A DNN implementation on FPGAs from the existing DNN framework using HLS

Hyunjae Kim[1,a)] Ryota Yamamoto[1] Shinya Honda[2] Masato Edahiro[1]

Abstract: DNN (Deep Neural Network) implementation on FPGAs has been actively conducted recently. Although some frameworks can use high speed and memory saving methods such as quantization, only a few frameworks are user friendly. Neural Network Console (NNC) is one of a few frameworks providing a user-friendly experience by GUI. In this paper, we explore implementing the network designed by the NNC on FPGAs. We used various optimization techniques using high-level synthesis, and evaluated execution time. In the case study, we implemented a network where all parameters were quantized to 8 bit integers. By implementing without using any floating-point arithmetic, we confirmed that the accuracy of inferences was comparable to the original network.

1. Introduction

Recently, there is a demand for Deep Neural Network (DNN) applications in embedded systems. For the convenience of DNN development, several DNN frameworks have been developed. Some frameworks can accelerate the calculation by utilizing an external computing device. This is due to CPU is designed for serial generic operations, which is not suitable for matrix calculation often required by DNN. Furthermore, the memory bandwidth of the CPU is relatively slower than others, which might lead to a bottleneck issue. For example, the bandwidth of a modern x86 CPU using DDR4 SDRAM can have a maximum bandwidth of 35.2GB/s. However, the bandwidth of modern GPU using GDDR6X SDRAM can have a maximum bandwidth of 1TB/s (Reference [1], [2]). For these reasons, a popular framework such as Tensorflow accelerates the calculation by utilizing the high bandwidth memory and parallel computing power of GPU. However, frameworks that utilize non-GPU hardware such as FPGA are relatively rare. Additionally, TensorFlow Lite for Microcontrollers [3] is available to implement a DNN inference application with quantized parameters. Nonetheless, it does not support FPGA.

Since many frameworks do not support implementing the network to embedded systems with FPGA, our final goal has set to make an automated tool that implements the network created by the existing DNN framework to FPGA. This tool is represented in **Figure 5**. As a pre-work, we review implementing networks created and trained by existing DNN frameworks in this paper. We chose Neural Network Console (NNC) as the target of its implementation. One of the reasons for choosing the NNC is that NNC provides to quantize its parameters and data that flow between layers. NNC supports quantization at the stage of training. This feature is important because calculating with floating-point arithmetic requires more cost comparing the calculation of an integer. Furthermore, by utilizing lower bit quantization, we could make the circuit size of FPGA relatively small. Moreover, NNC provides a user-friendly experience by utilizing its GUI. Therefore, if we can automate the implementation of NNC's network, we will be able to achieve productivity gains.

Implementation of FPGA using register-transfer level (RTL) language might be time-consuming and it can be very difficult for further optimization. Therefore, we used a system-level design toolkit named SystemBuilder with High-level synthesis (HLS) for development[4], [5]. We will present more detail about this process in section 3.

The contribution of this paper are as follows:

- The possibility of the implementation of network trained by existing DNN frameworks.
- A case study shows that inferences from implementation were the same as the original network from the framework.
- Testing the optimization technique using HLS.

2. Related Work

There are various studies on customized DNN accelerators using FPGAs. Zhang and Ye studied the methods to explorer optimal FPGA design in [6]. They developed a framework named DNNExplorer for modeling and exploring a design. They claimed that existing FPGA-based DNN accelerators typically fall into two design paradigms. The one is a paradigm that adopts a generic reusable architecture to support various DNN networks. This means this design sacrifices further performance and efficiency for generic reuse. The other is a paradigm that adopts a layer-wise tailor-made architecture to optimize a layer-specific demand. Each layer in this design is combined into pipeline implementation. This design loses scalability as it has a limited

[1] Nagoya University, Nagoya, Aichi 464–8603, Japan
[2] Nanzan University, Nagoya, Aichi 464–8673, Japan
[a)] hyunjae@ertl.jp

Table 1 Used programs inside SystemBuilder

Name	Version
SysGen	2.0
Vivado HLS	2019.1
GCC	6.3.1
Questa Sim	2019.10
Qemu	2.11.1

number of DNN layers that can be supported. And deeper DNN means fewer resources for each layer, which might lead to performance degradation.

The key idea of DNNExplorer is to use both paradigms at the same time. DNNExplorer utilizes the second's pipeline design for the fist up to the split-point (SP) layer and uses the first's generic accelerator for the rest. DNNExplorer explores design architecture to find out optimal split-point and it maps resources for each design architecture.

As we discussed in the Introduction, our final goal, including the work in this paper, is to create a tool to automatically implements the network from NNC. We believe this approach is a worthy reference for future design exploration of our models.

FINN is one of the DNN framework for an FPGA by Xilinx Inc [7], [8]. This framework targets quantized neural network, and the framework provides training environment to set bit width. Moreover, PYNQ (Python productivity for ZYNQ)[9] is a hardware platform to develop DNN inference application in Python. Developers can develop DNN inference more easily because they can develop on browser with Jupyter notebook.

Nakahara et al. developed GUINNESS (GUI based Neural Network Environment SyntheSizer) for DNN inference on an FPGA [10], [11]. GUINNESS also provides train front-end with Chainer, and supports from training to C++ code for HLS. The C++ code is inputted to SDSoC [12], then HDL code are generated for target FPGA. The hardware receives an activation from LINUX on CPU, however, the hardware co-work with small-scale and low overhead RTOS. Therefore, our framework has advantages for real-time systems and automotive systems.

3. High-Level Synthesis Using SystemBuilder

For the convenience of development, we used a system-level design toolkit named SystemBuilder. SystemBuilder is a toolkit that abstracts the interface between hardware and software such as the device interface and bus interface. Due to such abstraction, a designer can have better productivity by eliminating the need to do such low-level designs themselves.

In SystemBuilder, we call processes that run either hardware or software as functional units. As a result of the abstraction of the interface, SytemBuilder provides four types of communication channels between functional units. These channels are named as a communication primitive. Since the functional unit could be either hardware or software process, these primitive can also be used not only between hardware and software but also between hardware and hardware, or between software and software.

Designers should design the functional units and the connection between units using the provided communication primitive. The unit will be implemented either a hardware module or software process based on the system specification that the designer

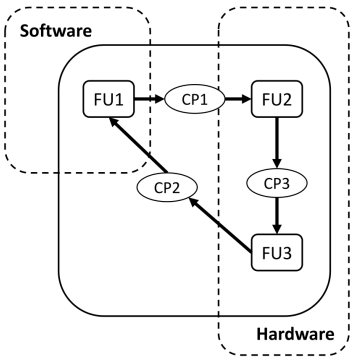

Fig. 1 Example of partitioning Functional Units

defined. We present more details of the described methods in section 3.1

SystemBuilder then generates RTL hardware, RTOS specific software, and interface of these. Synthesis is being run by an external tool that SystemBuilder executes. Since SystemBuilder does partitioning of hardware and software, a designer can now efficiently explore partitioning of hardware/software.

We present the detail of synthesis used in SystemBuilder in section 3.2. In section 3.3, we will explain the co-simulation feature that SystemBuilder provides. To summarize, we will discuss the entire workflow in section 3.4.

Since SystemBuilder is a toolkit, it includes several programs inside. For reference, we listed the program we used in **Table 1**. We used Ubuntu 18.04.5 for the host OS.

3.1 Design description

The designer should write follow files to use SystemBuilder.

- Functional units written in C language.
- Description of the design target. This file is named as System DeFinition (SDF) file. SDF file includes follow information:
  - Partitioning scheme of functional units.
  - The number and names of communication primitives.
  - Connection relations between functional units and communication primitives.

SystemBuilder reads these files and follows the steps presented in section 3.4.

3.1.1 Partitioning

Designers using SystemBuilder should define partition scheming in SDF File. This means the designer can decide which functional units to run on hardware or software. To test another partition scheming, the designer should edit the SDF file and rerun SystemBuilder. In this way, the designer now efficiently explore the partition scheming of the hardware/software. To test the implementation works as expected, the designer can run a co-simulation. Designers can even run faster testing by building all functional units as software. Since simulating the software is much faster than simulating the hardware, it is expected to be more efficient than running co-simulation. Co-simulation/simulation is explained in section 3.3.

3.1.2 Functional Units (FU)

As described in section 3.1.1, the designer can decide whether to implement the functional unit as hardware or software. However, there is a limitation. SystemBuilder uses an external tool

for HLS. It means the functional units to be implemented as a hardware module, must be subject to restrictions of the external HLS tool. On the contrary, units to be implemented as a software process does not have those restrictions. This means that all units can be built as a software process, and it is possible to test the design at a functional level.

3.1.3 Communication Primitives

SystemBuilder supports four communication primitive as follows.

Non-Blocking Communication Primitives (NBC)
It corresponds to a shared variable in software and a register in hardware.

Blocking Communication Primitives (BC)
It corresponds to OS's communication feature in software and a FIFO in hardware. Access is executed by blocking in one direction.

Memory Primitives
It corresponds to a global array in software and a dual-port memory in hardware. Access is executed by non-blocking.

PreFetch BC (PFBC)
It is a memory designed between external memory and hardware process. The purpose of PFBC is to hide latency while reading external memory. PFBC reads the bulk of data from external memory and provides fast internal memory to hardware processes connected.

These primitives can be accessed as a function of C language when designing the unit.

3.2 Synthesis

SystemBuilder takes SDF and C programs as an input and generates software, hardware, and interface.

3.2.1 Software Synthesis

Software synthesis takes place by the compiler. It will be compiled as software running on either ITRON or ATK. Communication primitives between the unit and software process are implemented as software.

3.2.2 Hardware Synthesis

Hardware modules and communication primitives between them will synthesis as hardware. SystemBuilder uses a commercial tool to synthesis hardware modules. SystemBuilder supports follow HLS tool.

- eXCite
- CyberWorkBech[13]
- Vivado HLS[14]

These HLS tools support various optimization techniques such as:

- Pipelining the loop to implement the operations in a loop in a concurrent manner.
- Unrolling the loop to exploit parallelism between loop iterations.
- Expanding the array signal into the individual variable.
- Unrolling multi-dimensional array by a dimension designer configured.
- Inlining the function.

To take advantage of these features, it is required for designers to design modules corresponding to it. In this paper, we had a brief compare test of pipelining the network represented in **Table 2** and **Table 3**.

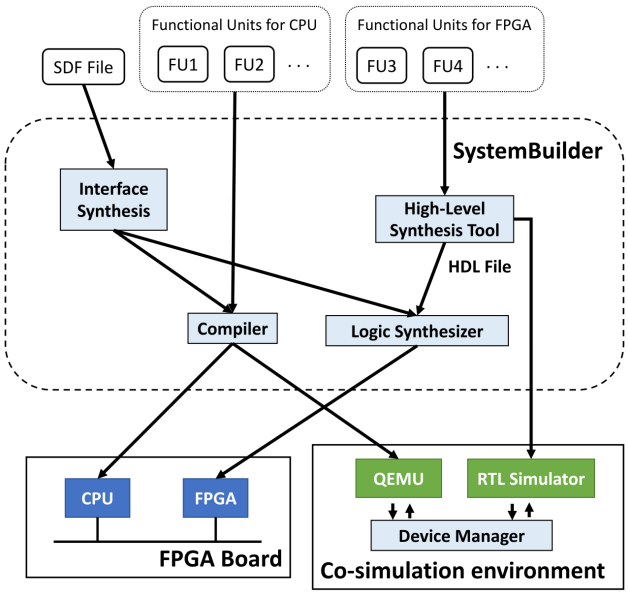

Fig. 2 Workflow inside SystemBuilder.

3.2.3 Interface Synthesis

SystemBuilder automatically generates a hardware/software interface based on the SDF file. For the software, the device driver is generated. The device driver includes access functions, interrupt handlers, and semaphores for synchronization between hardware and software. For the hardware, the interface circuit described in HDL is generated. This interface includes glue logic that allows accessing the bus and the circuit that corresponds to communication primitives such as buffer or memory. The glue logic has an interface to a generic bus named VBUS. Therefore, SystemBuilder generates a circuit that converts protocol to connect VBUS.

3.3 Co-simulation

To test the design, SystemBuilder supports hardware and software co-simulation. It runs RTOS and RTL simulator and then coordinating them to perform co-simulation. It connects the RTOS simulator and RTL simulator by a protocol named device manager. The device manager runs on the host computer then connects to the RTOS simulator using an inter-process communication function from the host pc. While the RTOS simulator tries to communicate with the RTL simulator, the device manager sends a signal to the RTL simulator. The device manager uses Foreign Language Interface (FLI) to connect VBUS connected to the RTL simulator.

SystemBuilder supports running software modules either on QEMU or directly from the host computer. Functional Units that run on RTOS, are compiled and linked with the RTOS model. Then SystemBuilder generates an object code which is directly executable on the host computer [15]. Due to native support, the speed of co-simulation is much faster than other simulators that simulate at the instruction-set level.

In this paper, we used QEMU for RTOS simulator and Questa Sim for RTL simulator.

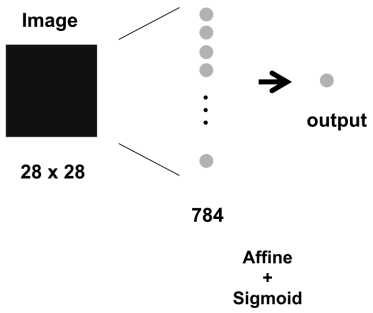

Fig. 3 Logistic regression.

3.4 Workflow

We represent the workflow inside SystemBuilder in this section. **Figure 2** and the following steps are an overview of workflow inside SystemBuilder.

(1) SystemBuilder reades SDF Files, and determines the partition scheme of hardware/software of each functional unit.
(2) SystemBuilder synthesis the interface between hardware and software.
(3) For software, the compiler compiles the functional units with the device driver generated in step 2.
(4) For hardware, HLS Tool synthesis the code to HDL files.
(5) Users now can choose whether to run co-simulation or implement the results to FPGA.
Co-simulation This process is described in section 3.3.
FPGA Logic synthesizer synthesis the HDL file created and device interface generated in step 2.

4. Case Study

4.1 DNN Models and Implementation

For the case study, we used two DNN models provided by NNC. The one is network named logistic regression. The other is network named Binary CNN. These network gets handwriting images of 4 or 9 as an input, and then determine whether it is 4 or 9. The input image size is 28x28 pixels with one color channel. Since our main goal in this paper is to verify the implementation, we did not implement one process per one layer but implemented all layers in one process.

To implement the network, we followed the following steps.

(1) Implement network from running on the x86 computer. All network was written from scratch.
(2) Checked whether the network we created makes the same inference as NNC.
(3) Modify the code for SystemBuilder.
(4) Run the functional level simulation.
(5) Confirm our network using co-simulation. We used ITRON as RTOS, and we used Vivado HLSto synthesize the hardware module.

All network we used can be referred to at reference [16].

4.1.1 Quantization

Using floating-point arithmetic on FPGA would have performance degradation compared to using an integer. Therefore we did not want to use any floating-point arithmetic. We edited the networks to use quantization. We quantized all parameters and all data that flow between layers. We quantized all data to 8-bit integers, and we set the size of 1-bit as the power of two. After quantization of the data, data now will be expressed as follows:

$$\text{original data} = \text{quantized data} \times \delta, \quad \delta = 2^{-n} \ (n = 1, 2, \ldots)$$

δ stands for the size of 1 bit in the quantized integer. Since we are using δ as the power of two, we can calculate between the quantized data with low cost. It is because we can use shift operator for adjusting δ.

4.1.2 Logistic regression

Logistic regression is a simple network that is formed with two layers. This network can be referred to from **Figure 3**. For the sigmoid layer, We used the Taylor series of sigmoid functions that can be referenced in [17]. However, we could not get the correct result by following the equation in [17]. We expect this is because the calculations have occurred with the low precision of 8-bit integers. To solve this problem we made exception handling for the input that did not match the correct result.

4.1.3 Binary CNN

After successful implementation of logistic regression in section 4.1.2, we tried to implement another network. We used the DNN Model which includes the convolution layer. It is because we believed that the convolution layer has room for further optimization with a loop statement. Which could be useful to test the optimization provided by the HLS tool as future work. We used Binary CNN from example networks provided by NNC. This network can be referred from **Figure 4**.

We used a lookup table for the activation layer such as tanh and sigmoid. It is because as we present in section 4.1.2, it was difficult to achieve the correct result while calculating with the low precision of 8-bit integers.

4.2 Training and Accuracy

To train and evaluate the networks, we used MNIST dataset [18]. We used 1500 images for training. After training the network using NNC, we extract the trained parameter from the result. Then we implement the parameter to the network we build. While NNC stores parameters in 32-bit float, we converted to an integer by quantized parameter we set.

To evaluate the network, we used 500 images. For logistic regression, we got 95.2% of inference accuracy from NNC, and 98.0% of accuracy for binary CNN. We used memory primitives for the interface between hardware and software. After implementing the parameter, we confirmed that the same inference by co-simulation.

4.3 Results

We have confirmed that our implementation shows the same inference as the original. Next, we measured the performance of our implementation. We measured two kinds of data. One is the entire latency from data transmission to output of the FPGA. This data is listed in Table 2. For logistic regression, we calculate the average of 100 inference, and for binary CNN we used the average of 5 inference. The other measured performance is latency without data transmission, which is the pure computation latency of the network. This data is listed in Table 3

For the logistic regression, we have tested the optimization

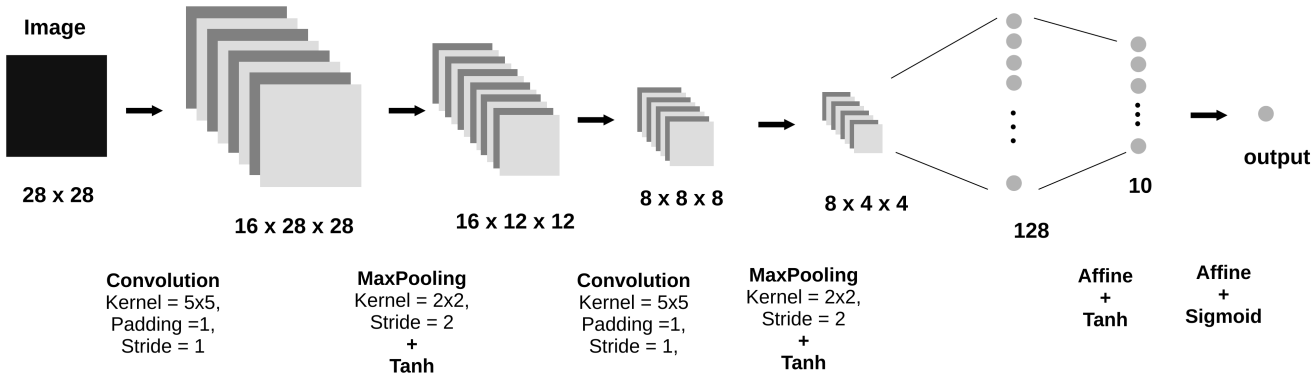

Fig. 4 Binary CNN.

Table 2 A result of case study including loading time of input image.

Target	Network	Optimization	Latency (μs)	Throughput (fps)	BRAM Usage (kB)
ZYBO @ 100MHz	Logistic Regression	-	347	2884	162
		pipeline	330	3031	162
	Binary CNN	-	18593	53.8	209

Latency for logistic regression : `average execution time of 100-discussion`
Latency for binary CNN : `average execution time of 5-discussion`

technique using the HLS tool described in section 3.2.2. We have pipelined the loop statement in the affine layer. As a result, we had about 5 percentage of performance improvement (Table 2). However focusing on the computation time itself, there was a 35 percent performance improvement.

4.4 Discussion

Comparing Table 2 and Table 3, it shows a difference in latency of approximately 284 μs. This latency is the time required for the FPGA to read the image from memory. For binary CNN, it is about 1.5 percentage of entire latency, but for logistic regression, it is about 82 percent of entire latency. Memory bandwidth is being a huge bottleneck for logistic regression. For future work, testing other communication primitives such as prefetch BC will be worth it.

Implementing a sigmoid layer or a tanh layer from the definition might require to realize the exponential function. However, implementing an exponential calculation of quantized numbers was a difficult task for us. Consequently we built the sigmoid function using the Taylor series in section 4.1.2. We believed the loss using the Taylor series would be negligible. It is because we thought the loss in quantizing the results would be much bigger to be able to hide the loss in the Taylor series. We used multiple Taylor series for multiple domains. We found out that at the boundary of each domain, the output does not monotonically increase. We expect this is due to the low precision calculations of 8-bit integers.

To solve this problem we took a different approach in section 4.1.3. We prepared 256 (= 2^8) size output an array that corresponds to 256 cases of input. In other words, we made a lookup table.

Apart from the difficulty of making, and in terms of performance and circuit size, we believe that comparing the lookup table and implementing it as a function is worthy of discussion.

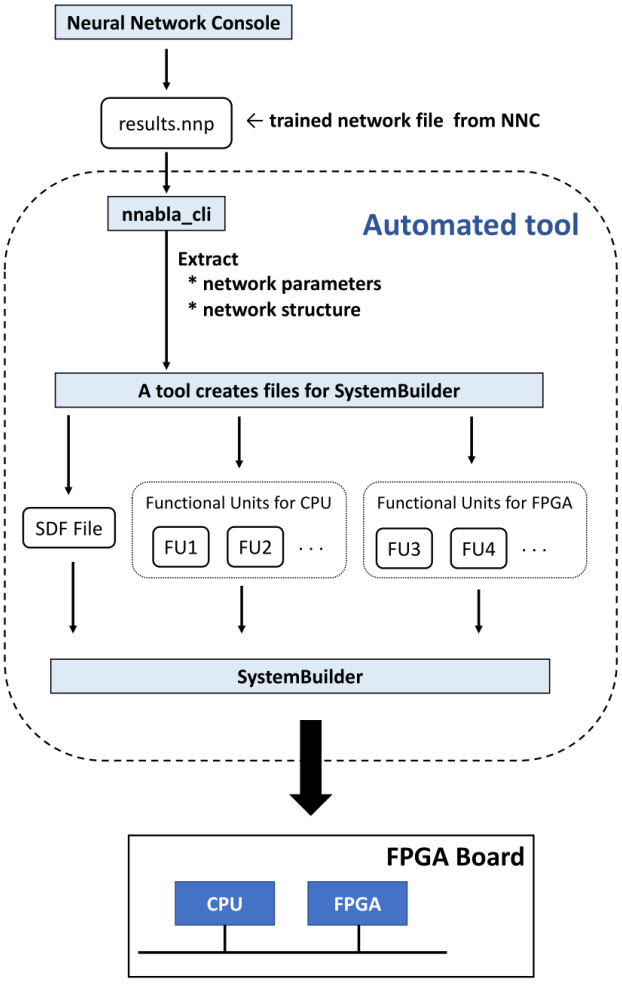

Fig. 5 Automated tool.

5. Conclusion

In this paper, we reviewed the possibility of the implementation of existing DNN frameworks. To achieve this, we implemented

Table 3 A result of case study without loading time of input image.

Target	Network	Optimization	Latency (μs)	Throughput (fps)	BRAM Usage (kB)
ZYBO @ 100MHz	Logistic Regression	-	61.5	16260	162
		pipeline	45.5	21978	162
	Binary CNN	-	18311	54.6	209

Latency : `execution time for 1-discussion`

two networks created and trained from NNC. In order not to use floating-point arithmetic, all the data in the network was quantized from the stage of training. By implementing two networks from scratch, We confirmed that inferences from implementation were the same as the original from NNC.

Our final goal is to create automated tool that implement the network created and trained in NNC. These are some works that should be done in the future.

- Optimize each layer using the feature provided by HLS, such as pipelining the loop. In which explained in section 3.2.2.
- Test various communication primitives described in section 3.1.3.
- Implement every layer that NNC supports.
- Build a tool that extracts network structures and create files for SystemBuilder.
- Explore a design of the network, and make the optimal decision.

References

[1] Micron Technology: Memory Speeds and Compatibility, `https://www.crucial.com/support/memory-speeds-compatability`. Accessed: 2020-11-19.

[2] Micron Technology: GDDR6X, `https://www.micron.com/products/ultra-bandwidth-solutions/gddr6x`. Accessed: 2020-11-19.

[3] Google: TensorFlow Lite for Microcontrollers, `https://www.tensorflow.org/lite/microcontrollers`. Accessed: 2020-11-16.

[4] Honda, S., Tomiyama, H. and Takada, H.: SystemBuilder: A system level design environment (in Japanese), *The IEICE Transactions on Information and Systems(Japanese Edition)*, Vol. 88, No. 2, pp. 163–174 (2005).

[5] Honda, S., Tomiyama, H. and Takada, H.: RTOS and codesign toolkit for multiprocessor systems-on-chip, *2007 Asia and South Pacific Design Automation Conference*, IEEE, pp. 336–341 (2007).

[6] Zhang, X., Ye, H., Wang, J., Lin, Y., Xiong, J., Hwu, W.-m. and Chen, D.: DNNExplorer: A Framework for Modeling and Exploring a Novel Paradigm of FPGA-based DNN Accelerator, *arXiv preprint arXiv:2008.12745* (2020).

[7] Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P., Jahre, M. and Vissers, K.: FINN: A Framework for Fast, Scalable Binarized Neural Network Inference, *FPGA 2017*, ACM, pp. 65–74 (2017).

[8] Blott, M., Preußer, T. B., Fraser, N. J., Gambardella, G., O ' brien, K., Umuroglu, Y., Leeser, M. and Vissers, K.: FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks, *TRETS 2018*, Vol. 11, No. 3, p. 16 (2018).

[9] Xilinx: PYNQ - Python productivity for Zynq, `http://www.pynq.io/`. Accessed: 2020-11-19.

[10] Nakahara, H., Fujii, T. and Sato, S.: A fully connected layer elimination for a binarizec convolutional neural network on an FPGA, *FPL 2017*, IEEE, pp. 1–4 (2017).

[11] Yonekawa, H. and Nakahara, H.: On-chip memory based binarized convolutional deep neural network applying batch normalization free technique on an FPGA, *IPDPSW 2017*, IEEE, pp. 98–105 (2017).

[12] Xilinx: SDSoC Development Environment, `https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html`. Accessed: 2020-11-19.

[13] Wakabayashi, K.: CyberWorkBench: integrated design environment based on C-based behavior synthesis and verification, *In VLSI Design, Automation and Test, 2005. (VLSI-TSA-DAT)*, IEEE, pp. 173–176 (2005).

[14] Xilinx: Vivado High-Level Synthesis, `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`. Accessed: 2020-9-26.

[15] Honda, S., Wakabayashi, T., Tomiyama, H. and Takada, H.: RTOS-centric hardware/software cosimulator for embedded system design, *Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 158–163 (2004).

[16] Sony: Project - Neural Network Console, `https://dl.sony.com/project/`. Accessed: 2020-11-19.

[17] Çetin, O., Temurtaş, F. and Gülgönül, Ş.: An application of multilayer neural network on hepatitis disease diagnosis using approximations of sigmoid activation function., *Dicle Medical Journal/Dicle Tip Dergisi*, Vol. 42, No. 2 (2015).

[18] LeCun, Y., Cortes, C. and Burges, C. J.: The MNIST Database, `http://yann.lecun.com/exdb/mnist/`. Accessed: 2020-11-18.