# AutoDeepHLS: Deep Neural Network High-level Synthesis using fixed-point precision

Mohammad Riazati, Masoud Daneshtalab, Mikael Sjödin, and Björn Lisper
Heterogeneous Systems Research Group, Mälardalen University, Västerås, Sweden
{mohammad.riazati, masoud.daneshtalab, mikael.sjodin, bjorn.lisper}@mdh.se

***Abstract*—Deep Neural Networks (DNN) have received much attention in various applications such as visual recognition, self-driving cars, health care, etc. Hardware implementation, specifically using FPGA and ASIC due to their high performance and low power consumption, is considered an efficient method. However, implementation on these platforms is difficult for neural network designers since they usually have limited knowledge of hardware. High-Level Synthesis (HLS) tools can act as a bridge between high-level DNN designs and hardware implementation. Nevertheless, these tools usually need implementation at the C level, whereas the design of neural networks is usually performed at a higher level (such as Keras or TensorFlow). In this paper, we propose a fully automated flow for creating a C-level implementation that is synthesizable with HLS Tools. Various aspects such as performance, minimal access to memory elements, data type knobs, and design verification are considered. Our results show that the generated C implementation is much more HLS friendly than previous works. Furthermore, a complete flow is proposed to determine different fixed-point precisions for network elements. We show that our method results in 25% and 34% reduction in bit-width for LeNet and VGG, respectively, without any accuracy loss.**

***Index Terms*—Deep Neural Network, Accelerator, High-Level Synthesis, Fixed-Point, Quantization**

## I. Introduction

Deep Neural Networks (DNN) are neural networks with more than one hidden layer. Running a DNN (a.k.a. inference) can be done on a variety of platforms such as CPU, GPU, FPGA, and ASIC, each of which has its own advantages and disadvantages. Since FPGAs provide high performance, low power consumption, and fast prototyping, their application for inference is considered by users. In this paper, FPGA is usually used in descriptions and experimental results, but virtually all procedures, methods, and capabilities can also be used for ASIC.

The main issue of implementing a DNN on an FPGA is that DNN design tools use high-level languages and libraries such as Keras, TensorFlow, PyTorch, and Caffe in Python and do not provide an output for FPGA. To fill the gap between very high-level design of DNNs and low-level implementations for FPGA, HLS tools were created. However, these tools do not accept such very high-level descriptions, and most of them need a design in C. Besides, not all the C implementations are synthesizable to hardware. DNN designers, who usually have limited knowledge of hardware, need an intermediary flow to create a proper implementation for FPGA.

In this paper, a fully automatic flow is presented that receives an input in Keras and produces an implementation of the inference without any need for user intervention. The main contributions of this paper are as follows:

- A complete toolchain is proposed, implemented, and presented in detail that covers all the steps needed to implement a DNN on FPGA. Steps include generating C code from Keras, validating the C implementation, quantization effect analysis, and finally synthesizing the neural network using HLS tools.
- Various implementation knobs are considered to make it possible to easily set the data type used to implement the whole design or every specific layer.
- The created C code is in standard ANSI C, which is supported by most commercial or open-source HLS tools.
- In creating the C code, it is tried to use minimal calculations, temporary arrays, and non-local variables. They could downgrade the performance of the generated circuit.
- Features for verifying the accuracy of the generated C code are added to ensure that the C implementation is functionally equal to its very high-level counterpart in Keras.
- A complete procedure is proposed in detail to help designers find the appropriate size and configuration of the fixed-point data types for each data element in a way that has the least impact on accuracy.

## II. Related work

Implementing DNNs on FPGAs has been addressed in many works. Some of them propose creating a code at the Register Transfer Level (RTL) [1], [2]. RTL implementations can be directly synthesized on an FPGA using conventional synthesis tools. Some others, e.g., [3]–[5], propose implementation on heterogeneous systems. Operations, such as addition, multiplication, or convolution, are implemented in the programmable logic and are controlled and scheduled in the CPU using frameworks like OpenCL [6]. Another related approach is implementing a domain-specific processor in the programmable logic, which executes the DNN related operations with a higher performance than general-purpose CPUs. Two examples are Xilinx Deep Learning Processor Unit (DPU) and Intel Vision Processing Unit (VPU), which are configured and implemented using Vitis and OpenVINO, respectively.

The first problem with these methods is that the user cannot modify the generated output by considering the limitations or freedoms in accuracy, performance, area utilization, or quantization level. Secondly, in practice, they cannot be used by neural network designers, who are mostly users with limited hardware knowledge [7].

Another category of works transforms a DNN to a lower-level implementation in C/C++. It can then be implemented on hardware platforms using HLS tools. Using these methods, the end-user may have the freedom to alter and tweak the design to get the desired performance, accuracy, and area utilization. The user may prefer to sacrifice the accuracy for lower latency or the latency for lower area utilization. It is required that the generated code be synthesizable. For instance, four of these works generated code that could not be synthesized by HLS [8]–[11], and their main target was embedded CPUs. There were only a few cases that could be synthesized by the HLS. They create a synthesizable C++ version of a neural network [12]–[14]. They do not provide features like verification, testbench, or finding the proper quantization size and configuration. Besides, some of them can only be used for small networks such as LeNet [15], and large networks like VGG [16] are not supported. Finally, the method in [17] generates a synthesizable C implementation, but only one data type for all data elements is supported.

Another topic that should be considered in the related work is the quantization technique or using Fixed-Point (FxP) data type instead of Floating-Point (FlP) data type. If FxP is used, then its configuration (including the size and the Fixed-Point position) becomes important. In [2] and [18], 16-bit data type using eight bits for integer and fractional portions is used for all data elements. Many others use 16-bit data types without explicitly mentioning the exact configuration [1], [4], [19]. Some others have added the support for FxP implementation, but the configuration should be determined by the user [7], [17], [20]. To the best of our knowledge, this is the first work that extensively analyzes the impact of FxP data types and proposes a systematic method for determining the proper FxP configuration.

## III. AutoDeepHLS Flow

DNNs are usually designed in very high-level languages and libraries. These descriptions are not synthesizable on FPGAs by HLS tools and require methods to automatically create an equivalent instance in a lower level such as C. In this paper, we consider the input to be in Keras [21]. For output, ANSI C [22] is generated, which is supported by almost every HLS tool. AutoDeepHLS flow receives input in Keras and, through several steps and by examining various aspects, creates an optimal implementation in C to be synthesized on an FPGA. Figure 1 shows the overall flow of AutoDeepHLS. In this section, each of the steps is explained in detail.

***Preprint submitted to IEEE International Conference on Artificial Intelligence Circuits and Systems (AICAS 2022)***

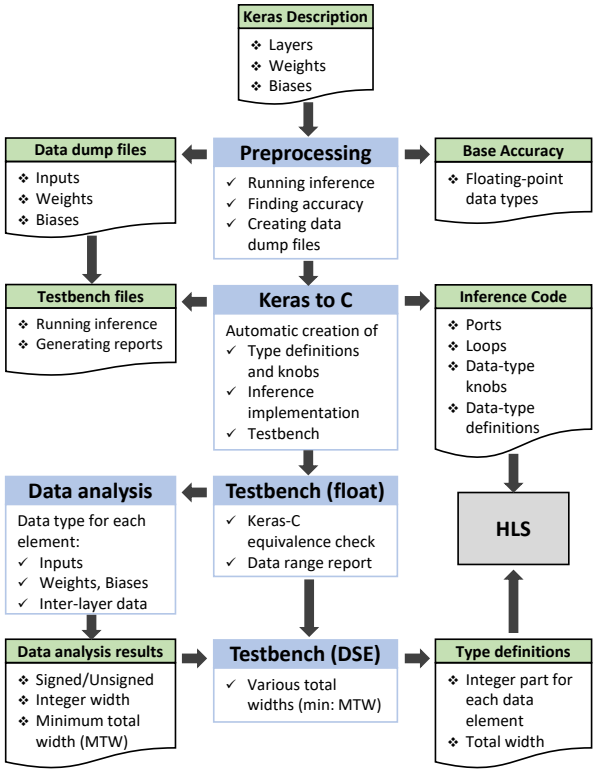


Fig. 1. The overall view of the proposed method

### A. Preprocessing

Before creating the C implementation begins, it is needed to extract the required data for the DNN under process. It is assumed that the DNN is already designed, trained, and tested. Preprocessing stage is implemented in Python to enable the extraction of the data seamlessly in the same environment that the neural network is implemented and trained. Preprocessing receives a trained DNN, which contains layers' specifications, weights, and biases.

In this stage, input data (e.g., images), weights, and biases of the DNN are converted to multidimensional arrays in standard C format. Another output of this stage is the base accuracy of the DNN using these specific input data, weights, and biases. Keras uses floating-point data type (FlP) for processing the DNN, and hardware implementations mostly use the fixed-point data type (FxP). The accuracy obtained at this step (base accuracy) using FlP is later used for two purposes: conversion validation and FxP impact analysis.

### B. Keras to C

This stage is the main part of creating the C implementation. In this stage, Keras implementation is processed to extract all the required information. A DNN description in Keras is relatively concise, and most of the information about the layers is not explicitly provided. However, to generate a C code, it is necessary to have various information about the network layers, including layer type (e.g., convolutional or fully connected), the number of filters, kernel size, stride size, padding type (valid or same), type of activation function (e.g., relu), and layer input and output sizes. They can either be directly extracted from the Keras function for a specific layer or by analyzing previous layers.

In this stage, two outputs are created. The first output is the C implementation of the DNN inference that is ready to be synthesized on the FPGA by the HLS tool. It includes ports, loops, data definitions, etc. It also comes with several knobs in order to simply switch between various data types. These knobs and how they can be set are explained later. The second output is testbench files. Testbench, like a wrapper, is placed around the inference code, and in addition to applying input data (e.g., input image), is responsible for monitoring and recording inputs, outputs, and internal values of the network, and it ultimately creates the necessary reports.

In creating C implementation, several measures are taken to enhance the usability and performance of the circuit which is being generated based on it:

- The first and most crucial aspect to consider is that the code is created such that it can be adequately synthesized on an FPGA. The HLS processes are very sensitive to the loop layout, use of variables and constants instead of accessing memory arrays, and locality of the accesses.
- The generated C code is able to use various data types for each of the data elements, such as inputs, outputs, weights, biases, and layer data, through easy-to-use knobs.
- The code created for inference does not only perform the main math operations but also embodies features that, when the testbench executes it, it can monitor and save all the data elements. They are required in the future stages.
- In a neural network, different data elements, including inputs, outputs, biases, weights, and layer data, are needed. In the case of layer data, both internal FPGA storage (such as BRAM) and external memory (such as DDR) can be used. In C code generation, both of these approaches are supported to be able to implement even very large networks with many layers or large layers dimensions.
- Input data sets, especially when many of them are to be applied, can be very large. The number of weights and biases can also be very high for large DNNs. To ensure scalability, file-to-memory mechanisms are included in the testbench.
- In some of the previous works, the code for each layer or operation (such as convolution) is implemented as a function. But in this work, the inference code to be synthesized is fully flat (without using functions) and in a single file. This is very important because, firstly, calls to the functions add extra states and latency when synthesized, and secondly, a flat implementation allows the HLS tool to make inter-layer optimizations.
- All for-loops have labels. This will lead to more readable HLS reports. Additionally, it allows the designers or tool developers to use external HLS directive files, e.g., directives.tcl file for Xilinx Vivado HLS.

### C. Running testbench in floating-point mode

As mentioned earlier, testbench, like a wrapper, is placed around the inference code and connects the input and output data to it. Before we can run the testbench, hence the inference code, we need to determine the data type to be used for each of the data elements in the network through the knobs that are defined. These data types may be all FlP or FxP, or different FxP settings for each of them. In this stage, testbench is executed using the FlP mode for all network data elements (including inputs, outputs, weights, biases, and layer data). There are two main objectives for this execution.

- When a Keras description of a DNNs is executed on a CPU or a GPU, FlP is used. Therefore, if the inference in C is executed using the same input data and the same type of data, it should have the same accuracy as the one obtained in the preprocessing stage as the base accuracy. In fact, by executing the testbench in FlP mode, an equivalence check is performed to ensure that the conversion is done correctly.
- The main question about using FxP is that what configuration for each FxP data type should be selected. This configuration includes the total width, the length of the integer portion, and the length of the fractional portion. During the execution of the inference in FlP mode, all the values stored in data elements are monitored, and the required information for finding a proper FxP configuration is extracted and stored. They will be used in the later stages.

### D. Data Analysis

At this stage, various data that were extracted and stored in the stage of running testbench in FlP mode are analyzed to select appropriate data types. The required knobs to determine the type of data for each of the elements in the network are already integrated into the code, and in this stage (and the next), we are going to find how to adjust those knobs. Data elements include inputs, outputs, weights, biases, and data for each layer separately.

One of the three modes for the implementation and execution of the inference can be selected.

- Floating-point mode: In this mode, for all data elements in the DNN, the FlP data type is chosen.

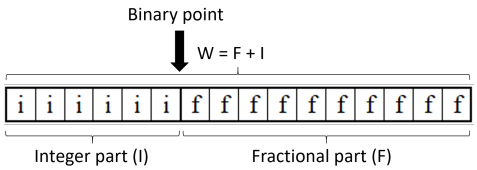


Fig. 2. Fixed-point format and its constituent parts

- Single Fixed-point mode: An FxP data type configuration is determined by three main characteristics: number of bits in its integer portion (I), number of bits in its fractional portion (F), and signedness (S). The total width (W) equals I + F. The integer part can be signed or unsigned, and it determines the type of the whole number. Figure 2 shows how an FxP is made of these parts. In this mode, all data types have the same FxP configuration, i.e., identical W, I. In this mode, all data types are signed. For example, as will be discussed later, a LeNet network on the MNIST data set can be implemented in this mode with W and I equal to 16 and 8, respectively, without losing any accuracy.
- Multiple Fixed-point mode: In this mode, like the previous mode, FxP data type is used for all data elements, with the difference that each of the FxP types can have different I and S. Note that W is equal for all types. The rest of the stages in this section cover how the configuration for each of the data types is determined.

To determine I and signedness of a specific data element, all the values that a (scalar or array) variable has received should be analyzed to find their minimum and maximum. These values are monitored and stored in the stage of running testbench in FlP mode. Then, based on them, Equation 1 is used to determine I. Regarding the sign, if the minimum value is smaller than zero (i.e., at least once, the element has received a value less than zero), then a signed type should be chosen, and otherwise, an unsigned data type suffices.

$$I = max(\lfloor \log_2 max(|\text{MinValue}|, |\text{MaxValue}|) \rfloor, -1) + 1 \qquad (1)$$

Another result from this stage is the minimum data width. Since W is common for all FxP types, it should be able to fit (at least) the integer part of all types. Therefore, when I is calculated for all types, the maximum I is chosen as Minimum Total Width (MTW). Most likely, the W of the DNN will be larger than MTW. It will be found in the next stage.

### *E. Running testbench for various data widths*

In the previous stage, the width for the integer part of each data element was found. Then, we need to find the whole width (and, as a result, the width for the fractional part of each of them). This stage could not be performed by analysis or a formula, and we need to examine different widths to evaluate their impact on the network accuracy. Therefore, testbench is executed for various values for W beginning from MTW. It continues until the accuracy drop is perceived as negligible.

### *F. Running HLS*

At this point, the configuration for each of the data types is known. These configurations are used to set the knobs already integrated into the design. Now, all the ingredients, including the source files and the knobs, are ready to run the HLS.

## IV. Experimental results

To demonstrate the performance of the presented work, we first provide the results of the stages for two well-known networks. Then, to explain how HLS friendly is our generated code, we compare the proposed method with a recent work.

### *A. Tool stages results*

To verify the performance of the presented method, we tested it on two well-known networks. The first, LeNet [15], is a seven-layer network that, using MNIST [23] dataset, is designed to detect digits in black and white photos with dimensions of 32x32. The second one, VGG [16], consists of 21 layers and receives colored images with dimensions of 224x224 pixels and classifies them in one of 1,000 possible categories. The reason for choosing these two is that they are among the smallest and largest networks that have been used in most articles that provide hardware implementation for neural networks. After completing the Preprocessing and KerasToC stages, the next stage is to run the inference in C in floating-point mode on the same data that was used in the Preprocessing stage. The obtained accuracy in this stage should be compared with the accuracy obtained in the Preprocessing stage. This is to ensure that the conversion is done without error. This was done for both networks. Another output of this stage is a list of minimum and maximum stored values in every variable or array. The obtained values are then examined in the Data Analysis stage to find the data type configurations, including the required number of bits to store the integer part, according to Equation 1, as well as being signed or unsigned. Tables I and II show the results for LeNet and VGG, respectively. Note that the values shown in the table are rounded.

TABLE I
DATA ANALYSIS RESULT FOR LENET

| Data Element | MinValue | MaxValue | Signedness | Int. Width (bits) |
|---|---|---|---|---|
| **Biases** | -0.050506 | 0.132952 | Signed | 1 |
| **Weights** | -0.603276 | 0.942556 | Signed | 1 |
| **Inputs** | 0 | 1 | Unsigned | 1 |
| **Data (Layer 1)** | 0 | 6.844573 | Unsigned | 3 |
| **Data (Layer 2)** | 0 | 6.844573 | Unsigned | 3 |
| **Data (Layer 3)** | 0 | 13.107732 | Unsigned | 4 |
| **Data (Layer 4)** | 0 | 13.107732 | Unsigned | 4 |
| **Data (Layer 5)** | 0 | 15.539978 | Unsigned | 4 |
| **Data (Layer 6)** | 0 | 13.975531 | Unsigned | 4 |
| **Data (Layer 7)** | 0 | 18.596546 | Unsigned | 5 |

TABLE II
DATA ANALYSIS RESULT FOR VGG

| Data Element | MinValue | MaxValue | Signedness | Int. Width (bits) |
|---|---|---|---|---|
| **Biases** | -1.027151 | 9.431553 | Signed | 5 |
| **Weights** | -0.6714 | 0.608516 | Signed | 1 |
| **Inputs** | -123.68 | 151.061005 | Signed | 9 |
| **Data (Layer 1)** | 0 | 1056.574219 | Unsigned | 11 |
| **Data (Layer 2)** | 0 | 4527.906738 | Unsigned | 13 |
| **Data (Layer 3)** | 0 | 4527.906738 | Unsigned | 13 |
| **Data (Layer 4)** | 0 | 7646.894043 | Unsigned | 13 |
| **Data (Layer 5)** | 0 | 14836.139648 | Unsigned | 14 |
| **Data (Layer 6)** | 0 | 14836.139648 | Unsigned | 14 |
| **Data (Layer 7)** | 0 | 16868.339844 | Unsigned | 15 |
| **Data (Layer 8)** | 0 | 15843.737305 | Unsigned | 14 |
| **Data (Layer 9)** | 0 | 17463.654297 | Unsigned | 15 |
| **Data (Layer 10)** | 0 | 17463.654297 | Unsigned | 15 |
| **Data (Layer 11)** | 0 | 12117.775391 | Unsigned | 14 |
| **Data (Layer 12)** | 0 | 6396.614746 | Unsigned | 13 |
| **Data (Layer 13)** | 0 | 4099.385742 | Unsigned | 13 |
| **Data (Layer 14)** | 0 | 4099.385742 | Unsigned | 13 |
| **Data (Layer 15)** | 0 | 2423.161377 | Unsigned | 12 |
| **Data (Layer 16)** | 0 | 1097.153931 | Unsigned | 11 |
| **Data (Layer 17)** | 0 | 566.926697 | Unsigned | 10 |
| **Data (Layer 18)** | 0 | 566.926697 | Unsigned | 10 |
| **Data (Layer 19)** | 0 | 111.201981 | Unsigned | 7 |
| **Data (Layer 20)** | 0 | 24.104248 | Unsigned | 5 |
| **Data (Layer 21)** | 0 | 30.676727 | Unsigned | 5 |

According to the data obtained from data analysis, the maximum integer width of all data elements will be the minimum total width (MTW) required for all elements. Therefore, MTW for LeNet and VGG will be 5 and 15 bits, respectively.

Before running the HLS, the last stage of data width determination is to find a total width that has the least impact on accuracy. For example, for the LeNet network, if the total width is equal to 5 bits, i.e., equal to MTW, there will be four bits remaining for the fractional part of biases, weights, and inputs and two, one, and zero fractional bits for the layer data of the first, sixth, and seventh layers, respectively. Therefore, in the last stage, we will test various total widths beginning from MTW bits until the accuracy loss is minimal.

Figure 3 illustrates the accuracy loss for both single- and multi-fixed-point modes for LeNet. As shown, in multi-fixed-point mode, at least 14 bits are needed to have an accuracy loss of zero, and the accuracy loss for 12 and 13 bits is negligible. As for the conventional method of single-fixed-point mode, which uses a common configuration for all of the data elements, even by using 16-bit data types, there is a slight accuracy loss, and with 12-bit data types, the network, with 59% error, is virtually unusable.

In the case of a large network like VGG, the difference is much more considerable. As can be observed in figure 4, in single-fixed-point mode, even by using 32-bit data types, there is some accuracy loss (equal to 1%), and using a data width less than 31 bits makes the network much different from the execution using FlP. This contrasts with the fact that the accuracy loss is zero when 23 bits are used in the multi-type method, and with a 20-bit width, the accuracy loss can be ignored for some applications.

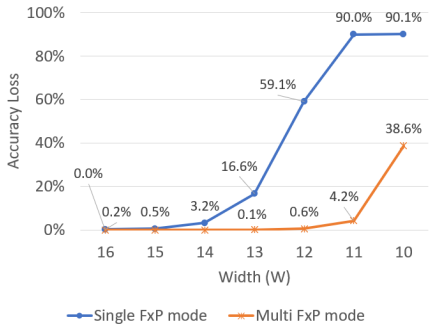


Fig. 3. Accuracy loss as a function of total width (LeNet)

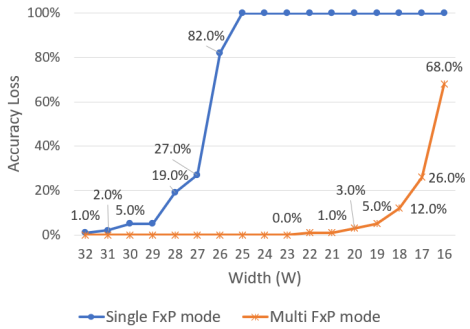


Fig. 4. Accuracy loss as a function of total width (VGG)

After determining the appropriate data type configurations for network elements, the C implementation is ready to be synthesized using HLS. In order to show the impact of the data width on the resource utilization, the Lookup Table (LUT) and Flip Flop (FF) utilization for two different sizes are also shown in Table III. To produce these results, we used Xilinx Vivado HLS tool on a Xilinx FPGA (xczu7ev-ffvc1156-2-e).

It should be noted that in this work, in order to make the results reproducible, we used the datasets, weights, and biases provided by the Keras library without any modification or improvement. The results can differ if quantization-aware training [24] or post-training quantization [25] were used. These enhancements can result in lower data widths, e.g., 16 bits for VGG.

TABLE III
RESOURCE UTILIZATION AFTER HLS SYNTHESIS

| | LeNet | | VGG | |
|---|---|---|---|---|
| **Fixed-point width (bits)** | 16 | 12 | 32 | 21 |
| **Accuracy loss** | 0.00% | 0.60% | 0% | 1% |
| **LUT** | 38661 | 31623 | 240644 | 186711 |
| **FF** | 8754 | 7325 | 86537 | 64398 |

### *B. Comparison with previous work*

In order to compare the results of our tool with similar works, we investigated various publications and tools. Some of them did not generate pure C implementations, and some others were only applicable to just small DNNs. To the best of our knowledge, [14] is the only work that can convert large deep neural networks, like VGG, to a pure C implementation. It uses an LLVM-based approach by connecting various third-party tools to create the final C code. It does not provide testbench, verification, and quantization determination, and therefore, we just compare the base C implementation results without applying any optimizations. Their generated C code uses eight bits for both weights and internal layer data. We configured our knobs similarly and implemented the network on the same device (Xilinx VU9P FPGA) with a similar clock frequency (200 MHz). As shown in Table IV, the latency of the present work for VGG is slightly lower, and the resource utilization is much lower. This is because a minimal number of C statements, calculations, temporary arrays, and non-local variables are used. In essence, the generated code by our tool is much more HLS friendly.

TABLE IV
COMPARING RESULTS WITH [14]

| | Latency (clk. cycles) | BRAM | DSP | FF | LUT |
|---|---|---|---|---|---|
| **ScaleHLS [14]** | 695,607,608 | 311 | 14 | 5760 | 16029 |
| **AutoDeepHLS** | 694,406,880 | 82 | 1 | 2186 | 6070 |
| **Gain** | 0.17% | 73.63% | 92.86% | 62.05% | 62.13% |

## V. CONCLUSION AND FUTURE WORK

In this work, we proposed a complete flow through which a DNN in Keras description could be converted to an implementation in C to be synthesized by an HLS tool. The flow began with generating the implementation and verifying the result and continued with finding proper fixed-point configurations for every data element in the DNN. We assumed that all the data types are supposed to have the same bit-width, and the only difference is in the binary point position. However, we believe that for some elements, like VGG biases, even a smaller bit-width will not cause accuracy loss. One of the future tracks is finding the proper bit-widths for each data element individually.

## REFERENCES


[1] J. Qiu *et al.*, "Going deeper with embedded fpga platform for convolutional neural network," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016, pp. 26–35.

[2] S. I. Venieris *et al.*, "fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas," *IEEE transactions on neural networks and learning systems*, vol. 30, no. 2, pp. 326–342, 2018.

[3] A. Ghaffari *et al.*, "Cnn2gate: Toward designing a general framework for implementation of convolutional neural networks on fpga," *arXiv preprint arXiv:2004.04641*, 2020.

[4] C. Zhang *et al.*, "Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38, no. 11, pp. 2072–2085, 2018.

[5] P. G. Mousouliotis *et al.*, "Cnn-grinder: From algorithmic to high-level synthesis descriptions of cnns for low-end-low-cost fpga socs," *Microprocessors and Microsystems*, vol. 73, p. 102990, 2020.

[6] J. E. Stone *et al.*, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[7] Y. Guan *et al.*, "Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 152–159.

[8] Aljabr0: from-keras-to-c. [Accessed Mar. 29, 2022]. [Online]. Available: https://github.com/aljabr0/from-keras-to-c

[9] Dobiasd: frugally-deep. [Accessed Mar. 29, 2022]. [Online]. Available: https://github.com/Dobiasd/frugally-deep

[10] gosha20777: keras2cpp. [Accessed Mar. 29, 2022]. [Online]. Available: https://github.com/gosha20777/keras2cpp

[11] pplonski: keras2cpp. [Accessed Mar. 29, 2022]. [Online]. Available: https://github.com/pplonski/keras2cpp

[12] T. Aarrestad *et al.*, "Fast convolutional neural networks on fpgas with hls4ml," *Machine Learning: Science and Technology*, vol. 2, no. 4, p. 045015, 2021.

[13] Ai transformer - keras to c code converter. [Accessed Mar. 29, 2022]. [Online]. Available: https://twitter.com/aitransformer

[14] H. Ye *et al.*, "Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation," in *The 28th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022.

[15] Y. LeCun *et al.*, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[16] K. Simonyan *et al.*, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[17] M. Riazati *et al.*, "Deephls: A complete toolchain for automatic synthesis of deep neural networks to fpga," in *2020 27th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. IEEE, 2020, pp. 1–4.

[18] S. I. Venieris *et al.*, "fpgaconvnet: A framework for mapping convolutional neural networks on fpgas," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2016, pp. 40–47.

[19] K. Guo *et al.*, "Angel-eye: A complete design flow for mapping cnn onto customized hardware," in *2016 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*. IEEE, 2016, pp. 24–29.

[20] H. Sharma *et al.*, "From high-level deep neural models to fpgas," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–12.

[21] A. Gulli *et al.*, *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[22] H. Schildt, *The annotated ANSI C Standard American National Standard for Programming Languages—C: ANSI/ISO 9899-1990*. McGraw-Hill, Inc., 1990.

[23] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[24] Tensorflow - quantization aware training. [Accessed Mar. 29, 2022]. [Online]. Available: https://www.tensorflow.org/model_optimization/guide/quantization/training

[25] Tensorflow - post-training quantization. [Accessed Mar. 29, 2022]. [Online]. Available: https://www.tensorflow.org/lite/performance/post_training_quantization