2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM) | 978-1-6654-3555-0/20/$31.00 ©2021 IEEE | DOI: 10.1109/FCCM51124.2021.00010

A Safari through FPGA-based Neural Network Compilation and Design Automation Flows

Patrick Plagwitz[1], Frank Hannig[1], Martin Ströbel[2], Christoph Strohmeyer[2], and Jürgen Teich[1]
[1]Department of Computer Science, Friedrich-Alexander University Erlangen-Nürnberg (FAU), Germany
[2]Schaeffler Technologies AG & Co. KG, Herzogenaurach, Germany

***Abstract*—Thanks to the enormous computing power of GPUs, Machine Learning (ML) based on artificial neural networks has found its way into many important application fields. Sophisticated compiler infrastructures facilitate the task of mapping neural networks onto these accelerators. Recently, new developments have also led to compilation and design automation flows that target FPGA-based accelerators. Although not being as mature as their GPU counterparts, there exists a multitude of published and actively developed approaches with differing support levels for network classes, file formats, and target platforms. Neural network exchange file formats advance jointly with the modeling frameworks. In this paper, we take a quick safari through the jungle of neural network compilation flows for FPGA-based targets by reporting qualitative and quantitative metrics. For comparison, we study the classes of supported neural network architectures of each approach, and the corresponding compatibility of exchange formats, emphasizing ONNX, by examining available conversion tools. Besides, we look at several non-functional properties, including FPGA resource utilization and performance numbers for selected neural networks, but also soft criteria such as licensing, community support, and development activity. Finally, we also assess and discuss some deficiencies currently still affecting some approaches. We hope that our study supports interested readers to orient themselves in the jungle of available flows concerning both functionality and usability, as well as to guide further development and research activities in the endeavor of automated ML acceleration on FPGAs.**

I. Introduction

A. Motivation

Since the construction of the first Artificial Neural Network (ANN), Machine Learning (ML) research has advanced a lot into multiple directions and provided breakthroughs in several domains where it now forms a major part of the available staple techniques. The current omnipresence of Neural Networks (NNs) has been made possible by sufficient and affordable computing resources for accelerating their training and application. Naturally, this was in part due to Moore's Law. But most important to research, it is in the form of more or less specialized computer architectures such as Graphics Processing Units (GPUs) and Tensor Processing Units (TPUs). Their ease of programming, their construction as a regular array of processing units, and their excellent support for floating-point computations are a good match, especially for Convolutional Neural Networks (CNNs).

But in general, considerable costs and high power consumption make GPUs much less attractive in the area of embedded systems and edge processing. Example applications include speech recognition for human-machine interaction, biological signal processing on wearable devices for medical and fitness applications, on-the-fly sensor processing for visual environment understanding applications such as those used in robotics and advanced driver assistance systems, or anomaly detection for predictive maintenance in industrial processes. These scenarios involve design restrictions regarding power consumption and cost that differentiate them fundamentally from cloud-based contexts where vast computational resources are readily available. However, the design of energy- and cost-efficient programmable hardware implementations, e. g., using Field-Programmable Gate Arrays (FPGAs) or CPU-FPGA heterogeneous platforms or System-on-Chip (SoC) solutions is difficult due to either tight resource constraints, or in the difficulty of automatically compiling a given trained NN to a plethora of emerging platforms and available accelerator IP cores (tool support).

做AI算法自动化

Thus, the demand for domain-specific design automation and state-of-the-art ML algorithms is apparent. As a comparison, the development of software compilers took decades to advance to its current modern state and is still subject of ongoing research and an important factor for the success of new technologies and industry trends. For CPU and GPU targets, there already exist mature compilation flows starting from a well-known environment for neural network design, e. g., PyTorch or TensorFlow. In contrast, there is much less support to automatically map a given net to a SoC which includes one or a few CPUs and an embedded FPGA.

In this paper, we take a safari through the jungle of approaches for (semi-)automatically mapping NNs to SoCs, i. e., FPGA-based targets. Of particular interest to us are promising approaches following an open-source model inspired by other compiler toolchains like GCC. From experimentation with these flows, we conclude that available approaches are far from being mature or general enough in terms of either supported class of nets or in terms of the supported class of target architectures.

As one reason for this discrepancy, we identify the high diversity of different hardware accelerator solutions proposed recently including (a) *dedicated circuit designs*, often based on libraries of building blocks, and (b) *overlay-based co-designs* from different vendors and academic sources. In the first case, a given net is implemented fully in hardware. Therefore, it often requires heavy net tuning in terms of quantization of weights, inputs, and number of neurons before hardware synthesis due to severe resource constraints. In the second case, one or multiple accelerator cores, e. g., a tensor core, is instantiated in hardware, often called an *overlay* architecture. By proper partitioning of a given net into a sequence of accelerator calls, the net can then be executed by a generated control program running on the CPU subsystem to manage the accelerator calls and the shuffling of data through the overlay.

We conclude that still a lot of work is needed in the future to generalize available compilation-based techniques to enable a wide support of target architectures and provide higher coverage of NN support.

In Section I-B, we classify the investigated target architectures

978-1-6654-3555-0/21/$31.00 ©2021 IEEE
DOI 10.1109/FCCM51124.2021.00010

and approaches. Section III presents the qualitative results of our evaluation of a selection of design flows following the essential techniques explained in Section II-C to compile NNs to SoCs. In Section IV, we examine the suitability of different compilation flows by end-to-end testing with several real-world models. Section IV-C provides concluding remarks and an outlook on which further research is needed.

B. Fundamentals

1) Tool Flow Classification: Using the cloud, data is sent via a network to a remote server system that performs the computation. Server systems generally can have more memory and more powerful processing capabilities than embedded devices, albeit with reduced energy efficiency. Recent research has led to some solutions to these disadvantages: Weight compression techniques are used to overcome low memory capacity and custom hardware implementations lead to more performant embedded systems for application-specific workloads, opening up the possibility of edge-computing NNs.

In edge computing, no network communication is necessary for the computation. Instead, data is used at the site it is accrued, e. g., in a mobile phone or a control unit as part of an assembly line. In [1], the advantages of FPGA-based SoCs to reduce the well-known automation pyramid for the three levels of field, cell, and company have been identified, giving an enormous potential for cost savings. Moreover, for many devices operating at the edge, the analysis of huge amounts of data close to the sensors is indispensable due to real-time constraints and limited network capacities, often by orders of magnitude. Finally, also energy can be saved when not transporting vast amounts of data.

For the implementation of NN applications, GPUs are mostly suited for the cloud, while in both CPU and FPGA categories, there are products suited for both cloud and edge. Due to their flexibility, the most interesting recent developments happened on FPGA-based SoC architectures like the Xilinx Zynq product line [2], which is also investigated as an example in this paper.

Whereas compilers for mapping NNs to CPUs have shown to be quite mature already, e. g., [3], starting a network hardware design using a Hardware Description Language (HDL) might be much too tedious and error-prone. Beneficial, particular to SoC targets, would therefore be a design flow starting with an established NN design framework such as TensorFlow [4] or PyTorch [5], or in Open Neural Network Exchange (ONNX) format [6] (see Figure 1).

Using an NN compilation approach, a vast set of optimization techniques becomes available using the domain and high-level knowledge of the neural network model. A second benefit is that even developers without detailed architectural knowledge could program and deploy an efficient ML system. Lastly, different SoC architectures could be more easily retargeted using the same frontends. For instance, a model could be prototyped on an ARM CPU with software-based test signal generation before being implemented as an FPGA hardware design, all without changing the model itself.

As mentioned in the introduction, two classes of edge-computing approaches for NNs on SoCs can be distinguished: those building a dedicated circuit implementation using IP libraries and those based on programmable accelerator overlays (see Figure 1). Overlay architectures consist of one or more

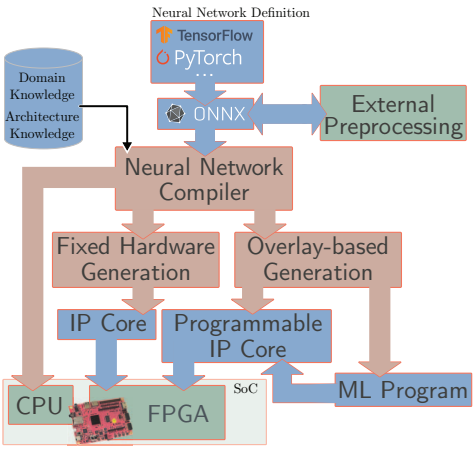

Fig. 1: Illustration of a compilation-based tool flow from NN framework to SoC implementation. Different approaches can be classified into dedicated hardware generation (left path) and overlay-based acceleration (right path).

programmable accelerator cores instantiated in the FPGA fabric to accelerate tensor operations such as Vitis AI [7] and VTA [8]. On-chip bus systems interconnect these to fetch and store data to the environment, e. g., PCI Express or AMBA AXI4. A driver program on a CPU controls the activation of the accelerator units. With dedicated hardware solutions, we characterize approaches that construct a direct implementation of the NN operations, typically resulting in a layer-parallel circuit design.

Figure 3 shows the different components of a full design flow; in this case, Vitis AI and VTA, both overlay-based approaches. As mentioned before, frameworks for defining and training networks provide only the first step. For the next step, there exist numerous different import formats. One attempt to unify these is the ONNX format [6], which now can be regarded as an accepted standard for model interchange.

The second step is the platform setup for both host and target. From the most general perspective, there can be three platforms involved: (1) the development machine for compiling the NN, (2) the host part of the target, which executes software and controls the FPGA, and (3) the FPGA itself that is used for acceleration.

Part of the design flow is the generation of the hardware design bitstream. Dedicated hardware flows use a compiler to produce this, while in overlay-based systems, it is preconfigured. Parameters involved in this process affect operator support and FPGA performance results like resource usage and latency.

The last step is shell compilation, where a driver program for controlling the hardware is built around the synthesized network accelerator. E.g., this might be implemented in the form of a simple C program calling software libraries belonging to the tool flow.

2) Performance Metrics: We evaluate the presented tool flows regarding several qualitative and quantitative metrics. First, a system would ideally support all types of NNs. In practice, only a subset of them can be compiled or adequately accelerated. Many research projects report on image classification solved by CNNs due to their regular structure, lending itself to acceleration. However, other network types are of similar research interest as well as practical usefulness. Approaches for accelerating Long Short-Term Memory (LSTM) architectures on FPGAs have also

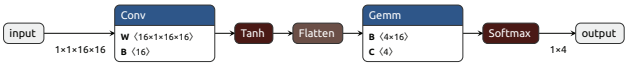

Fig. 2: Shallow neural network VoltageNet as an example used for testing tool flows.

been reported (e. g., [9], [10]).

In this context, we want to compare the flexibility of each approach: Which networks are accepted, and in what formats?

Also of interest is the development model and vendor affiliation of the project. Is it a proprietary or an open-source solution? How active is its development? Based on this, one can estimate the technological trajectory, i. e., how a tool of this kind further matures as well as the level of support one can expect when incorporating it into a larger system or when trying to modify it. Last, we compare the set of platforms supported by each framework.

For the purpose of testing scope, supported formats, and the applicability of each design flow, we use the following real-world NN models: ResNet-50-V2 [11], ShuffleNet-V2 [12], Tiny YOLOv3 [13], BiDAF [14], and VoltageNet throughout the paper. VoltageNet is a shallow NN created by ourselves for testing purposes (see the architecture in Figure 2). It can classify voltage time series of 256 numbers into four classes: constant, sawtooth, triangle, and sine signal curves. It has an accuracy of 96% when using single-precision floating-point numbers.

For a detailed overview of less recent tools, also see [15].

II. Accelerator Designs

In this section, we present the characterization of design flows according to the hardware acceleration approach as overlay-based or dedicated circuit design.

A. Overlay-based Accelerators

Overlay-based accelerator approaches adopt the coprocessor paradigm, where a hardware/software co-design is used to operate the periphery and performs the computations. Example target platforms include the Xilinx Zynq family of Programmable System-on-Chips (PSoCs), where an ARM processor controls the execution of portions of a net (ideally the whole) on an overlay-accelerator IP core architecture synthesized in the FPGA. In principle, the whole net could be executed also purely in software, but at no acceleration.

Tool flows following this approach are Vitis AI by Xilinx [7], VTA [8] as part of the TVM compiler [3], the Matlab Deep Learning Processor [16], and the sensAI stack by Lattice Semiconductor [17]. All of these contain a compiler as part of their package, thus being self-contained tool flows. There are also overlays without compiler support, for example, MARLANN [18] or TinBiNN [19].

One notable further project but not evaluated here is Brainwave by Microsoft [20]. It has support for a wide variety of network types, including CNNs, Gated Recurrent Unit (GRU), and LSTMs. It is deployed on Microsoft cloud computing centers and uses heavily parallelized FPGAs internally, such as in an overlay-based compilation flow but is otherwise dissimilar from the approaches presented here, which are focused on edge computing.

Refer now to Figure 4 for a generalized view of accelerator overlays on SoCs. As can be seen here, DRAM is used to store activation, weight, and instruction data. The accelerator runs inside the programmable logic and is controlled via memory-mapped

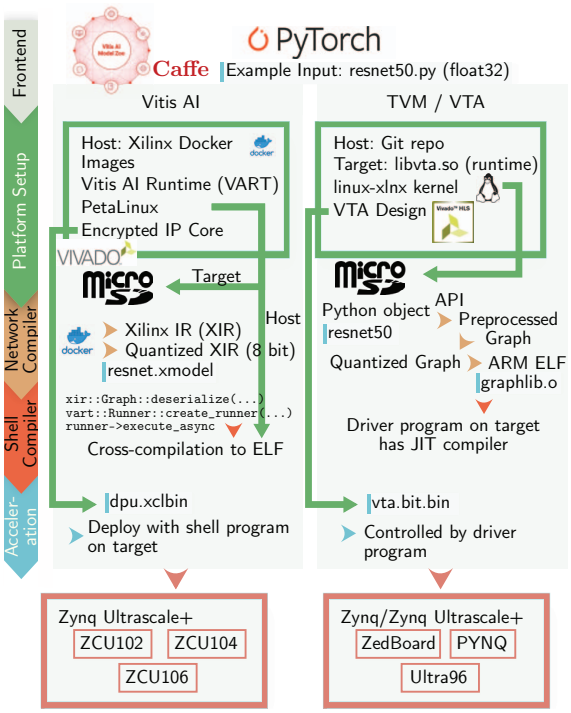

Fig. 3: Illustration of FPGA-supported NN acceleration using the overlay-based tool flows Vitis AI and TVM/VTA. In the case of Vitis AI, a compiler-specific intermediate representation is exposed to the user during the compilation steps. For VTA, a deployment script on the host platform transforms the framework model via a high-level API to a cross-compiled ELF object.

registers from the software side. Data can be transferred by a Direct Memory Access (DMA) mechanism, depending on the actual platform, into smaller BRAM buffers.

How and when to transfer data is an essential factor for performance because NN computations can quickly become I/O-bound, nullifying computation resources. Techniques like operator fusion and data compression heavily affect this but are handled by the compiler and otherwise are outside the scope of our discussion. For further details, we refer, for instance, to [21].

The accelerator must have some load and store units feeding and being fed, respectively, by the actual compute unit. Synchronization between them can follow traditional RISC pipelining or otherwise and is up to the specific design. The overlay architectures contain at least a General Matrix-Multiply (GEMM) core consisting of a systolic array. Since FPGAs are synchronized circuits, only small matrices can be processed without intermediate registers. The compiler can handle the coordination and partitioning of larger computations. Other specialized compute units are also possible: For example, ReLU and pooling layers can be treated separately.

A characteristic is that overlays can be reprogrammed by loading new instructions into memory and transferring them to the FPGA. No reconfiguration of the FPGA is necessary although possible. As such, overlay-based approaches can use one and the same accelerator configuration for the implementation of multiple networks or multiple parts of a single network by merely reloading the instruction queue. However, the performance thus obtained might not be optimal without adapting the accelerator compute units.

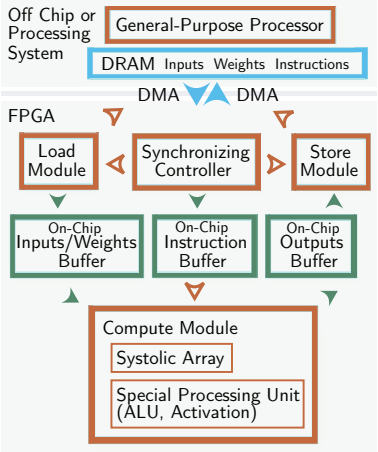

Fig. 4: Schematic illustration of an overlay-based accelerator architecture for deep learning. Solid arrows show the data flow, while unfilled ones represent control signals. Green blocks are normally implemented as BRAM primitives.

Due to their use of compilers, all overlay-based solutions can be categorized as layer-sequential processing approaches. Being Application-Specific Instruction Set Processors (ASIPs), they also benefit from ASIP generation techniques. The open-source tools currently evaluated use C++ High-Level Synthesis (HLS) but no further automated high-level generation like in [22].

B. Dedicated Hardware Solutions

Dedicated circuit solutions pick preconfigured IP cores from a library and connect them to produce a direct and dedicated hardware implementation of an NN. They are not as reliant on a compiler and generally not as flexible during runtime since the accelerator cannot be reprogrammed. The generator of the IP core is not an optimizing compiler in the sense that it applies generalized transformation passes, but it does convert from a high-level representation to one closer to hardware. Examples for this include FINN, a research project by Xilinx [23], fpgaConvNet [24], and smaller work from various research publications (e. g., [25], [26]).

Proposed solutions include layer-sequential as well as layer-parallel circuit implementations. For instance, in FINN, each layer is converted to an HLS design and the whole network subsequently into a *stitched* IP, composed of these subcomponents. Of course, further optimization also happens during synthesis, which can cross layer boundaries. Also, operator fusion is an important task of the compiler.

Dedicated accelerators still need a driver program to transfer activation and possibly weight data, but the transfer is not triggered by a specific instruction sequence. Minimizing driver footprint increases portability, while the use of vendor- or chip-specific primitives inside the library will decrease it.

The upside of these approaches is that the IP library can be tailored to specific properties of certain networks and that the accelerator design is therefore more application-specific than overlays and therefore can, in general, be better optimized. After combining the parts, a hardware design can be built that uses the chip resources in such a way as to maximize performance for exactly one network. In principle, no computing resources are wasted this way, and also the runtime overhead is reduced.

C. Compilation

Starting from the NN framework model, some NN modeling frameworks demand preparations to be applied already on the network specification level. Quantization is especially important for FPGA targets because they are not at all suited for efficient floating-point computations. This is done by the compiler for overlay-based solutions, while dedicated circuit approaches are often not automated in this regard.

For example, FINN is specialized to support Binarized Neural Networks (BNNs), where weights and activation numbers are stored as a single bit, making multiply-add operations especially efficient [27]. This also has the advantage of avoiding memory bottlenecks, being a very good compression and drastically reducing the amount of data to transfer. However, network accuracy will inevitably be lost when choosing these optimizations.

Moreover, converting a network from typically a floating-point number format to binary is not trivial and generally requires retraining and must therefore be performed in a separate step. Conversions of important networks, like AlexNet, have been proposed [28]. When not reducing to 1-bit resolution of coefficients and weights, 8-bit fixed-point numbers are often chosen as a trade-off between remaining accuracy and resource usage.

Advanced NN compilers, like TVM, combine graph-based and loop-based optimization techniques. TVM can also target multiple platforms, among them CPUs and GPUs, all benefitting from target-agnostic optimizations. Graph-based optimization works on the data-flow graph that is the typical representation of a NN in TensorFlow or ONNX. Here, layers can be fused, or quantization can be applied. Loop-based optimization is one abstraction level lower and is facilitated by research on iteration space representations, manifested in frameworks like Hipacc [29, 30] or Halide [31], the latter of which is a basis for TVM. There also exist recent developments for complete compilers in the literature (see, e. g., [32], which is intended to support multiple existing accelerators).

III. Evaluated Tool Chains

This section lists the evaluated design flows in greater detail. See Table I for a structured view of the metrics gathered for each solution.

A. Overlay-based Approaches

1) Vitis AI: Vitis AI [7] is a commercially developed design tool by Xilinx for their Alveo cards and Zynq UltraScale+ chips. The former is intended to be used in cloud setups while the latter also as edge-computing devices. Vitis AI is a complete tool flow that includes a closed-source compiler and configurable accelerator IP core with internal parallelism. It follows the general outline explained in Section II-A. Supported networks are provided in the Xilinx Model Zoo [33] and include all essential ImageNet networks as well as object detection networks. All of them are CNNs, although a Recurrent Neural Network (RNN) version of the accelerator is also available. Custom networks can be read from Caffe, TensorFlow, or PyTorch definitions. Graph partitioning for flexibly placing nodes in the compute graph on CPU or accelerator is possible. Networks are typically quantized to 8 bits. No special support for other bit widths is currently available.

We refer to Figure 3 for an overview of the overall design flow. Host platforms must be set up with a runtime system that

TABLE I: Qualitative evaluation of design flows distinguished according to type of accelerator design, and supported platforms and network types as well as network modeling frameworks (frontends).

Type	Design Flow	Platforms	Network Types	Development	Frontends
Overlay-based	Vitis AI [7]	Xilinx Zynq / UltraScale+	CNN / RNN ([33])	Proprietary (Xilinx)	Caffe, TensorFlow, PyTorch
	VTA [8]	Xilinx Zynq, Intel DE10	CNN (ResNet-like)	Scientific and open-source (active project)	ONNX, PyTorch, and others
	Matlab DLP [16]	Xilinx / Intel	CNN	Proprietary (MathWorks)	ONNX, Keras, Caffe
	sensAI [17]	Lattice FPGAs	CNN	Proprietary (Lattice Semiconductor)	Caffe, TensorFlow
	MARLANN [18]	Lattice iCE40	N/A (no compiler)	Open-source (last commit 2018)	N/A (no compiler)
	TinBiNN CNN [19]	Lattice iCE40	CNN (CIFAR-10)	No source code available	N/A (no compiler)
Dedicated	FINN [23]	Xilinx Zynq / UltraScale+	BNN and CNN	Scientific and open-source (active project)	PyTorch
	fpgaConvNet [24]	Xilinx Zynq	CNN	No source code available	Custom
	FINN-L [9]	Xilinx Zynq	LSTM	Open-source (last commit early 2020)	Custom
	hls4ml [34]	Xilinx	CNN	Open source (18 commits during Nov 2020)	Keras
	CaFPGA [35]	Xilinx	CNN	No source code available	Caffe

is provided in packaged format by Xilinx in Docker containers. Likewise, the compiler is provided in a Docker container. Shell applications can be written in C++ by using the Vitis AI Runtime (VART) library that has functions for controlling the accelerator, a so-called Deep Learning Processing Unit (DPU).

2) VTA: Versatile Tensor Accelerator (VTA) [8] is similar to Vitis AI in principle but is part of the open-source compiler TVM. Originally implemented as a proof-of-concept backend, it is still in active development in connection with TVM, although its GitHub commit history shows it has not been updated for three months at the time of writing.

A VTA instance consists of a vector-matrix GEMM core and an Arithmetic Logic Unit (ALU) core supporting operations on matrix operands, both with configurable sizes. The compiler is TVM and can be called from Python code. Thus, all models supported by TVM on its other target platforms can be deployed on VTA in principle. However, VTA is intended to be used for networks following general architecture of ResNet or MobileNet as these are its main case study [8]. Currently, other models are likely not supported without modification.

See also Figure 3 for an overview of the design flow. The target platform (e. g., Xilinx Zynq) must be set up with a suitable Linux kernel and operating system. As can be seen, VTA is preconfigured as a Vivado HLS design and synthesized accordingly.

3) Matlab Deep Learning Processor [16]: Matlab is a proprietary IDE and programming language for scientific and engineering tasks. It has only recently included support for deep learning models. In version R2020b, an accelerator and corresponding compiler implementation were released [16]. It also supports a full compilation tool flow, including quantization, and can target any platform the Matlab HDL Coder can. This includes the Xilinx Zynq and Zynq UltraScale+ platforms.

The IP core has a matrix multiplication block as well as a unit for fully connected layers, making it possible to accelerate parts of networks that are left for the CPU in VTA or Vitis AI. Although Matlab has its own network design frontend, it can also process ONNX, which makes it interoperable with other frameworks. A tool flow from e. g., PyTorch to Xilinx Zynq is therefore possible with Matlab.

4) sensAI [17]: The sensAI stack is a proprietary tool flow developed by the FPGA producer Lattice Semiconductor [17]. Notable here is that their accelerator can work on the very small iCE40 FPGA, which is interesting for low-power applications [36].

5) Others: Other notable developments for very low-power contexts, namely targeting the iCE40 FPGA, are two projects. MARLANN is an overlay accelerator that lacks a compiler but is open-source and documented [18]. TinBiNN is very similar, but no source code is available [19]. It is based on the ORCA RISC-V softcore processor.

B. Dedicated Hardware Approaches

1) FINN: FINN was initially published in 2017 [23], but subsequent publications have been incorporated into its GitHub repository [37] recently in 2020. This also includes extending its primitives to quantization bit widths other than one bit [38].

Networks that are supported out-of-the-box are MNIST- and CIFAR-10-solving ones in various configurations. Custom models can be imported from preprocessed ONNX by calling FINN from a Python script. To produce these FINN-specific ONNX files, an included PyTorch library, named Brevitas, must be used. Using FINN involves a significant effort as each network layer must be treated separately, and transformation passes must be called manually, so that correct output is produced. As such, substantial familiarity with the code and techniques is required, although the project also provides example scripts for this.

2) FINN-L: FINN-L [9] is part of the FINN research project but targets different applications and is not part of the same code base. It aims to accelerate LSTM models and works as

a thin code generation layer written in Python using an LSTM IP library. It is not in active development anymore and has no generalized way of importing new models.

3) fpgaConvNet: fpgaConvNet [24] is a research project that has yielded good results for various CNNs using a template library for convolutional layers. It can support both fixed-point as well as floating-point representation of activation and weight data. No source code has been published, but the authors explicitly reserve this possibility for the future.

4) hls4ml: hls4ml [34] is an open-source compiler for CNNs to dedicated hardware via Vivado HLS. It can read multiple input formats. However, in an end-to-end flow, only Keras is fully supported. Layers are implemented by choosing and configuring HLS modules from a template library. This also provides the possibility of performing software simulation of the compilation product, which is covered in the hls4ml programming interface.

5) Others: A straightforward but still partially automated way of implementing a Neural Network design is the Matlab HDL Coder, which can be used from its graphical programming interface Simulink [39]. VoltageNet, for example, can be turned into a Simulink model and HDL code generated from them. This is a viable alternative to specialized compilers although not as configurable and flexible. However, existing design flows compatible with Matlab can be integrated more easily in this way.

Another approach for generating dedicated designs is CaFPGA [35]. It can perform a design space exploration considering the objectives execution time, resource consumption, and memory interfacing. The authors consider VGG [40] and AlexNet modeled in Caffe for their case study. Support for other frontend interfaces is unclear, and no source code is available.

IV. Deployment Testing

In this section, we test the applicability of the presented design flows on a selection of real-world models, taking into account all necessary preparatory steps and evaluating the level of support offered during all stages of deployment. As target platforms, we use a ZedBoard and PYNQ-Z1 board with Xilinx XC7Z020 chips for all flows except for Vitis AI. Here, we use a ZCU106 board with a Xilinx XCZU7EV-2FFVC1156 Ultrascale+ MPSoC.

A. Framework Support and Format Conversions

When compiling a given network model, the first stop is to translate it into a format that can be read by the compiler. There exist several NN formats connected to the different modeling frameworks and in general, compilers only support a subset. For example, PyTorch is well-supported by Vitis AI and is necessary for FINN. The hls4ml compiler [34] can only process Keras models end-to-end. If pre-trained ONNX models could easily be imported into these frameworks, it would open up the corresponding tool flows towards many real-world models.

Therefore, we tested existing conversion support first to verify that it works correctly. We did this by using ONNX as an intermediate format: We converted each of the networks presented in Section I-B2 to another format, then back to ONNX. Afterwards, we used the onnxruntime software [41] to input a set of test data to verify that the behavior of the model did not change. In case the conversion from ONNX did not work, we tried to find a substitute model in the corresponding framework to test conversion towards ONNX. For all except VoltageNet, we used the ONNX models from the ONNX Model Zoo [42].

Table II shows which software packages provide which conversion functionality. As can be seen here, PyTorch, unfortunately, cannot import ONNX models. All other frameworks have some sort of support. We also cite the tool versions, which were chosen to be the most recent ones available to us, so our results can be reproduced. For the same reason, the source code and input for our tests can be found at https://www.cs12.tf.fau.de/forschung/projekte/ml-lib.

Refer now to Table III for the quality of the conversion algorithms. Conversion towards Keras and back works as expected for VoltageNet and ResNet. When trying to import ShuffleNet, an error message referring to incompatible shapes in one of the convolutional layers was yielded. This could not be solved even when explicitly specifying input shapes with the top-level API. As a replacement, we used a Keras implementation from [51] to test the export. However, Keras and ONNX evaluation results differ. Finally, the problem with the other two networks, Tiny YOLOv3 and BiDAF, could be tracked to issue #32 of the onnx2keras Github project, which is unresolved and also leads to a mismatch of operation input shapes. We also found that the onnx2keras software wraps activation functions in Keras Lambda layers even when dedicated layers exist in the library. Lambda layers are not portable and cannot be read by all frameworks.

Export in PyTorch works well for the first three networks for which an implementation from the PyTorch `torchvision` package or our own could be used, respectively.

Unfortunately, MXNet cannot import any of the selected test models. The errors that result here are strongly related to the reported problems in issue #13774 of the MXNet Github issue tracker. Because import did not work, the MXNet Gluon ResNet-50 version was used to test the export. However, this also resulted in an error message describing a faulty specification of a BatchNormalization layer.

TensorFlow conversion was also lacking when it comes to ShuffleNet-V2, Tiny YOLOv3, and BiDAF, the latter two erroring out on unsupported operators: a case analysis and CategoryMapper, respectively.

It can be noted that BiDAF, an LSTM, can neither be imported nor exported by any framework. Also ShuffleNet and YOLO cannot both be imported and exported by any of the considered frameworks. Thus, conversion support is unfortunately hardly universal and leaves much room for improvements.

B. Compiler Support

Table IV gives a summary of the compilation capabilities of four of the evaluated design flows. We choose two test models: VoltageNet and ResNet-50. For the test, we converted each network to the most suitable input format for each flow, e. g.,

TABLE II: Summary of conversion tools between ONNX and network modeling framework-specific formats. Unless otherwise mentioned, the tools are 3rd-party software projects.

Framework	ONNX Import	ONNX Export
Keras [43]	onnx2keras [44]	keras2onnx [45]
PyTorch [46]	No support	Native
TensorFlow 2 [47]	onnx-tf [48]	tf2onnx [49]
MXNet [50]	Native	Native

TABLE III: Conversion support between modeling frameworks and exchange formats for selected models. ONNX was used as an intermediate format and both import (**I**) and export (**O**) support was tested, as explained in Section IV-A.

Model		Keras	PyTorch	MXNet	TensorFlow
VoltageNet	**I**	Functional	No support	issues/13774	Functional
	O	Functional	Functional	N/A	Functional
ResNet-50	**I**	Functional	No support	issues/13774	Functional
	O	Functional	Functional	Bad node spec	Functional
ShuffleNet-V2	**I**	Incompatible shape of Conv	No support	issues/13774	Functional
	O	Differing behavior	Functional	No substitute in MXNet	Does not terminate
Tiny YOLOv3	**I**	issues/32	No support	issues/13774	Error in operator conversion
	O	No substitute in Keras	No substitute in PyTorch	No substitute in MXNet	No substitute in TF
BiDAF	**I**	issues/32	No support	Operator not implemented	Operator not implemented
	O	No substitute in Keras	No substitute in PyTorch	No substitute in MXNet	No substitute in TF

Keras for hls4ml. We then imported and compiled the models the way it is recommended in the corresponding software manuals.

Table V shows selected FPGA implementation statistics that resulted from these processes, including the resource usage of the overlay / dedicated hardware and classification latency. Note that the target platforms do differ due to differing support of the tools. Also, depending on configuration and tuning, a tool flow can produce vastly different results. Refer to the respective publications for other results. In the following, each tool flow is characterized qualitatively.

1) TVM/VTA: TVM can import models from a variety of network modeling frameworks. As ONNX is well-supported and is of special interest as an exchange format, we used this for the specification of both test networks. In TVM, a short Python script (about 200 lines of code) is used for loading and compiling (see also Figure 3). This includes quantization from 32-bit floating-point to 8-bit integer numbers.

While there exist many different algorithms for NN quantization, quantization support can be generally categorized into three types: (1) Simple integer scaling, (2) calibrated quantization with a calibration data set to determine value ranges during execution, and (3) quantization-aware training where the whole network must be re-trained. TVM can use simple integer scaling and also optionally supports calibration.

When targeting FPGAs with a VTA overlay-based design, certain modifications must be performed on the compute graph of a net (see Figure 2 for an example), unlike when compiling for, e. g., GPU. These modifications are fully developed for ResNet-like networks, but we found them lacking for models that have other types of layers and other combinations of layers.

As a result, a fully connected version of VoltageNet did not synthesize right away, yielding errors as to the input shapes, which are not NCHW (batch, channel, height, width) as expected. This is why the fully connected hidden layer was translated into a convolutional layer (see Figure 2). But also here, we could not accelerate the model because the necessary VTA micro operations could not be matched against the input graph. This rather resulted in a correctly compiled ARM binary that, however, does not invoke the accelerator. ResNet-50 is an explicitly supported network and works out of the box also in the compilation script used here.

TABLE IV: Compiler support for two of the test networks described in Section I-B2.

Design Flow	Input Format	VoltageNet	ResNet-50
TVM/VTA	ONNX	No accel. possible	Functional
Vitis AI	PyTorch	Functional	Functional
hls4ml	Keras	Not synthesizable	Compilation fails
FINN	ONNX	Functional	Compilation fails

The VoltageNet implementation shown in Table V was done for comparison with explicit TVM tensor expressions that are compiled to an accelerated binary with a separate TVM API. The overlay configuration used here uses a GEMM core with 1×16 activation and 16×16 weight tensors. Also, buffers are configured to use almost all BRAMs of the FPGA and likewise for the ResNet-50 implementation.

2) Vitis AI: Vitis AI deployment consists of two steps: Preparing the model and compiling a shell program. Our chosen input format was PyTorch in this case. Model preparation included quantization and target-specific optimization, like in TVM. We used a Python script for the former, running our PyTorch object against the Xilinx API. This required a significant amount of calibration data for quantization to work. The rest is taken care of by the external Xilinx compiler binary.

Unlike in TVM, where the acceleration is launched from the host platform over the network, the recommended execution strategy in Vitis AI is to download a cross-compiled and self-contained binary onto the target. As a result, a cross-compilation toolchain is used to produce a driver program using the Xilinx API, see Figure 4 for illustration.

ResNet-50 is supported completely in the Xilinx Model Zoo. However, this model also works when input externally from the PyTorch `torchvision` package as was done here. Similarly,

TABLE V: Comparison of design flows. Latency is the time it takes to perform one inference with batch size 1.

LUT	Reg.	BRAM	URAM	DSP	Latency [ms]
ResNet-50 with Vitis AI (ZCU106, Base Clock Frequency: 300MHz)					
57,873 (25.1 %)	106,891 (23.2 %)	145 (46.5 %)	46 (47.9 %)	562 (32.5 %)	12.4
ResNet-50 with VTA (ZedBoard, Clock Frequency: 100MHz)					
25,943 (48.8 %)	27,018 (25.4 %)	131 (93.6 %)	—	220 (100.0 %)	749.9
VoltageNet with VTA (ZedBoard, Clock Frequency: 100MHz)					
25,943 (48.8 %)	27,018 (25.4 %)	131 (93.6 %)	—	220 (100.0 %)	0.11
VoltageNet with FINN (PYNQ-Z1, Clock Frequency: 100MHz)					
11,239 (21.1 %)	11,811 (11.1 %)	2 (1.4 %)	—	0 (0 %)	3.3

VoltageNet works as expected. We can produce a quantized Intermediate Representation (IR) file from it correctly runnable on the target platform.

The Vitis AI processor uses the B4096 configuration and employs URAM on the ZCU106. As can be seen in Table V, the processor leaves room for potentially more cores on the chip for parallel processing. Classification, when compared to VTA on the ZedBoard, is 60 times as fast for ResNet-50 but the Vitis AI processor would not even fit on the chip with this configuration, which is shown for one DPU core. The Xilinx UltraScale+, of course, has a different fabric and provides URAM cells. On it, the DPU, as a multi-clock design, works with a 300MHz and a 600MHz clock.

3) hls4ml: VoltageNet was fed into hls4ml as a Keras model converted from ONNX. The first problem with this is that the Softmax activation function is wrapped in a Lambda layer (as explained in Section IV-A). This cannot be processed in hls4ml, leading to an error. After turning it into a Keras Softmax layer, HLS code is generated from the model. However, it is not synthesizable due to excessive loop unrolling in several operator implementations.

For ResNet-50, we used once a Keras model converted from the ONNX model zoo and once the equivalent Keras pre-trained model. In both cases, we got an error that a `ZeroPadding2D` layer is not supported. hls4ml thus is not fit for arbitrary complete network architectures although it can process several simpler examples as shown in its documentation. Integer scaling quantization is also done by hls4ml itself during compilation and can be configured per-layer.

4) FINN: We ran FINN by calling its API from a Python script with about 300 lines of code. This is not dissimilar from the other compilers but involves explicitly listing transformation and optimization passes that the compiler has to perform and therefore requires in-depth knowledge if modifications are necessary. In its documentation, it is stated: "If there are substantial differences [compared to the examples], you will most likely have to write your own Python scripts that call the appropriate ... functions" [52].

First, FINN can only process PyTorch models that have passed quantization-aware training with the included Brevitas library. This step requires a redefinition of the network with Brevitas layers, which correspond to standard PyTorch layers. E.g., there is a `QuantLinear` layer type. We replaced the tanh activation function with a `QuantReLU` layer because tanh could not be imported into FINN even though it exists in Brevitas. After training with Brevitas and 8-bit weights, our accuracy is reduced from 96% to 70%. This result can certainly be improved with some tuning.

After this preprocessing, VoltageNet compiled correctly and could be executed on the PYNQ board with a configuration similar to FINN's example networks TFC and CNV. However, the measured accuracy on the board showed a further reduction to just 54%. Table V shows that albeit the resulting dedicated implementation is much smaller than the VTA overlay, it is much slower and also comes at the price of a reduced accuracy. Both compilation flows are potentially very flexible. For example, folding of multiplications can be configured in both VTA and FINN. Together with quantization options, a wide variety of design points can thus be reached. Note also that software auto-tuning is an important component when compiling NNs to FPGAs, which can lead to different results than the standard configurations chosen here [8].

Finally, our attempt to compile the ResNet-50 failed during the mandatory Streamline transformation pass where multiplications could not be absorbed into the FINN Thresholding Unit [23].

When compared to a VTA implementation, VoltageNet uses very few resources on the ZedBoard but takes 30 times as long to process. Both measurements excluded the Softmax activation function, which has been executed in software.

5) Summary: In summary, one can say that no compiler can be called with simple command-line options to produce a running implementation fully automatically. All require more or less complex compilation scripts and/or quantization-aware training. Needless to say, the quality of the results is therefore very dependent on the expertise of the user. Finally, by far not all models are supported universally—only Vitis AI was able to compile the two test networks ResNet-50 and VoltageNet correctly.

C. Discussion and Concluding Remarks

As could also be seen, most of the existing modern design flows supporting the compilation of NNs to FPGA-based targets are focused on CNNs, which is, of course, due to the popularity of this network type in past research. For this reason, CNN acceleration also has the most mature tool flows available, like VTA and Vitis AI. While interesting for many embedded applications, RNNs are seen mostly in custom manual implementations in the literature, and not yet fully supported in compilation-based flows, although libraries such as FINN-L also exist.

But even TVM, which has a user-friendly interface for CPU- and GPU-targeted compilation and a sophisticated compilation approach, is difficult to use for FPGAs. The compilation flow for its single FPGA-based backend, VTA, only supports networks following a ResNet-like architecture. Vitis AI is more flexible but does not provide extensibility through the open research community due to being closed source. It is currently undoubtedly the best choice for a practical application if large Xilinx FPGAs are the target. For lower-resource platforms, sensAI is suitable but also vendor-dependent. FINN provides an extensive library of compilation techniques but is quite difficult to use universally.

Matlab is also not bound to a single vendor and can target a vast set of FPGA platforms. It can be useful as an intermediate compiler step or also for integrating existing frameworks. Unfortunately, among the non-commercial solutions, only VTA and hls4ml are actively developed. Other publications were not accompanied by source code or are considered to be complete.

In conclusion, we have evaluated a set of modern FPGA-targeting NN compilation flows. Our selection has been on those which offer an attractive class of supported networks, input formats, and target platforms. The state-of-the-art will certainly advance very fast with a lot of novel opportunities and bridge the currently observed pitfalls and restrictions in our evaluation safari. To this end, the effect of a common open-source development platform for new techniques, including supported exchange formats, should not be underestimated. Here, we see TVM as an excellent candidate for further research.

ACKNOWLEDGMENTS

This work was partially supported by the Schaeffler Hub for Advanced Research at Friedrich-Alexander University Erlangen-Nürnberg (SHARE at FAU) and the German Federal Ministry for Education and Research (BMBF) within project "KISS" (01IS19070B).

REFERENCES

[1] F. Streit et al. "Data acquisition and control at the edge: a hardware/software-reconfigurable approach." In: *Production Engineering* 14.3 (2020), pp. 365–371. DOI: 10.1007/s11740-020-00964-x.

[2] Xilinx. *Zynq-7000 SoC*. URL: https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html (visited on 03/2021).

[3] T. Chen et al. "TVM: An automated end-to-end optimizing compiler for deep learning." In: *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. (Carlsbad, CA, USA). USENIX Association, Oct. 8–10, 2018, pp. 578–594.

[4] M. Abadi et al. "TensorFlow: a system for large-scale machine learning." In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. (Savannah, GA, USA). USENIX Association, Nov. 2–4, 2016, pp. 265–283.

[5] A. Paszke et al. "PyTorch: An imperative style, high-performance deep learning library." In: *Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS)*. (Vancouver, BC, Canada). Dec. 8–14, 2019, pp. 8026–8037.

[6] The Linux Foundation. *Open Neural Network Exchange (ONNX)*. URL: https://onnx.ai (visited on 03/2021).

[7] Xilinx. *Vitis AI*. URL: https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html (visited on 03/2021).

[8] T. Moreau et al. "VTA: An open hardware-software stack for deep learning." In: *The Computing Research Repository (CoRR)* (Apr. 21, 2019). arXiv: 1807.04188v2 [cs.LG].

[9] V. Rybalkin et al. "FINN-L: Library extensions and design trade-off analysis for variable precision LSTM networks on FPGAs." In: *The Computing Research Repository (CoRR)* (July 11, 2018). arXiv: 1807.04093 [cs.CV].

[10] S. Han et al. "ESE: Efficient speech recognition engine with sparse LSTM on FPGA." In: *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. (Monterey, CA, USA). ACM, Feb. 22–24, 2017, pp. 75–84.

[11] K. He et al. "Identity mappings in deep residual networks." In: *Proceedings of the 14th European Conference on Computer Vision (ECCV)*. (Amsterdam, The Netherlands). Springer, Oct. 11–14, 2016, pp. 630–645.

[12] X. Zhang et al. "ShuffleNet: An extremely efficient convolutional neural network for mobile devices." In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. (Salt Lake City, UT, USA). IEEE, June 18–22, 2018, pp. 6848–6856. DOI: 10.1109/CVPR.2018.00716.

[13] J. Redmon and A. Farhadi. "YOLOv3: An incremental improvement." In: *The Computing Research Repository (CoRR)* (Apr. 8, 2018). arXiv: 1804.02767 [cs.CV].

[14] M. Seo et al. "Bidirectional attention flow for machine comprehension." In: *The Computing Research Repository (CoRR)* (Nov. 5, 2016). arXiv: 1611.01603 [cs.CL].

[15] A. Shawahna et al. "FPGA-based accelerators of deep learning networks for learning and classification: a review." In: *IEEE Access* 7 (Dec. 28, 2018), pp. 7823–7859. DOI: 10.1109/ACCESS.2018.2890150.

[16] MathWorks. *Deep Learning Processor IP Core*. URL: https://www.mathworks.com/help/deep-learning-hdl/ug/deep-learning-processor-ip-core.html (visited on 03/2021).

[17] Lattice Semiconductor. *Lattice sensAI Stack*. URL: https://www.latticesemi.com/sensAI (visited on 03/2021).

[18] Symbiotic EDA. *MARLANN – a simple FPGA machine learning accelerator*. URL: https://github.com/SymbioticEDA/MARLANN (visited on 03/2021).

[19] G. G. F. Lemieux et al. "TinBiNN: Tiny binarized neural network overlay in about 5,000 4-LUT and 5mW." In: *The Computing Research Repository (CoRR)* (Mar. 5, 2019). arXiv: 1903.06630 [cs.DC].

[20] J. Fowers et al. "A configurable cloud-scale DNN processor for real-time AI." In: *Proceedings of the ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. (Los Angeles, CA, USA). IEEE, June 1–6, 2018, pp. 1–14. DOI: 10.1109/ISCA.2018.00012.

[21] L. Petrica et al. "Memory-efficient dataflow inference for deep CNNs on FPGA." In: *The Computing Research Repository (CoRR)* (Nov. 14, 2020). arXiv: 2011.07317 [cs.AR].

[22] P. Plagwitz et al. "Compiler-based high-level synthesis of application-specific processors on FPGAs." In: *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. (Cancun, Mexico). IEEE, Dec. 9–11, 2019, pp. 1–8. DOI: 10.1109/ReConFig48160.2019.8994778.

[23] Y. Umuroglu et al. "FINN: a framework for fast, scalable binarized neural network inference." In: *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*. (Monterey, CA, USA). ACM, Feb. 22–24, 2017, pp. 65–74. DOI: 10.1145/3020078.3021744.

[24] S. I. Venieris and C.-S. Bouganis. "fpgaConvNet: a framework for mapping convolutional neural networks on FPGAs." In: *Proceedings of IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. (Washington, DC, USA). IEEE, May 1–3, 2016, pp. 40–47. DOI: 10.1109/FCCM.2016.22.

[25] Z. Li et al. "Laius: An 8-bit fixed-point CNN hardware inference engine." In: *Proceedings of the 16th IEEE International Conference on Ubiquitous Computing and Communications (IUCC)*. (Guangzhou, China). IEEE, Dec. 12–15, 2017, pp. 143–150. DOI: 10.1109/ISPA/IUCC.2017.00030.

[26] C. Zhang et al. "Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks." In: *Proceedings of the 35th International Conference on Computer-Aided Design (ICCAD)*. (Austin, TX, USA). ACM, Nov. 7–10, 2016, 12:1–12:8. DOI: 10.1145/2966986.2967011.

[27] T. Simons and D.-J. Lee. "A review of binarized neural networks." In: *Electronics* 8.6 (June 12, 2019). DOI: 10.3390/electronics8060661.

[28] S. Zhou et al. "DoReFa-Net: Training low bitwidth convolutional neural networks with low bitwidth gradients." In: *The Computing Research Repository (CoRR)* (June 20, 2016). arXiv: 1606.06160 [cs.NE].

[29] R. Membarth et al. "HIPAcc: a domain-specific language and compiler for image processing." In: *IEEE Transactions on Parallel and Distributed Systems* 27.1 (Jan. 21, 2015), pp. 210–224. DOI: 10.1109/TPDS.2015.2394802.

[30] O. Reiche et al. "Generating FPGA-based image processing accelerators with Hipacc." In: *Proceedings of the 36th International Conference on Computer Aided Design (ICCAD)*. (Irvine, CA, USA). IEEE, Nov. 13–16, 2017, pp. 1026–1033. DOI: 10.1109/ICCAD.2017.8203894.

[31] J. Ragan-Kelley et al. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines." In: *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, June 16–19, 2013, pp. 519–530. DOI: 10.1145/2491956.2462176.

[32] Y. Xing et al. "DNNVM: End-to-end compiler leveraging heterogeneous optimizations on FPGA-based CNN accelerators." In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.10 (July 23, 2019), pp. 2668–2681. DOI: 10.1109/TCAD.2019.2930577.

[33] Xilinx. *Vitis AI Model Zoo*. URL: https://github.com/Xilinx/AI-Model-Zoo (visited on 03/2021).

[34] J. Duarte et al. "Fast inference of deep neural networks in FPGAs for particle physics." In: *Journal of Instrumentation* 13.7 (July 27, 2018). DOI: 10.1088/1748-0221/13/07/p07027.

[35] J. Xu et al. "CaFPGA: An automatic generation model for CNN accelerator." In: *Microprocessors and Microsystems* 60 (Mar. 31, 2018), pp. 196–206. DOI: 10.1016/j.micpro.2018.03.007.

[36] Lattice Semiconductor. *iCE40 LP/HX/LM – Low-power, high-performance FPGA*. URL: http://www.latticesemi.com/iCE40 (visited on 03/2021).

[37] *FINN*. Version af783db8d (commit). URL: https://github.com/Xilinx/finn.

[38] M. Blott et al. "FINN-*R*: An end-to-end deep-learning framework for fast exploration of quantized neural networks." In: *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* 11.3 (Dec. 2018), 16:1–16:23. DOI: 10.1145/3242897.

[39] MathWorks. *HDL Coder*. URL: https://www.mathworks.com/products/hdl-coder.html (visited on 03/2021).
[40] K. Simonyan and A. Zisserman. "Very deep convolutional networks for large-scale image recognition." In: *The Computing Research Repository (CoRR)* (Sept. 4, 2014). arXiv: 1409.1556 [cs.CV].
[41] *ONNX Runtime*. Version 1.3.0. URL: https://github.com/microsoft/onnxruntime.
[42] *ONNX Model Zoo*. URL: https://github.com/onnx/models.
[43] *Keras*. Version 2.4.3.
[44] *onnx2keras*. Version 881dcbd9 (commit). URL: https://github.com/nerox8664/onnx2keras.
[45] *keras2onnx*. Version 1.7.0. URL: https://github.com/onnx/keras-onnx.
[46] *PyTorch*. Version 1.6.0.
[47] *TensorFlow*. Version 2.2.0.
[48] *onnx-tf*. Version 1.7.0. URL: https://github.com/onnx/onnx-tensorflow.
[49] *tf2onnx*. Version 1c9c02d220 (commit). URL: https://github.com/onnx/tensorflow-onnx.
[50] *MXNet*. Version 1.7.0.post1. URL: https://github.com/apache/incubator-mxnet.
[51] *Keras ShuffleNet*. URL: https://github.com/scheckmedia/keras-shufflenet.
[52] Xilinx. *Getting Started – FINN documentation*. URL: https://finn.readthedocs.io/en/latest/getting_started.html (visited on 03/2021).