IP that can be used to create components that run software functionally (without performance data) as it would in real hardware [23]. Furthermore, the Synopsys solution also contains already written SystemC TLM models simulating IP such as CPUs and peripherals, that can be connected together with the DLA's model to run software, get quick performance data and make design choices within the design exploration space [24]. This enables running software in a functional and performance model of the SoC to be developed and accelerates time-to-market, since hardware decisions can be made before having RTL and both SW and HW can be developed in parallel.

Finally, an alternative development platform worth mentioning is Xilinx's PYNQ [25]. It consists of an FPGA development board based on the ZYNQ architecture and a software toolchain that allows for the synthesis of high-level software in C/C++ to RTL hardware, wrapped in a Python interface so it can be used by the programmer from the Python language. The FPGA board contains a programmable-logic block where the HW optimization blocks are synthesized, and a series of hard IP including interconnects, memory and a CPU, all within the same SoC. Thus, it is possible to create highly specific RTL designs for a particular software workload [26]. This platform allows for a wide exploration of the HW-SW co-design and experimentation at smaller scales than a full AI engine.

1.3 Proposed solution

This work is focused on performing a co-simulation of the NVDLA using both hardware in an FPGA as well as emulation and high-level modelling technologies such as QEMU [27] and the SystemC/TLM standard [28], [29]. The NVDLA accelerator is particularly convenient since it is open-source, and thus all the design, integration, verification and virtual platform specifications are available to the public [30].

Figure 2 shows the block diagram of the NVDLA. The Configuration interface block receives the location in memory of the instructions to be executed, as well as the required data and the parameters that control the execution, through the CSB interface. The Memory interface block then communicates with memory and receives the weights and data to be processed. The blocks that perform the network's operations are the following: the Convolution core (CONV, in Figure 2), which convolutes the activations with the weights, the SDP block, that can perform a variety of activation functions and bias addition, the PDP block, which executes pooling operations if needed, the CDP block, that does the LRN operation over activations, the RUBIK block, which changes the shape of the data in memory when necessary, and the BDMA, that serves as a communication link between the system DRAM memory and the NVDLA's dedicated SRAM memory, if included in the design.

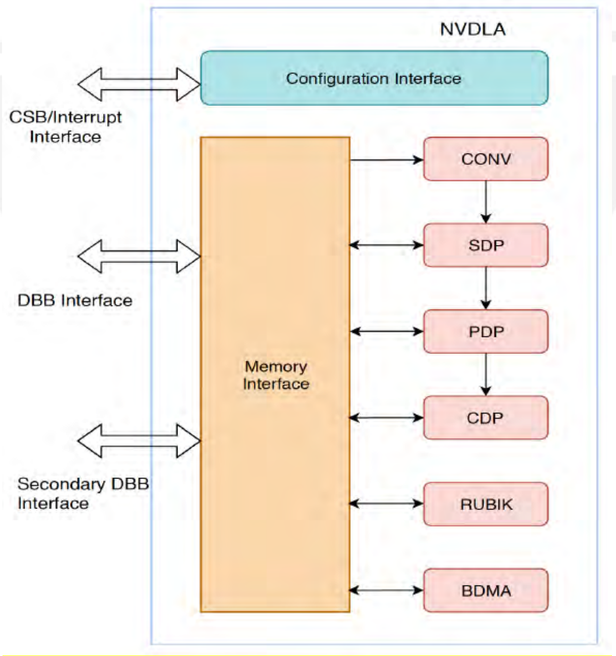


Figure 2. NVDLA block diagram. It shows the CSB and DBB interfaces for configuration and memory access respectively, as well as the processing blocks. The arrows between the blocks show the possible directions in which data can flow during processing [31].

All this access to its inner workings allows for the analysis and experimentation with the architecture. The NVDLA project also offers runtime software, which is composed of two drivers which allows the user to send DNN inference jobs to the DLA [32].

This software can be set up to run and execute inference over its virtual platform using its C behavioral model. However, this software does not allow for simulation using the RTL hardware description, which also includes timing and other implementation details. Therefore, it is limited to testing the software intended for the NVDLA but not the NVDLA hardware. With a VP-FPGA co-simulation, it is possible to test both, thus giving way to faster development and prototyping of new, improved versions of the NVDLA.

1.4 NVDLA development platform

Hardware accelerators require an external memory and a CPU in order to receive control commands and extract and store the necessary data, as they would have if being part of a SoC. The RTL model of the NVDLA will be used in this work. For the rest of the components, high-level models in SystemC/TLM and the QEMU emulator will be employed. In order to run a user application on the NVDLA, a software stack composed of an operating system, the NVDLA drivers and a command line interface (CLI) is needed. This software stack will run on the emulated QEMU Guest CPU.

To be able to run the NVDLA RTL model as well as the rest of the hardware components and the software stack, the Elastic & Agile Computing (EC2) AWS service will be used [1]. This service provides remote access to hardware instances that contain storage, CPUs and FPGAs, and are capable of running Linux on top. These instances can be accessed through an SSH connection, which will be the main interface to communicate with the remote hardware.

1.5 Objectives

1.5.1 Main objective.

Validate the NVDLA architecture by running a DNN on its AWS Virtual Platform.

1.5.2 Specific objectives.

1. Set up the NVDLA hybrid platform (QEMU+FPGA) in the AWS framework in order to run co-simulation tests.
2. Run hardware regression tests on the NVDLA to verify the sanity and integrity of each stage of the NVDLA RTL.
3. Run software sanity tests in order to demonstrate the proper setup of the hybrid platform employing the NVDLA runtime software.
4. Run the AlexNet DNN in the AWS QEMU-FPGA co-simulation environment using the nv_small configuration of the NVDLA architecture.

Chapter 2: Background

Deep neural networks and, specifically, convolutional neural networks, can have several different architectures. Nonetheless, most of them involve either a subset or the whole set of the following mathematical operations: convolutions, matrix multiplications, activation functions (linear or non-linear), pooling and normalization [33]. Since developing a DLA for all this functionality and a SoC to put it into can take significant time, it becomes highly beneficial to be able to parallelize the development of software for the target hardware. For this, high-level abstraction technologies such as SystemC, TLM, and CPU emulators can be considered.

2.1. Operations involved in neural network inference

This subsection illustrates some of the typical operations that shape the layers of a neural network which are relevant to understanding the work done afterwards.

2.1.1. Matrix multiplication.

This operation is mainly present in fully connected layers. In this type of layer, every neuron from the previous layer is connected to each neuron of the current layer. Therefore, if R is the number of neurons on the $(j\text{-}1)_{th}$ layer and Q is the number of neurons on the j_{th} layer, there are $R \times Q$ connections. Each connection represents a weight of the j_{th} layer, so the matrix $\boldsymbol{W}_{R\times Q}$ represents all the layer's weights. If the activations of the $(j\text{-}1)_{th}$ layer are represented by $\boldsymbol{X}_{R\times 1}$, then the matrix multiplication performed in the j_{th} fully connected layer is given by Equation (2.1) [34]:

$$o^j = \boldsymbol{W}_{R\times Q}^T \times \boldsymbol{X}_{R\times 1} \tag{2.1}$$

Where $\boldsymbol{W}_{R\times Q}^T$ is the transposed weight matrix, and o^j is the output of the operation corresponding to the j_{th} layer. Figure 3 gives an example of a fully connected layer.

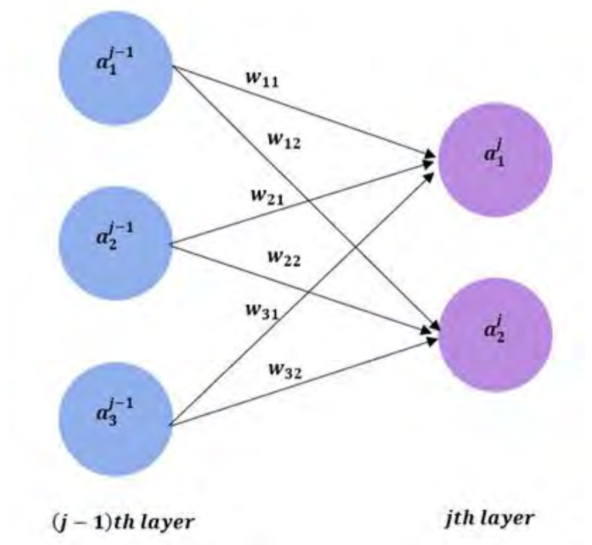


Figure 3. Example of a fully connected layer. Here, the *(j-1)_{th}* layer contains three neurons (*R*=3) and the j_{th} layer contains two (*Q*=2). X is a (*3x1*) vector, and W is a (*3x2*) matrix. a_i^l is the activation corresponding to the i_{th} neuron on the l_{th} layer, which is the result after applying the biases and activation functions to the output of equation 2.1.

2.1.2. Direct convolution.

There is a wide variety of convolution operations being used on deep neural networks. Nevertheless, the direct convolution is by far the most common convolution operation and serves as a starting point to understand all the others [35].

The direct convolution is an operation between two kinds of tensors. Firstly, the input tensor which contains the features that are to be convolved. Secondly, the kernels that have the layer's weights, which are used to perform convolution over the input tensor [36]. The dimensions of each component will be illustrated as follows:

- Kernel width and height = *k x k*
- Kernel depth = *d*
- Number of kernels = *N*
- Input width and height = *i x i*
- Output width and height = *o x o*

The constraints for the direct convolution are the following:

- The depth of every kernel must match the depth of the input tensor.
- As a result of the operation, the depth of the output will be the same as the number of kernels.

For simplicity, it is assumed that both the kernels and the input tensor have their widths the same as their heights, which is the configuration adopted by most DNNs implementations.

Figure 4 shows the convolution of a 2D *3x3* kernel with a 2D *4x4* input tensor. The operation starts on the top-left side as shown on step (1) of that same illustration. The input tensor is colored blue, and the kernel is placed in the dark-blue areas. The output is a 1D *2x2* tensor, colored green.

In step (1), the kernel is placed in the top-left corner of the input tensor, and an element-wise multiplication between the kernel elements and the overlapped input elements is performed. The nine partial results are added into the first element of the output, colored in dark-green. Then, the kernel slides rightwards one element and the multiplication process is performed once again. When the kernel reaches the rightmost possible position, it slides downwards one element and returns to the leftmost position, as shown in step (3).

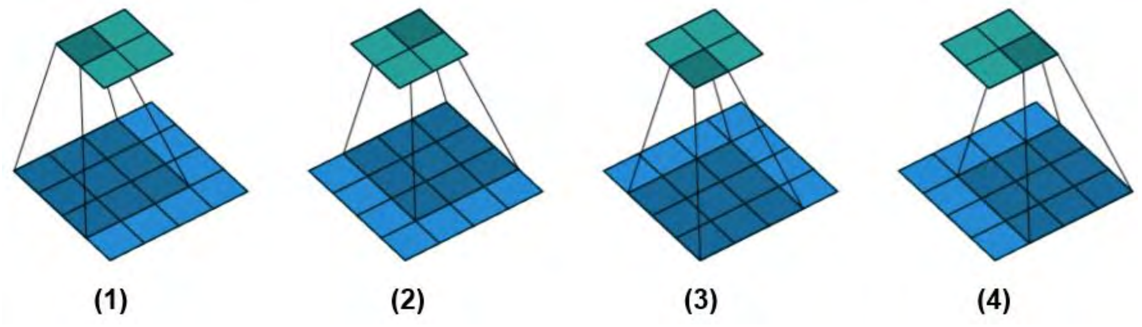


Figure 4. Example of a direct convolution. The convolution has a 2D *4x4* input tensor (*i*=4), in blue, and a 2D *3x3* kernel (*k*=3, *d*=1), in dark-blue. The output is a 2D 2x2 tensor (*o*=2) [36].

Provided *N* kernels with *kxkxd* dimensions, the convolution is performed for each kernel applied to the same *ixixd* input resulting into an *oxoxN* output tensor (e.g., *2x2xN* applied to the example above).

Furthermore, there are two additional important parameters of the direct convolution: padding and stride. On one hand, padding represents the number of elements (generally zero) that are added to the outer part of the input tensor, in terms of the height and width dimensions. For instance, a 3D input tensor of dimensions $[i \times i \times d]$, with a padding of 1, results in a $[(i+1) \times (i+1) \times d]$ tensor. The convolution is then applied to that new tensor. The stride, on the other hand, refers to the number of rows or columns that will be skipped when sliding the kernel during convolution [36]. In Figure 4, the stride is 1 which means no skipping.

In general, the values of the output width and height are given by Equation 2.1:

$$o = \left[\frac{i+2p-k}{s}\right] + 1 \tag{2.2}$$

Where p and s represent the padding and stride parameters, respectively. Figure 5 shows an example of a convolution with a stride of 2 and a padding of 1.

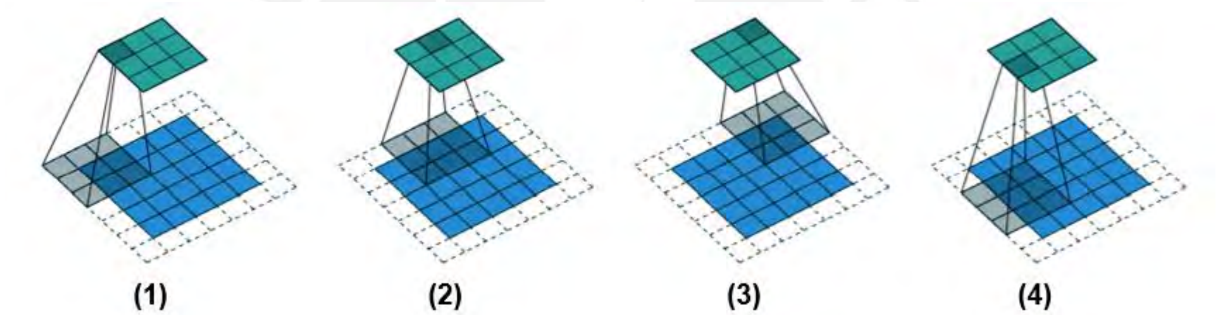


Figure 5. Example of a padded convolution. The convolution has a 2D 5x5 input tensor (i=5), in blue, and a 2D 3x3 kernel (k=3, d=1), colored in gray overlapping the input tensor to which a padding of 1 was applied (doted-lined borders, p=1). The output is a 2D 3x3 tensor (o=3) [36].

After the convolution, a bias is added to the output tensor to generate the final output. This is further explained in the next subsection.

2.1.3. Bias addition.

The bias is a common parameter present in several types of layers, such as fully connected and convolutional layers. In the case of fully connected layers, for each neuron there is a bias. In a convolutional layer, there is one bias per kernel. When a kernel is convoluted with the input of the layer, a single bias is then added element-wise to each element of the resulting tensor [36].

2.1.4. Activation functions.

To the output of a layer, a function is commonly applied in and elementwise fashion, which may be a linear or non-linear operation. Common activation functions include the Sigmoid function [37], hyperbolic tangent [38] and ReLU [39].

2.1.5. Pooling.

Pooling operations are used to reduce the size of a tensor. There are three main types of pooling operations: max pooling, min pooling and average pooling. Max pooling takes the output tensor of a previous layer and extracts the maximum value out of a subset of its elements [36]. This is illustrated in Figure 6.

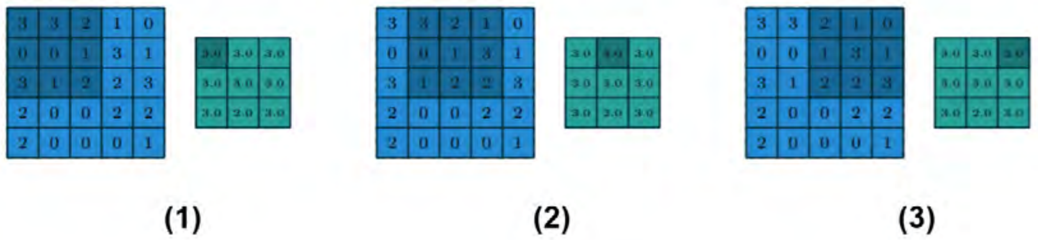


Figure 6. Example of a max pooling operation. It is performed over a 2D *5x5* input tensor, shown in blue, with stride 1 and a 3x3 kernel (k=3), shown in dark-blue. The output is the green matrix where the result of each step is highlighted in dark-green. The first three steps out of nine are shown [36].

Just as in the convolution, the kernel starts sliding from the top left corner to the bottom right corner of the input tensor, with the difference that the kernel operation picks the maximum value out of the group of overlapped elements colored in dark- blue in the example above. Pooling operations also work with the stride parameter, which has the same effect as in direct convolution. Likewise, the min pooling operation picks the minimum value out of the elements overlapped by the kernel. Finally, the average pooling operation outputs the average value of the overlapped elements at each step [36].

2.2. The NVDLA hardware architecture

The NVDLA is an open-source deep learning inference accelerator created by NVIDIA [18]. It is designed to support a wide range of common operations, like the ones mentioned in section 2.1. At the same time, it can be customized in many different configurations in order to adjust to the power, performance, and area requirements of each application. It is meant to form part of an SoC, together with memory blocks and at least one CPU [40].

2.2.1. Block description.

Figure 7 shows a block diagram of the NVDLA, where its different interfaces are highlighted, as well as the different blocks that perform the operations of DNNs.

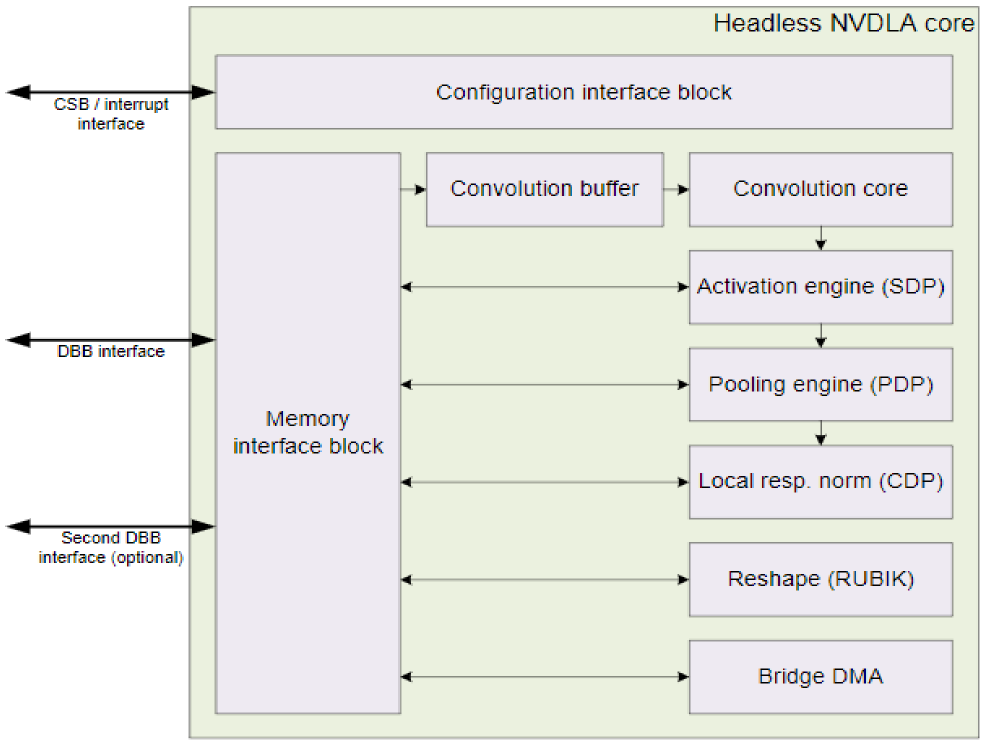


Figure 7. Block diagram of the NVDLA deep learning accelerator [18].

The overview of each block is as follows:

- **Configuration interface block:**

  This block contains the CSB control buffer, through which the configuration and programmability data come from the CPU. Each module that performs operations connects to this block to receive relevant data. Moreover, this interface block also contains one interrupt signal for the CPU, which asserts when an error has occurred or when a layer has finished executing [40].

- **Memory interface block:**

  This block contains the interface through which all operation blocks read and write weights and activations in system memory [40].

- **Convolution pipeline (convolution core and convolution buffer):**
  The convolution pipeline performs the fetching of the input features and weights and convolutes them according to the operation parameters like stride and padding, in a 5-stage process. This is shown in Figure 8, which encompasses the blocks of convolution core and convolution buffer.

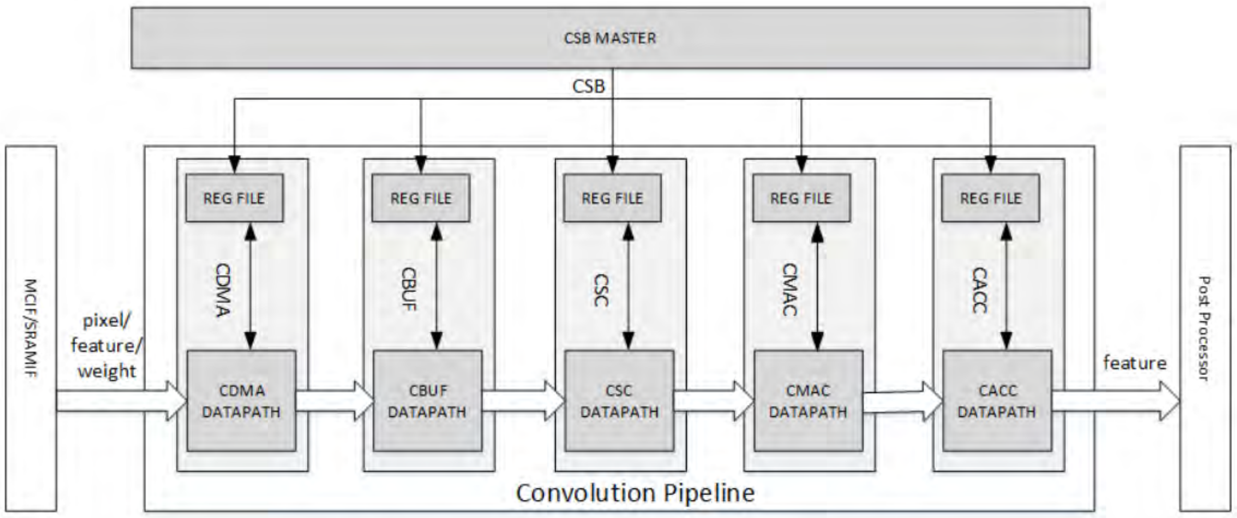


Figure 8. Block diagram of the 5-stage convolution pipeline with its interfaces [41].

Each stage has a register file (REG FILE, in the image), which receives from the CSB bus the configuration for each subblock for the execution of a convolutional layer. Then, this configuration passes to the block's registers and the operations are executed.

First, the CDMA fetches the input features and weights from memory, which are then passed on to the CBUF, which consists of a configurable-size set of banks of SRAM, where the data and weights are stored. Then, the CSC stage, which receives status information from the CDMA and the loaded data and weights from the buffer, generates the scheduling for the convolution operation, and finally sends the parameters to the CMAC.

The CMAC is an array of MAC cells that can parallelize the convolution operation. Once the MAC operations are done, the results are passed on to the CACC. This submodule contains a bank of SRAM blocks to store accumulative sums. Once the operation is finished, the output is truncated to preserve the precision format for which the NVDLA is configured and is stored into a second bank of SRAM blocks. Finally, the output is sent block by block out of the CACC, and so out of the convolution pipeline [41].

- **Activation engine (SDP):**
  This module is responsible for executing the activation functions in a layer, as well as other operations like bias addition and batch normalization. It supports non-linear activation functions like hyperbolic tangent and sigmoid, implemented with LUTs, while supported linear functions are implemented with logic [41].
- **Pooling engine (PDP):**
  In this block, max, min, and average pooling operations are supported. This allows for pooling layers to be executed here [40].
- **Local response normalization (CDP):**
  This module performs operations along the channel direction. More specifically, it can execute the local response normalization operation [41].
- **RUBIK:**
  This block reshapes data in memory and stores it in the new desired format. It can perform contraction, splitting and merging of data into a single block. It is used for operations such as deconvolution [41].
- **Bridge DMA:**
  When a dedicated SRAM for the NVDLA is present, this block enables transferring data between the system DRAM and the dedicated SRAM. It can also move data to and from each memory independently [41].

Each block connects to the CSB master bus to receive control and configuration signals and, except for the Bridge DMA, each block has a small DMA module to interface with memory [41].

2.2.2. Configurability and programmability.

The NVDLA has multiple configurable parameters. It can also be set to contain only a subset of the blocks on Figure 7, depending on the targeted application. Among the most relevant configurable features, there is the data format. It can support multiple precision formats, such as int8, int16, int32, fp16, fp32 and fp64. Moreover, it can have support for several types of convolution, like direct and Winograd convolutions, for weight compression, and for different sizes of the MAC array on CMAC [18].

Once a configuration is chosen, the NVDLA executes inference in a command-execute-interrupt scheme. This means, that first it receives the data to execute one layer of the DNN. That is, the values the configuration registers in all the blocks to be used for that layer must have in order to perform the operations. Then, the NVDLA receives an "activate" signal to begin the operation, and it executes the layer. When it is done, it asserts an interrupt to indicate its completion [18].

2.3. The NVDLA software stack

The NVDLA software environment consists of a compilation build that transforms the deep learning model from a supported high-level framework to a format executable by the NVDLA called the Loadable. The NVDLA also includes two drivers, the User Mode Driver (UMD) and the Kernel Mode Driver (KMD), which take the loadable and drive the execution of the DNN in the NVDLA hardware [32]. The whole stack is shown in Figure 9.

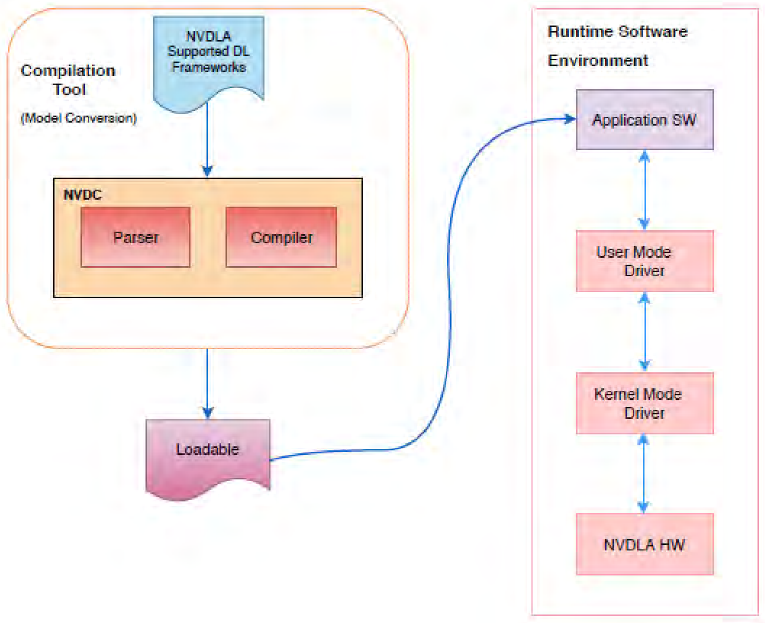


Figure 9. Software stack of the NVDLA [31].

2.3.1. The NVDLA compilation build.

The NVDLA compilation tool, as said before, takes the DNN pre-trained model from a supported high-level framework, which until now it is only Caffe [42], and outputs an executable loadable file. The compilation build is comprised of a parser and a compiler. The parser first takes the Caffe model and creates an intermediate representation of several layers, checking if the model is compatible with the NVDLA [31]. Then, the compiler takes a series of user-defined parameters, such as the specific NVDLA hardware configuration to use, the data type and the calibration file if needed, and maps those software layers from the parser to the actual hardware layers in the NVDLA, in order to create the executable loadable file [31], [43].

2.3.2. The Loadable format.

The Loadable is a binary file format created by NVIDIA for the NVDLA project which contains execution information about the compiled neural network [18]. That includes details about how data is stored and mapped in memory, order of execution of operations and information about the tensors involved in each layer of the neural network [44]. This loadable will then be passed on to the NVDLA drivers for the execution in the hardware itself.

2.3.3. The NVDLA runtime software.

During runtime, the execution of DNN inference in the NVDLA is performed by two drivers provided by the NVDLA project which run over the host operating system. These drivers are the following:

- UMD: this driver can be accessed by the graphical user interface of the host operating system, namely the command line interface. Once it is initialized, it is possible to retrieve some information about the NVDLA HW configuration, load the loadable file and send an inference job with it to the KMD driver. It therefore allows the user to interact with the hardware during runtime execution [32].
- KMD: the kernel mode driver runs on the background, takes the loadable file and schedules the execution of the inference job, configuring the NVDLA internal registers and receiving back the NVLDA signals like the interrupts [32].

Therefore, after the loadable format is acquired for the program to execute on the NVDLA, the UMD will be started, and an inference job will be sent to the KMD.

2.4. VP-FPGA co-simulation

The NVDLA project comes with a co-simulation environment intended for running application software using the NVDLA RTL model in an FPGA together with the rest of components that would be in a SoC, such as a CPU and interconnects, modelled in SystemC/TLM and QEMU.

2.4.1. Co-simulation basics and benefits.

The process of having an RTL model of a given hardware IP being developed that fulfills both the functionality requirements as well as performance demands can take a long time. Having the full SoC that includes that IP block can take even more. Therefore, technologies to have a model of the hardware that focuses only on functionality and leaves out implementation details can be created much faster and allow for early SW development for the target IP [45].

In the co-simulation case used in this work, the NVDLA's RTL model is already developed. Thus, it can be programmed into an FPGA and verified in hardware. However, in order to run a full-stack simulation of the hardware and the software developed for the NVDLA, other components such as interconnects, and a CPU are needed. For this, in the VP-FPGA co-simulation the RTL model is combined with high-level models and emulators of the interconnects and CPU blocks needed to run the application software on the NVDLA. This further leverages the usefulness of the simulation environment, as it now has the following benefits as well:

1. Since the NVDLA runtime software is now ran on the RTL model in the FPGA, it is tested at a level closer to what it would be to execute it in silicon.
2. Both the software and the hardware now can be developed and tested quickly. This is because changes to the runtime software can be rapidly executed in the actual hardware of the NVDLA, and also upon low-level verification, changes in the NVDLA hardware can be tested with real application software as if the whole SoC had been developed already.

Thus, the co-simulation further parallelizes the development of an IP block, since the RTL of the IP does not have to be ready in order to start writing software for it, and similarly the whole SoC does not have to be ready in order to keep developing the IP's RTL.

2.4.2. The SystemC and Transaction Level Modelling.

SystemC is a library written in C++ which enables the high-level modelling of hardware. It allows for the description of several processes, signals, ports and other components in a way that resembles hardware description languages. It is possible then to run event-driven simulations based on the interactions between several processes described in the SystemC language and leaving out several implementation details involved in RTL hardware description [46]. Thus, modelling in SystemC can be done much quicker and can cut simulation time by several orders of magnitude with respect to RTL simulation [47].

However, in order to perform proper functional simulation with systems of high complexity, as an SoC with several parts communicating with different protocols, a standard technique is necessary in order to do this efficiently. Transaction Level Modelling (TLM) is a high-level modelling technique and standard that builds on the SystemC language (though it's not bounded by it) to make it easier to simulate communication between several IP blocks [29]. It hides the complexity of implementation and individual protocols by modelling communication mechanisms as channels between the wrappers of each simulated IP block, and simulating transactions as function calls dependent only on the data and location to and from where it is being transferred [48]. TLM then completes the set of tools required in order to have a functional fast simulation of an SoC system, with several hardware components communicating with each other. This type of simulation is characteristically functionally complete, with loose or approximate timing, fast, available months before the RTL model and desirably accurate enough to stay relevant after the RTL is available [49].

2.4.3. The QEMU CPU emulator.

The Quick Emulator (QEMU) is an open-source machine emulator and virtualizator. It is capable of running emulation of a guest CPU on a host CPU quickly. The way it works is by using dynamic binary translation (DBT), translating continuously the binary executable files (that is, the instructions) intended for the host CPU architecture into binaries executable in the guest CPU, increasing speed execution [27]. By adding a SystemC wrapper with TLM interfaces it can then be integrated to interact with the rest of the components of the virtual platform.

2.5. The AlexNet CNN architecture

AlexNet is a convolutional neural network comprised of five convolutional layers and three fully-connected layers, which also includes three max pooling operations [5]. It contains around 60 million parameters and was primarily intended for image classification [6]. Figure 10 illustrates the network's architecture. The model receives a 224x224 RGB input and in this case it produces an output of size 2, representing 2 possible classes.

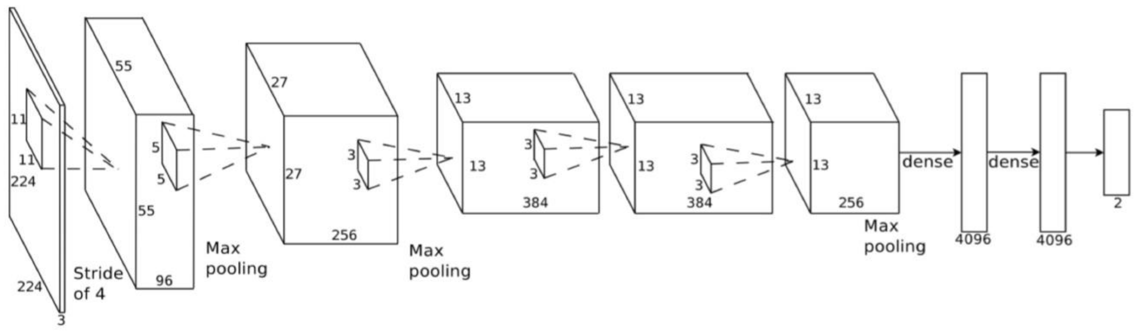


Figure 10. Architecture of the AlexNet CNN [50].

Since AlexNet is a powerful, sizeable model comprised of several operations such as pooling, convolution, matrix multiplication and activation functions, it is a good reference to put the NVDLA system to the test.

Chapter 3: Methodology

The NVDLA is a configurable, flexible hardware project. That means that it has several parameters to be set which will determine its size, power consumption and performance. In order to verify the NVDLA and test it with a real application, it is necessary to first choose a specific configuration. After that, a proper software stack must be implemented that takes the neural network with its input and drives its execution in the NVDLA. On the hardware side, the NVDLA needs both a memory to get the weights from and store the results in, and a host CPU which schedules the operations and drives the execution [31]. Once all that is implemented, the NVDLA can be verified with an established test plan.

3.1. The NVDLA configuration

In this work, we intend to run simulation and testing of real NVDLA hardware together with necessary system IP, employing emulation technology for the CPU. Therefore, our NVDLA parameter selection is intended to just provide the basic hardware in order to complete the aforementioned task.

3.1.1. Configurable parameters and the specification file.

As mentioned before, the NVDLA has several parameters that can be set in order to get different versions of the hardware. This parameters can determine aspects like whether the rendered NVDLA hardware contains entire submodules or leaves them out, whether it can handle a dedicated SRAM memory for increased performance, types of convolutions supported, size of internal buffers, bandwidth capabilities for fetching and storing data and data types of each submodule. The method for setting up the configurable parameters is through a specification file, which is a list of #define statements that represent fixed values.

Table 2. shows some relevant configurable parameters of the NVDLA. For a full list, please see the “nv_small.spec” annex.

Table 2.
Most relevant configurable parameters for the NVDLA for functionality and size.

Parameter (s)	Description	nv_small value (s)
Weight data type	Data type supported for the DNN's weights.	INT8
Feature data type	Data type supported for the DNN's activations.	INT8
Second memory bus, Rubik and BDMA feature support	Whether the NVDLA has a secondary memory interface, the Rubik and BDMA blocks or not.	No, no, no
Atomic-C sizing, Atomic-K sizing	Number of parallel MAC operations in input and output feature channel dimensions. Multiplied, give the number of MACs in the NVDLA.	8, 8
BUFF bank #	Number of banks in the convolutional buffer.	32
BUFF bank size	Size of banks in the convolutional buffer.	4 KB

Note: Taken from "Implementation of a Deep Learning Inference Accelerator on the FPGA Implementation of a Deep Learning Inference Accelerator on the FPGA", Ramakrishnan, 2020 [31].

3.1.2. The nv_small configuration.

This work employs the nv_small configuration. This configuration provides the minimum required NVDLA hardware, and thus functionality in order to perform DNN inference over it. Also, since this work is only concerned with DNN inference, whose real-world application is mostly in mobile and IoT devices, a more portable NVDLA configuration is preferred. Figure 11 provides a block diagram of the nv_small hardware.

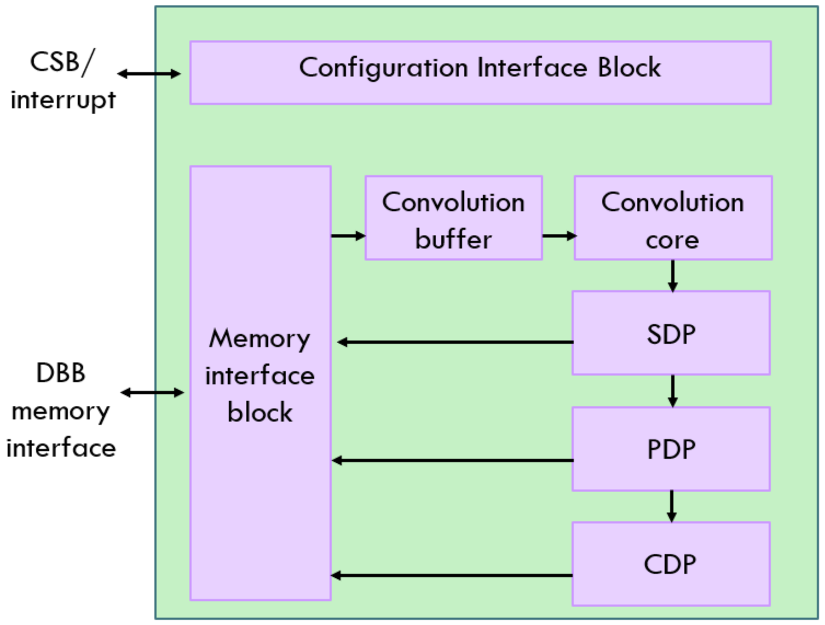


Figure 11. Block diagram nv_small configured hardware.

All this existing blocks are configured according to the specification file detailed in the "nv_small.spec" annex. The omission of the RUBIX, BDMA and second memory interface blocks contribute to saving area and power during execution. Thus, the standard implementation of this DLA in an SoC would be as shown in Figure 12.

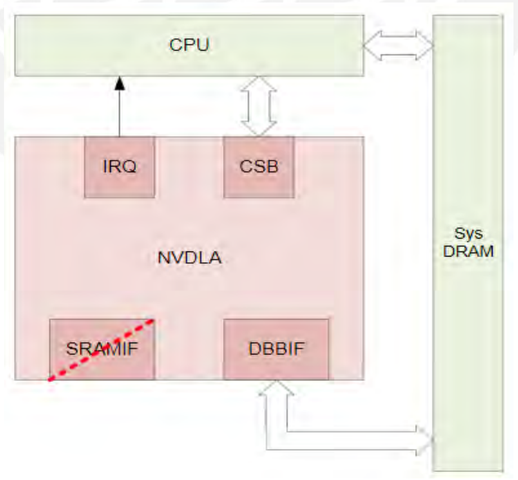


Figure 12. Standard SoC architecture for the nv_small configuration of the NVDLA. It contains only one memory interface for the system DRAM (DBBIF) and one CPU to drive the execution [18].

Furthermore, the nv_small configuration files already set the NVDLA to run at a clock frequency of 100 MHz. Once the usage of the nv_small configuration is decided, appropriate tests can be designated for it.

3.2. The NVDLA AWS Hybrid Platform

As detailed in Figure 12, in order to run an application in the NVDLA configured as nv_small, external IP such as a CPU, system DRAM and an interconnect is needed. The NVDLA provides a virtual plattform that uses the QEMU CPU emulator and several other SystemC/TLM components that can be used to send instructions to the NVDLA real hardware and drive its execution, through a TLM/PCIe converter. In this case, the NVDLA virtual plattform integrated into the AWS Elastic Compute Cloud (AWS EC2) web service is used, given that the AWS service allows for the remote use of FPGA instances that can fit well the NVDLA nv_small hardware, as explained below.

3.2.1. AWS EC2 F1 instances.

AWS EC2 is a web service that allows for remote interaction, programmability and job execution on real hardware instances (called F1) in the cloud. In this case, the f1.2xlarge instances are employed. They contain an 8-core CPU, together with 30 GiB of storage and an FPGA configurable logic block [1]. The FPGA is based on the Xilinx Virtex-7 Ultrascale+ VU9P FPGAs [51]. From [31], the NVDLA nv_small together with system IP has a resource utilization of 78437 LUTs and 85266 FFs at the post-implementation stage. The Virtex-7 VU9P FPGA contains more than a million LUTs and more than 2 million FFs [52]. Therefore, the hardware provided by the f1.2xlarge instances is more than sufficient for this application. The remote connection to the F1 instance is done via a SSH connection and then it’s possible to interface with the hardware through the CLI.

3.2.2. The NVDLA virtual platform on AWS.

The NVDLA project provides tools to integrate their virtual plattform with the AWS EC2 service. The full stack from application to hardware is shown in Figure 13.

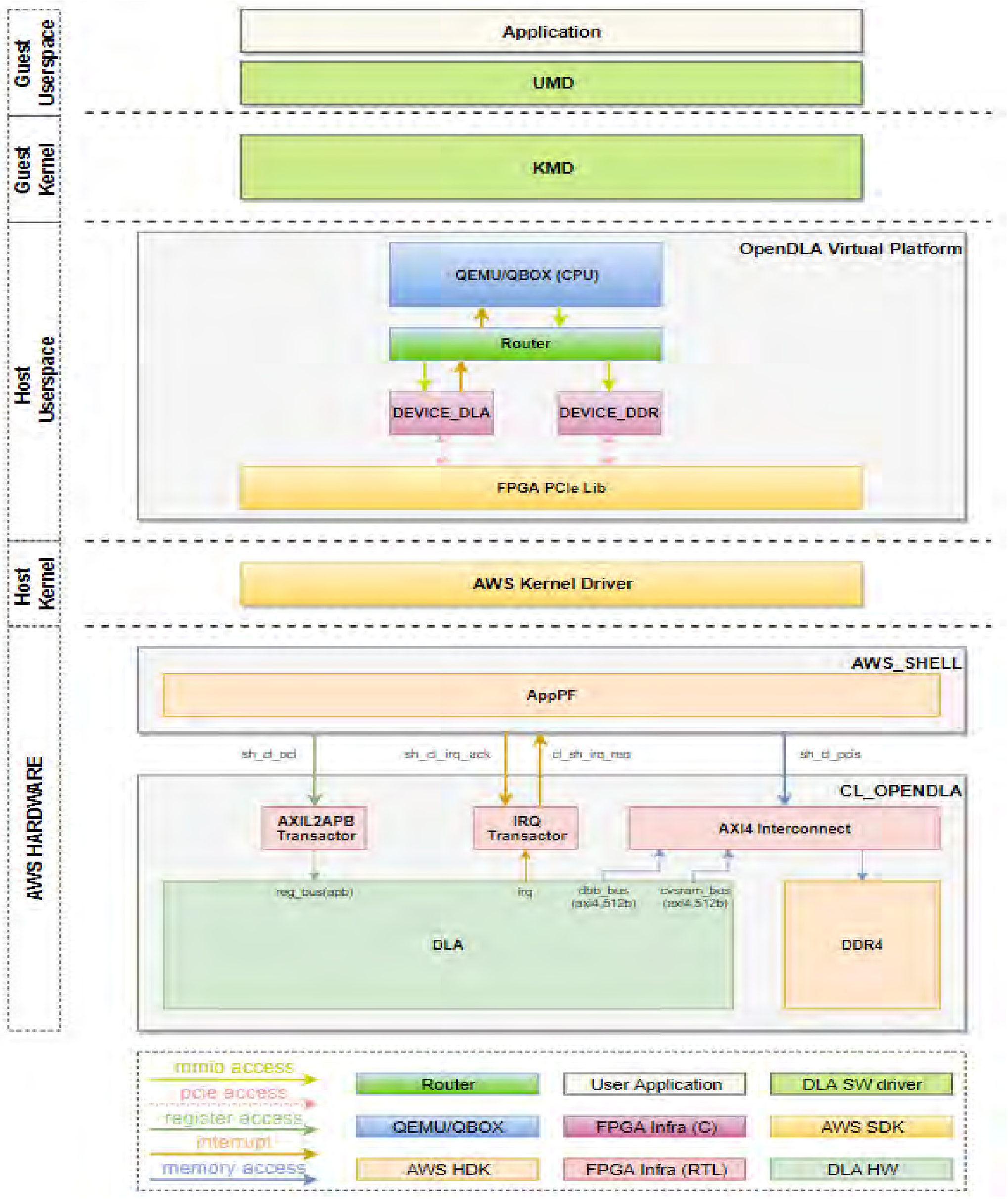


Figure 13. Full HW/SW stack of the NVDLA virtual plattform integrated with the AWS EC2 service [53].

On the top of the stack lies the Guest Userspace. This basically consists of the high-level application, that is, the CLI we are running together with the compilation tools and DNN we intend to run on the HW, and also the UMD, which allows us to send jobs to the KMD and receive logs. The Guest Kernel is simply the KMD. Below it, the Host Userspace follows. This contains the totality of the NVDLA virtual plattform. The guest CPU runs on the QEMU emulator, which in this case is the 64-bit Armv8 architecture, with a SystemC TLM wrapper. Then, there are the native SystemC components such as the router, the NVDLA (DEVICE_DLA) and DRAM (DEVICE_DDR), which interface with each other by means of register accesses and memory-mapped input/output. AWS SDK then provides the FPGA PCIe Lib SystemC module, that integrates the virtual plattform with the greater AWS environment.

The host kernel in the host CPU, that is, the 8-core CPU of the f1.2xlarge instance, contains the AWS Kernel Driver from AWS itself. Finally, the latter drives all the signals into de AWS Hardware. There, a similar setup as in Figure 12 is noticeable. The DLA block is the NVDLA (nv_small configuration) hardware implemented on the FPGA. Attached to its interfaces are a series of system IP, such as the AXIL2APB and IRQ protocol translators, and an AXI interconnect that allows the NVDLA to access the DRAM (DDR4 block). This whole setup then allows for the VP-FPGA co-simulation of actual NVDLA hardware configured on the FPGA together with the VP running on the AWS virtual environment.

With the full stack, it is now feasible to run a high-level application on the NVDLA hardware, using a mixture of real and virtualized hardware. In order to configure this setup, to components are required in the AWS plattform:

- Amazon Machine Image (AMI): the AMI is a software template that has the information required to launch an AWS EC2 instance. It also contains a stack of software from the operating system to packages and applications. In this case, the AMI nvdla_vp_fpga_ami_ubuntu from the NVDLA project is used, which contains Ubuntu 14.04 and all other environment packages and libraries to run the NVDLA virtual plattform [53].
- Amazon FPGA Image (AFI): this template contains the rendered hardware for the NVDLA nv_small configuration and must be registered in the AWS plattform. For this thesis, the AFI nv_small: agfi-05d68b424ef03f66e is used, which is also registered in AWS as part of the NVDLA project [54].

3.3. Verification and testing

Once the NVDLA virtual plattform is set up and running in AWS, it is possible to run a series of tests to verify the NVDLA at a system level. The first step is to run hardware regression tests to check the proper configuration and operation of the NVDLA nv_small architecture on the FPGA. Subsequently, software sanity tests are executed in the VP in order to check the correct functionality of the whole stack, from application software to hardware. Finally, the last step is to attempt inference on the NVDLA.

3.3.1. AlexNet execution.

The AlexNet model is chosen as the target DNN for inference because of its relevance in several aspects. First, as mentioned in previous chapters, it is a sizable network which contains a variety of deep learning operations typically present in other DNN architectures. Therefore, it puts to the test most of the NVDLA hardware, such as the convolution, SDP and PDP blocks, as well as the management of a large amount of memory accesses. Second, AlexNet is the first model to achieve a fair amount of accuracy in real world scenarios, as outlines in Chapter 1. Therefore, successfully running it means that, functionally, the NVDLA hardware and its runtime software are ready for real world, demanding applications.

In this thesis, the AlexNet file executed is the loadable from the annex "NN_L0_1_small_fbuf", which contains a pre-compiled AlexNet network together with its input embedded in the file.

Chapter 4: Results

In order to run the VP-FPGA co-simulation of the NVDLA, a testing procedure is followed. First, the NVDLA hardware itself is first verified through a series of testbenches. Then, the virtual plattform with the SystemC/TLM components is tested with sanity tests that check for the correct behavior of every functional block in the nv_small hardware, the CPU emulator and the other described components of the system. Finally, the AlexNet CNN is executed in the virtual platform in order to test the system as a whole running the intended application: a complete neural network. The steps followed to setup the NVDLA AWS VP were 2.1 through 2.5 from the "Virtual Platform On AWS FPGA" guide form the NVDLA documentation [53].

4.1. Hardware regression tests on the FPGA

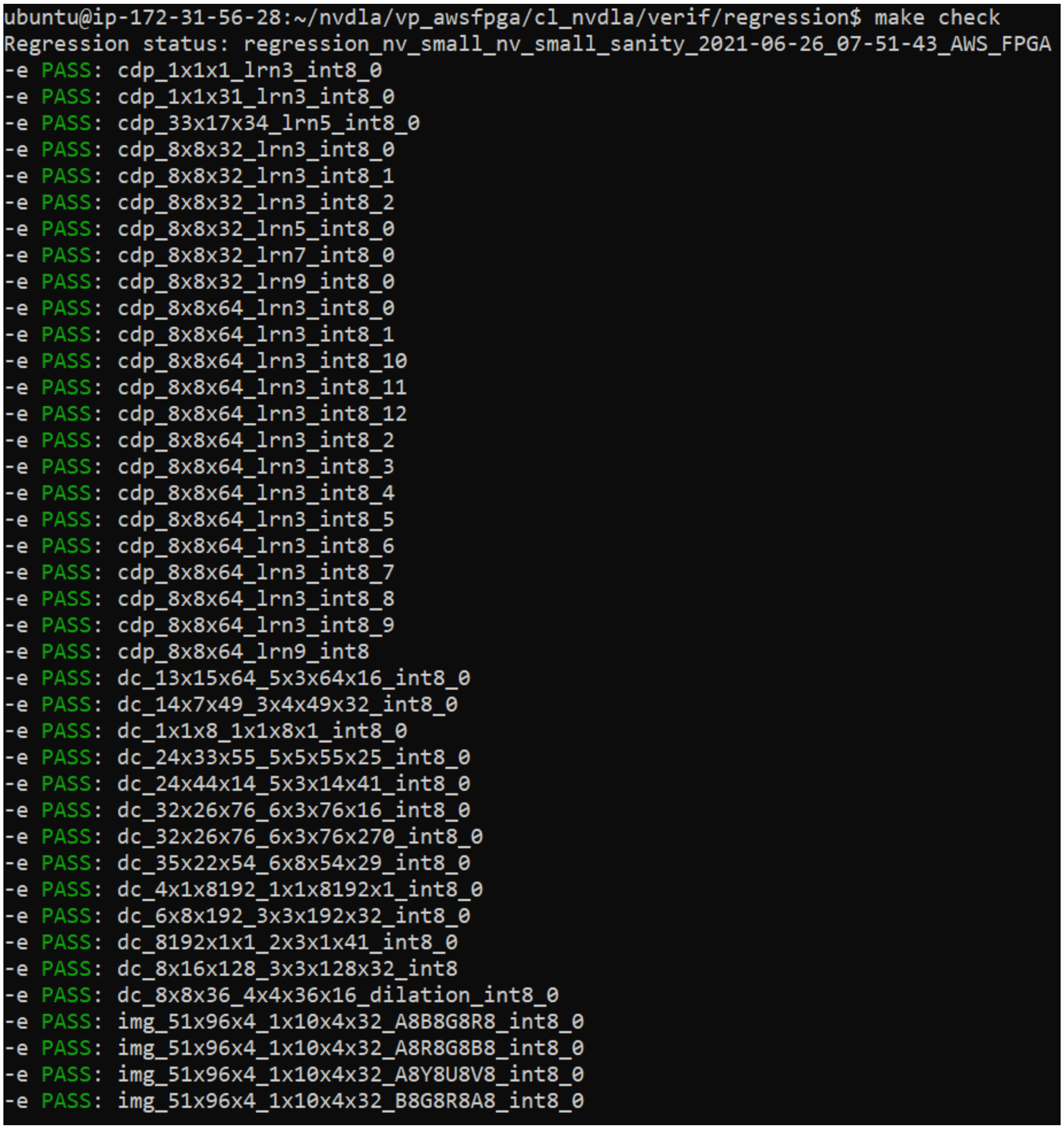
The NVDLA project contains a regression tool set with a series of direct tests in trace format that can be found at *vp_awsfpga/cl_nvdla/verif/regression/nv_small_sanity* in the nvdla/vp_awsfpga repository [54]. These regressions verify the nv_small hardware at the levels described in Table 3 below.

Table 3.
Test levels of regressions ran on the NVDLA hardware.

Level	Functionality
L0	Read and write from memory.
L1	Common function cases, testing main data paths and simulating a single layer of a DNN.
L2	Corner cases such as minimum and maximum tensor sizes (1x1x1 and 8192x1x1 in width, height and depth respectively).

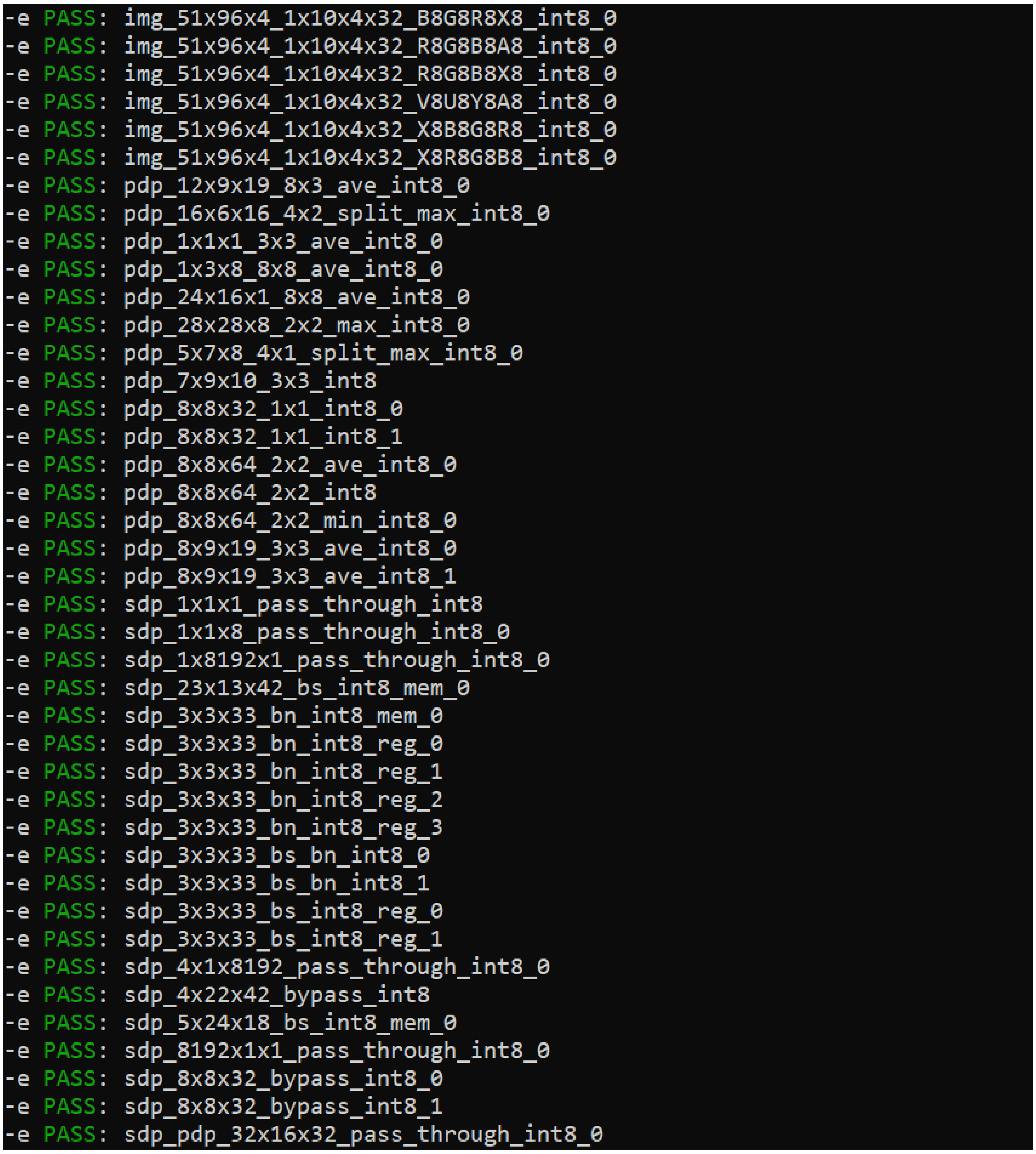
Note: Taken from "NVDLA Verification Suite User Guide", Nvidia Corporation, 2018 [55].

To run these regressions, the step 3.5.1 from [53] were followed. The tests, their individual results and traces of each test can be found in the annex “hw_regressions_nv_small”. Figures 14 and 15 show the results of running the *make check* command. Its log can be found in the annex “hw_regressions_make_check.log”.



```
ubuntu@ip-172-31-56-28:~/nvdla/vp_awsfpga/cl_nvdla/verif/regression$ make check
Regression status: regression_nv_small_nv_small_sanity_2021-06-26_07-51-43_AWS_FPGA
-e PASS: cdp_1x1x1_lrn3_int8_0
-e PASS: cdp_1x1x31_lrn3_int8_0
-e PASS: cdp_33x17x34_lrn5_int8_0
-e PASS: cdp_8x8x32_lrn3_int8_0
-e PASS: cdp_8x8x32_lrn3_int8_1
-e PASS: cdp_8x8x32_lrn3_int8_2
-e PASS: cdp_8x8x32_lrn5_int8_0
-e PASS: cdp_8x8x32_lrn7_int8_0
-e PASS: cdp_8x8x32_lrn9_int8_0
-e PASS: cdp_8x8x64_lrn3_int8_0
-e PASS: cdp_8x8x64_lrn3_int8_1
-e PASS: cdp_8x8x64_lrn3_int8_10
-e PASS: cdp_8x8x64_lrn3_int8_11
-e PASS: cdp_8x8x64_lrn3_int8_12
-e PASS: cdp_8x8x64_lrn3_int8_2
-e PASS: cdp_8x8x64_lrn3_int8_3
-e PASS: cdp_8x8x64_lrn3_int8_4
-e PASS: cdp_8x8x64_lrn3_int8_5
-e PASS: cdp_8x8x64_lrn3_int8_6
-e PASS: cdp_8x8x64_lrn3_int8_7
-e PASS: cdp_8x8x64_lrn3_int8_8
-e PASS: cdp_8x8x64_lrn3_int8_9
-e PASS: cdp_8x8x64_lrn9_int8
-e PASS: dc_13x15x64_5x3x64x16_int8_0
-e PASS: dc_14x7x49_3x4x49x32_int8_0
-e PASS: dc_1x1x8_1x1x8x1_int8_0
-e PASS: dc_24x33x55_5x5x55x25_int8_0
-e PASS: dc_24x44x14_5x3x14x41_int8_0
-e PASS: dc_32x26x76_6x3x76x16_int8_0
-e PASS: dc_32x26x76_6x3x76x270_int8_0
-e PASS: dc_35x22x54_6x8x54x29_int8_0
-e PASS: dc_4x1x8192_1x1x8192x1_int8_0
-e PASS: dc_6x8x192_3x3x192x32_int8_0
-e PASS: dc_8192x1x1_2x3x1x41_int8_0
-e PASS: dc_8x16x128_3x3x128x32_int8
-e PASS: dc_8x8x36_4x4x36x16_dilation_int8_0
-e PASS: img_51x96x4_1x10x4x32_A8B8G8R8_int8_0
-e PASS: img_51x96x4_1x10x4x32_A8R8G8B8_int8_0
-e PASS: img_51x96x4_1x10x4x32_A8Y8U8V8_int8_0
-e PASS: img_51x96x4_1x10x4x32_B8G8R8A8_int8_0
```

Figure 14. First batch of results from the regression tests.



```
-e PASS: img_51x96x4_1x10x4x32_B8G8R8X8_int8_0
-e PASS: img_51x96x4_1x10x4x32_R8G8B8A8_int8_0
-e PASS: img_51x96x4_1x10x4x32_R8G8B8X8_int8_0
-e PASS: img_51x96x4_1x10x4x32_V8U8Y8A8_int8_0
-e PASS: img_51x96x4_1x10x4x32_X8B8G8R8_int8_0
-e PASS: img_51x96x4_1x10x4x32_X8R8G8B8_int8_0
-e PASS: pdp_12x9x19_8x3_ave_int8_0
-e PASS: pdp_16x6x16_4x2_split_max_int8_0
-e PASS: pdp_1x1x1_3x3_ave_int8_0
-e PASS: pdp_1x3x8_8x8_ave_int8_0
-e PASS: pdp_24x16x1_8x8_ave_int8_0
-e PASS: pdp_28x28x8_2x2_max_int8_0
-e PASS: pdp_5x7x8_4x1_split_max_int8_0
-e PASS: pdp_7x9x10_3x3_int8
-e PASS: pdp_8x8x32_1x1_int8_0
-e PASS: pdp_8x8x32_1x1_int8_1
-e PASS: pdp_8x8x64_2x2_ave_int8_0
-e PASS: pdp_8x8x64_2x2_int8
-e PASS: pdp_8x8x64_2x2_min_int8_0
-e PASS: pdp_8x9x19_3x3_ave_int8_0
-e PASS: pdp_8x9x19_3x3_ave_int8_1
-e PASS: sdp_1x1x1_pass_through_int8
-e PASS: sdp_1x1x8_pass_through_int8_0
-e PASS: sdp_1x8192x1_pass_through_int8_0
-e PASS: sdp_23x13x42_bs_int8_mem_0
-e PASS: sdp_3x3x33_bn_int8_mem_0
-e PASS: sdp_3x3x33_bn_int8_reg_0
-e PASS: sdp_3x3x33_bn_int8_reg_1
-e PASS: sdp_3x3x33_bn_int8_reg_2
-e PASS: sdp_3x3x33_bn_int8_reg_3
-e PASS: sdp_3x3x33_bs_bn_int8_0
-e PASS: sdp_3x3x33_bs_bn_int8_1
-e PASS: sdp_3x3x33_bs_int8_reg_0
-e PASS: sdp_3x3x33_bs_int8_reg_1
-e PASS: sdp_4x1x8192_pass_through_int8_0
-e PASS: sdp_4x22x42_bypass_int8
-e PASS: sdp_5x24x18_bs_int8_mem_0
-e PASS: sdp_8192x1x1_pass_through_int8_0
-e PASS: sdp_8x8x32_bypass_int8_0
-e PASS: sdp_8x8x32_bypass_int8_1
-e PASS: sdp_pdp_32x16x32_pass_through_int8_0
```

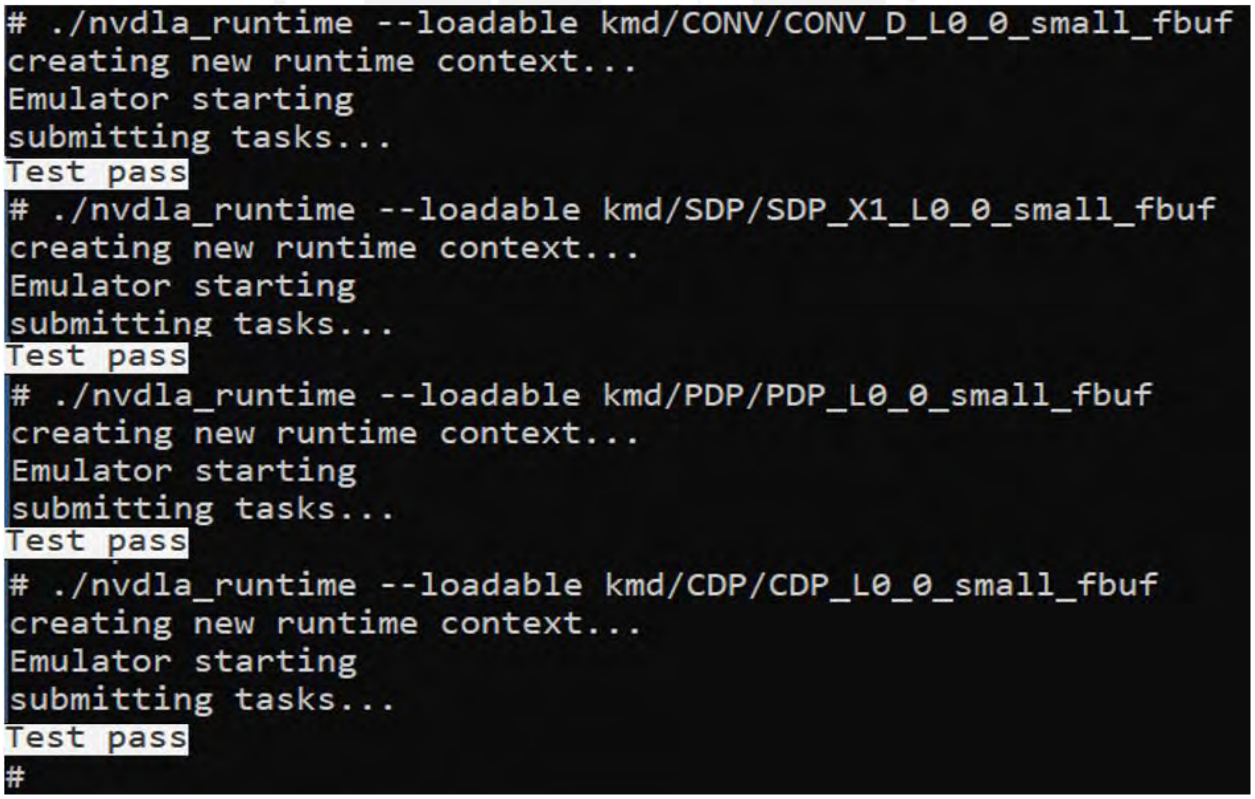
Figure 15. Second batch of results from the regression tests.

It is noticeable that every test ran gave a PASS result, indicating that the NVDLA hardware has been verified.

4.2. Virtual platform sanity tests

The sanity tests in the virtual platform run on the whole stack of the system, from the application software all the way to the emulated CPU and the NVDLA hardware on the FPGA. Thus, they test the hardware, the SystemC components and the NVDLA drivers. There is one test per functional unit of the NVDLA nv_small configuration, which are the CONV, SDP, PDP and CDP blocks.

These tests are precompiled loadables that contain internally the input data and the instructions to be executed at each functional block. Once the file is executed, an output is generated, over which an MD5 checksum value is produced. If this value coincides with the golden data from the respective test, the result is a “Test pass”. Figure 16 shows the outcome of running these sanity tests on the NVDLA virtual platform. The full results are in the annex “sw_sanity_tests_execution.log”.



```
# ./nvdla_runtime --loadable kmd/CONV/CONV_D_L0_0_small_fbuf
creating new runtime context...
Emulator starting
submitting tasks...
Test pass
# ./nvdla_runtime --loadable kmd/SDP/SDP_X1_L0_0_small_fbuf
creating new runtime context...
Emulator starting
submitting tasks...
Test pass
# ./nvdla_runtime --loadable kmd/PDP/PDP_L0_0_small_fbuf
creating new runtime context...
Emulator starting
submitting tasks...
Test pass
# ./nvdla_runtime --loadable kmd/CDP/CDP_L0_0_small_fbuf
creating new runtime context...
Emulator starting
submitting tasks...
Test pass
#
```

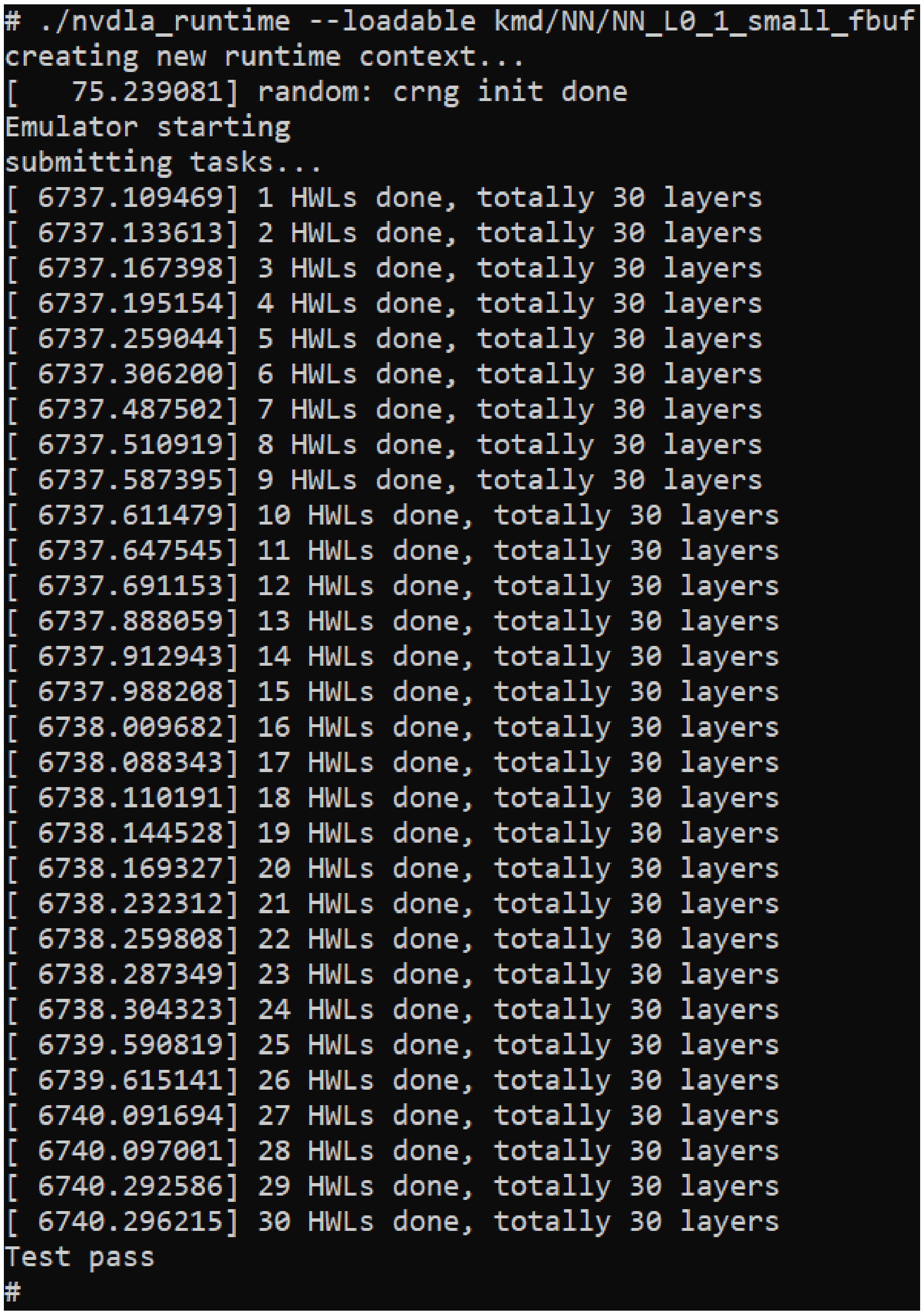
Figure 16. Passed results for each sanity test ran, verifying the functionality of running a loadable from the user application all the way to the nv_small hardware on each block (only the essential part of the log is shown to make it more understandable).

4.3. AlexNet execution on the hybrid platform

As mentioned in Chapter 3, the AlexNet loadable file used was the one in the annex "NN_L0_1_small_fbuf". This loadable contains the instructions as well as the input data to the network. Once the execution was done, similarly to the sanity tests, a MD5 checksum value was calculated over the output and compared to a golden data value. If these two matched, the AlexNet DNN would have been correctly executed by the NVDLA on the virtual platform. The command to send the loadable to run on the NVDLA was simply:

```
./nvdla_runtime --loadable NN_L0_1_small_fbuf
```

Its processing in the hybrid platform took around 112 minutes and gave a successful outcome, as can be seen in Figure 17. The log file can also be found in the annex "alexnet_execution.log".



```
# ./nvdla_runtime --loadable kmd/NN/NN_L0_1_small_fbuf
creating new runtime context...
[   75.239081] random: crng init done
Emulator starting
submitting tasks...
[ 6737.109469] 1 HWLs done, totally 30 layers
[ 6737.133613] 2 HWLs done, totally 30 layers
[ 6737.167398] 3 HWLs done, totally 30 layers
[ 6737.195154] 4 HWLs done, totally 30 layers
[ 6737.259044] 5 HWLs done, totally 30 layers
[ 6737.306200] 6 HWLs done, totally 30 layers
[ 6737.487502] 7 HWLs done, totally 30 layers
[ 6737.510919] 8 HWLs done, totally 30 layers
[ 6737.587395] 9 HWLs done, totally 30 layers
[ 6737.611479] 10 HWLs done, totally 30 layers
[ 6737.647545] 11 HWLs done, totally 30 layers
[ 6737.691153] 12 HWLs done, totally 30 layers
[ 6737.888059] 13 HWLs done, totally 30 layers
[ 6737.912943] 14 HWLs done, totally 30 layers
[ 6737.988208] 15 HWLs done, totally 30 layers
[ 6738.009682] 16 HWLs done, totally 30 layers
[ 6738.088343] 17 HWLs done, totally 30 layers
[ 6738.110191] 18 HWLs done, totally 30 layers
[ 6738.144528] 19 HWLs done, totally 30 layers
[ 6738.169327] 20 HWLs done, totally 30 layers
[ 6738.232312] 21 HWLs done, totally 30 layers
[ 6738.259808] 22 HWLs done, totally 30 layers
[ 6738.287349] 23 HWLs done, totally 30 layers
[ 6738.304323] 24 HWLs done, totally 30 layers
[ 6739.590819] 25 HWLs done, totally 30 layers
[ 6739.615141] 26 HWLs done, totally 30 layers
[ 6740.091694] 27 HWLs done, totally 30 layers
[ 6740.097001] 28 HWLs done, totally 30 layers
[ 6740.292586] 29 HWLs done, totally 30 layers
[ 6740.296215] 30 HWLs done, totally 30 layers
Test pass
#
```

Figure 17. Results of running AlexNet on the NVDLA AWS virtual platform, using the nv_small configuration (only relevant parts of the log are shown so as to increase readability).

The log shows that the NVDLA runtime is dividing the execution of the 8 layers of the AlexNet DNN into 30 different layers. Ultimately, the neural network is computed successfully, demonstrating the capabilities of the system to execute DNN workloads.

Conclusions

In this thesis, the NVDLA was implemented on a hybrid platform using the F1 instances provided by the EC2 service from AWS, with the main objective of running inference of a DNN in a co-simulation. The configuration chosen for the NVDLA was the more portable nv_small, due to the fact that it is the most suitable one for inference on the emerging mobile and IoT technologies and therefore was considered of greater relevance. The hardware/software co-simulation of the system involved multiple technologies, such as actual hardware in the case of RAM memory and the NVDLA, the QEMU CPU emulator and a SystemC TLM/PCIe converter, as well as the Linux Ubuntu OS and the NVDLA drivers.

As a consequence, the complexity of the project required verification and testing at several layers of the hardware-software full stack. From the testbenches ran on the NVDLA hardware, it can be concluded that the nv_small RTL was properly installed and configured in the FPGA on the F1 instance of the AWS EC2 service, having passed every verification test.

Furthermore, the successful execution of the sanity test loadables on the virtual platform confirmed the proper functionality of the hardware-software co-simulation system running on AWS. This means that the emulated Guest ARM CPU was able to fetch the instructions and data from the RAM memory and execute them in the NVDLA hardware as driven and scheduled by the UMD and KMD drivers. Finally, a full neural network comprising a multiplicity of deep learning layers and operations, the AlexNet CNN, was properly executed by the NVDLA in its AWS virtual platform environment.

It is also worth mentioning that the execution of AlexNet on the virtual platform took around 112 minutes to complete, which is quite quick for a fully functional co-simulation running a heavy runtime application that enables testing the system’s functionality.

Finally, it is important to highlight the importance of virtual prototypes on the SoC design flow since it enables early software development while the hardware is still being designed. At the same time, that allows for changes on the SoC RTL in the early phases of the design process. It is noteworthy then, that the NVDLA platform allows for the quick co-simulation of hardware and software, while using the NVDLA’s RTL design combined with the rest of the components using virtual-prototype models. This goes in contrast to the virtual prototype solutions showed in Chapter 1, where either both the DLA and the rest of the IP had to be in RTL or hard-IP (already manufactured blocks, as in PYNQ), or they all had to be in SystemC/TLM models. Therefore, this gives added value to the NVDLA platform, since the specific component of interest, the DLA, can be at an advanced stage for real testing, while the rest of the components can be just virtual-prototype models, which are faster and easier to implement than RTL.

Recommendations And Future Work

As a recommendation, it would be highly beneficial to set up the GNU Project Debugger (GDB) [56] in order to be able to debug in the virtual platform. The GDB tool would add visibility into the inner workings of the VP-FPGA co-simulation, thus creating an environment more prepared to run more complex applications such as different DNNs or other workloads.

Furthermore, the 112-minute co-simulation of the AlexNet execution in the NVDLA shows its proper functionality when running a full DNN. Future work could include working with a cycle-accurate virtual environment, such as FireSim [57] in order to also test the performance of the NVDLA in terms of computational speed. FireSim can also run with the F1 instances of the EC2 AWS service, making it a natural next step for building on this work, and being able to test the NVDLA capabilities beyond just functionality and into performance.

References

[1] Amazon Web Services, “Amazon EC2 F1 Instances,” 2021. https://aws.amazon.com/ec2/instance-types/f1/ (accessed May 22, 2021).

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA: MIT Press, 2016.

[3] S. Han, “Efficient Methods and Hardware for Deep Learning,” Stanford University, 2017.

[4] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778, [Online]. Available: https://ieeexplore.ieee.org/document/7780459.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” *Adv. Neural Inf. Process. Syst. 25*, pp. 1097–1105, 2012, [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[6] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, “Hardware for Machine Learning: Challenges and Opportunities,” in *IEEE Custom Integrated Circuits Conference (CICC)*, 2017, pp. 1–8, doi: doi: 10.1109/CICC.2017.7993626.

[7] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *3rd Int. Conf. Learn. Represent. ICLR 2015 - Conf. Track Proc.*, pp. 1–14, 2015.

[8] S.-C. Lin *et al.*, “The Architectural Implications of Autonomous Driving: Constraints and Acceleration,” *ASPLOS*, vol. 18, 2018, doi: 10.1145/3173162.3173191.

[9] J. Redmon and A. Farhadi, “YOLO9000: Better, Faster, Stronger,” 2017, doi: 10.1109/CVPR.2017.690.

[10] R. Huang, J. Pedoeem, and C. Chen, “YOLO-LITE: A Real-Time Object Detection Algorithm Optimized for Non-GPU Computers.”

[11] A. Bartolo and W. Hwang, “Domain-Specific Accelerator Design & Profiling for Deep

Learning Applications.” Accessed: Sep. 29, 2020. [Online]. Available: https://stanford.edu/~bartolo/assets/dsa-design-dl.pdf.

[12] McKinsey & Company, “McKinsey on Semiconductors: Creating value, pursuing innovation, and optimizing operations,” no. 7. McKinsey & Company, 2019.

[13] X. Li, G. Zhang, K. Li, and W. Zheng, “Deep Learning and its Parallelization,” in *Big Data*, Elsevier, 2016, pp. 95–118.

[14] I. Bratt, J. Brothers, and F. Sijstermans, “Machine Learning I.” Hot Chips: A Symposium on High Performance Chips, Cupertino, CA, 2018, [Online]. Available: https://www.hotchips.org/archives/2010s/hc30/.

[15] Arm, “Arm Ethos-N78 Processor Highly scalable and efficient second-generation Machine Learning processor.” Arm, 2020, [Online]. Available: https://armkeil.blob.core.windows.net/developer/Files/pdf/ML on Arm/arm-ethos-n78-product-brief.pdf.

[16] A. Lavin and S. Gray, “Fast Algorithms for Convolutional Neural Networks,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 4013–4021, doi: 10.1109/CVPR.2016.435.

[17] Nvidia Corporation, “NVDLA,” 2018. http://nvdla.org/ (accessed Oct. 14, 2020).

[18] Nvidia Corporation, “NVDLA Primer — NVDLA Documentation,” 2018. http://nvdla.org/primer.html (accessed Sep. 22, 2020).

[19] E. C. Culau, G. C. Marchesan, N. R. Weirich, and L. L. de Oliveira, “An Efficient Single Core Flexible Processor Architecture for 4096-bit Montgomery Modular Multiplication and Exponentiation,” in *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2018, pp. 1–5, doi: 10.1109/ISCAS.2018.8351190.

[20] Cadence Design Systems; Leibniz Universität Hannover, “Cadence Tensilica Product Overview,” no. September. Hannover, 2019, [Online]. Available: https://www.ims.uni-hannover.de/fileadmin/ims/aktivitaeten/Tensilica_Day/2019/1_td19_cadence.pdf.

[21] Cadence Design Systems, “Xtensa® Instruction Set Architecture (ISA) Summary.” San

Jose, pp. 31–36, 2022, [Online]. Available: https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/ip/tensilica-ip/isa-summary.pdf.

[22] Synopsys, “Virtual Prototyping Models,” 2023. https://www.synopsys.com/verification/virtual-prototyping/virtual-prototyping-models.html.

[23] Synopsys, “DesignWare TLM Library.” 2018.

[24] Synopsys, “SystemC TLM Models,” 2023. https://www.synopsys.com/verification/virtual-prototyping/virtual-prototyping-models/systemc-tlm-models.html.

[25] Advanced Micro Devices, “PYNQ PYNQ: Python productivity for Adaptive Computing platforms.” 2022, [Online]. Available: https://pynq.readthedocs.io/en/latest/index.html#.

[26] Advanced Micro Devices, “PYNQ Overlays.” 2022, [Online]. Available: https://pynq.readthedocs.io/en/latest/pynq_overlays.html.

[27] F. Bellard, “QEMU, a Fast and Portable Dynamic Translator.”

[28] “IEEE Standard for Standard SystemC Language Reference Manual,” *IEEE Std 1666-2011 (Revision IEEE Std 1666-2005)*, pp. 1–638, 2012, doi: 10.1109/IEEESTD.2012.6134619.

[29] Open SystemC Initiative, “OSCI TLM-2.0 LANGUAGE REFERENCE MANUAL,” 2009.

[30] Nvidia Corporation, “nvdla repository.” NVIDIA, Accessed: Sep. 22, 2020. [Online]. Available: https://github.com/nvdla.

[31] S. Ramakrishnan, “Implementation of a Deep Learning Inference Accelerator on the FPGA Implementation of a Deep Learning Inference Accelerator on the FPGA,” Lund University, 2020.

[32] NVIDIA, “Software Manual — NVDLA Documentation,” 2018. http://nvdla.org/sw/contents.html (accessed May 23, 2021).

[33] V. Sze, Y.-H. Chen, T.-J. Yang, and J. Emer, “Efficient Processing of Deep Neural Networks : A Tutorial and Survey,” 2017, [Online]. Available: https://arxiv.org/pdf/1703.09039.pdf.

[34] A. Ram and C. C. Reyes-aldasoro, “The relationship between Fully Connected Layers and number of classes for the analysis of retinal images,” 2020, [Online]. Available: https://arxiv.org/pdf/2004.03624.pdf.

[35] J. Hany and G. Walters, “Best Practices for Model Design and Training,” in *Hands-On Generative Adversarial Networks with PyTorch 1.x: Implement next-generation neural networks to build powerful GAN models using Python*, Birmingham: Packt Publishing Ltd., 2019, p. 58.

[36] V. Dumoulin and F. Visin, “A guide to convolution arithmetic for deep learning,” pp. 1–31, 2016, [Online]. Available: http://arxiv.org/abs/1603.07285.

[37] K. Chakraborty, S. Bhattacharyya, R. Bag, and A. A. Hassanien, “Sentiment Analysis on a Set of Movie Reviews Using Deep Learning Techniques,” in *Social Network Analytics: Computational Research Methods and Techniques*, 2019, pp. 127–147.

[38] A. Hosseinzadeh Namin, K. Leboeuf, R. Muscedere, H. Wu, and M. Ahmadi, “Efficient hardware implementation of the hyperbolic tangent sigmoid function,” 2009, doi: 10.1109/ISCAS.2009.5118213.

[39] A. F. M. Agarap, “Deep Learning using Rectified Linear Units (ReLU),” 2019, [Online]. Available: https://arxiv.org/pdf/1803.08375.pdf.

[40] Nvidia Corporation, “Hardware Architectural Specification — NVDLA Documentation,” 2018. http://nvdla.org/hw/v1/hwarch.html (accessed Sep. 22, 2020).

[41] Nvidia Corporation, “Unit Description,” 2018. http://nvdla.org/hw/v1/ias/unit_description.html.

[42] Y. Jia *et al.*, “Caffe: Convolutional architecture for fast feature embedding,” in *MM 2014 - Proceedings of the 2014 ACM Conference on Multimedia*, Nov. 2014, pp. 675–678, doi: 10.1145/2647868.2654889.

[43] P. NVIDIA; Gaikwad, “sw/LowPrecision.md at master · nvdla/sw,” 2019. https://github.com/nvdla/sw/blob/master/LowPrecision.md (accessed Jul. 05, 2021).

[44] Skymizer, “Porting ONNC to NVDLA,” 2020. https://www.youtube.com/watch?v=bOfRYYIAVwA&t=388s (accessed May 22, 2021).

[45] D. G. Bailey, “The advantages and limitations of high level synthesis for FPGA based image processing,” in *ACM International Conference Proceeding Series*, Sep. 2015, vol. 08-11-Sep-2015, pp. 134–139, doi: 10.1145/2789116.2789145.

[46] Electronic Systems group, “Version 2.0 User’s Guide Update for SystemC 2.0.1 SystemC 2.0 User’s Guide iii,” 1996. Accessed: Jul. 05, 2021. [Online]. Available: http://www.systemc.org.

[47] Microsemi; LegUp Computing Inc., “Benefits of High-Level Synthesis for FPGA Design,” 2020. Accessed: Jul. 05, 2021. [Online]. Available: www.legupcomputing.com.

[48] F. Doucet, R. K. Shyamasundar, I. H. Krüger, S. Joshi, and R. K. Gupta, “Reactivity in SystemC Transaction-Level Models.”

[49] J. Aynsley, “What is TLM-2.0?,” 2009. https://www.youtube.com/watch?v=ocniBsPNRwk&t=173s (accessed Jul. 05, 2021).

[50] A. Pedraza, J. Gallego, S. Lopez, L. Gonzalez, A. Laurinavicius, and G. Bueno, “Glomerulus classification with convolutional neural networks,” in *Communications in Computer and Information Science*, 2017, vol. 723, pp. 839–849, doi: 10.1007/978-3-319-60964-5_73.

[51] Amazon Web Services, “Acceleration in the AWS Cloud,” 2021. https://www.xilinx.com/products/design-tools/acceleration-zone/aws.html (accessed May 22, 2021).

[52] Xilinx, “UltraScale+ FPGA Product Tables and Product Selection Guide,” 2015.

[53] NVIDIA, “Virtual Platform On AWS FPGA — NVDLA Documentation,” 2018. http://nvdla.org/vp_fpga.html#setup-aws-ec2-instance-machine (accessed May 23, 2021).

[54] NVIDIA, “vp_awsfpga/README.md at master · nvdla/vp_awsfpga · GitHub,” 2018.

https://github.com/nvdla/vp_awsfpga/blob/master/README.md (accessed May 23, 2021).

[55] NVDLA, “NVDLA Verification Suite User Guide — NVDLA Documentation,” 2018. http://nvdla.org/hw/v2/verif_guide.html#preparation (accessed May 23, 2021).

[56] I. Free Software Foundation, “GDB: The GNU Project Debugger.” https://www.gnu.org/software/gdb/ (accessed Jul. 05, 2021).

[57] S. Karandikar *et al.*, “FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud,” doi: 10.1109/ISCA.2018.00014.