Bare-Metal RISC-V + NVDLA SoC for Efficient Deep Learning Inference

Vineet Kumar, Ajay Kumar M, Yike Li, Shreejith Shanker[†], Deepu John

School of Electrical and Electronic Engineering, University College Dublin, Dublin, Ireland

[†]Department of Electronic and Electrical Engineering, Trinity College Dublin , Dublin, Ireland

vineet.bitsp@gmail.com, ajay.kumarm@ucdconnect.ie, yike.li@ucdconnect.ie, shankers@tcd.ie, deepu.john@ucd.ie

Abstract—**This paper presents a novel System-on-Chip (SoC) architecture for accelerating complex deep learning models for edge computing applications through a combination of hardware and software optimisations. The hardware architecture tightly couples the open-source NVIDIA Deep Learning Accelerator (NVDLA) to a 32-bit, 4-stage pipelined RISC-V core from Codasip® called μRISC_V. To offload the model acceleration in software, our toolflow generates bare-metal application code (in assembly), overcoming complex OS overheads of previous works that have explored similar architectures. This tightly coupled architecture and bare-metal flow leads to improvements in execution speed and storage efficiency, making it suitable for edge computing solutions. We evaluate the architecture on AMD's ZCU102 FPGA board using NVDLA-small configuration and test the flow using LeNet-5, ResNet-18 and ResNet-50 models. Our results show that these models can perform inference in 4.8 ms, 16.2 ms and 1.1 s respectively, at a system clock frequency of 100 MHz.**

Index Terms—**System-on-chip, RISC-V, NVDLA, Hardware accelerators, Deep learning, FPGA**

I. Introduction

The growing computational demands of AI workloads and the limitations of edge devices have driven the need for specialized hardware accelerators. The rise of open-source hardware has enabled the development of accelerators like the NVIDIA Deep Learning Accelerator (NVDLA) [1]–[4]. NVDLA is a scalable, configurable, open-source inference engine suited for edge AI. Its integration with RISC-V presents a compelling solution for deep learning acceleration, as explored in several studies.

Previous works [5]–[7] have examined the integration of RISC-V with NVDLA to enhance deep learning inference efficiency and flexibility. However, these studies primarily focus on simulation-based implementations rather than real hardware deployments, such as FPGAs. In [8], an FPGA-based prototype incorporating multiple instances of NVDLA and a RISC-V core is presented, but details on resource utilization and integration methodologies are not provided. Additionally, these studies [5]–[8] rely on a Linux-based kernel to execute neural network models, requiring NVDLA drivers and resulting in significant software overhead. Few works [10]–[12] have demonstrated FPGA-based implementations of NVDLA integrated with existing processor cores in SoCs, often utilizing Linux-based environments such as PetaLinux [10]. Other works [13]–[16] have explored the use of NVDLA in various research applications but without a focus on integrating the accelerator with a RISC-V core. The reliance on Linux kernel for executing deep learning workloads introduces additional performance and storage overhead, making these solutions less suitable for resource-constrained edge devices.

In this paper, we present the design of an open-source NVDLA and RISC-V based SoC, which takes a neural network model as input and executes it on NVDLA using RISC-V assembly code without relying on a Linux kernel. Moreover, the SoC is demonstrated on FPGA by running neural network models. Instead of using a Linux kernel-managed driver stack, we leverage configuration files (traces) to directly configure NVDLA's registers, serving as an execution control sequence. The official NVDLA release provides pre-generated configuration files for basic hardware tests (e.g., sanity checks, convolution, and pooling layer tests). However, no guidelines are available on how these files were generated or how to create them for arbitrary neural networks. This work addresses this gap by proposing a methodology to generate configuration files for arbitrary Caffe-based neural networks. These files are then converted into RISC-V assembly code, enabling direct hardware configuration of NVDLA. The key contributions of this work include:

– *Design of an SoC architecture based on NVDLA and RISC-V and its implementation on FPGA*
– *Automated generation of configuration files and weight extraction for arbitrary Caffe neural network models* [1]
– *Tightly coupled hardware architecture and bare-metal assembly-based execution, eliminating the need for a Linux kernel and additional storage*

For system design, we integrate NVDLA with a Codasip μRISC_V core and implement the design on an AMD ZCU102 FPGA board. The system is validated using LeNet-5, ResNet-18, and ResNet-50 neural network models.

Corresponding Author: Vineet Kumar. This work was partly funded by Taighde Éireann – Research Ireland through the Research Ireland Centre for Research Training in Machine Learning (18/CRT/6183)

[1] https://github.com/vineetbitsp/riscv-nvdla-sw

II. Related Works

Several prior studies have explored the integration and evaluation of NVDLA within different computing environments, primarily focusing on simulation-based approaches and Linux-kernel based FPGA implementations. Gem5-NVDLA [6] serves as a valuable tool for analyzing design tradeoffs and evaluating NVDLA's performance in a simulated environment. However, this work is limited in scope, as it does not support the small configuration of NVDLA (nv_small) [4]. Gonzalez and Hong [7] conducted a comparative study of the NVDLA and Gemmini accelerators within the Chipyard framework, assessing their respective advantages for deep learning workloads. While insightful, this work is framework-specific, restricting its applicability to Chipyard users.

Farshchi et al. [5] investigated the integration of NVDLA with RISC-V-based SoCs using FireSim, a cycle-accurate simulation platform, to evaluate performance in object detection tasks. However, their study is limited to simulation-based analysis and does not address the practical challenges of FPGA-based deployment or a custom ASIC design. Notably, their simulation assumes an unrealistic NVDLA operating frequency of 3.2 GHz—equivalent to the CPU clock—due to FireSim's constraints, whereas in practical FPGA implementations, NVDLA operates at frequencies below 100 MHz [8].

Giri et al. [8] proposed an open-source embedded system platform for agile heterogeneous SoC design and demonstrated FPGA-based prototypes incorporating multiple instances of NVDLA alongside the Ariane RISC-V 64-bit processor core. However, their work does not provide details on FPGA resource utilization or integration methodologies.

To the best of our knowledge, all prior works [5]–[12] require a Linux kernel to configure and operate NVDLA. In contrast, our work employs bare-metal assembly programming to directly configure NVDLA registers for a given neural network. Furthermore, our implementation supports both small and full configurations (nv_small and nv_full) of NVDLA.

III. The Proposed system

This section outlines the architecture of the proposed SoC, followed by detailed software and hardware development in the subsequent section. Fig. 1 presents the software generation workflow, which converts a trained neural network model into RISC-V assembly code and a corresponding weight file. As this process is model-specific and performed only once, it is executed offline using NVDLA's virtual platform (VP) in conjunction with the software development methodology described in Section IV-B.

Fig. 2 illustrates the architectural design of the proposed SoC. The system integrates the NVDLA accelerator with a μRISC-V core through a system bus, an arbiter, and a custom NVDLA wrapper. The system bus—comprising an internal decoder and arbitration logic—enables communication between the μRISC-V core and two memory-mapped slave devices: the NVDLA engine and DRAM-based data memory. Given the shared access to data memory, an arbiter manages potential conflicts between the core and NVDLA. The NVDLA wrapper encapsulates the accelerator hardware alongside interface bridges and a data width converter to address mismatches between the μRISC-V and NVDLA interfaces. Specifically, an AXI data width converter connects the NVDLA's 64-bit data backbone (DBB) interface to the 32-bit data memory. The μRISC-V core employs an AHB-Lite interface for access to both program and data memory. Communication with NVDLA's configuration space bus (CSB) requires an AHB-Lite to APB bridge, leveraging the existing APB-to-CSB adapter provided by the NVDLA package. The AHB-APB bridge, available as an open-source ARM design, facilitates this integration. Furthermore, an AHB-Lite to AXI bridge enables connectivity between the core and AXI-compliant data memory. The system bus decoder assigns distinct address spaces to each slave device (NVDLA and DRAM) to ensure efficient memory-mapped communication.

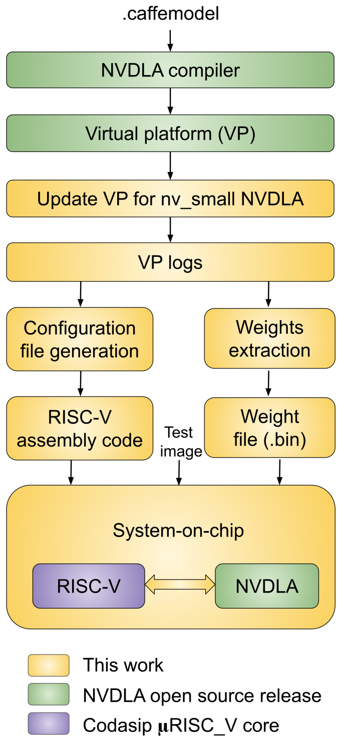

Fig. 1. The proposed system and software development flow.

IV. Methodology

A. Hardware Development Workflow

1) *NVDLA Hardware Generation*: Parameterized Verilog code from the official NVDLA GitHub repository [17] is used to generate hardware configurations via the hardware tree build process, as outlined in the documentation [4].
2) *System Integration*: The NVDLA was integrated with a RISC-V processor using Codasip Studio. A custom wrapper component was developed to encapsulate the NVDLA core, interface bridges, and data-width converters, ensuring seamless compatibility. The system bus, arbiter, and

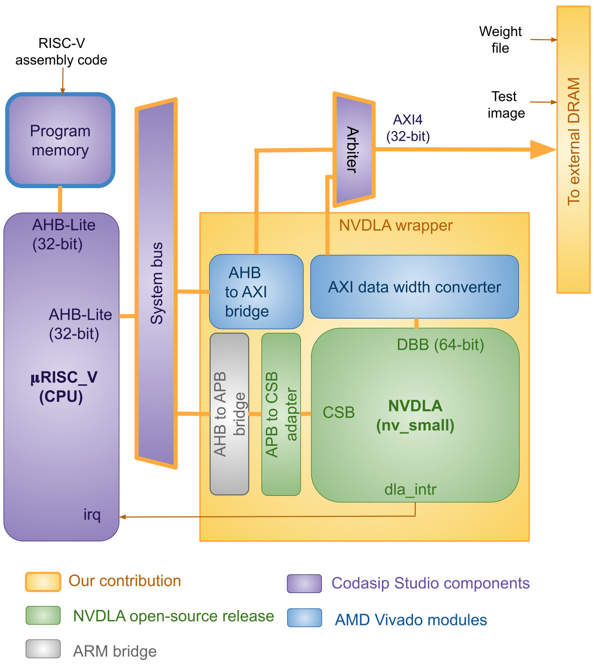

Fig. 2. The system-on-chip.

memory were interconnected through Codasip Studio's `testbench` construct to generate synthesizable RTL, which was subsequently imported into Vivado design suite (Fig. 2). The system bus decoder allocates two dedicated address spaces for the slave devices:

- *NVDLA:* Address range `0x0 -- 0xFFFFF`, covering all configuration register addresses of the NVDLA
- *DRAM:* Address range `0x100000 -- 0x200FFFFF`, providing access to 512 MB of DRAM data memory

This memory mapping enables the RISC-V processor to program the NVDLA using its standard load and store instructions for writing configuration registers and reading their status, eliminating the need for custom RISC-V instructions. The arbiter component coordinates DRAM access between the NVDLA (via its DBB interface) and the RISC-V processor (via its AHB interface), ensuring mutual exclusion and efficient memory utilization. This tightly coupled hardware interface enables bare-metal assembly programming for neural network execution.

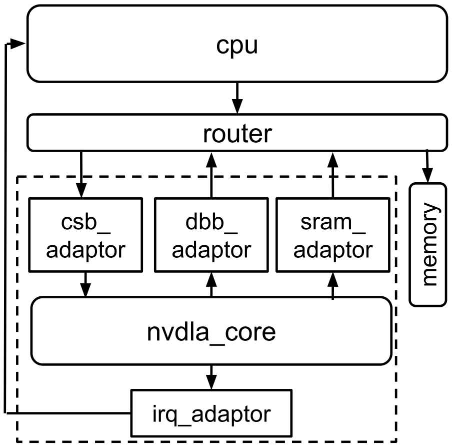

Fig. 3. NVDLA virtual platform.

3) *Simulation and FPGA Prototyping*: Behavioral simulation was performed in Vivado using RTL from the previous step along with Vivado IP cores for bridges and converters, while software binaries and neural network weights were loaded into memory. After successful simulation, the design was synthesized and deployed on the FPGA board, utilizing the onboard DDR memory for input and weight storage. Various DNN models were executed to evaluate system performance.

B. Software Development Flow

The software flow generates RISC-V machine code and extracts neural network weights from a Caffe model (Fig. 1). The Github repository[1] provides Python scripts, Linux commands, and detailed instructions for generating bare-metal RISC-V software through the following steps:

1) *Execution on Virtual Platform*: The Caffe model is compiled using the NVDLA compiler and executed on NVDLA's VP, which provides a cycle-accurate co-simulation environment using QEMU and SystemC [4]. Interface-level transactions (CSB, DBB) are logged during execution (Fig 3).
2) *Configuration File Generation*: A Python script processes the VP log file by extracting lines containing the keyword `nvdla.csb_adaptor`. Each entry represents a register transaction, categorized as read or write based on the `iswrite` flag:
   - Read operations (`iswrite=0`) are converted into `read_reg` commands, which store the expected register values.
   - Write operations (`iswrite=1`) are converted into `write_reg` commands, specifying the target register address and the corresponding data value.

   The generated command sequence constitutes the configuration file, which is subsequently converted into RISC-V assembly code. The assembly code is compiled into machine code using the RISC-V core SDK in *Codasip Studio* and loaded into program memory for execution.
3) *Weight Extraction*: To extract neural network weights, the python script filters VP log entries containing the keyword `nvdla.dbb_adaptor`. Each entry corresponds to a data transaction:
   - Read operations (`iswrite=0`) indicate memory fetches, which correspond to weights.
   - Write operations (`iswrite=1`) specify addresses and values being written to memory.

   Finally, duplicate address entries in the weight file are deleted by retaining the first occurrence, as they are the original weights.

[1] https://github.com/vineetbitsp/riscv-nvdla-sw

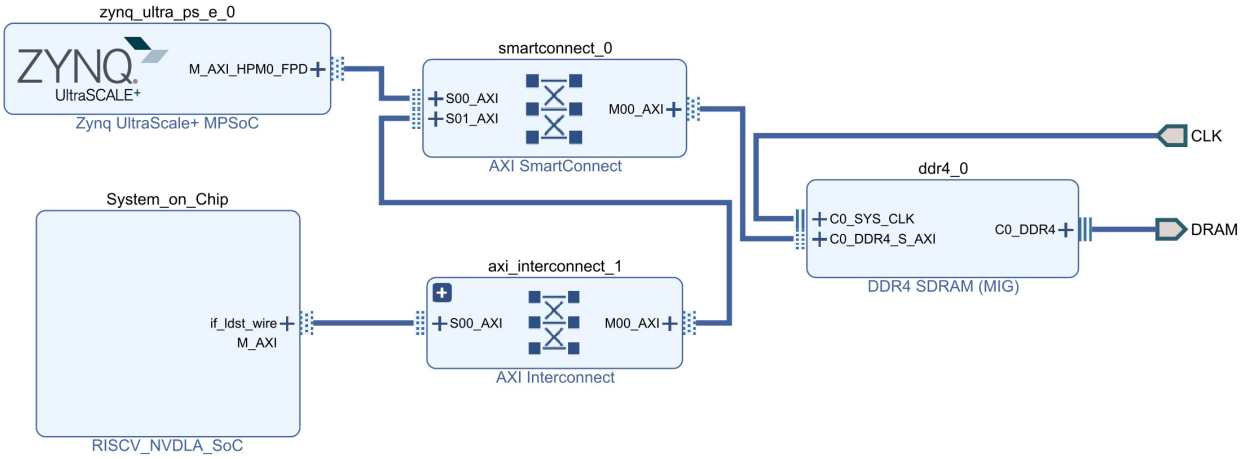

Fig. 4. Set-up to test our SoC (Vivado block design of overall system set-up).

TABLE I
FPGA RESOURCE UTILIZATION (AMD'S ZCU102 EVALUATION BOARD)

Major Components	**CLB LUTs**	**CLB Regs**	**CARRY8**	**F7 Muxes**	**F8 Muxes**	**CLBs**	**BRAM Tiles**	**DSPs**
(FPGA)	(274080)	(548160)	(34260)	(137040)	(68520)	(34260)	(912)	(2520)
Overall System Set-up (Fig. 4)	96733	102823	1825	3719	1133	19898	323.5	39
MIG DDR4	8651	10260	56	164	0	1754	25.5	3
AXI SmartConnect	5546	7860	0	0	0	1137	0	0
Our SoC (Fig. 2)	81986	83659	1762	3555	1133	17025	298	36
nv_small NVDLA	74575	79567	1569	3091	1048	15734	66	32
uRISC_V core	6346	2767	173	419	67	1297	0	4
Program Memory	241	6	0	45	18	148	232	0

TABLE II
EVALUATION OF OUR SOC CONTAINING NV_SMALL NVDLA (FPGA IMPLEMENTATION RESULTS)

Model	**Layers**	**Input**	**Model Size**	**Proc. Time @100MHz**	**Proc. Time @50MHz** [8]
LeNet-5	9	1×28×28	1.7 MB	**4.8 ms**	263 ms
ResNet-18	86	3×32×32	0.8 MB	**16.2 ms**	NA
ResNet-50	228	3×224×224	102.5 MB	**1.1 s**	2.5 s

TABLE III
EVALUATION OF OUR SOC CONTAINING NV_FULL NVDLA (SIMULATION RESULTS)

Model	**Input size**	**Model size**	**Number of clock cycles**	**Processing time @100 MHz (ms)**
LeNet-5	1x28x28	1.7 MB	143188	1.4
ResNet-18	3x32x32	813.5 KB	324387	3.2
ResNet-50	3x224x224	102.5 MB	26565315	265
MobileNet	3x224x224	17 MB	22525704	220
GoogleNet	3x224x224	53.5 MB	40889646	408
AlexNet	3x227x227	243.9 MB	35535582	355

V. EVALUATION AND TESTING

This section presents the performance evaluation of the proposed SoC, including its FPGA implementation and testing with standard neural network models. Initial functional validation was performed via behavioral simulation using standard NVDLA test traces such as sanity, convolution and memory tests available from the NVDLA Github repository. These were translated into RISC-V assembly and used to verify the correctness of the integrated SoC design.

To support larger models, external DRAM was connected to the SoC through a DDR4 memory controller (MIG DDR4), and is initialized via the ARM core of the Zynq UltraScale+ MPSoC on the ZCU102 board. This configuration enables access to 512 MB of DDR4 memory from the programmable logic. The system architecture was implemented in AMD Vivado, with a high-level interface diagram shown in Fig. 4. The Zynq core initializes the DRAM with both the weight file and input image. At any given time, the DRAM is connected either to the Zynq core or the SoC using an AXI SmartConnect, which functions as a multiplexer. Additionally, an AXI Interconnect is placed between the SoC and MIG DDR4 to reconcile frequency mismatches, since the SoC operates at 300

MHz while the DDR4 runs at 100 MHz. The complete block design was synthesized and deployed on the FPGA.

The SoC was successfully tested with standard deep learning models, including LeNet-5, ResNet-18, and ResNet-50. During execution, the DRAM is preloaded with weight and image files in `.bin` format. The RISC-V program memory, implemented using FPGA block RAMs, is loaded with machine code generated from the configuration file in `.mem` format.

While Table I shows the FPGA resource utilization for the complete system set-up, our SoC, and its major components, Table II reports the execution times at a system clock frequency of 100 MHz. The execution speed outperforms previous work, where NVDLA was integrated on a 64-bit RISC-V–based platform, as shown in the table. Table III presents simulation results for the `nv_full` configuration of NVDLA, including total cycle counts and processing times at 100 MHz. Although the `nv_full` configuration delivers higher performance than `nv_small`, it is an enormous design and does not fit on most FPGAs, including the ZCU102 FPGA board used in this work. For this device, the LUTs overutilization was quite substantial for `nv_full` as observed during synthesis.

The `nv_small` configuration supports only INT8 precision, while `nv_full` additionally supports FP16 computations. The models included in the Table III shows computation times with FP16 precision. The performance comparison of ResNet-50 on both configurations highlights that `nv_full` is substantially faster, as it integrates a larger number of MAC units. A limitation of this work is that the `nv_small` configuration currently supports only a limited set of models, primarily due to the lack of INT8 calibration tables. Future work will address this limitation to broaden model support.

Future Work

Future development will focus on extending model support for the nv_small configuration to include additional deep learning models such as MobileNet, GoogleNet, and AlexNet. Two promising directions are:

1) Generating INT8 calibration tables required by the NVDLA compiler, which are not currently provided but are partially described in the NVDLA GitHub documentation [17].
2) Integrating the ONNC compiler [18] to generate NVDLA-compatible loadable files from ONNX models, enabling broader deployment through execution on the NVDLA VP.

VI. Conclusion

This work presents a custom SoC integrating the NVDLA accelerator with a RISC-V processor, implemented and validated on an FPGA platform. The current design leverages the nv_small configuration, with the flexibility to support nv_full by modifying parameters such as the AXI interface width (e.g., from 64-bit to 512-bit). The SoC operates without the need for a Linux kernel, enabling a lightweight, standalone execution model ideal for edge AI applications requiring low latency and constrained resources. FPGA synthesis results demonstrate the feasibility of this design on low- to mid-range devices.

References

[1] K. Asanović and D.A. Patterson, ”Instruction sets should be free: The case for risc-v,”EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, 2014.

[2] G. Gupta, T. Nowatzki, V. Gangadhar, and K. Sankaralingam, ”Kickstarting semiconductor innovation with open source hardware,” IEEE Computer, vol. 50, no., pp. 50–59, June 2017.

[3] S. Greengard, ”Will RISC-V revolutionize computing?,” Communication of ACM, vol. 63, no. 5, pp. 30–32, April 2020.

[4] NVIDIA Deep Learning Accelerator (NVDLA) open-source project, 2017. Available at https://nvdla.org. (Accessed on 23/07/2025)

[5] F. Farshchi, Q. Huang and H. Yun, ”Integrating NVIDIA deep learning accelerator (NVDLA) with RISC-V SoC on FireSim,” 2nd Workshop on Energy Efficient Machine Learning and Cognitive Computing for Embedded Applications, Washington, DC, USA, 2019, pp. 21-25.

[6] C. Lai and W. Zhang, ”Gem5-NVDLA: A simulation framework for compiling, scheduling, and architecture evaluation on AI system-on-chips,” ACM Trans. Des. Autom. Electron. Syst., vol. 29, no. 5, Article 84, 20 pages, September 2024.

[7] A. Gonzalez, and C. Hong, ”A Chipyard comparison of NVDLA and Gemmini,” Berkeley, CA, USA, Tech. Rep. EE, pp.290-2, 2020.

[8] D. Giri, K.L. Chiu, G. Eichler, P. Mantovani, N. Chandramoorth, and L.P. Carloni, ”Ariane+ NVDLA: Seamless third-party IP integration with ESP,” In Workshop on Computer Architecture Research with RISC-V, May 2020.

[9] H. Afzali-Kusha and M. Pedram, ”X-NVDLA: Runtime accuracy configurable NVDLA based on applying voltage overscaling to computing and memory units,” IEEE Transactions on Circuits and Systems I: Regular Papers, vol. 70, no. 5, pp. 1989-2002, May 2023.

[10] S. Ramakrishnan, Implementation of a deep learning inference accelerator on the FPGA, 2020. Master's thesis available at https://lup.lub.lu.se/luur/download?func=downloadFile&recordOId=9007070&fileOId=9007133 (Accessed on 23/07/2025)

[11] G. Cesarano, FPGA implementation of a deep learning inference accelerator for autonomous vehicles, 2018. Master's thesis available at https://webthesis.biblio.polito.it/9033/ (Accessed on 23/07/2025)

[12] Zynq-NVDLA. Available at https://github.com/LeiWang1999/ZYNQ-NVDLA (Accessed on 23/07/2025)

[13] M.T. Sanic, C. Guo, J. Leng, M. Guo, and W. Ma, ”Towards reliable AI applications via algorithm-based fault tolerance on NVDLA,” In Proc. of 18th IEEE International Conference on Mobility, Sensing and Networking, pp. 736-743, December 2022.

[14] L. Liu, Z. Ren and T. Chong, ”Research and optimization of neural network accelerator based on NVDLA,” In Proc. of 7th International Conference on Control Engineering and Artificial Intelligence, pp. 37-42, January, 2023.

[15] Y. Chen, D. Ma, and J. Mao, ”A hardware accelerator for sparse computing based on NVDLA,” In Proc. of IEEE Conference of Science and Technology for Integrated Circuits, pp. 1-3, March, 2024.

[16] M. Meena, D. Vaithiyanathan, P. Verma, and B. Kaur, ”Hardware analysis on NVDLA using ResNet50,” In Proc. of IEEE International Conference on Advances in Modern Age Technologies for Health and Engineering Science, pp. 1-5, May, 2024.

[17] NVDLA Github page, 2017. Available at https://github.com/nvdla/ (Accessed on 23/07/2025)

[18] ONNC NVDLA Github page, 2020. Available at https://github.com/ONNC/onnc (Accessed on 23/07/2025)