DeepHLS: A complete toolchain for automatic synthesis of deep neural networks to FPGA

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

***Abstract*—Deep neural networks (DNN) have achieved quality results in various applications of computer vision, especially in image classification problems. DNNs are computational intensive, and nowadays, their acceleration on the FPGA has received much attention. Many methods to accelerate DNNs have been proposed. Despite their performance features like acceptable accuracy or low latency, their use is not widely accepted by software designers who usually do not have enough knowledge of the hardware details of the proposed accelerators. HLS tools are the major promising tools that can act as a bridge between software designers and hardware implementation. However, not only most HLS tools just support C and C++ descriptions as input, but also their result is very sensitive to the coding style. It makes it difficult for the software developers to adopt them, as DNNs are mostly described in high-level languages such as Tensorflow or Keras. In this paper, an integrated toolchain is presented that, in addition to converting the Keras DNN descriptions to a simple, flat, and synthesizable C output, provides other features such as accuracy verification, C level knobs to easily change the data types from floating-point to fixed-point with arbitrary bit width, and latency and area utilization adjustment using HLS knobs.**

***Index Terms*—Deep Neural Networks, Convolutional neural networks, CNN, accelerator, high-level synthesis**

I. Introduction

In recent years, Deep Neural Networks (DNN) have become very successful in computer vision and classification problems. Numerous tools are used to design and implement this type of networks, most notably Tensorflow and related libraries such as Keras, PyTorch, and Caffe. These tools provide the necessary facilities from the design stage of the neural network layers to the training and testing stages. After that the designing, training, and testing are complete, there exist two main outputs. First, the structure of the network layers is specified. It includes the dimensions of the layers, the dimensions of the kernels, the number of input and output layers, the activation functions associated with each layer, etc. Second, the weights and biases required for the inference stage are finalized and ready to be used. From this stage onwards, the use of neural networks in various applications such as classification begins. Performance and energy consumption are usually two of the most important parameters in a DNN hardware implementation. DNN demands high computational power. Processors usually do not provide the necessary performance for the inference step. Although GPUs are high-performance inference accelerators, they consume a lot of energy and cannot be deployed in many battery-based embedded applications. Therefore, the synthesis of the deep neural network on FPGA or ASIC seems promising. They offer a high level of parallelism and are much more efficient in terms of power consumption and performance [1]. Note that in this paper, we use FPGA to refer to hardware implementation, but the presented methods can also be used for ASIC.

The main challenge of implementing DNNs on FPGA is that neural network software designers usually have little knowledge of the underlying hardware and digital design, and this gap limits the use of FPGAs by them. The HLS was invented to fill this gap to allow developers to meet their needs without having to know the hardware details. However, HLS tools usually only support low-level languages such as C or C++, while DNN designers typically use high-level tools such as Tensorflow, Keras, PyTorch, or Caffe, among others, to describe their networks.

In this paper, to fill the gap between a high-level description of a DNN network in Keras and its low-level description in C, we provide a full toolchain that, in addition to the conversion, offers features such as conversion result verification and straightforward user-configurable quantization. This toolchain provides DNN users with rapid prototyping of their algorithms without worrying about the hardware details. In addition, the description in C is entirely flat. By flat, we mean that it does not use any functions and function calls. This is very useful for the implementation of the network by HLS and allows designers to easily add the HLS directives to the generated C design in a fine-grained manner. Moreover, a flat implementation is the most suitable type of design description for HLS tools as it allows them to apply both inter- and intra-layer optimizations. It should be noted that although the proposed process is based on the Keras input, with a some extensions, it can also be used for other high-level model descriptions such as Caffe. In summary, the main contributions and features of this work are:

- Providing a full toolchain including the Keras to C conversion and conversion accuracy verification.
- The generated C has easy-to-use knobs to switch the HLS implementation from floating point to fixed point with arbitrary quantization levels.
- The generated C code is in ANSI C, and thus, it can be used in almost all the open-source and commercial HLS tools.
- The C code is flat, allowing the HLS users to easily add directives such as *pipeline* and *unroll* in a fine-grained way. This also enables the intra-layer optimizations.
- For verification and accuracy evaluation, a file-to-memory interface is implemented that allows using the tool for large networks with many parameters and large datasets.

Preprint submitted to International Conference on Electronics Circuits and Systems (ICECS'20)

II. Related work

The methods to implement the inference computation of a DNN on the FPGA can be divided into three categories [2], [3]. In this section we discuss some of the most important and recent works in each category. The first way to run a DNN on an FPGA is to directly implement the neural network at the Register Transfer Level (RTL). These implementations are directly synthesized to the FPGA [4], [5]. Although they usually offer an implementation with acceptable performance and quality, in practice, using them is not possible for neural network designers, which are usually users with little hardware knowledge [6]. Moreover, the generated results have specific features, like quantization, accuracy, and area utilization, which are not easily alterable.

The second category includes the HW/SW co-design of a DNN. The control and scheduling of the execution of the hardware elements in the hardware part are performed by the software running on the CPU. They may use various technologies and tools to handle the hardware elements and the HW-SW communication. Some of them such as [1] and [6] used OpenCL [7]. In [8], the Pthread feature of the Legup synthesis tool [9] was used. Authors of [10] proposed their own Instruction Set Architecture (ISA), and [11] and [12] used the Xilinx SDAccel [13] and SDSoC [14] respectively. The first problem with these methods is that the generated output cannot be tweaked or optimized, i.e. the user cannot modify the design by considering the limitations or freedoms in performance, area utilization, or quantization level. The second drawback of these methods is that the users are imposed and limited to using heterogeneous environments in which both CPU and FPGA exist, as the CPU is responsible for controlling the hardware elements.

The third category, relevant to our contribution, are those that transform the DNN inference to a C/C++ or LLVM code. In this case, the end user may have the freedom to change and tweak the design to get the desired performance, accuracy, and area utilization. This is an important feature because the user may prefer to sacrifice the accuracy for lower latency or sacrifice the latency for lower area utilization. By searching through the articles and tools related to this category, we found seven works. Four of these works generated code that could not be synthesized by HLS [15]–[18]. The main target for these works was embedded CPUs. There were only three cases that could be synthesized by the HLS. The first was a tool that converts a design in Tensorflow to LLVM code with the help of Google XLA compiler [19], [20]. The LLVM code is then synthesized to FPGAs using the Legup synthesis tool [9]. Due to the use of LLVM, tweaking the generated code or setting high-level HLS knobs is impossible for the user. Besides, Legup is an open-source tool which is limited to specific Altera FPGAs and a few high-level synthesis and scheduling algorithms [21]. The second tool provided a C++ version of neural network, but was limited to multi-layer perceptron (MLP) networks and did not support convolution layers [22]. Another tool in this area was an online tool that supported DNN networks and produced synthesizable output in C language [23]. This tool only converted layers as functions, as opposed to our flat implementation, and did not provide features like verification or adjusting the quantization levels. On top of that, it could only be used for small networks such as LeNet [24], and medium or large networks like AlexNet or VGG was not supported. Table I summarizes the works in this category.

TABLE I
Summary of the most related works

Work	HLSs/FPGAs	Tweakable	CNN	Flat	Verification	Large Networks
[19], [20]	Legup/Altera	No	Yes	Yes	No	Yes
[22]	All	Yes	No	No	No	No
[23]	All	Yes	Yes	No	No	No

III. DeepHLS toolchain

DNN network designers and programmers often use a high-level language such as Keras or TensorFlow to define, train, and test their neural networks. On the other hand, implementing these networks on the FPGA requires a low-level language or library like C or OpenCL. In this work, we have considered Keras [25] as the input description, although the toolchain can be extended to also support other languages like TensorFlow and Caffe. Keras is an easy-to-learn, easy-to-use deep learning library that allows fast implementation and experimentation while allowing integration with TensorFlow functionality. DeepHLS generates ANSI C to model the output since it can be synthesized by most HLS tools and to all FPGAs, either SoC or non-SoC ones. The toolchain seeks to satisfy all the user's needs in a complete process from just after the network description in Keras is finished by the designer to the end of creating a desired C model. Figure 1 shows the overall flow of DeepHLS. The input of this process is a DNN that has been modeled, trained, and tested in Keras plus the test data that was used in the DNN design phase. The three blocks at the center of this figure are the contributions of DeepHLS.

A. Preprocessing

The first tool of the toolchain is the initial processing of the network and its test data. This tool is fully implemented in Python, to be seamlessly and fully compliant with DNN design in Keras. The output of this step is the files containing the memory dump of the test data and network parameters, including weights and biases.

B. Synthesizable C code generation

In this step, the input Keras code is processed and the specifications of each layer are extracted [26]. The Keras code is a brief description that, for the sake of simplicity of description, does not specify many of the layer's specifications. Most of the layers' information must be inferred based on the type of a layer, the parameters, and the previous layers. Figure 2 shows a description of Keras from a Lenet-5 network [24] as an example input. Correspondingly, Table II shows all the information extracted from this description. After the structure extraction stage, the output code generation begins.

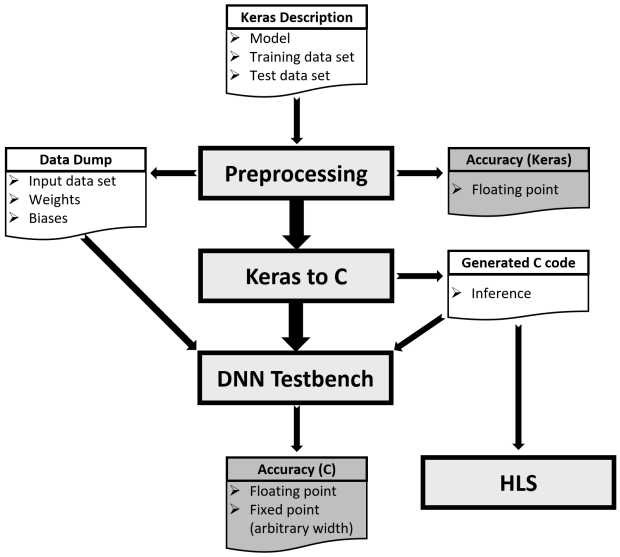

Fig. 1. The overall view of the proposed toolchain

```
model.add(Conv2D(filters=6, kernel_size=(5,5),
        activation='relu', input_shape=(28, 28, 1)))
model.add(MaxPool2D(strides=2))
model.add(Conv2D(filters=16, kernel_size=(5,5),
        activation='relu'))
model.add(MaxPool2D(strides=2))
model.add(Flatten())
model.add(Dense(120, activation='relu'))
model.add(Dense(84, activation='relu'))
model.add(Dense(10, activation='softmax'))
```

Fig. 2. Keras description of the LeNet-5

For each layer, in addition to defining the layer output array, the necessary loops to perform the layer output calculation are automatically implemented.

TABLE II
EXTRACTED INFORMATION FROM KERAS DESCRIPTION IN FIGURE 2

Layer	Layer	Padding	Filters	Kernel	Stride	Activation	Input	Output
1	Conv2D	Valid	6	(5, 5)	(1, 1)	Relu	(28, 28, 1)	(24, 24, 6)
2	MaxPool2D	-	-	(2, 2)	(2, 2)	-	(24,24,6)	(12,12,6)
3	Conv2D	Valid	16	(5, 5)	(1, 1)	Relu	(12,12,6)	(8,8,16)
4	MaxPool2D	-	-	(2, 2)	(2, 2)	-	(8,8,16)	(4,4,16)
5	Flatten	-	-	-	-	-	(4,4,16)	(256,1,1)
6	Dense	-	-	-	-	Relu	(256,1,1)	(120,1,1)
7	Dense	-	-	-	-	Relu	(120,1,1)	(84,1,1)
8	Dense	-	-	-	-	Softmax	(84,1,1)	(10,1,1)

While producing the output C file, various features are considered so that the output will be easier to handle for creating an optimized implementation on FPGA.

- At a single point of the output, a new data type is defined, and all the data required in the code, including input data, weights, and biases, are defined based on this specific type. The user can easily adjust the DNN data type before final synthesis on FPGA only at this one point, and accordingly, the type of all input data, parameters, and intermediate data are adjusted. This can be specifically very helpful for changing the network data type from floating-point to fixed-point with any arbitrary precision. Obviously, these changes may affect network accuracy. These effects can be easily observed and evaluated by the next tool of the toolchain, described in the next section.
- All layers are defined in one file and in one function. This allows HLS to perform universal and intra-layer optimizations (unlike tools that define each layer as a function with specific parameters).
- All loops are labeled. This not only makes the code and the HLS output report more readable, but also allows the end user to be able to apply HLS directives to each individual loop by addressing it in a separate directive file (such as the directive.tcl file in Xilinx HLS tool).

C. DNN Testbench

In the previous sections, the steps to create a ready-to-synthesize C file was explained. In this section, we describe the DNN testbench tool, which makes it possible for the designers to test and simulate the created file. The designer can obtain the accuracy of the produced code by executing it on the same test data that was used in the preprocessing stage. If the data type is exactly the same as what was used in the Keras design phase (e.g., the 32-bit floating-point), the accuracy must be exactly the same.

This tool is essential for two reasons. The first reason is that the designer can make sure that the code created by the previous tool is correct. On the other hand, by changing the data type from floating-point to fixed-point of arbitrary sizes, the designer can see its effect on accuracy and determine the desired fixed-point size before sending it to HLS for synthesis. Examples of these settings are described in the next section.

IV. EXAMPLES AND RESULTS

To ensure the correct operation of the entire toolchain, we tested it on three well-known DNNs including LeNet-5 [24], AlexNet [27], and VGG [28], as small, medium, and large scale DNNs respectively. We applied all the tools in the chain and finally synthesized the generated C code using the Xilinx Vivado HLS tool on Xilinx Kintex-7 (xc7k480t-ffv901-1), which is a medium-size non-SoC FPGA. The results showed that the generated C code obtained exactly the same accuracy as the original Keras design when the same data type (32-bit floating point) was chosen. Regarding the latency, in the absence of any HLS optimizations and without changing the data types to fixed-point, our toolchain generated a result with a latency of 905,772 clock cycles, in contrast to 1,275,700 clock cycles in [19], for the first layer of the LeNet-5 network. Table III shows the area utilization of the floating-point implementation of the experimented networks.

TABLE III
SYNTHESIS RESULTS OF THREE DNNS OF VARIOUS SIZES

CNN	BRAM	FF	LUT
VGG	616	7991	21636
AlexNet	326	4200	11257
LeNet-5	16	2348	5763

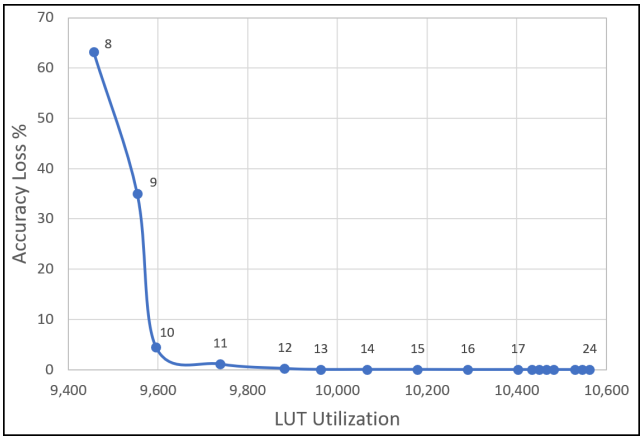

Fig. 3. Accuracy Loss and LUT Utilization as a function of the number of fractional bits

Additionally, as an application of the DNN testbench tool, we tested the AlexNet network for fixed-point data types with various sizes. We changed the fixed-point fraction size from 24 bits to 8 bits to assess its influence on the accuracy. We also synthesized all versions to be able to compare the area utilizations. Figure 3 shows the results. The numbers beside each point imply the number of bits considered for the fraction in a fixed-point representation. As can be seen, by choosing 12 bits for the fraction, the accuracy loss is almost zero. Choosing more bits for the fraction has no effect on the accuracy and only increases the area utilization.

V. Conclusion and Future Work

In this paper, we proposed an integrated toolchain that in addition to converting the Keras DNN descriptions to a simple, flat, and synthesizable C output, provides features like accuracy verification, data type knobs to support floating-point and fixed-point types, and enabling the user to adjust the latency and area utilization using HLS knobs. We tested all the tools on DNNs of three various sizes. The results showed the feasibility of the flow and its scalability as well as various outputs it can generate to facilitate the decision making during the acceleration of the DNNs. It should be noted that the toolchain currently does not support Recurrent Neural Networks (RNN), and in the future, we are going to extend our work to support them too.

References

[1] A. Ghaffari and Y. Savaria, "Cnn2gate: Toward designing a general framework for implementation of convolutional neural networks on fpga," *arXiv preprint arXiv:2004.04641*, 2020.

[2] A. Shawahna, S. M. Sait, and A. El-Maleh, "Fpga-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2018.

[3] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions," *arXiv preprint arXiv:1803.05900*, 2018.

[4] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song *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.

[5] S. I. Venieris and C.-S. Bouganis, "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.

[6] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "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.

[7] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in science & engineering*, vol. 12, no. 3, pp. 66–73, 2010.

[8] J. H. Kim, B. Grady, R. Lian, J. Brothers, and J. H. Anderson, "Fpga-based cnn inference accelerator synthesized from multi-threaded c software," in *2017 30th IEEE International System-on-Chip Conference (SOCC)*. IEEE, 2017, pp. 268–273.

[9] A. Canis, J. Choi, B. Fort, B. Syrowik, R. L. Lian, Y. T. Chen, H. Hsiao, J. Goeders, S. Brown, and J. Anderson, "Legup high-level synthesis," in *FPGAs for Software Programmers*. Springer, 2016, pp. 175–190.

[10] Y. Xing, S. Liang, L. Sui, X. Jia, J. Qiu, X. Liu, Y. Wang, Y. Shan, and Y. Wang, "Dnnvm: End-to-end compiler leveraging heterogeneous optimizations on fpga-based cnn accelerators," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[11] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, "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.

[12] P. G. Mousouliotis and L. P. Petrou, "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.

[13] L. Wirbel, "Xilinx sdaccel: a unified development environment for tomorrow's data center," *The Linley Group Inc*, 2014.

[14] V. Kathail, J. Hwang, W. Sun, Y. Chobe, T. Shui, and J. Carrillo, "Sdsoc: A higher-level programming environment for zynq soc and ultrascale+ mpsoc," in *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. New York, NY, USA: Association for Computing Machinery, 2016, p. 4. [Online]. Available: https://doi.org/10.1145/2847263.2847284

[15] Aljabr0: from-keras-to-c. [Accessed Sep. 20, 2020]. [Online]. Available: https://github.com/aljabr0/from-keras-to-c

[16] Dobiasd: frugally-deep. [Accessed Sep. 20, 2020]. [Online]. Available: https://github.com/Dobiasd/frugally-deep

[17] gosha20777: keras2cpp. [Accessed Sep. 20, 2020]. [Online]. Available: https://github.com/gosha20777/keras2cpp

[18] pplonski: keras2cpp. [Accessed Sep. 20, 2020]. [Online]. Available: https://github.com/pplonski/keras2cpp

[19] D. H. Noronha, K. Gibson, B. Salehpour, and S. J. Wilton, "Leflow: Automatic compilation of tensorflow machine learning applications to fpgas," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 393–396.

[20] D. H. Noronha, B. Salehpour, and S. J. Wilton, "Leflow: Enabling flexible fpga high-level synthesis of tensorflow deep neural networks," in *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*. VDE, 2018, pp. 1–8.

[21] Legup 4.0 documentation. [Accessed Sep. 20, 2020]. [Online]. Available: http://legup.eecg.utoronto.ca/docs/4.0/faq.html

[22] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba, M. Pierini, R. Rivera, N. Tran *et al.*, "Fast inference of deep neural networks in fpgas for particle physics," *Journal of Instrumentation*, vol. 13, no. 07, p. P07027, 2018.

[23] Ai transformer - keras to c code converter. [Accessed Sep. 20, 2020]. [Online]. Available: http://www.aitransformer.com

[24] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[25] A. Gulli and S. Pal, *Deep learning with Keras*. Packt Publishing Ltd, 2017.

[26] V. Dumoulin and F. Visin, "A guide to convolution arithmetic for deep learning," *arXiv preprint arXiv:1603.07285*, 2016.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[28] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.