UML-Based Design Flow for Systems with Neural Networks

Daniel Suarez
TEISA dept.
Universidad de Cantabria
Santander, Spain
suarezd@teisa.unican.es

Hector Posadas
TEISA dept.
Universidad de Cantabria
Santander, Spain
ORCID: 0000-0002-1427-7524

Víctor Fernández
TEISA dept.
Universidad de Cantabria
Santander, Spain
ORCID: 0000-0003-0614-151X

***Abstract*—Artificial intelligence has demonstrated its ability to solve lots of critical tasks, but at the cost of high computational requirements. Different hardware has been proposed to provide this computational power, each one with its benefits and drawbacks. However, the exploration of the different alternatives in an easy an integrated way is still a complex task. To solve so, this paper proposes a UML-based design flow where neural networks are initially specified and then automatically generated and trained using TensorFlow. The approach also enables automatic mapping of models to CPU, GPU and FPGAs, using Xilinx's Deep Learning Processor Units (DPUs). The framework also generates the communication codes required to connect the other system components with the implementation selected. This approach addresses design-space exploration challenges, system architecture definition, and improves implementation and training processes by saving time and effort.**

***Index Terms*—AI, CNN, FPGA, UML, automatic generation, design space exploration**

I. Introduction

Artificial Intelligence (AI) has emerged as a critical research and development area due to the impressive performance of Deep Neural Networks in multiple applications [1]. However, these capabilities come with high computational requirements. Satisfying them is a complex task. High-end microprocessors incorporate multiple CPU cores and co-processors for parallel operations, such as SIMD [2] instructions and GPUs. AI-specific implementations on FPGAs have also been proposed.

AI applications on cloud-based systems have limitations in terms of real-time needs and data privacy, leading to a trend of deploying AI models directly to edge devices. This approach offers advantages like reduced latency, improved privacy and security, and cost savings [3]. Engineers face challenges in mapping AI components to resources, especially with FPGAs.

The difficulty of using FPGAs limits their widespread adoption, as they require hardware description languages like Verilog or VHDL. High-level synthesis (HLS) tools provide an alternative, allowing code generation from languages like C/C++ [3]. However, working at higher abstraction levels can reduce circuit efficiency and demands a good understanding of the underlying hardware layer and tool capabilities.

In the AI field, simplicity is favored, with frameworks like TensorFlow [4], PyTorch, and Keras in Python being widely used. However, the difference between VHDL and Python programming poses an additional challenge.

FPGA implementations offer more alternatives than CPUs or GPUs, making the exploration of cost and performance complex. A holistic approach that considers application details and platform capabilities is necessary. Flexible design flows capable of generating multiple implementations across different hardware resources (CPUs, GPUs, FPGAs) with low design effort are required, covering the entire design process from specification to implementation.

This paper proposes a UML-based design flow where neural networks are initially specified and then automatically generated and trained using TensorFlow. The approach also enables automatic mapping of models to FPGAs using Xilinx's Deep Learning Processor Units (DPUs) [5]. This approach addresses design-space exploration challenges, system architecture definition, and improves implementation and training processes by saving time and effort.

In summary, this paper introduces improvements in specifying neural networks in UML and automating the generation, training, and implementation of network models on different devices, including FPGA implementations.

II. State of the art

Deep neural networks (DNN) are commonly implemented on CPU, GPU, and FPGA devices. Tools like TensorRT [20], ONNX Runtime [21], TVM [22], nGraph [23], Core ML [24], and OpenVINO [25] facilitate the generation and optimization of AI models for various hardware platforms. However, the use of FPGAs in these tools is not fully covered, leading to the emergence of research solutions that span cloud-based and edge-based deployments.

In cloud-based deployments, [6] surveys FPGAs in the cloud and introduces the Open Cloud Testbed, which incorporates network-attached FPGAs. At the edge level, [2] and [1] propose hardware and software co-design implementations on the Ultrascale+ ZCU102 platform.

This work has been supported by Project PID2020-116417RB-C43, funded by Spanish MCIN/AEI/10.13039/501100011033 and by the KDT JU agreement No 101007273 ECSEL DAIS, funded by EU H2020 and by Spanish pci2021- 121988

979-8-3503-0385-8/23/31.002023$IEEE$

Several works explore application-independent architectures and optimizations. For example, [7] accelerates the writing digital neural network of the MNIST dataset on an FPGA platform, while [8] review computations and optimizations in DNN models across different hardware platforms.

To address the challenge of manual coding, alternatives propose automatic generation of hardware description language (HDL) codes for DNN implementation [9], [10], [11]. Some works integrate hardware accelerators from C codes, as [12] with Xilinx Vitis-HLS and [13] using Xilinx SDAccel for parallel acceleration of DNNs with OpenCL.

Recent approaches with Xilinx FPGAs have shifted from ad-hoc accelerators to using Xilinx's Deep-learning Processing Units (DPUs) for efficient AI service implementation. Various alternatives, including [14], [15], [16], and [17], employ DPU-based CNN models on FPGAs for hardware acceleration.

Using DPUs enables a systematic approach to explore different design alternatives. [18] analyzes runtime, energy consumption, and tradeoffs in accuracy, runtime, cost, and energy consumption when using different DNN topologies, DPU configurations, and FPGA models.

However, there is a lack of approaches that integrate FPGA-based accelerators with CPUs and GPUs in the entire system, particularly with the software subsystem, and explore mapping alternatives for DNNs across different resources. This paper proposes automatic generation from UML models as a foundation for this exploration process. Previous works have explored using UML models as inputs for different steps of the design process, as virtual simulation [19] and code synthesis [27], but not specifically for DNNs or component implementation on different resources, including FPGAs. The idea of specifuing DNNs directly in the UML model, instead of using sparated solutions [27] enables integrating the DNN design flow together with the rest of the system design. This solution improves activities such as maintenance or system redesign, since less files and tools are required.

Design space exploration and automatic code synthesis from UML models have received significant interest in recent years, although none of the existing approaches focus on DNNs or component implementation on diverse resources like FPGAs. The subsequent sections present a novel approach that addresses these objectives.

III. Proposed design flow

In the context of automating the generation and deployment of deep learning solutions across various platforms, several challenges in neural network and runtime software generation need to be addressed. The proposed design flow aims to be a comprehensive solution to these challenges and an overview is shown in Fig. 1. The design flow comprises three phases labeled on the figure 1 as system specification, software generation, and platform deployment.

The proposed solution employs a UML (Unified Modeling Language) diagram in its first phase for a comprehensive description of neural networks as an integral component of a complete system, encompassing its detailed specifications and training characteristics. This UML diagram provides a graphical representation of the neural network structure, layers, connections, and training configurations.

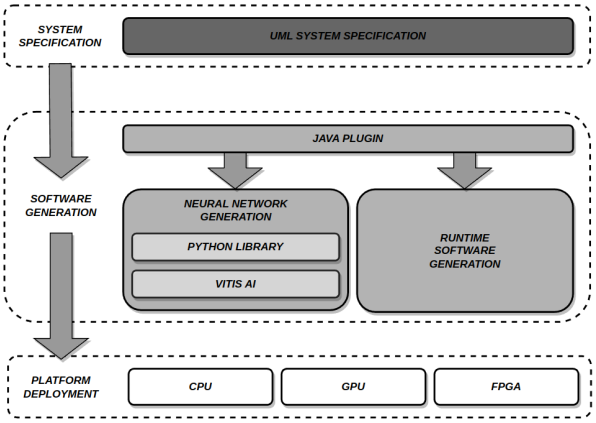

Fig. 1. Design flow.

During the software generation phase, third-party proprietary tools (Vitis AI) and a developed Python library are used to automatically generate the neural network graph and the corresponding runtime software based on the target platform. This automation streamlines the implementation and deployment process, ensuring compatibility and efficiency.

In the platform deployment phase, the generated neural network graph and runtime software are ready for performing data inference on the selected device, enabling the deployment of deep learning solutions across different environments.

The following sections will provide a more detailed explanation of the different phases.

IV. UML Model

The proposed flow begins with generating the UML model of the system, which includes the neural network information along with other functional components, interconnections with external components, hardware mapping, and the environment model. UML modeling is conducted using the Eclipse platform with the Papyrus plugin. The UML model serves as input for a dedicated plugin developed for this research, which generates the necessary configuration files and executable codes for the subsequent design flow steps.

A. UML system modeling

The UML modeling methodology used is based on the approach proposed in [27]. It defines system functionality by specifying the corresponding C++ code files. Communications are represented by service calls encapsulated within interfaces located in ports. Ports define both required and provided services of each component.

This methodology is extended in this research to support the modeling of neural networks as system components. The application is described by creating instances (UML "Properties") of these network components, being their ports interconnected using UML connectors. These connectors represent function-call connections between components requiring services and

the components providing (implementing) them. It is important to note that a component can act both as a client and a server if it has multiple ports for multiple connections.

The model is further enriched by adding the software and hardware resources that constitute the HW/SW platform. The main SW resource is the operating system where different components are assigned, while HW resources include the modeling of processing elements such as GPP, DSP, and GPUs.

Additionally, the methodology allows capturing buses, memories, caches, and other relevant attributes to characterize them. The HW/SW platform is described by creating and interconnecting instances of these HW/SW resources. This additional information is not used within the flow presented in this paper, but it can be used for other purposes, such as documentation or performance simulation.

B. Modeling AI components

In the presented proposal, neural networks are modeled as UML components within the system. These components are specified using the "NNComponent" stereotype and described using a Composite Structure Diagram that represents the network architecture. The diagram consists of multiple properties representing different layers of the network and their characteristics. The layers are interconnected using arrows (abstractions) to indicate their placement in the overall model.

A network component is defined with a single port including the execution of the network inference as a service. The service is defined as a function with three arguments: the number of inputs for inference in a call, a buffer containing the input data, and a buffer for the inference outputs. Buffer sizes are calculated based on the first and last layers of the network. Additionally, in the composite diagram, the port is connected to the first and last layers using arrows (abstractions).

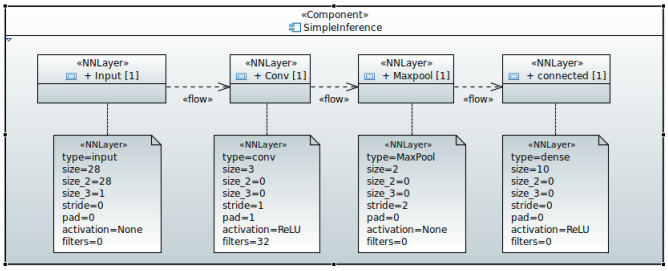

Fig. 2. UML Neural Network component.

The network description can also be hierarchical. In complex network models, blocks of layers can be repeated to increase network accuracy. In this methodology, these blocks can be created as new components representing sub-networks composed of these layers. Each sub-network component has its own diagram describing the block of layers as mentioned above. The sub-network port is optional, but none, one, or two ports can be used to improve the visualization of the network architecture. Layers modeling a network (not a sub-network) are specified with the "NNLayer" stereotype (see an example in Fig. 2), including several parameters to describe the layer details such as type, filters, size, stride, batch normalization, and activation function.

In convolutional network design, a set of six fundamental nodes is proposed to fulfill most computational and topological requirements. These nodes include convolution, connected, maxpool, and averagepool. They can be grouped based on their computation and layer contribution to the network.

The convolution and connected nodes can introduce up to three layers and are responsible for the network's main computation. The max pooling and average pooling nodes introduce a single layer each, specifically dedicated to their respective pooling operations.

In addition to the computational nodes, two topological nodes have been proposed: the residual block and the inception block. These nodes introduce one layer each and offer alternative topological configurations compared to sequential network design. The residual block combines two branches by adding their outputs, while the inception block concatenates the outputs of multiple input branches.

V. Software Generation and Platform Deployment

The second and third phases of the design flow are covered in detail in this section. Various devices are considered for the development of both phases, including CPU, GPU, SoC FPGA, and PCIe FPGA. This variability in the target device results in distinct workflows for both the runtime software and the neural network graph generation process.

A. Runtime Software generation

The generation of runtime software is a crucial step in executing the neural network graph on the target platform. To generate device-specific functional Python/C++ software, a Java plugin has been developed. The UML system specification provides essential information for the generation of runtime software. Once the runtime software is generated, it is used to create software components responsible for tasks such as neural network graph loading, data input handling, inference execution, and output management.

The execution process varies depending on the target device. GPU/CPU implementations utilize the TensorFlow inference engine for graph computation, while FPGA implementations rely on proprietary framework. For FPGA-based devices, the runtime procedure is facilitated by the XIR (Xilinx Intermediate Representation) and VART (Vitis AI Runtime) libraries. These libraries are essential in enabling the following steps:

1) Implementing the DPU on the FPGA platform.
2) Loading the model weights and instructions from the host into the FPGA.
3) Transferring input data from the host to the memory of the FPGA device.
4) Initiating the DPU inference process by sending a starting signal.
5) Waiting for the inference to complete, indicated by an inference end signal from the DPU.
6) Reading the data inferred by the DPU.

The depicted steps in Fig. 3 offer a view of the tasks executed by the runtime software in both SoC FPGA and PCIe FPGA devices. These steps emphasize the collaboration between the CPU and FPGA during the inference process.

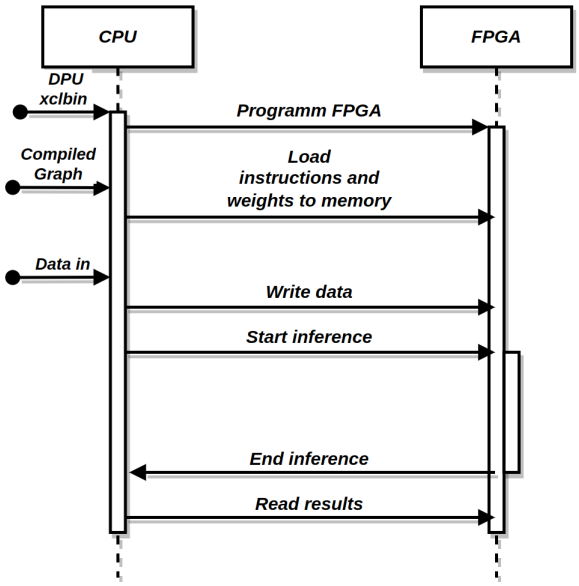

Fig. 3. CPU-FPGA execution flow.

B. Model generation

The course of action for neural network graph generation varies significantly depending on whether it is designed for CPU/GPU or FPGA deployment. Both methodologies share an initial part, but the FPGA-specific workflow includes additional steps. The initial part, shared by both workflows, consists of 4 stages represented by square white boxes in Fig. 4. The resulting model obtained from this part can be directly used on CPU/GPU. For FPGAs, three additional stages numbered from five to seven in Fig. 5 are required. To implement the tasks and stages common to both targets, a Python library and a Java plugin have been developed. For the FPGA-centric process flow, the specific stages are implemented using the Vitis AI framework provided by Xilinx for developing AI solutions on their commercial products.

The first stage, implemented as a Java plugin, translates the UML model specification into several configuration files that serve as a starting point for the next stage. These files capture the details of the desired neural network, including its architecture, layer types, neuron numbers, activation functions, and other training features.

The next stage generates two TensorFlow graphs: one for training, including nodes for cost function, optimization algorithm, and network architecture with trainable parameters; and another graph for inference, without the training-specific nodes. This stage decodes the configuration files into Python objects and generates TensorFlow nodes to form the graphs.

The third stage involves training the generated graph using the specified dataset. After training, a post-processing step adjusts the graph nodes storing the optimized parameters for inference, ensuring consistency between different implementations of the same layer. However, this stage can be skipped if an optimized weight file is provided as part of the UML specification beforehand

In the final stage, a process known as ”weight freezing” is performed. This involves consolidating the optimized weights or parameters obtained during training and replacing the *tf.variable* nodes in the graph with *tf.constant* nodes. This ensures that the model's weights remain constant during inference, reducing memory consumption and improving efficiency. The resulting network model can be used on CPUs and GPUs, while FGPAs requires additional steps.

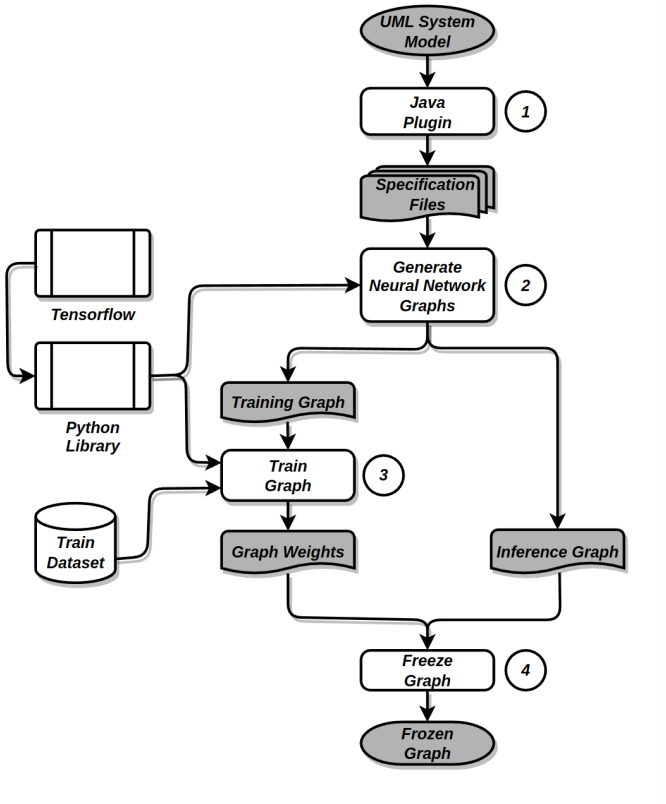

Fig. 4. CPU/GPU neural network graph design flow.

For FGPA implementation, next, the ”decent q” tool, added to TensorFlow's ”contrib” module, performs graph quantization. This process converts the graph's parameters and activations into lower-precision data representations, reducing storage space.

For the following stages, specific Xilinx tools are necessary. These tools are compatible with various machine learning frameworks, but our design exclusively utilizes TensorFlow. To achieve compatibility, Xilinx employs an intermediate representation language that standardizes graphs from different machine learning frameworks.

In the sixth stage, the quantized graph in TensorFlow format is converted into a common intermediate representation

language compatible with the Xilinx compiler, making it independent of the specific framework used for graph design at the software level.

The sixth stage standardizes graphs from different machine learning frameworks into a common intermediate representation language compatible with the Xilinx compiler. This enables seamless integration and optimization for deployment on Xilinx platforms.

The seventh and final stage involves compiling the graph in the XIR format to the instruction set of the specified DPU. This compilation process translates the intermediate representation into specific instructions that the DPU can efficiently execute. Compiling the graph to the DPU's instruction set enables seamless deployment and execution on the target hardware, maximizing performance and efficiency.

These stages collectively ensure a comprehensive design flow for model generation and deployment, taking into account the specific requirements of different target devices.

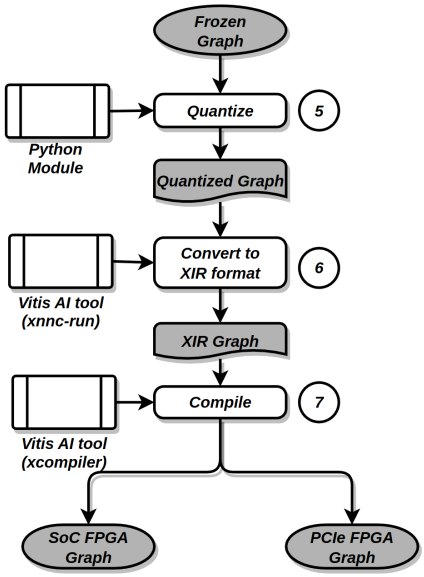

Fig. 5. FPGA neural network design flow.

VI. Results

Three different neural networks and their respective runtime software have been generated for four different platforms to validate the preservation of performance metrics compared to manual implementations. The chosen ones for implementation, namely VGG, ResNet, and GoogLeNet, are popular state-of-the-art architectures in the field of Convolutional Neural Networks (CNNs). Residual Networks (ResNet) and Inception Networks, such as GoogLeNet, represent significant variations in CNNs when compared to classical architectures like VGG. While traditional approaches focused on improving performance by increasing the network size, ResNet and Inception introduced innovative network topology to enhance performance and capacity without disproportionately increasing the number of parameters.

Each network has been deployed on different platforms: Intel Xeon Gold 6138 with 20 cores and 40 threads, Nvidia Quadro RTX 4000, PCIe ALVEO card U50 with AMD UltraScale+ architecture, and ZCU102 board with Zynq UltraScale+ MPSoC. In the FPGA platforms, different DPU have been utilized. The ALVEO U50 implements a single core of the DPUCAHX8H with three process engines. In contrast, the ZCU102 has a single instance of the DPUCZDX8G configured for maximum operation with 4096 MACs per clock cycle. The performance of each network on each platform, measured in frames per second, is reported in Table 1.

They have been tested with a Fashion MNIST dataset consisting of 28x28 pixel resolution images, and the frames per second values have been obtained from the execution of 1000 images.

TABLE I
FPS MEASURED FOR THE PROPOSED METHODOLOGY

Devices	**FPS performance**		
	ResNet50	***VGG16***	***GoogLeNet***
Quadro RTX 4000	2487.0	10832.9	40713.6
Intel Xeon 40 core CPU	428.1	1588.6	9479.9
ALVEO U50	840.8	634.3	3785.9
ZCU102	197.1	115.0	1385.0

The results in Table 1 are presented as evidence of the validity of the applied methodology, without the intention of comparing speed performance between different platforms, which is beyond the scope of this work. The executions on FPGA systems have not been optimized to achieve the maximum frames per second ratio allowed by the technology. The strength of the proposed methodology lies in its ability to automate the deployment of neural networks as part of a larger system on different platforms without compromising accuracy across implementations.

VII. Conclusions

This research proposes a comprehensive design flow that overcomes the challenges of implementing deep neural networks on different platforms. It combines UML modeling, automatic code generation, and deployment processes to address the limitations of traditional approaches.

UML modeling enables precise definition of neural networks, which can be converted into TensorFlow graphs. Automatic code generation reduces knowledge requirements and coding times, enabling easier exploration of the different design alternatives. TensorFlow is directly used to execute the networks on CPUs and GPUs. For FPGA implementations, specialized code generation tools transform the neural networks into FPGA-optimized graph formats. FPGA mapping benefits from the utilization of Xilinx Deep Learning Processor Units (DPUs), streamlining the design process and improving efficiency. Thus, the proposed methodology, combining both elements, demonstrates exceptional adaptability to diverse

hardware configurations, including CPUs, GPUs, and FPGAs. This flexibility allows users to choose the most suitable hardware and deployment options for their specific requirements, making the approach highly robust and versatile.

The research successfully implemented three popular neural network architectures (VGG, ResNet, and GoogLeNet) on four platforms (Quadro RTX 4000, Intel Xeon 40-core CPU, ALVEO U50, and ZCU102) and presented performance results in terms of achieved FPS. While throughput and other performance metrics are essential for evaluating a solution's effectiveness, our focus was primarily on showcasing the versatility and scalability of our proposed approach.

Our paper aims to demonstrate the feasibility and potential of our methodology across various hardware configurations, including FPGAs, without delving into the intricacies of FPGA optimization. Results emphasize the preservation of performance metrics compared to manual implementations and highlighting the efficient implementations achieved across different platforms.

VIII. Acknowledgments

The work has been partially supported by DAIS project, funded from Key Digital Technologies Joint Undertaking (KDT JU) under grant agreement No 101007273. The KDT JU receives support from the European Union's Horizon 2020 research and innovation program and Sweden, Spain, Portugal, Belgium, Germany, Slovenia, Czech Republic, Netherlands, Denmark, Norway, Turkey. This work reflects only the author's view and the Commission is not responsible for any use that be made of the information it contains.

References

[1] G. Sciangula, F. Restuccia, A. Biondi and G. Buttazzo, "Hardware Acceleration of Deep Neural Networks for Autonomous Driving on FPGA-based SoC," 2022 25th Euromicro Conference on Digital System Design (DSD), Maspalomas, Spain, 2022, pp. 406-414, doi: 10.1109/DSD57027.2022.00061.

[2] W. Liu and K. Tan, "Face Landmark Detection Based on Deep Learning Processor Unit on ZYNQ MPSoC," 2022 7th International Conference on Intelligent Computing and Signal Processing (ICSP), Xi'an, China, 2022, pp. 415-419, doi: 10.1109/ICSP54964.2022.9778436.

[3] Max Kelsen, Neural Network Inference on FPGAs, towardsdatascience.com, 2021, https://towardsdatascience.com/ neural-network-inference-on-fpgas-d1c20c479e84

[4] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[5] Vitis-AI DPU, Xilinx Corporation. Available: https://github.com/Xilinx/Vitis-AI/tree/1.2.1/DPU-TRD, May 2023

[6] M. Leeser, S. Handagala and M. Zink, "FPGAs in the Cloud," in Computing in Science and Engineering, vol. 23, no. 6, pp. 72-76, 1 Nov.-Dec. 2021, doi: 10.1109/MCSE.2021.3127288.

[7] H. Xiao, K. Li and M. Zhu, "FPGA-based scalable and highly concurrent convolutional neural network acceleration,"2021 IEEE International Conference on Power Electronics, Computer Applications (ICPECA), Shenyang, China, 2021, pp. 367-370, doi: 10.1109/ICPECA51329.2021.9362549.

[8] Joo-Young Kim, "FPGA based neural network accelerators", Advances in Computers, Elsevier, Volume 122, 2021, Pages 135-165, doi: 10.1016/bs.adcom.2020.11.002.

[9] Y. Zhao et al., "Automatic Generation of Multi-Precision Multi-Arithmetic CNN Accelerators for FPGAs,"2019 International Conference on Field-Programmable Technology (ICFPT), Tianjin, China, 2019, pp. 45-53, doi: 10.1109/ICFPT47387.2019.00014.

[10] Zhiqiang Liu, Yong Dou, Jingfei Jiang and Jinwei Xu, "Automatic code generation of convolutional neural networks in FPGA implementation,"2016 International Conference on Field-Programmable Technology (FPT), Xi'an, 2016, pp. 61-68, doi: 10.1109/FPT.2016.7929190.

[11] Thang Huynh, "Deep neural network accelerator based on FPGA", NAFOSTED Conference on Information and Computer Science, November 2017, DOI: 10.1109/NAFOSTED.2017.8108073

[12] A. Misra, C. He and V. Kindratenko, "Efficient HW and SW Interface Design for Convolutional Neural Networks Using High-Level Synthesis and TensorFlow,"2021 IEEE/ACM International Workshop on Heterogeneous High-performance Reconfigurable Computing (H2RC), St. Louis, MO, USA, 2021, pp. 1-8, doi: 10.1109/H2RC54759.2021.00006.

[13] Li Luo, Yakun Wu, Fei Qiao, Yi Yang, Qi Wei, Xiaobo Zhou, Yongkai Fan, Shuzheng Xu, Xinjun Liu, Huazhong Yang, "Design of FPGA-Based Accelerator for Convolutional Neural Network under Heterogeneous Computing Framework with OpenCL", International Journal of Reconfigurable Computing, vol. 2018, Article ID 1785892, 10 pages, 2018. https://doi.org/10.1155/2018/1785892

[14] A. Perepelitsyn, H. Fesenko, Y. Kasapien and V. Kharchenko, "Technological Stack for Implementation of AI as a Service based on Hardware Accelerators," 2022 12th International Conference on Dependable Systems, Services and Technologies (DESSERT), Athens, Greece, 2022, pp. 1-5, doi: 10.1109/DESSERT58054.2022.10018615.

[15] F. Restuccia and A. Biondi, "Time-Predictable Acceleration of Deep Neural Networks on FPGA SoC Platforms" 2021 IEEE Real-Time Systems Symposium (RTSS), Dortmund, DE, 2021, pp. 441-454, doi: 10.1109/RTSS52674.2021.00047.

[16] T. Jeong, E. Ghasemi, J. Tuyls, E. Delaye and A. Sirasao, "Neural network pruning and hardware acceleration"2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), Leicester, UK, 2020, pp. 440-445, doi: 10.1109/UCC48980.2020.00069.

[17] K. -H. Chen, C. -B. Wu, Y. -T. Hwang, H. -L. Chen, J. -M. Lin and C. -P. Fan, "Hardware Acceleration of YOLO-based Convolutional Neural Network Detector by Deep Learning Processor Unit for Intelligent Autonomous Mover," 2022 IET International Conference on Engineering Technologies and Applications (IET-ICETA), Changhua, Taiwan, 2022, pp. 1-2, doi: 10.1109/IET-ICETA56553.2022.9971562.

[18] R. Kedia, S. Goel, M. Balakrishnan, K. Paul and R. Sen, "Design Space Exploration of FPGA-Based System With Multiple DNN Accelerators" in IEEE Embedded Systems Letters, vol. 13, no. 3, pp. 114-117, Sept. 2021, doi: 10.1109/LES.2020.3017455.

[19] Ebeid, E. ; Fummi, F. ; Quaglia, D., "A Toolchain for UML-based Modeling and Simulation of Networked Embedded Systems", Conference on Computer Modelling and Simulation (UKSim), 2013

[20] NVIDIA TensorRT, NVIDIA. Available: https://developer.nvidia.com/tensorrt, May 2023

[21] Optimize and Accelerate Machine Learning Inferencing and Training, Microsoft. Available: https://onnxruntime.ai/, May 2023

[22] Apache TVM, The Apache Software Foundation. Available: https://tvm.apache.org/, May 2025

[23] nGraph, Intel Corporation. Available: https://www.intel.com/content/www/us/en/artificial-intelligence/ngraph.html, May 2023

[24] Core ML: Integrate machine learning models into your app, Apple Inc. Available: https://developer.apple.com/documentation/coreml, May 2023

[25] OpenVINO, Intel Corporation. Available: https://docs.openvino.ai/latest/home.html, May 2023

[26] H. Posadas, P. Peñil, A. Nicolás, E. Villar, "Automatic synthesis of embedded SW for evaluating physical implementation alternatives from UML/MARTE models supporting memory space separation", Microelectronics Journal, 2014, https://doi.org/10.1016/j.mejo.2013.11.003.

[27] V. Gola, M Saja , P. Mobadersany, "TensorFlow-GUI", https://github.com/vikasgola/tensorflow-gui