Assessing the Configuration Space of the Open Source NVDLA Deep Learning Accelerator on a Mainstream MPSoC Platform

Alessandro Veronesi, Davide Bertozzi, Milos Krstic

Assessing the Configuration Space of the Open Source NVDLA Deep Learning Accelerator on a Mainstream MPSoC Platform

Alessandro Veronesi[1], Davide Bertozzi[2], and Milos Krstic[1]

[1] IHP - Leibniz-Institut für innovative Mikroelektronik,
Frankfurt Oder 15236, Germany,
veronesi@ihp-microelectronics.com, krstic@ihp-microelectronics.com
[2] Universitá degli Studi di Ferrara, Department of Engineering,
Ferrara 44122, Italy,
davide.bertozzi@unife.it

Abstract. Deep neural networks (DNNs) are computationally and memory intensive, which makes them difficult to deploy on traditional hardware environments. Therefore, many dedicated solutions have been proposed in the literature and market. However, most of them remain proprietary or lack maturity, thus preventing the adoption of deep-learning (DL) based software in new application domains. The Nvidia Deep-Learning Accelerator (NVDLA) is a free and open architecture that aims at promoting a standard way of designing deep neural network (DNN) inference engines. Following an analogy with open-source software, which is downloaded and executed, open hardware is likely to use FPGAs as reference implementation platform. However, tailoring accelerator configuration to the capacity of cost-effective reconfigurable logic remains a fundamental challenge for their actual deployment in system-level designs. This chapter presents an overview of the hardware and software components of the NVDLA inference framework, and reports on the exploration of its configuration space. It explores the resource utilization-performance trade-offs spanned by the main precompiled NVDLA accelerator configurations on top of the mainstream Zynq UltraScale+ MPSoC. For the sake of comprehensive end-to-end performance characterization, the inference rate of the software stack and of the accelerator hardware are matched, thus identifying current bottlenecks and promising optimization directions.

Keywords: Deep-Learning, Reconfigurable Logic, Bare-Metal Software, Open Hardware, Configurable Accelerator

1 Introduction

In recent years, the market request for efficient hardware supporting deep-learning inference and training is increasing. As a result, computing accelerators in the context of heterogeneous hardware platforms are becoming key enablers

for the large-scale adoption of Artificial Intelligence (AI)-based solutions. While graphics processing units (GPUs) are still the most commonly used platform for accelerating deep neural networks (DNNs), more specialized hardware may enhance the overall power and time execution budget [1], [17].

Among the several options that are currently investigated, Field-Programmable Gate Arrays (FPGAs) are gathering momentum because of their capability to fit the tight power budgets of embedded applications [15] and to shorten the development cycle. Unlike their ASIC counterparts, they can be reconfigured on-the-field, which plays a fundamental role in modern context-sensitive IoT scenarios with changing input datasets over time and frequent model updates. As a result, FPGAs are gaining ground in assisting CPUs for deep neural network (DNN) acceleration [14], [26].

The growing popularity of FPGAs well beyond their traditional prototyping function as mainstream computing platforms well matches another emerging trend: open hardware. Inspired by the successful trajectory of open-source software, open hardware typically refers to the publicly availability of RTL models of IP components, so that interested users can feed them to a physical implementation flow. The industry itself may leverage open hardware as a means of speeding up the adoption of disruptive technologies, thus triggering a virtuous cycle that potentially results in broader market opportunities. This paradigm enables companies and academia to reduce development times and to take advantage of emerging technologies without massive investments. It is the case of the RISC-V Foundation, which aims at promoting the free and open RISC-V instruction set architecture, together with its hardware and software ecosystem [34].

There is a strong parallelism between open hardware and open software at various level of development and deployment process. In particular, as open software can be easily downloaded and compiled for a general-purpose CPU, an open hardware IP can be easily synthesized for an FPGA target, with a number of additional benefits. First, the released hardware models undergo an intensive customization effort by developers, either to specialize them for the application at hand or to augment the level of security of the hardware realization, since anybody can inspect and test the design. Second, prototyping on specific FPGA platforms may be an integral part of the business model for open hardware, to enable its validation on the field. Third, the resulting implementation on reconfigurable logic may already hit the desired compromise between performance, power and development cost to make it the target platform for deployment.

From the above brief observations, it is possible to assess how open hardware may play a central role in deep-learning-based ecosystems. Besides the availability of various commercial deep-learning accelerator (DLA) designs, closed DLA solutions' reserved nature and demanding license cost prevent broad adoption of those architectures and, more generally, of AI in new application domains. Therefore, the availability of open hardware DLA may be a key enabler for future innovative AI-based applications.

The more relevant examples in this sense are represented by the Nvidia Deep-Learning Accelerator (NVDLA) [32] and the agile systolic array generator Gemmini [4]. NVDLA is an open-source industry-grade inference engine that has also been integrated into the Jetson Xavier SoC platform [30]. NVDLA is an end-to-end software-hardware stack from the high-level deep-learning framework to the actual hardware implementation through the Runtime environment. The accelerator is highly configurable to adapt to various computing applications with different resource budgets.

On the other hand, Gemmini is an academic design implementing a parametric systolic array architecture for general matrix-to-matrix multiplication (GEMM) acceleration. In Gemmini, the systolic array hardware is expanded with some elementary post-processing units and a scratchpad memory. Given their open source nature, both NVDLA and Gemmini are influential contributions to the abundance of new machine learning accelerators, which raises interest in their comparison against state-of-the-art common benchmarks. The work in [5] has proven NVDLA to be 3.77× faster than Gemmini on an equivalently sized configuration running ResNet-50. Due to its promising performance and to its industry-proven nature, the focus of this chapter will be on the NVDLA accelerator and on its concrete FPGA deployment.

Each target FPGA platform can impose a different resource budget, which forces developers to trade architectural complexity and performance for lower resource utilization to some extent. On the one hand, hardware and algorithm optimizations for performance are pursued to make up for the inherently limited reconfigurable fabric speed. On the other hand, the limited programming density of such fabrics may cause full-featured hardware architectures not to fit the FPGA resources at hand. These conflicting requirements are exacerbated by the recent advancements of DNN algorithms (e.g. Winograd and FFT convolutions, weight compression), which come with their associated hardware extentions [2], [11], [28].

This chapter moves from the observation that the flexible NVDLA accelerator has mainly undergone virtual or physical prototyping so far while validating only single design points. Moreover, although the accelerator gained ample popularity since its initial release, the active support received from the community remains below expectations. One of the main reasons is the design complexity and poor software documentation. Moreover, few works correlate its wide configuration space to the capacity of mainstream reconfigurable fabrics and achievable performance. At the same time, like many other open hardware cores, NVDLA comes with testbenches for direct hardware performance evaluation. However, they may turn out to be misleading since they ignore the overhead of the software stack, including user-mode and kernel-mode drivers, the Runtime environment and portability layers for compatibility across computing platforms.

In order to facilitate the assessment, the further development and the concrete deployment of NVDLA, this chapter expands our previous work [20] by providing a more detailed introductory guide to the hardware and software stack of this DLA and an accurate in-sight of the available NVDLA Compiler's work-

flow and capabilities. This chapter will analyze the Compiler's workflow and functional capabilities (Section 4), the Runtime environment and its inference performance (Section 5), and will correlate hardware configurations to the resource capacity of a mainstream FPGA platform (Section 6).

The Xilinx's Zynq UltraScale+ MPSoC family (more precisely, the ZCU102 board) has been considered as implementation target. A bare-metal porting guide on top of the board is described in Section 5.1. On the other hand, Section 6.2 describes the synthesis results of available pre-compiled configurations on top of the board's programmable logic. Last, Section 7 provides a comparison between software and hardware throughput, thus, identifying major bottlenecks and pointing to some easily affordable optimizations to enhance the architectural support for FPGA targets.

2 Related Work

There is a surge of interest in FPGAs as implementation platforms for DNN accelerators [14], [25], [26]. The availability of high-level synthesis (HLS) tools from FPGA vendors lowers the programming hurdle and allows sophisticated end-to-end hardware-software co-design flows [13], [21]. Flexibility of reconfigurable logic is typically exploited to directly map hardwired DNN models for better performance [16], [25], [26].

A more general approach consists of supporting the mathematical operations at the core of deep-learning inference in the accelerator hardware. This approach is better suited for virtualized environments and frequent model updates. Along this direction, the NVIDIA deep-learning accelerator project promotes interoperability with the majority of modern deep-learning networks [32]. Early works on this architecture mainly resulted in extensive prototyping effort: some of them targeted virtual platforms [8], [27], while others targeted FPGAs [29]. However, none of them initially have gone beyond a functional verification of the design or the straightforward reporting of FPGA resource utilization, thus lacking of in-depth analysis.

Only the work in [12] provides performance evaluation. For instance, assuming the YOLO-v1 neural-network model as a benchmark, with a 256-MAC NVDLA at 150 MHz, they reach 8 frames per second. Unfortunately, a single design point is investigated (e.g., there is no discussion on the feasibility of fitting larger NVDLA configurations into the target board) and the Runtime overhead was not taken into account. To date, the most informative parametric study is still the one reported on the NVDLA website [32], which is however referred to ASIC technology, is far from exhaustive, and ignores the overhead of the software stack.

More recently, the focus has shifted on system integration and customization. NVDLA has in fact been synthesized and combined with different MCU architectures [3], [6]. For instance, in work [6], authors have integrated NVDLA with a RISC-V processor core, while keeping under control the power consumption of the accelerator. In the works [22], [23], the NVDLA's convolutional core has

been extended with additional hardware to implement an error detection routine working in parallel to the standard convolution operation. With only 30% of extra hardware and power consumption in the convolutional core, they can identify up to 99% of the injected faults during convolutions.

Udupa et al. [19] noticed that the NVDLA's dataflow is not optimal for 2D Convolutions and Pooling operations, which are quite common in mobile-oriented CNNs. Therefore, they propose a modification of the baseline MAC cells to enhance resource usage during those operations.

While the hardware architecture has been the focus of explorative research only since recently, the NVDLA Compiler has been targeted for optimization since the original release. An Open-Source tool enabling both the compilation of a broader range of neural networks than initially supported and their mapping onto different NVDLA configurations is presented in Ref. [9].

Last, the scalable SIMBA deep-learning accelerator [24] is based on an array of processing elements that are directly derived from the NVDLA architecture. The SIMBA DLA targets the provisioning of scalable compute power within package-level integration through multi-chip modules rather than large monolithic dies.

This chapter provides an introduction to the NVDLA architecture and aims to correlate the configuration space of the accelerator to the configuration space of the NVDLA accelerator to the performance-resource utilization trade-offs spanned on a commercial FPGA platform. In particular, this work has a distinctive focus on the interplay between hardware and software layers in determining inference performance, and investigating the NVDLA's synthesis outcome on a mainstream FPGA platform.

3 The NVDLA Architecture

The NVDLA project is composed of three main GitHub repositories, providing the source code of:

- The Virtual Prototyping platform;
- The Software Stack, composed of two Runtime Drivers and a Neural Network Compiler;
- The Hardware RTL and SystemC models.

The *Virtual Prototyping* repository is mainly composed by an NVDLA SystemC model and by a Linux Kernel image running on a QEMU platform and containing both drivers and Compiler. It is worth highlighting that the available SystemC model is an RTL model. Therefore, more complex networks may have a long simulation time.

The *Software* repository contains the Runtime environment, composed of a User-Mode Driver (UMD) and a Kernel-Mode Driver (KMD). The NVDLA's Compiler is released as part of the UMD code, since the two software components share many data structures. Both User-Mode and Kernel-Mode Drivers

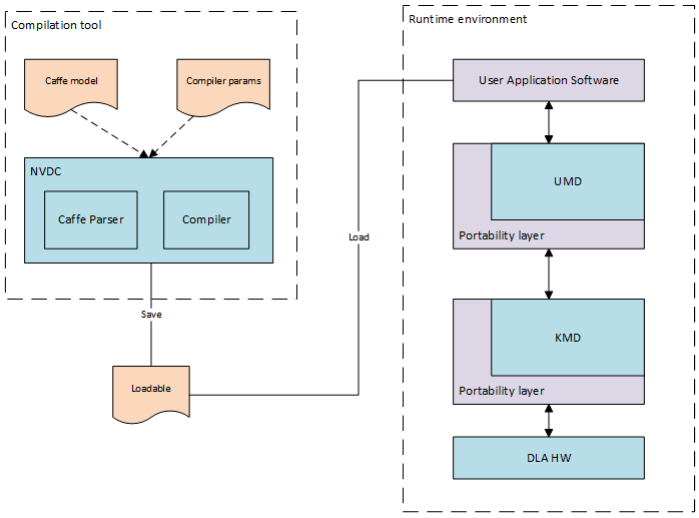

Fig. 1. NVDLA software stack.

are largely hardware-independent, and the needed modifications to make them run with different NVDLA configurations are few and simple.

The *Hardware* repository is structured into three different branches: the *"nvdlav1"* contains the "Full" configuration, entirely supported by the Compiler and containing all NVDLA functional units; the *"nv_small"* branch contains two "Small" and one "Large" configurations, suitable for more scaled applications; last, the *"master"* branch contains several custom configurations, most of them only partially verified or not mature for an ASIC implementation. All the available branches contain both a SystemC and a Verilog source code of the accelerator instances. Those descriptions are dynamically generated by a script-based environment which revolves around a *".spec"* Specification File of the configuration.

For our purpose, we focus on the software stack and the Verilog RTL model.

3.1 Software Environment

The software environment is composed by a Network Compiler and by a Runtime Environment. The Compiler acquires a pre-trained Neural Network written in *Caffe* as well as an NVDLA configuration and produces a *".loadable"* file, containing the list of hardware layers to be executed (see Fig. 1). A hardware layer is a set of operations to be scheduled on each functional unit of the NVDLA architecture. The current baseline Compiler fully supports only the "Full" configuration (also referred as *"NVDLA_v1"* online), while other configurations are only partially supported (see Section 4.2).

The loadable file is given to the Runtime Environment, which is composed by a *User Mode Driver (UMD)* and a *Kernel Mode Driver (KMD)*. The former

loads the network into the main memory (which has to be shared between the two drivers and the hardware), reads and unpackages the image file and executes data preparation, which mainly consists of re-interpreting the input pixel format into an accelerator-specific memory mapping. The KMD contains the functional unit scheduler and the hardware abstraction layer.

Instead, the network's weights are already prepared by the Compiler, which also takes care of their memory reservation. Network's execution information are thus serialized and packed into the *".loadlable"* file. More precisely, the "loadable" file consists of a byte sequence, containing network's data and a network's compiled model description structured as a list of executable hardware layers.

UMD starts its execution copying in DRAM the loadable file content. Together with the pre-processed input image, the loadable file's data are used by the UMD to prepare and send an "NVDLA Task" to the KMD. Thus, the inference is performed in a single NVDLA Task, through a layer-by-layer execution.

The OS-driver interactions are wrapped into a Portability Layer structure and hidden to the drivers' main routines. Currently, no actively supported bare-metal implementations exist for the NVDLA drivers, a gap that the work reported in this chapter will bridge not to include the OS overhead into the NVDLA performance evaluation. Both the UMD and the KMD are originally written for Linux and only support *JPEG* and *PGM* image formats.

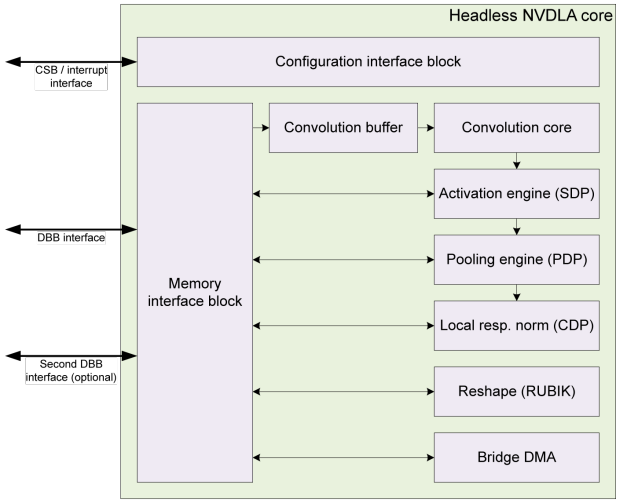

Fig. 2. NVDLA hardware block diagram.

3.2 Hardware Architecture

The NVDLA architecture is composed of several functional units. As shown in Fig. 2, NVDLA revolves around a sophisticated Convolution Pipeline (CONV), which is augmented by an activation engine (Single-Point Data Processor, SDP) and a pooling engine (Planar Data Processor, PDP). There are also more specialized hardware units to extend the range of compatible deep-learning applications. They include a Cross-Channel Data Processor (CDP) for local response normalization and a Reshape Engine (RUBIK) for simple image manipulation.

The accelerator has two interfaces to the outside world. On the one hand, the Configuration Space Bus (CSB) is a slave interface connected to the host processor and is used for accelerator configuration together with an interrupt signal. On the other hand, the data backbone (DBB) is a link with the main system memory, which is shared with the host processor. The CSB is a simple, memory-like interface, thought to be easily connected to a different standard bus. As an instance, the NVDLA's GitHub hardware repository already contains a CSB-to-AMBA APB bridge. Last, it is possible to instantiate an optional DBB interface, which typically consists of a high-speed, dedicated SRAM, onto which a Bridge DMA moves data from the main memory in a software-controlled way.

Beyond supporting the processing of a wide range of DNNs, NVDLA targets instantiation flexibility: in fact, most of the above hardware units are optional and highly configurable (e.g., number of MACs, convolution buffer size, batch size, number of operations per convolution round, activation function, etc.). Even when all functional units are supported, there are still significant degrees of freedom as to the deep-learning processing features, such as the convolution algorithm and the weight compression option. A detailed list of configuration options is reported in Section 6.1.

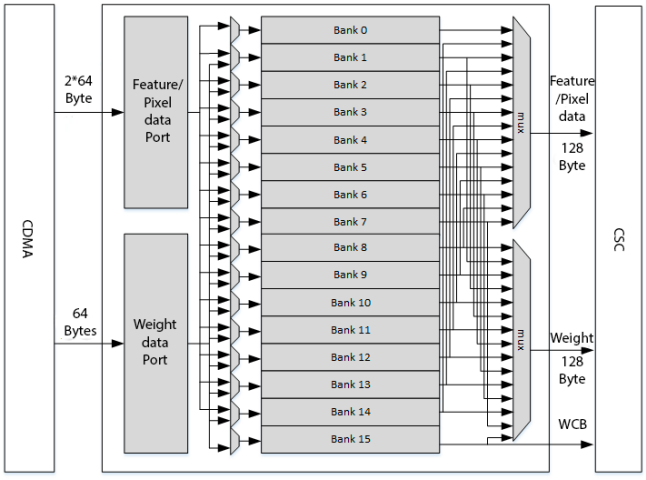

Fig. 3. NVDLA CBUF block diagram in the "Full" configuration.

The Convolutional Pipeline (CONV unit) is composed of five stages: Convolution DMA (CDMA), Convolution Buffer (CBUF), Convolution Sequence Controller (CSC), Convolution MAC array (CMAC), and Convolution Accumulator (CACC). The system designer can configure the CONV pipeline by enabling optional extra features (Winograd convolution support, Multi-Batch size and weight compression support) and choosing CBUF and CMAC sizes.

CBUF block comprises several SRAM banks (see Fig. 3) which can be configured in number and size. Each bank acts as a circular buffer, where new data has incremental entry address, and if the address reaches the end, it wraps to zero and starts increasing again.

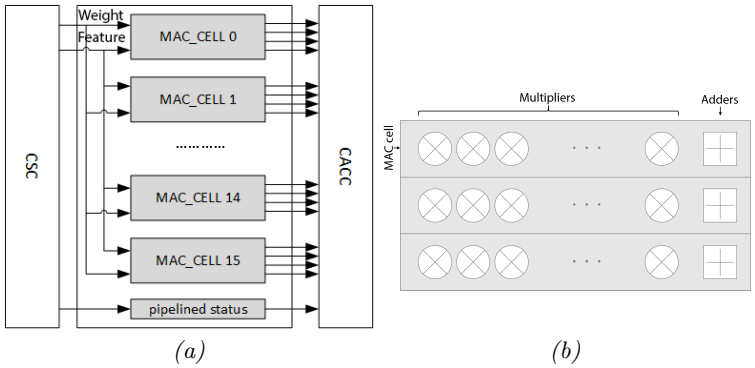

Fig. 4. NVDLA CMAC unit. An array of MAC cells shares a broadcast input (a). Each MAC cell is composed of an array of multipliers and the result is sent to the adder tree in CACC (b).

The computing core of the CONV unit is the CMAC. The CMAC unit is structured as an array of *Atomic_K* MAC_cells, where each cell is composed of an *Atomic_C* number of multipliers (see Fig. 4). Data are fed from the CBUF and broadcast among different MAC cells. The two above parameters define the simplest atomic convolution supported, where *Atomic_C* is the maximum number of accepted input tensor channels and *Atomic_K* is the maximum number of accepted kernels in a single layer. Multiplication results are sent to the accumulation unit (CACC) and summed thanks to an adder tree. Last, it is essential to remember that the CONV unit output has no direct access to the memory interface. Instead, CONV results are sent to the SDP unit as inputs.

In NVDLA "Full", the Single-Point Data Processor (SDP) can perform bias addition, batch normalization, element-wise operations among different input tensors and non-linear functions applications, mainly to perform activations. Non-linear functions may be both natively supported by SDP hardware (like ReLU and PReLU) or emulated through a Look-Up Table unit (like Sigmoid and Tanh in the "Full" configuration). Other configurations differ per type of supported operations and sub-units throughput (see Section 4.2 and Section 6.1).

The Planar Data Processor (PDP) is capable of *min*, *max* and *mean* pooling operations. The PDP engine operates on different planes of the input feature data cube. No interferences are present on different channels of the same feature data cube. PDP can acquire data both from the memory interface and the SDP output (also referred to as *"fused"* mode or *"on-the-fly"* mode).

The Cross-Channel Data Processor (CDP) is specifically designed to implement those functions that need to cross-combine values coming from separate channels. To perform them, CDP uses a LUT-based engine. In the the "Full" configuration the Compiler can configure it to perform Local-Response Normalization (LRN) functions. LRN layers are present in models like AlexNet [7] and GoogLeNet [18], but are becoming less and less popular in recent DNN models.

NVDLA configuration takes place exclusively through the CSB interface. Every unit has a CSB slave where, to hide the programming latency, two sets of configuration registers are present, handled in a ping-pong manner.

NVDLA units can operate sequentially or pipelined. In particular, CONV, SDP and PDP units can be configurred as a single execution chain (also referred to as *"fused mode"*), while CDP and RUBIK cannot, since they differ in data format. Otherwise, functional blocks can be configured independently (also referred to as *"independent mode"*) and perform memory-to-memory operations. This causes a per-block round-trip through main memory. However, online documentation poorly describes those operating modes. More details will be provided in Section 4.2.

4 NVDLA Network Compiler

The NVDLA's *Compiler* takes care of the Neural Network (NN) framework interpretation and of the operations for functional units mapping. Its routine revolves around two primary operations: input files parsing and execution code generation. An NVDLA environment refers to the compiled file containing the information for the complete execution of a Deep-Learning model as *".loadable"* file.

It is worth noticing that the NVDLA's original Compiler is compatible only with the *Caffe* Neural Network Framework. Nevertheless, different alternative solutions are available online. Talking about the direct support by NVIDIA, the TensorRT library [31] is an open-source AI toolchain and runtime software developed for Jetson boards. It fully supports the ONNX framework and is capable of advanced NN compiling features (e.g., kernel auto-tuning, layers and tensors fusion, etc.). Although, it natively targets NVIDIA Jetson boards, thanks to its open-source availability, it may be adapted to different NVDLA-based SoCs.

Regarding the third parties support, Skymizer developed an ONNX Compiler with a dedicated NVDLA backend [9]. The Open Neural Network Compiler (ONNC) is capable of the same features of the initially developed Compiler by NVIDIA, but with a greater focus on customization simplicity and flexibility.

Additionally, it has been reported to support more networks than the original NVIDIA Compiler (see Ref. [8]).

4.1 Compiler Workflow

The NVDLA Compiler is released as a set of APIs available in the UMD source code. The Compiler is structured into a frontend and a backend. The frontend coincides with the NN model parser, the backend instead implements compiling routines, which are composed, among other things, of the network model-to-hardware units assignment procedure, the memory allocation routine and the graph optimization. NVIDIA provides a Compiler executable for *Caffe*-based Neural Network models which relies on Google's Protocol Buffer library.

The NVDLA Compiler execution always starts with the parsing operation. The Compiler takes a pre-trained *Caffe* network (composed of a *".prototxt"* file containing the model description and a *".caffemodel"* file containing the trained weights), and produces a first intermediate representation of the network. This first representation consists of an oriented graph where nodes are network layers, and edges are features and weights data cubes. Next, the Compiler realizes a second inspection of this graph and associates to every network layer a different hardware unit to perform it, compliant to the layer behaviour. Last, several optimization cycles are performed on top of this graph to resolve data and control dependencies.

During the optimization cycles mentioned above, many essential routines take place. One relevant example is the convolutions splitting process. Since NVDLA's CMAC unit size is user-defined, it is expected that bigger convolutions are not fitting it. Thus, the Compiler separates them in several different hardware layers to be processed in sequence (see Fig. 5).

As stated in Section 3, the convolution engine has no write-dedicated DMA, and every convolution operation result is sent to the SDP engine. Since data coming from the CONV engine are not automatically retrieved from the SDP, the Compiler appends to every CONV layer an SDP "no-operation" (NOP), to make it write out its input. Since most of the convolutions in NNs are followed by activations, the Compiler then optimizes the graph by merging adjacent SDP operations, whenever possible. As a result, the number of scheduled SDP operations is always equal or bigger than the number of convolutions performed (see Fig. 5).

NVIDIA chose to make its DLA compatible with the FP16 and INT8 data types. However, many NN frameworks (and *Caffe* as well) rely on a 32-bit Floating-Point data type (FP32) to improve training precision.

The conversion between the original FP32 data and an FP16 or even INT8 data is automatically handled by the Compiler. However, a calibration table should be used to simplify the work that may not turn out successful in the most complex transformations (i.e., from FP32 to INT8, see Ref. [33]). Finally, when the entire network's data are converted, the Compiler reserves memory entries for them in a virtualized memory environment.

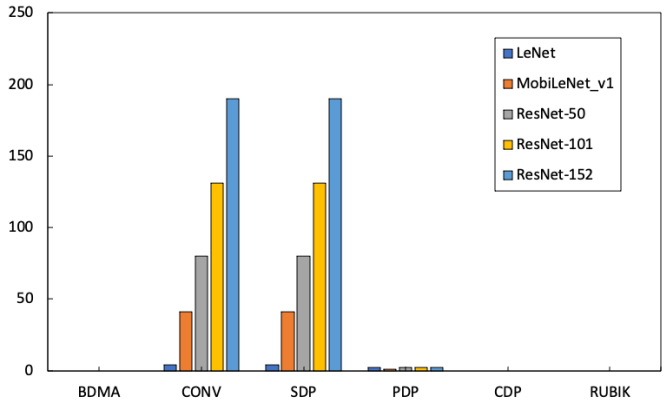

Fig. 5. Hardware Layers programmed for each functional unit (hereby compiled for the "Full" configuration). The CONV operations number is always equal or bigger than the network's convolutions number because of the splitting mechanism. The number of SDP is never less than the CONV operations, due to the NOPs appended to every CONV operation performed.

Together with the above operations, the Compiler performs additional transformations. Among them, this software updates the tensors sizes for optimizing element-wise operations in the SDP engine or pre-processes the network's auxiliary data.

Last, the Compiler can also support operations that are not natively accelerated by the NVDLA's hardware. This feature is mainly available for the *"SoftMax"* non-linear function. To achieve the SoftMax support, the Compiler produces a particular hardware layer labelled as "emulator-dedicated". EMU-dedicated hardware layers are network operations not supported by the hardware that must be executed by the Runtime environment instead and, thus, by the CPU where the software is running on.

When all the above operations are successfully performed, the Compiler serializes the network's graph representation and translates the whole data structure into a list of addresses and configuration values. The whole memory stack containing the network compiled information is finally saved into the loadable file.

4.2 Supported Features

The original NVDLA Compiler supports only the "Full", "Large" and "Small" configurations, but can be extended to additional configurations by customizing the proper data structure present in the source code. However, NVIDIA fully verified all the three available configurations ("Full", "Large" and "Small") only for the MNIST, ResNet-18 and ResNet-50 models, providing an INT8 calibration table only for ResNet-50.

Table 1. NVDLA's Compiler supported features per data path (a) and supported functional units per configuration backend (b)

Functional Unit	Feature	FP16	INT8
CONV	Direct Conv.	Yes	Yes
	Dilatation Conv.	Yes	Yes
	Winograd Conv.	Yes	Yes
	Deconvolution	Yes	Yes
	Fully Connected	Yes	Yes
	Group Conv.	Yes	No
	Winograd Group Conv.	Yes	No
PDP	Pooling (min/max/avg)	Yes	Yes
SDP	Bias Addition	Yes	Yes
	Batch Norm.	Yes	Yes
	Scale	Yes	Yes
	Sigmoid	Yes	No
	Tanh	Yes	No
	EltWise SUM	Yes	Yes
	EltWise SUB	No	No
	EltWise MIN	Yes	No
	EltWise MAX	Yes	No
CDP	LRN	Yes	No

(a)

Unit	Full	Large	Small
Winograd	No	No	No
Compress Weights	No	No	No
PDP	Yes	Yes	Yes
CDP	Yes	Yes	Yes
SDP Bias	Yes	Yes	Yes
SDP BatchNorm	Yes	Yes	Yes
SDP EW	Yes	Yes	No
SDP LUT	Yes	Yes	No
BDMA	Yes	No	No
RUBIK	Yes	No	No

(b)

Even if the Compiler can support a vast amount of different features, not all of them are implemented in practice (see Tab. 1). For example, the procedures to perform Winograd Convolutions are available in both the presented data path, but to date none of the available Compiler's backends can produce a loadable file supporting this algorithm. The same happens for the weight compression capability.

Different considerations hold for the LUT sub-engines in SDP and CDP units. The software must configure those units in order to use non-linear activation functions on them. To offload the Runtime and speed-up the execution, the Compiler already determines the configuration register values for the LUT programming.

Therefore, to implement a non-linear function in the LUT engines, the Compiler must contain a quantized version of the selected function and a dedicated subset of routines determining the configuration registers' values to program it into the hardware. Even if the Compiler supports LUT engines in both the "Full" and "Large" configurations, the functions' quantized descriptions are not available for all the datapaths (see Tab. 1). More precisely, *Sigmoid* and *Tanh* activations functions are available for the "Full" configuration, but not for the "Large" configuration.

The same happens with the LRN operations in the CDP engine. Since this unit is LUT-based, a quantized version of the non-linear function must be provided and programmed into the hardware. Like Sigmoid and Tanh, a quantized version of the LRN is available in the Compiler's code for the FP16 data path, but not for the INT8 data path, thus no LRN support is available in the "Large" configuration.

Another Compiler's duty consists of serializing the network's graph representation. That means the Compiler determines the precise functional units' execution order and optimizes it. Since NVDLA can combine the CONV, SDP and PDP engines, the Compiler decides if those units are operating independently (*independent mode*) or in a single pipelined chain (*fused mode*, or *"on-the-fly pro-*

cessing" in the online programming guide). Independent and fused modes are always available and supported since all the provided NVDLA configurations contain the involved units.

The CONV and SDP pipelining is always available since the CONV unit has no direct write back to the memory interface but always passes through the SDP engine. Nevertheless, currently the fused mode is only supported for *"ReLU"* activations, while executing a *"No Operation"* in SDP to write back the convolution result in other cases.

Last, if CONV and SDP units are pipelined, the Compiler always tries to append the PDP operations whenever possible. However, this is possible only when the PDP buffer size is compliant to the SDP output size. This means that if the software splits a convolution layer into more hardware layers, PDP pipelining is not possible.

5 Software Runtime Analysis

NVDLA's Runtime is composed of two drivers, named UMD and KMD. While UMD mainly takes care of data preparation, KMD contains the functional unit scheduler and the hardware abstraction layer.

UMD is available to the developer as a set of C++ APIs. NVIDIA provides a sample main routine for image recognition tasks, containing the input image pre-processing and basic routines calling the UMD APIs for network loading and task submission.

User-Mode Driver contains an *"Emulator"*, a software extension to DLA's capabilities. Thanks to the Emulator, the UMD can offload the execution of specific layers to the CPU. In the original UMD and Compiler, this is used to perform *SoftMax* activation functions.

Kernel-Mode Driver contains the functional units scheduler and the hardware abstraction layer. Since the network model interpretation and resource assignment have already been made by the Compiler, the functional unit scheduling restrict itself to a temporal assignment of the hardware layers to free units.

One of the main goals of this chapter is to shed light on the performance of the Runtime environment. A porting of the whole software stack on the industry-standard Zynq UltraScale+ MPSoC (ZCU102) is presented to assess it, including a bare-metal implementation of the Runtime software in order to hide the additional overheads introduced by the Operating System.

5.1 Bare-Metal Implementation

A bare-metal version of the NVDLA firmware can be derived and ported to one core of the 64-bit ARM Cortex A53 host processor on this platform, running at 1.2 GHz. The board's DRAM memory sub-system revolves around a 4 GB Micron MTA4ATF51264HZ device configured to reach a peak bandwidth of 21.3 GB/s. This speed is enough for the requirements of the DNNs that will be

tested later [32]. Without a lack of generality, the board SD Card is used as the repository for the input loadable file and for the images to be processed.

According to NVDLA's drivers' structure, the porting must be accomplished mainly by re-writing their portability layers. To simplify the hardware management, Xilinx provides a group of software libraries (named "standalone" in case of bare-metal software) for the Zynq UltraScale+ MPSoC boards family. Thus, those libraries were exploited to achieve the file system support (to read images and loadable file from the I/O unit), interrupt control (to interact with NVDLA hardware) and dynamic memory management (extensively used by the UMD). The file system support is not essential, but allows to reuse a big part of the available software without strongly patching the Compiler and the Runtime.

Since the focus is on the performance assessment of UMD essential routines, the Emulator code section is not ported, containing the SoftMax non-linear function implementation. This even simplifies the UMD porting since no half-precision Floating-Point (FP16) support is required anymore.

The original Linux drivers implementation relies on file descriptors to manage the communication between the two drivers, using virtualized memory regions for data exchange. As a workaround, specific data structures are instantiated as an exchange area between UMD and KMD. The actual exchange routines take place via *memcpy()* operations.

The NVDLA's original drivers communicate with each other through an *ioctl()* mechanism. Therefore, the original *ioctl()* methods have to be converted into lightweight standard C routines, which can be invoked by the UMD-to-KMD interface functions.

Last, since the UMD hadn't been initially developed for a bare-metal environment, many routines in the UMD portability layer (that largely correspond to the UMD-to-KMD interface) are thought to be invoked during the UMD associated process creation and closure. Those procedures are mainly unnecessary for a single-process environment like the one considered in this chapter, while the memory reservation mechanism (easily replaced by a *malloc()*) and the initialization procedures (to initialize the interrupt handler and the hardware-related parameters like the NVDLA's address) can be relevant for other contexts.

After the Runtime's bare-metal porting, the latter consists of a unified execution flow where the developer can focus on key Runtime operations rather than on the driver entities that perform them. Therefore, key Runtime operations include:

- **Network loading**. It transfers a pre-compiled loadable file from an I/O device (in our case, the SD Card) to main memory.
- **Test operation**. It performs image reading, data preparation and fills up a data structure with task scheduling information.
- **Submit operation**. It schedules operations to the accelerator's functional units, with which interaction takes place through the hardware abstraction layer.

In order to assess the software stack of the NVDLA framework in isolation, the hardware execution should be assumed to occur in zero time. This can be

achieved by disabling write/read operations to/from the accelerator registers in the KMD, and by virtually driving interrupt responses of the accelerator to the hardware abstraction layer, under the assumption of instantaneous execution of scheduled commands.

Achieving the latter object is relatively easy. The inference always starts with the UMD sending an "NVDLA task" to the KMD. The task is a decompressed version of the loadable file, containing a list of hardware layers to be performed, their dependencies and a set of memory addresses pointing to the weights data allocated in main memory. The KMD performs the task in a functional unit scheduling wheel, where at each cycle, a new operation is programmed in the NVDLA's registers, then a "wait for interrupt" statement is performed. At the arrival of the interrupt, the KMD reads the functional units' status and determines which operations have completed. With this information, the KMD updates the task dependencies and programs the next operation to be executed.

To virtually drive the interrupts, the registers interactions are hidden and the operation programmed data saved directly in a variable. Then, the interrupt handler function is manually called, and the correct flag is set according to the saved data.

5.2 Runtime Performance

To evaluate the Runtime performance, Zynq Cortex's APU Global Counter can be used. If clocked at 100MHz, Runtime execution time measurement turn out to have a precision of 10 ns.

Regarding the tested DNNs, their Caffe models have been compiled with the baseline NVDLA Compiler provided by NVIDIA. According to what presented in Section 4.2, compiling for different NVDLA configurations mainly results in different functional units availability, different data path precision and convolutional pipeline characteristics. In order to characterize the software stack, all the DNNs under test have to be compiled for the NVDLA "Large" configuration of the accelerator hardware, with a batch size of 1.

As anticipated in Section 5.1, the key Runtime operations include compiled Network Loading, Test operation and task Submit. Next, the assessment of the performance of these operations is provided.

As shown in Fig. 6, the network Loading time dominates over that of other operations, due to reading of the loadable file from SD Card. On the target board, the loading time turns out to be slightly better than a similar operation performed from Flash memory, given the bandwidths of the devices under test: 100 MByte/sec for the SD card vs. a peak bandwidth of up to 90 MByte/sec for the Flash memory. It is worth recalling that network loading occurs only occasionally for DNN model update, while Test operation and Submit are performed at each inference. Therefore, from now on, our focus will be only on Test and Submit.

In contrast, the scheduler and the hardware abstraction layers (see "Submit" column) run much faster, mainly because the DNN interpretation has already been realized during the compilation phase. Thus, Runtime scheduling restricts

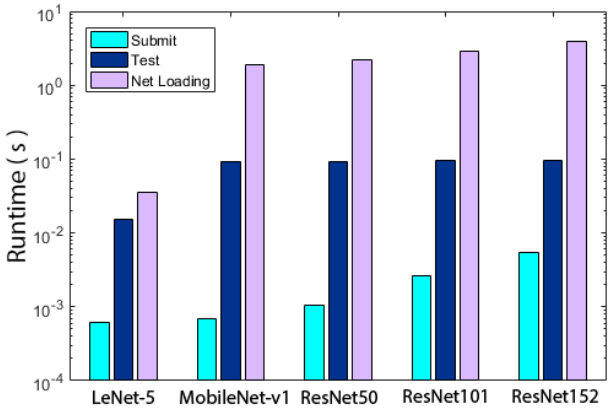

Fig. 6. Execution time of the key operations of the NVDLA Runtime for different DNNs under test. Image format: 28 × 28 pixels for LeNet-5, 224 × 224 pixels for the others.

only to task-to-functional unit assignment over time. As a result, the "Submit" time grows with the number of schedulable "hardware layers" the DNN is broken into by the Compiler, and that the KMD scheduler will process.

While the "Submit" time scales with the computational cost of the DNN under test, the Test operation time shows a tighter correlation with the format of the input dataset (MNIST for LeNet-5 vs. ImageNet for the remaining DNNs). This results from the breakdown in Fig. 7, which shows the contribution of the image loading time from SD card (in yellow). Clearly, the Test execution time is dominated by data preparation on the ARM and DRAM subsystems, and not by the I/O unit.

Next, under the assumption of instantaneous hardware execution, it is possible to characterize sustainable software throughput. In order to get realistic performance estimates, the assumption is that the image to be processed is already in DRAM. As a result, the Test routine remains the throughput bottleneck and does not enable a frame rate higher than roughly 20 frames-per-second for ImageNet-processing DNN models, and 120 fps for LeNet-5. The Submit operation alone would enable up to 1600 fps for LeNet-5 and slightly more than 200 fps for the most complex Resnet152.

It is important to observe that the Test operation is hardware-independent and memory-centric, thus the NVDLA Runtime environment gives rise to a significant software overhead. Therefore, using NVIDIA precompiled testbenches to project accelerator frame rate may be misleading, since limitations dictated by the software stack are not accounted for.

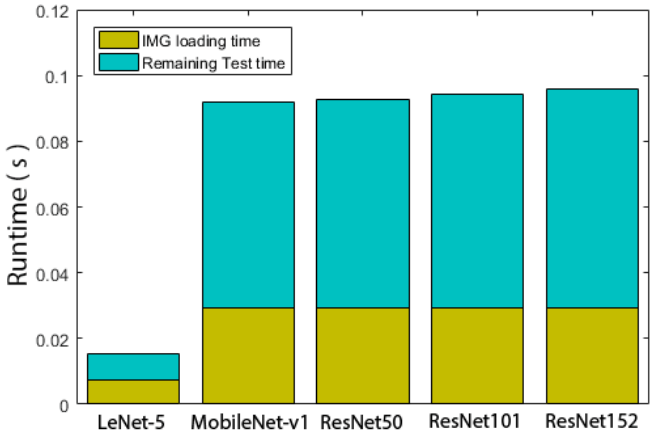

Fig. 7. Breakdown of the Test execution time.

6 Hardware Synthesis

One of the distinctive features of NVDLA is hardware configurability. For the sake of coherent analysis, the Xilinx's Zynq UltraScale+ MPSoC is also used as target platform as already done for the software stack. More in detail, NVDLA model was synthesized and mapped for the Programmable Logic (PL) of the ZCU102 board.

In all tested configurations, the accelerator is inferred as a single clock domain. No specific mapping optimization to heterogeneous FPGA resources was applied. Thus, absolute performance measurements reported in this chapter have to be considered as pessimistic.

6.1 Inspected Hardware Configurations

NVDLA currently comes with several pre-compiled hardware configurations. All the main stable configurations for an FPGA synthesis will be tested. A detailed description can be found in Tab. 2. They encompass "Small", "Medium" and "Large" or even "Full" instances. All of them share a common baseline feature: they instantiate all the functional units of the accelerator, except for RUBIK. Moreover, the usage of the SRAM interface is configuration-specific.

Instead, they mainly differ by the target data type, the number of instantiated MACs and the buffer size in the convolutional unit, batch size, and by DNN algorithmic features such as weight compression or Winograd convolution (which correspond to matching hardware units).

The "Full" version includes, among the other things, Winograd convolution, weight compression, a batch size of 32, 2048 MACs (organized as 16 MAC cells, 64 multipliers each), INT8/FP16 data path and a dedicated SRAM interface.

In this configuration, internal optimizations are enabled in the SDP, PDP and CDP in order to maximize throughput. The 16 FP16 MAC cells can be reconfigured to support multiple data path as 32 INT8 MAC cells. Overall, it is a high-end instantiation of a deep-learning accelerator aiming at memory-efficient high-performance inference. The "Large" configuration is directly retrieved from the "Full" configuration while relying only on an INT8 data path.

The "Small" configuration exhibits a baseline convolutional pipeline without the optional memory interface, and a tighter resource budget for reduced area and power requirements, including 64 MACs (organized as 8 MAC cells, 8 multipliers each), INT8 data path, and a batch size of 1. To further diminish the area footprint, the SDP engine cannot perform element-wise operations between two input feature data cubes and its LUT-based activation unit is not present (see Tab. 2 for all details). The "Medium" configuration directly enhances the "Small" configurations, enlarging the MAC array size and the other units' throughput.

Table 2. Tested hardware configurations. *Atomic_K* is always referred to INT8 data precision

	Data Type	Total MACs	Atomic C	Atomic K	CBUF (KB)	SDP	PDP	CDP	Winograd Support	Weight Compression	Batch Size
nv_small	Int8	64	8	8	128	Yes	Yes	Yes	No	No	1
nv_small_256	Int8	256	32	8	128	Yes	Yes	Yes	No	No	1
nv_medium_512	Int8	512	32	16	512	Yes	Yes	Yes	No	No	1
nv_large	Int8	2048	64	32	512	Yes	Yes	Yes	Yes	Yes	32
nv_full	Int8/Fp16	2048	64	32	512	Yes	Yes	Yes	Yes	Yes	32

	SDP E-W Support	SDP LUT Support	SDP Throughput (op/cycle)	SDP E-W Throughput (op/cycle)	PDP Throughput (op/cycle)	CDP Throughput (op/cycle)	RUBIK Support	SRAM Mem. If.
nv_small	No	No	1	-	1	1	No	No
nv_small_256	No	No	1	-	1	1	No	No
nv_medium_512	No	No	4	-	2	2	No	No
nv_large	Yes	Yes	16	4	8	8	Yes	Yes
nv_full	Yes	Yes	16	4	8	8	Yes	Yes

6.2 Implementation Synthesis Results

Tab. 3 reports FPGA resource utilization for the configurations under test. Interestingly, even the smallest accelerator instance ("Small" configuration) takes already 43% of available CLBs, an utilization that grows to 84% for the "Medium" configuration. From these tests, the "Large" and "Full" architectures are so overprovisioned that they do not fit the Programmable Logic of the Zynq UltraScale+ platform, which questions their deployment in cost-sensitive application doamins.

More in detail, the second column of Tab. 3 shows an extensive logic LUT utilization, which grows from 25% in the "Small" configuration to 47% in the "Medium" one. MAC units occupy only 10% of the LUTs in the "Small" baseline configuration and roughly add from 60 to 90 additional LUTs for each increment in the number of MACs, which depends on the specific configurations.

Table 3. Synthesized hardware resource utilization

	LUTs	**LUTRAMs**	**FFs**	**BRAMs**	**DSP slices**	**CLB occupation**
nv_small	68927	196	69347	100	41	43%
nv_small_256	90871	196	92830	165	41	57%
nv_medium_512	130532	200	116364	187	75	84%

Simultaneously, the number of instantiated DSP slices is quite limited: from 1.63% in "Small" to 2.98% in "Medium". This points to an inherent limitation of the released source code of the accelerator: it natively targets ASIC implementation and is not optimized to make extensive use of DSP slices of the target FPGA. The root cause for such inefficient use of DSPs may be identified in the logic-level specification of convolutional's unit multipliers. In facts, MAC cells multipliers' gate-level description biases the synthesis tool interpreter toward a LUT-based implementation of those hardware elements.

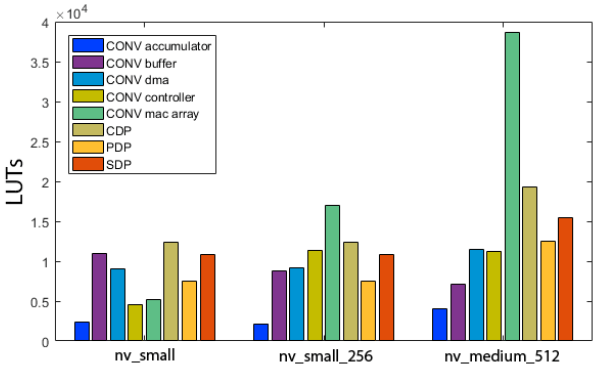

Fig. 8. LUT occupation for each NVDLA functional unit.

Allocated DSP slices are in the Convolution DMA, which uses them for address manipulation, and especially in the CDP engine. In those units, multiplications are described as arithmetic operations and modelled with a more abstract behavioural syntax, thus mapping to DSP slices is straightforward for the synthesis tool. As a result, when target CDP performance is boosted in the transition from "Small 256" to "Medium" configurations, hardware redundancy is used to meet the requirement, which results in a doubled allocation of DSP slices. Source code optimizations for better use of DSP slices in the convolutional pipeline is an active research and development area.

For each hardware configuration under test, Fig. 8 illustrates a breakdown of LUT occupation into the individual functional units. Only interfaces to external

busses are omitted since constant throughout the configurations and lightweight with respect to reported results (e.g., 300 LUTs for the CSB interface).

We can first notice that the LUT occupation of the MAC array significantly changes across configurations. In the "Small" one, the MAC array area is irrelevant compared to contributions of other functional units, while in the "Medium" configuration it becomes as large as 30% of the total LUT count.

Besides the MAC array, the CDP makes the most extensive usage of LUTs, followed by the SDP engine. In particular, the number of LUTs in both CDP, PDP, and SDP increases from "Small 256" to "Medium" due to the boost in these functional units' throughput requirements.

Fig. 9. NVDLA CBUF bank structure in three different configurations. nv_small, SRAM macros=8-Byte×256 each (a); nv_small_256, SRAM macros=8-Byte×128 each (b); nv_medium, SRAM macros=16-Byte×128 each (c)

Observing the Convolutional Buffer (CBUF), we can notice a counterintuitive trend in LUT (Fig. 8) and BRAM allocation (Tab. 3). Starting from the "Small" configuration, the CBUF LUT utilization decreases with the growth of the CBUF size (from 128kB to 512kB). At the same time, the BRAM utilization has a significant increase in the transition from "Small" to "Small 256" architectures (i.e., from 64 to 128 BRAMs, which determines a similar trend also for the total number of BRAMs allocated by the design as a whole). However, the allocated BRAMs remain stable in the transition from "Small 256" to "Medium" configurations.

This synthesis result is apparently unjustified observing the CBUF total size, which does not change between the "Small" configurations, but increases in the "Medium" one. The explanation can be found in the way the CBUF is organized. In all the tested configurations, NVDLA comes with 32 SRAM banks, but in the "Small" one they have a width of 8-Byte, which grows to 32-Byte data in the "Small 256" and "Medium" configurations.

The single buffer bank organization consists of a set of SRAM macros with a bitwidth of 64 in the two "Small" architectures (see Fig. 9). Therefore, when the MAC array needs to be fed with a 64-bit input ("Small" case), one macro would be enough. However, in order to increase the buffer depth, the "Small" configuration combines two SRAMs into a unique aggregate macro, while leaving the output bitwidth unaffected.

When the bitwidth increases to 256 bits ("Small 256" and "Medium" cases), the CBUF aggregates 4 64-bit macros to provide it in "Small 256", and 2 128-bit macros in "Medium".

The mapping of these convolution buffer configurations to FPGA turns out to be sub-optimal. In fact, both the 16kb SRAM macros in the "Small" configuration and the 8kb macros of the "Small 256" configuration are exclusively mapped to 36kb BRAMs of the FPGA. The result is twofold. On the one hand, this explains the increase in allocated BRAMs in Tab. 3 when moving from "Small" to "Small 256" despite the convolutional buffer size stays the same. On the other hand, this mapping results in a total of 2304Kb occupied memory for the "Small" and 4608Kb for the "Small 256" instances versus a request of only 1024Kb in both cases.

When moving to the "Medium" configuration, the SRAM macro size grows to 64Kb, which exceeds the BRAM size, hence justifying the FPGA synthesis and mapping tools (Vivado) choice to map each macro to two different 36Kb BRAMs. Thus, the number of occupied BRAMs is the same as in "Small 256" (see Tab. 3). However, memory utilization efficiency is better: 4608Kb of memory is occupied versus the requested 4096Kb. Overall, the "Medium" configuration makes more efficient use of memory resources than the lower-end ones.

Finally, the different convolution buffer organization leads to different complexity of the multiplexing and control logic. From Fig. 8, the "Small" configuration takes more LUTs in its CBUF due to an additional multiplexing layer to aggregate SRAM macros. Moreover, "Medium" reduces the used LUTs in CBUF with respect to "Small 256" because only two SRAM macros have to be combined to provide the target bitwidth instead of 4.

Table 4. Synthesized hardware timing

	Clock Speed	**ResNet-50 (frame/sec)**
nv_small	130 MHz	0.95
nv_small_256	130 MHz	5.98
nv_medium_512	80 MHz	7.44

Moving the discussion to the hardware execution time, we can observe the clock speed and inference performance results reported in Tab. 4. We observe a 38% reduction of the clock speed only for the "Medium" configuration.

In order to factor in the clock speed to assess inference performance, we assume the ResNet-50 DNN as a benchmark. From the "Small" to the "Small 256" configuration, an allocation of 4x the number of MACs results in an inference rate improvement by 6x.

When we move from "Small 256" to "Medium", the benefits of doubling the number of MACs, improving the convolutional buffer size and speeding up the functional unit throughput are partly offset by the more complex hardware and the lower operating speed. Therefore, the inference rate speedup is only 1.2x (instead of the theoretical 2x), achieved with a number of CLBs which is 1.4x, thus giving rise to an unbalanced cost-benefit trade-off for "Medium".

7 End-to-End HW/SW Performance and Optimizations

For a single inference of the ResNet-50 DNN, the hardware and software execution times have been combined. The software stack has been characterized for the "Large" configuration, but it is matched to the performance of a "Medium" accelerator instance since the only difference, i.e., the Submit time, would be negligible. Assuming a batch size of 1 (which is the only possible in the "Medium" and smaller configurations), 63ms have to be budgeted for the software stack (the Test operation takes two orders of magnitude longer than Submit) and 134ms for hardware inference in the "Medium" configuration. Overall, the resulting frame rate amounts to 5.13 fps. For the "Small" configuration, the frame rate can be as small as 1 fps.

As a future optimization, it is conceivable to parallelize data preparation with hardware operation. Since the total frame rate is currently hardware-dominated, the expected upper bound consists of the 7.44 fps derived in Tab. 4 for "Medium", while no noticeable improvement is expected for "Small". One way to quickly achieve this modification may be to restore the original UMD and KMD software entities and make them work in parallel on two different processing cores. Since original drivers act independently, their execution can be pipelined.

As far as hardware is concerned, future performance optimizations may come from better FPGA heterogeneous resource utilization. A significant re-writing of the MAC cells source code may involve DSP slices in the convolutional pipeline, enhancing the CMAC throughput. Second, onboard BRAMs can be better exploited to implement both CBUF's and other units' memories. Moreover, since those modifications reduce the LUT occupation and better use the already available FPGA resources, the total CLB usage will likely be reduced. Thus, this would enable smaller NVDLA configurations to fit the programmable logic of low-end devices as well.

Last but not least, since not all NVDLA functional units are always addressed (as seen in Section 4.2) and the Runtime is mainly configuration-independent, many NVDLA sub-units can be removed from the hardware "spec" file, without significant modifications of the hardware/software stack (e.g., CDP unit in "Small" configuration). This will lead to even smaller hardware footprints, without loss in performance or functionalities.

8 Conclusions

The NVIDIA Deep-Learning Accelerator is a promising project bringing an industrial-grade design to the open-source community. However, even if a rich hardware architecture has been released, the software's available support and documentation are reduced and must be enhanced by the community.

The exploration of the software stack reported in this chapter revealed the Compiler not to be optimized for hardware configurations different from "Full". At the same time, not all the functionalities available in hardware are adequately exploited in the software stack. LUTs engines are fully supported only

for the FP16-capable architectures, while the Compiler support for INT8-based architectures is reduced. Moreover, additional functionalities such as Winograd convolutions, weight compression or the additional memory interface for the "Large" configuration are unsupported even when available in the instantiated hardware.

Moving to the hardware implementation, this chapter´reports on the DNN accelerator mapping onto the commercial Xilinx UltraScale+ MPSoC and its integrated reconfigurable logic. The analysis revealed that the pre-compiled "Large" and "Full" configurations are overprovisioned for cost-effective Edge computing platforms. The "Medium" configuration makes effective use of FPGA memory resources and provides the best performance, but exhibits an unfavourable cost-benefit trade-off with respect to the "Small 256" configuration.

Finally, the chapter projects some optimization possibilities from the observation of the more evident bottlenecks. For instance, the frame rate on the target platform is currently hardware-dominated: it achieves roughly 5 fps for ResNet-50, potentially extended up to 8 fps, while future optimizations of hardware performance (including synthesis for ASIC) will bring a software-limited throughput of 20 fps to the forefront.

References

1. J. Chen and X. Ran: Deep Learning With Edge Computing: A Review. *Proceedings of the IEEE*, vol. 107, n. 8, 1655-1674pp (2019). doi:10.1109/JPROC.2019.2921977
2. R. DiCecco and G. Lacey and J. Vasiljevic and P. Chow and G. Taylor and S. Areibi: Caffeinated FPGAs: FPGA framework For Convolutional Neural Networks. *In Proceedings of the 2016 International Conference on FPT*, 265-268pp (2016). doi:10.1109/FPT.2016.7929549
3. F. Farshchi and Q. Huang and H. Yun: Integrating NVIDIA Deep Learning Accelerator (NVDLA) with RISC-V SoC on FireSim. arXiv preprint, (2019). arXiv: 1903.06495v2
4. H. Genc and A. Haj-Ali and V. Iyer and A. Amid and H. Mao and J. Wright and C. Schmidt and J. Zhao and A. Ou and M. Banister and Y. S. Shao and B. Nikolic and I. Stoica and K. Asanovic: Gemmini: An Agile Systolic Array Generator Enabling Systematic Evaluations of Deep-Learning Architectures. arXiv preprint, (2019). arXiv:1911.09925
5. A. Gonzalez and C. Hong: A Chipyard Comparison of NVDLA and Gemmini. Available online at: charleshong3.github.io/projects/nvdla_v_gemmini.pdf
6. C. Guoyu and P. Zhenjiang and F. Shanggong and W. Dawei and C. Jingwen and Z. Shengang: Research on the Architecture of Edge Computing SoC with Ultra-Low Power. *In Proceedings of the 2020 IEEE 3rd International Conference on Electronics Technology (ICET)*, 54-57pp (2020). doi:10.1109/ICET49382.2020.9119600
7. A. Krizhevsky and I. Sutskever and G. E. Hinton: ImageNet Classification with Deep Convolutional Neural Networks. *In Proceedings of Advances in Neural Information Processing Systems*, 84-90pp (2012).
8. W. Lin and C. Hsieh and C. Chou: ONNC-Based Software Development Platform for Configurable NVDLA Designs. *In Proceedings of the 2019 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 1-2pp (2019). doi:10.1109/VLSI-DAT.2019.8741778

9. W. Lin and D. Tsai and L. Tang and C. Hsieh and C. Chou and P. Chang and L. Hsu: ONNC: A Compilation Framework Connecting ONNX to Proprietary Deep Learning Accelerators. *In Proceedings of the 2019 IEEE International Conference on AICAS*, 214-218pp (2019). doi:10.1109/AICAS.2019.8771510
10. S. -M. Liu and L. Tang and N. -C. Huang and D. -Y. Tsai and M. -X. Yang and K. -C. Wu: Fault-Tolerance Mechanism Analysis on NVDLA-Based Design Using Open Neural Network Compiler and Quantization Calibrator. *In Proceedings of the 2020 International Symposium on VLSI Design, Automation and Test (VLSI-DAT)*, 1-3pp (2020). doi:10.1109/VLSI-DAT49148.2020.9196335
11. L. Lu and Y. Liang and Q. Xiao and S. Yan: Evaluating Fast Algorithms for Convolutional Neural Networks on FPGAs. *In Proceedings of the 25th IEEE International Symposium on FCCM*, 101-108pp (2017). doi:10.1109/FCCM.2017.64
12. S. Luo: Customization of a Deep Learning Accelerator. *In Proceedings of the 2019 International Symposium on VLSI-DAT*, 1-2pp (2019). doi:10.1109/VLSI-DAT.2019.8741855
13. T. Moreau and T. Chen and Z. Jiang and L. Ceze and C. Guestrin and A. Krishnamurthy: VTA: An Open Hardware-Software Stack for Deep Learning. arXiv preprint, (2018). abs/1807.04188
14. J. Qiu and J. Wang and S. Yao and K. Guo and B. Li and E. Zhou and J. Yu and T. Tang and N. Xu and S. Song and Y. Wang and H. Yang: Going Deeper with Embedded FPGA Platform for Convolutional Neural Network. *In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 26-35pp (2016). doi:10.1145/2847263.2847265
15. A. Shawahna and S. M. Sait and A. El-Maleh: FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review. *IEEE Access*, vol. 7, 7823-7859pp (2019). doi:10.1109/ACCESS.2018.2890150
16. N. Suda and V. Chandra and G. Dasika and A. Mohanty and Y. Ma and S. Vrudhula and J. S. Seo and Y. Cao: Throughput-Optimized OpenCL-Based FPGA Accelerator for Large-Scale Convolutional Neural Networks. *In Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 16-25pp (2016). doi:10.1145/2847263.2847276
17. V. Sze and Y. H. Chen and T. J. Yang and J. S. Emer: Efficient Processing of Deep Neural Networks: A Tutorial and Survey. *Proceedings of the IEEE*, vol. 105, n. 12, 2295-2329pp (2017). doi:10.1109/JPROC.2017.2761740
18. C. Szegedy and W. Liu and Y. Jia and P. Sermanet and S. Reed and D. Anguelov and D. Erhan and v. Vanhoucke and A. Rabinovich: Going Deeper with Convolutions. *In Proceedings of 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 1-9pp (2014). doi:10.1109/CVPR.2015.7298594
19. P. Udupa and G. Mahale and K. K. Chandrasekharan and S. Lee: Accelerating Depthwise Convolution and Pooling Operations on z-First Storage CNN Architectures. *In Proceedings of the 2020 IEEE International Symposium on Circuits and Systems (ISCAS)*, 1-5pp (2020). doi:10.1109/ISCAS45731.2020.9180863
20. A. Veronesi and M. Krstic and D. Bertozzi: Cross-Layer Hardware/Software Assessment of the Open-Source NVDLA Configurable Deep Learning Accelerator. *In Proceedings of the 28th IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*, 1-6pp (2020).
21. D. Wang and K. Xu and D. Jiang: PipeCNN: An OpenCL-based open-source FPGA accelerator for convolution neural networks. *In Proceedings of the 2017 International Conference on FPT*, 279-282pp (2017). doi:10.1109/FPT.2017.8280160

22. Z. Xu and J. Abraham: Design of a Safe Convolutional Neural Network Accelerator. *In Proceedings of* the 2019 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 247-252pp (2019). doi:10.1109/ISVLSI.2019.00053
23. Z. Xu and J. Abraham: Safety Design of a Convolutional Neural Network Accelerator with Error Localization and Correction. *In Proceedings of the 2019 IEEE International Test Conference (ITC)*, 1-10pp (2019). doi:10.1109/ITC44170.2019.9000149
24. S. S. Yakun and J. Clemons and R. Venkatesan and B. Zimmer and M. Fojtik and N. Jiang and B. Keller and A. Klinefelter and N. Pinckney and P. Raina and S. G. Tell and Y. Zhang and W. J. Dally and J. Emer and C. T. Gray and B. Khailany and S. W. Keckler: Simba: Scaling Deep-Learning Inference with Multi-Chip-Module-Based Architecture. *In Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 14-27pp (2019). doi:10.1145/3352460.3358302
25. J. Zhang and J. Li: Improving the Performance of OpenCL-Based FPGA Accelerator for Convolutional Neural Network. *In Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 25-34pp (2017). doi:10.1145/3020078.3021698
26. X. Zhang and J. Wang and C. Zhu and Y. Lin and J. Xiong and W. M. Hwu and D. Chen: DNNBuilder: An Automated Tool for Building High-Performance DNN Hardware Accelerators for FPGAs. *In Proceedings of the 2018 International Conference on Computer-Aided Design*, 1-8pp (2018). doi:10.1145/3240765.3240801
27. G. Zhou and J. Zhou and H. Lin: Research on NVIDIA Deep Learning Accelerator. *In Proceedings of 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification*, 192-195pp (2018). doi:10.1109/ICASID.2018.8693202
28. C. Zhuge and X. Liu and X. Zhang and S. Gummadi and J. Xiong and D. Chen: Face recognition with hybrid efficient convolution algorithms on FPGAs. *In Proceedings of the 2018 GLSVLSI Great Lakes Symposium on VLSI*, 123-128pp (2018). doi:10.1145/3194554.3194597
29. [Internet]: GitHub issue #110: NVDLA running on a FPGA platform;
Available at: github.com/nvdla/hw/issues
30. [Internet]: NVIDIA Jetson modules;
Available at: nvidia.com/autonomous-machines/embedded-systems
31. [Internet]: NVIDIA TensorRT library;
Available at: developer.nvidia.com/tensorrt
32. [Internet]: NVDLA open source project;
Available at: nvdla.org
33. [Internet]: NVDLA low precision support;
Available at: github.com/nvdla/sw/blob/v1.2.0-OC/LowPrecision.md
34. [Internet]: RISC-V Foundation;
Available at: riscv.org