GENERIC DESIGN FLOW OF PIPELINED HARDWARE IMPLEMENTATION OF DEEP NEURAL NETWORKS

El Hadrami Cheikh Tourad

École Mohammedia d'Ingénieurs, Mohammed V University in Rabat,
Rabat, Morocco
elhadrami_cheikhtourad@research.emi.ac.ma

Mohsine Eleuldj

École Mohammedia d'Ingénieurs, Mohammed V University in Rabat,
Rabat, Morocco
eleuldj@emi.ac.ma

Abstract

DNNs (Deep Neural Networks) have solved various deep learning tasks, including classification problems, natural language processing, and speech recognition. However, this success comes with increased computational and memory requirements. Furthermore, recent deep learning research indicates that hardware implementations such as FPGAs (Field Programmable Gate Arrays) are preferable for implementing DNNs, and they fulfill the requirements due to the integrated circuits with programmable logic gates and connections. This technique offers hardware implementation flexibility, which makes it appealing for a wide range of applications, and that flexibility differs from the standard circuit. As a result, FPGAs are becoming increasingly popular as a hardware solution for accelerating systems and processes. This article presents a generic version of the design flow for automatically implementing DNN models on hardware by generating pipelined HDL codes, which can overcome the implementation problem. The article compares the design flow to other recent similar tools. The design flow is validated using a DNN to detect the diabetic patient from the Pima Indians dataset classification. This paper shows a high performance by reducing the latency by 4x the non-pipelined VHDL code and 1000x the software implementation using the TensorFlow framework without affecting the model's accuracy. Also, this design flow can serve in the early prediction of diabetes in the future—finally, a presentation of the conclusion and future works.

***Keywords*: HDL; FPGA; DNN; Pipeline; Design Flow.**

1. Introduction

DNNs (Deep Neural Networks) have solved various deep learning challenges, including natural language processing, classification problems, and speech recognition. However, this progress increases computational and memory requirements due to their massive computations and intricate structures (nodes and layers), as shown in figure 1.

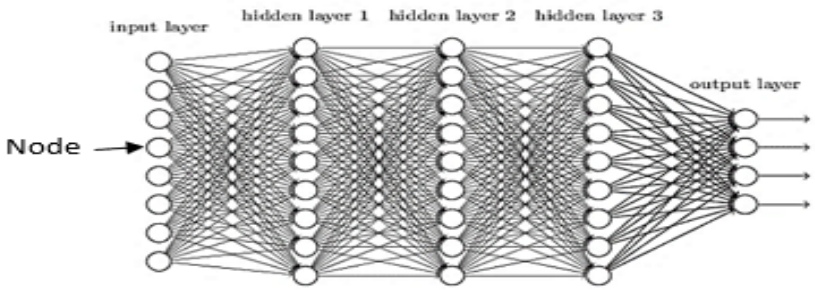

Fig 1. DNN[1].

As a result, implementing DNNs in data centers and real-time integrated devices is challenging. Furthermore, recent deep learning research indicates that hardware implementations such as FPGAs (Field Programmable Gate Arrays) are preferable for implementing DNNs, and they fulfill the requirements due to their power efficiency, flexibility, and computing performance. However, software engineers cannot obtain these characteristics when implementing DNN models onto general computing systems like GPU (Graphics Processing Unit) [1].

FPGAs (Field Programmable Gate Arrays) are integrated circuits with programmable logic gates and connections. This technique offers hardware implementation flexibility, which makes it appealing for a wide range of signal processing applications, and that flexibility differs from the standard circuit. As a result, FPGAs are becoming increasingly popular as a hardware solution for accelerating systems and processes.

A hardware device such as FPGAs contains a wide range of resources, including Flip-Flops, Block RAMs (BRAMs), connected Lookup tables (LUTs), and Digital Signal Processing (DSP) blocks [2]. These characteristics indicate that FPGA devices can be reconfigured and customized. Moreover, implementing a DNN accelerator using FPGA remains difficult. As a result, several challenges remain between designers and FPGAs, including VHDL and Deep Learning expertise. Therefore, to overcome this difficulty, the researchers attempt the DNNs model's automatic implementation on the FPGA devices from a high-level language like Python to the HDL code.

This article presents a new generic version of the design flow for automatically implementing DNN models on hardware by generating pipelined HDL codes. The design flow takes the model description in a graphic presentation as input and generates the HDL implementation. This presentation liberates the design flow from the model framework. Consequently, the generated HDL codes are generic, pipelined, and modular.

This article includes the following sections: Section II summarizes the state-of-the-art on DNN hardware implementations and reviews many related works. Next, section III describes the pipeline approach used in this work. Then, section IV explains the proposed design ow. Next, section V presents the case study results, and section VI compares the design flow and other frameworks. Finally, the last section contains the conclusion.

2. Related Works

Various prior works, including [3], [4], [5], and [6], have implemented DNNs on FPGA to accelerate the inference of deep learning models. In addition, Tourad et al. [1] explored numerous frameworks for automatically implementing DNN models on FPGAs. The rest of the section will focus on the essential frameworks from this research.

First, Guan et al. use a model mapper in FPDNN [7] to retrieve topology and parameter details from a TensorFlow [8] model. Then, on the network, deploy the C++/OpenCL RTL-HLS hybrid template. LeFlow [9] is a tool-flow that automatically transfers TensorFlow operations into FPGAs. The framework employs Google's XLA compiler to directly transfer Low-level Virtual Machine (LLVM) code from a Tensorflow specification. The code is then run via a synthesis tool like LegUp.

Moreover, snowflake [10] is among the most efficient CNN accelerators, with Torch-specific models and a single computational architecture optimized for Xilinx System on Chips (SoCs). hls4ml [3] [11] is a high-level FPGA neural network inference tool capable of producing high latency. This program is for particle physics research in which reaction time is critical. First, the model is constructed in Keras or PyTorch, then transferred to Vivado HLS, and finally, the FPGA RTL structure is generated in hls4ml using the synthesis and deployment toolkit.

DL2HDL [2] is a framework for mapping the DNNs on FPGA using Python and MyHDL (a low-level python language used for the FPGA). First, the models must be PyTorch or Keras models transformed into PyTorch models. Then a generation of the MyHDL model to proceed to the simulation and synthesis.

FpgaConvNet [12] [13] converts CNN models from Torch or Caffe to Xilinx Vivado HLS code. Other initiatives, such as ALAMO [14] [15], FFTCodeGen [16], and DnnWeaver [17], are attempting to implement DNNs on FPGAs. However, the frameworks in the study [1], including the tools listed earlier, are either not generic (particular tool or equipment or restricted templates) or include a large number of procedures and tool flows, such as hls4ml [11] and LeFlow [9].

Recently, Tourad et al. [6] have proposed the "DNN2FPGA." A design flow exploits a direct hardware mapping method. This method connects hardware blocks to the DNN model description elements, where the description is a graphic presentation. This approach allows the design flow to produce the HDL code according to the correct topology without dependence on the model framework.

This paper presents a new version of "DNN2FPGA." with an optimized deep learning engine and VHDL generator with pipelined approach implemented in the generated hardware code. Moreover, in this version, the case study model can serve in the early prediction of diabetes in the future, and the design flow uses a vast dataset for the test and layers with more nodes and several parameters to solve the diabetic patient classification problem.

3. Pipeline Approach

The concept is to break down a complicated process into smaller tasks that may be run in parallel, boosting throughput and lowering latency. The pipeline concept's primary premise is that each processing stage can run independently of the others. This method allows numerous stages to run simultaneously without waiting for each other's results. It also allows each step to process its input data without being dependent on the input data of previous stages. A Pipeline is an event occurring during overlapping time intervals, as shown in figure 2 and figure 3. The D1, D2, and D3 present the data, and T1, T2, and T3 present the tasks.

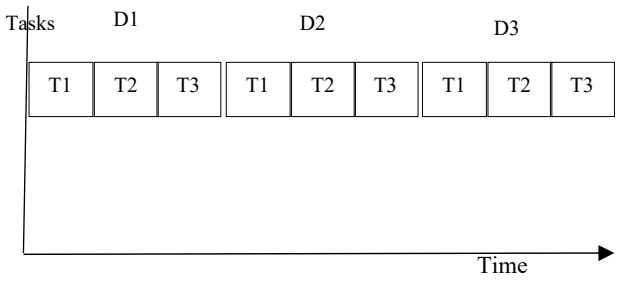

Fig. 2. Sequential Tasks.

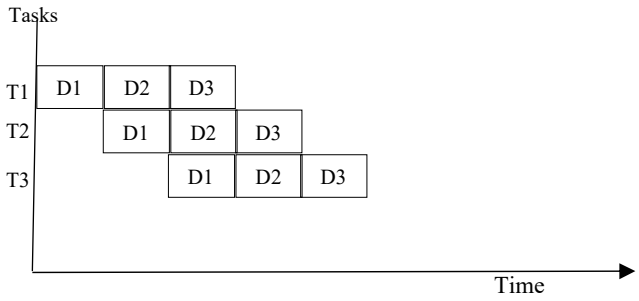

Fig. 3. Pipelined Tasks.

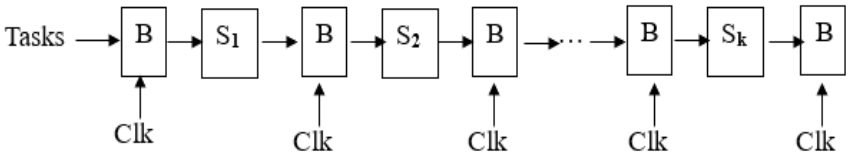

Fig. **4**. Detailed pipelined Tasks

From figure 4 we have these assumptions:
Si: stage i ($1 \leq i \leq k$), B: buffer and clk: clock

Time (Si) = τ_i

Time(B) = τ_0 and $\tau = \max \{1 \leq i \leq k, \tau_i\} + \tau_0$

n: number of tasks

T_1: time without pipeline and T_k: time with a pipeline of k stages

$T_k = \tau_0 + k\tau + (n - 1)\tau = (k + n - 1)\tau + \tau_0$

Suppose $T_1 = nk\,\tau$

Speed up = $T_1/T_k = nk\,\tau_i / [(k + n - 1)\tau + \tau_0] \approx nk/(k+n - 1)$

Using an integrated circuit with many flip-flops, such as FPGA, brings more attention to using a pipeline architecture to enhance the design's operating frequency. The pipeline architecture aims to place registers between each layer of combinatorial functions to decrease the critical time and, consequently, the time clock to set up.

The layers are dependent. As a result, it is impossible to execute all the network layers in a parallel scheme. To prevent the internal registers from being overwritten during operations, each layer must finish processing its current input (produce an output) before taking another set of inputs. [18]. As shown in figure 5, these master-slave D-flip-flops hold the data until the clock signal before transmitting the data to the next layer and receiving another data flow. This approach enhances the network throughput and allows the process to operate pipelined.

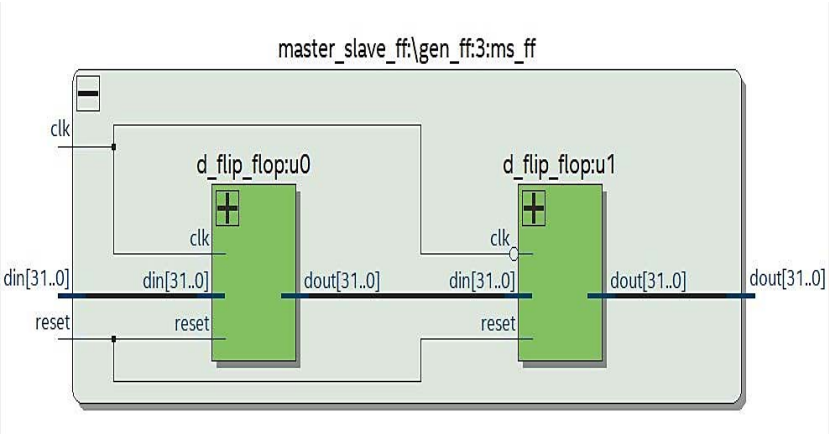

Fig 5. Master-slave D flip-flop.

As a result, the generated network's pipeline approach uses master-slave D-flip-flops between layers, as shown in figure 6. Figure 6 shows a part of the pipelined network where a set of master-slave D flip-flops follows the layer and takes the same number of inputs and outputs.

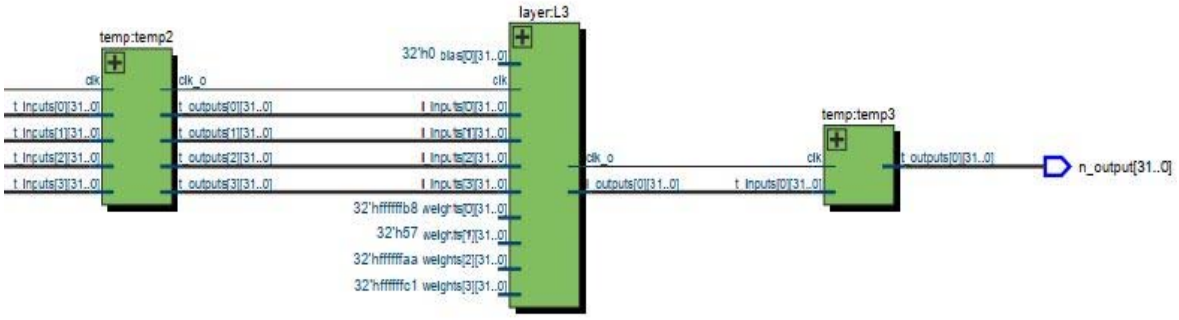

Fig 6. Pipeline Design.

4. GENERIC DESIGN FLOW

4.1. *Genericity*

The design flow maintains the genericity in several aspects:

- High-level language frameworks.
- The model description graphic formats.

- HDL version.
- Synthesis and simulation tool.
- The Hardware device.

This genericity presents in the independence of the high-level python frameworks such as TensorFlow or other frameworks due to the model description saving in an intermediate graphic format such as JSON or DOT. Moreover, the HDL generation independence of HDL versions like VHDL or Verilog and the synthesis and simulation tool such as Quartus. Finally, the genericity surrounds the hardware implementation independently from the type of the device, such as FPGA or ASIC. However, the work focuses on the FPGA.

4.2. *Design Flow*

As shown in figure 7, the design flow begins with the DLE (Deep Learning Engine), where the user can design the DNN using a high-level framework like Tensorflow and Keras. Using Keras, the DLE outputs the model description in a graphic format. The parser then extracts the layer computations, type (pooling, convolutional, or fully connected), and activation functions. After that, the model parser extracts the weights and biases. Finally, the data is saved as configuration files, then transmitted to the HDL generator.

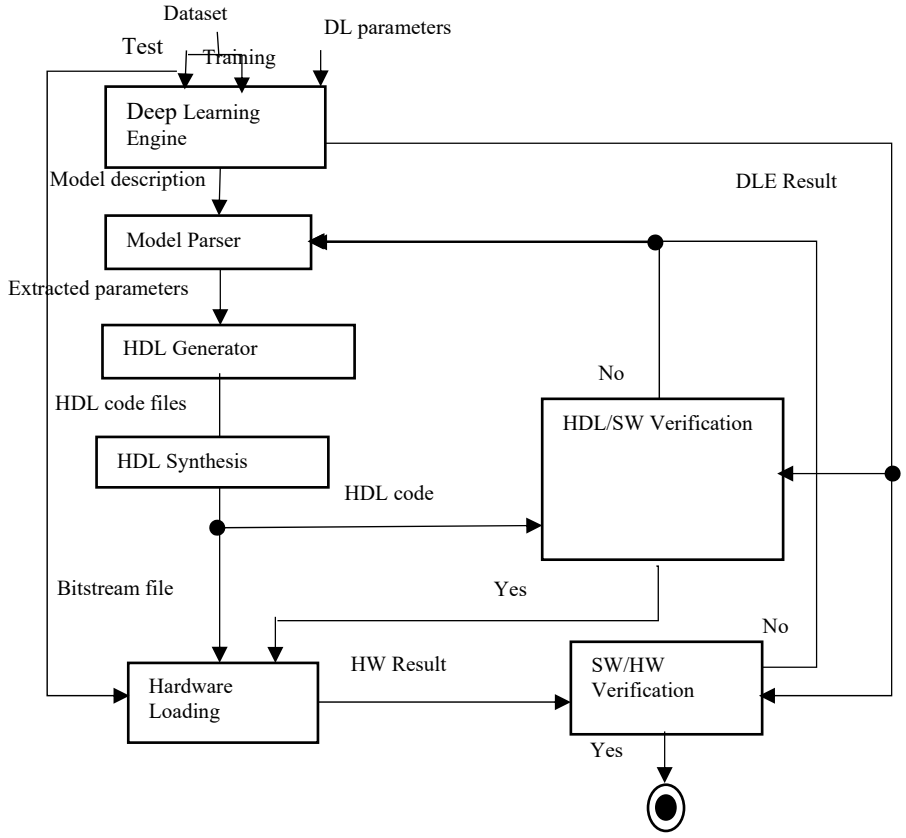

Fig 7. The design flow.

The HDL generator is in charge of several critical tasks: The scheduling step is responsible for the operation's synchronization and configuring the clock cycle for the master-slave d-flip-flops. In addition, this step manages the pipeline between layers. It divides the computations, memory, and interface access into clock cycles and as-signs them to the hardware units.

The HDL file components include entities, architectures, and processes, as shown in the HDL generation algorithm. Algorithm 1 describes all the necessary steps, from reading configuration files to creating hardware components (entities and architectures) to obtaining the final HDL code files.

After that, a synthesis tool analyzes and compiles the resulting HDL code. First, this step verifies the HDL syntax and the process statements. Then, the synthesis takes the high-level description down to low-level descriptions to produce the hardware mapping file, such as the bitstream file. Finally, the synthesis ensures that the HDL code corresponds to the targeted hardware device.

A simulation is then required to ensure the generated HDL results are exact before deploying the HDL on the device and confirm that the results are similar to the software implementation. If the achieved results are less than the DLE implementation, the entire process depicted in Figure 7 will replay from the model description creation. The operation will proceed to the FPGA device's synthesis and bitstream files if the opposite. The verification metrics are the model accuracy and the latency; the latency represents the time needed to infer the test data for each implementation (HDL and software).

Then, the loading of the bitstream file generated by the synthesis tool on the targeted device, specifically in the non-volatile memory. This file includes the logic, routing, and initial values of registers. It also contains all the hardware blocks and the instructions to manage the device resources. After downloading this file, the hardware is ready to exploit and test the network.

The hardware and software implementations comparison are required to verify the generated model's accuracy. Then, once the model implementation on the hardware ends, the inference evaluation is based on the test data to verify if the produced results match the DLE results, and if not, the entire process starts over from the DNN model description. The model accuracy and latency are the verification metrics; the latency represents the time needed to infer the test data for each implementation (hardware and software).

Algorithm1: HDL CODE GENERATION

Require: Extracted configurations, HDL Template

Ensure: HDL files

Create the generic entity for the network

While C in Configurations **do**

Read the Layers from the configurations

Read the Nodes from the configurations

Declare the main architecture

While L in Layers **do**

Instance the entity node

For N in Nodes **do**

Declare the parameters

Read the weights and bias of N

Chose corresponding activation function

Compute output of N

IF L != output layer **then**

While n in corresponding Nodes(L+1) **do**

Connect the output of N to the in-put of n

Create the master-slave d flip-flops with the same layer input, and the output is the next layer input

End while

5. Results

5.1. *Case Study*

Many recent works, including [19] [20], have implemented DNNs to classify diabetic patients by transforming the problem into a classification problem, the deep learning model's success area. In this case study, we use the Pima Indians diabetes dataset. It includes 768 patient health records data for Pima Indians. We use 30% of the dataset for the test and 70% for the training and prediction. The patient's dataset includes some relevant information, as mentioned in Table 1.

The DNN model used for the classification is an input layer followed by four fully connected layers, using the ReLU (Rectified Linear Unit) as an activation function in the three first layers and an output layer with a

Sigmoid activation function to ensure the output is between 0 and 1. The first two layers have eight nodes, four nodes in the third layer and one node for the output layer.

Figure 8 presents the model topology. The first layer is the input layer, where all the nodes are green, and the last layer is the output layer which contains one red node—the remaining hidden layers between the input and output layers, where the nodes are blue.

Column	Input data	Type of data
1	Number of times pregnant	Numeric
2	Plasma glucose concentration	Numeric
3	Diastolic blood pressure	Numeric
4	Triceps skinfold thickness	Numeric
5	2-h serum insulin	Numeric
6	Body mass index	Numeric
7	Diabetes pedigree function	Numeric
8	Age	Numeric
9	Diabetes diagnosis	(0: healthy, 1: diabetic)

Table 1. Features of Pima Indian's diabetes.

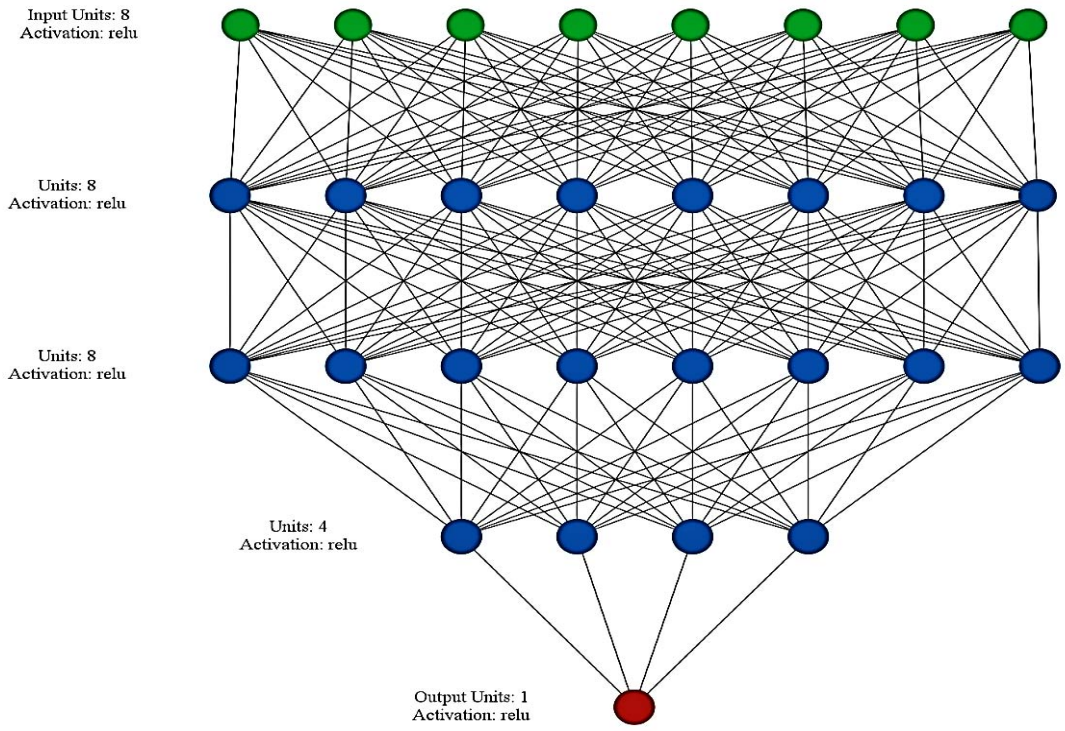

Figure 8. Model topology.

Table 2 regroups All the network hyperparameters. The network loss function is cross-entropy, generally used for binary classific and the Adam algorithm as an optimizer. Adam is a sophisticated version of gradient descent used in diverse problems.

Parameters	Configuration
Loss Function	Cross entropy
Optimizer	Adam
No of epoch	150
Batch Size	10

Table 2.Network hyperparameters.

Table 3 compares both hardware implementations and reports the resource consumption from the synthesis results using ALTERA Cyclone V. The results demonstrate that the pipeline implementation requires more resources than the non-pipelined version but substitutes by 4x latency and butter throughput.

	DNN2FPGA	Non pipelined version
DSP blocks	156	156
ALMs	1532	1878
Total registers	1082	32
Latency	30.5 ns	120 ns

Table 3. Comparison between DNN2FPGA and Non-pipelined version.

The speedup of the pipeline implementation is calculated using the simulation results and the formula in the pipeline approach section. Speed up = nk/k+n-1 (k=4, n=300) Speed up = 1200/303=3.96.
Table 4 compares the software implementation results and both hardware implementations. These results show the efficiency of the design flow without affecting the model's accuracy.

	DLE	DNN2FPGA	Non-pipelinedversion
Precision	Float32	Float32	Float32
Accuracy	90 %	90 %	90 %
Latency	128 ms	30.5 ns	120 ns

Table 4. Comparison between implementations.

5.2. *Results Comparison*

The hardware implementation results are compared to their equivalent produced by the DL2HDL and hls4ml tools. The paper explores these tools in the previous related works section. DNN2FPGA, on the other hand, offers direct HDL-to-Python mapping. However, this version is tested using the model mentioned in [1]. The tested design was a four-layer fully connected network, with ReLU activating the first three layers, whereas softmax activating the output layer [1]. After the HDL generation, the ALTERA Cyclone V 5CGXFC3B6F23C6 simulation uses Intel Quartus software. Table 5 compares the hls4ml, DL2HDL, and DNN2FPGA.

	hls4ml	DNN2FPGA	DL2HDL
#DSP48E	954	156	1069
#LUT + #FF	88797	2737	38146
Latency	75	30.5	70

Table 5. Comparison with hls4ml, DL2HDL frameworks.

Table 6 compares the design flow to the LeFlow [2] toolkit. The model used in the study is called dense a. It consists of a single dense layer with a single input, eight outputs, and bias and ReLU activation [2] [9]. The authors of [2] [9] employed an Altera Stratix IVEP4SGX290NF45C3, whereas the paper uses an ALTERA Cyclone V 5CGXFC3B6F23C6. DNN2FPGA, on the other hand, takes significantly less time inferring through the layers due to the pipeline approach used for the operations and correct parameters hardcoded in the logic elements. In addition, the model in this experience uses all the datasets to have a significant amount of testing

data. Moreover, Lc and LE in table 6 are similar. The LC indicates the logic cells, and LE indicates logic elements.

	DNN2FPGA			LeFlow		
The model	LC	DSP	Latency	#LE	MemB	Latency
Dense_a	2516	156	80	1743	1056	1421

Table 6. Comparison between LeFlow and DNN2FPGA framework.

6. CONCLUSION

The article presents a new generic version of the design flow for automatically implementing DNN models on hardware by generating pipelined HDL codes, which can overcome the implementation problem. Moreover, this design flow maintains its genericity in several aspects, such as Hig-level language frameworks, the model description graphic formats, HDL version, synthesis tool, and the Hardware device.

The article summarizes several relevant studies of a similar nature and illustrates the design flow and hardware implementation results. It also compares the design flow to other recent equivalent tools. Moreover, it proposes a case study of a DNN to detect diabetic patients from the Pima Indians dataset classification.

The design flow reduces the latency by more than 4x the non-pipelined version and 1000x the software inference time using the TensorFlow framework without affecting the model's accuracy. Also, the case study model can serve in the early prediction of diabetes in the future. The paper uses a vast dataset for the test and layers with more nodes and several parameters to solve the diabetic patient classification problem. More studies will be conducted to improve the implementation, implement more sophisticated activation functions, and handle more significant network concerns.

Conflicts of interest

The authors have no conflicts of interest to declare.

References

[1] El-Hadrami Cheikh Tourad, and Mohsine Eleuldj. Survey of Deep Learning Neural Networks Implementation on FPGAs. In 2020 International Conference on Cloud Computing and Artificial Intelligence: Technologies and Applications (CloudTech'20). 2020.
[2] Maciej Wielgosz and Michał Karwatowski. Mapping Neural Networks to FPGA-Based IoT Devices for Ultra-Low Latency Processing. In MDPI Sensors. 2019.
[3] Ahmad Shawahna, Sadiq M.Sait and Aiman El-Malehi. FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review In IEEE. 2018.
[4] Stylianos I Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. Tool flows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions. ACM Computing Surveys (CSUR) 51, 3 (2018), 56. 2018.
[5] Kaiyuan Guo, Shulin ZENG, Jincheng Yu, Yu Wang and Huazhong Yang. A survey of FPGA-based neural network inference accelerator. In ArXiv (integration December 2018).
[6] El-Hadrami Cheikh Tourad, and Mohsine Eleuldj. Generic Automated Implementation of Deep Neural Networks on Field Programmable Gate Arrays. In 2021 The Sixth International Conference on Smart City Applications.The Springer Lecture Notes in Networks and Systems LNNS Book Series. 27-28 October 2021
[7] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason 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). 2017.
[8] M. Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems. 2015.
[9] Daniel H. Noronha, Bahar Salehpour, and Steven J.E. Wilton. 2018. LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks. In arXiv 2018.
[10] Vinayak Gokhale, Aliasger Zaidy, Andre Xian Ming Chang, and Eugenio Culurciello. Snowflake: An Efficient Hardware Accelerator for Convolutional Neural Networks. IEEE ISCAS. 2017.
[11] J. Duarte, S. Han, P. Harris, S. Jindariani, E. Kreinar, B. Kreis, J. Ngadiuba,M. Pierini, R. Rivera, N. Trana and Z. Wue. Fast inference of deep neural networks in FPGAs for particle physics. In J. Instrum. 2018.
[12] Stylianos I. Venieris and Christos-Savvas Bouganis. fpgaConvNet: Automated Mapping of ConvolutionalNeural Networks on FPGAs (Abstract Only). InProceedings of the 2017 ACM/SIGDA International Symposium onField-Programmable Gate Arrays (FPGA' 17). ACM, 291–292. 2017.
[13] Stylianos I. Venieris and Christos-Savvas Bouganis. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs. In 2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). 2016.
[14] Yufei Ma, Naveen Suda, Yu Cao, Jae sun Seo, and Sarma Vrudhula. Scalable and Modularized RTL Compilation of Convolutional Neural Networks onto FPGA. In 2016 26th International Conference on Field Programmable Logic and Applications (FPL). 2016.
[15] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In Field Programmable Logic and Applications (FPL) IEEE 1-8. 2017
[16] Hanqing Zeng, Chi Zhang, and Viktor Prasanna. Fast Generation of High Throughput Customized Deep Learning Accelerators on FPGAs. In 2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig). 2017.

[17] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. From high-level deep neural models to FPGAs. In Micro Architecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium. IEEE, 1–12. 2016.
[18] Zhou, H., Myrzashova, R. & Zheng, R. Diabetes prediction model based on an enhanced deep neural network. J Wireless Com Network 2020, 148 (2020).
[19] Tawfik Beghriche, Mohamed Djerioui, Youcef Brik, Bilal Attallah, Samir Brahim Belhaouari. An Efficient Prediction System for Diabetes Disease Based on Deep Neural Network. Complexity, vol. 2021, Article ID 6053824, 14 pages, 2021.
[20] Adrián Alcolea and Javier Resano. FPGA Accelerator for Gradient Boosting Decision Trees. In MDPI Electronics. 2021
[21] Zhiqiang Liu, Yong Dou, Jingfei Jiang, and Jinwei Xu. Automatic Code Generation of Convolutional Neural Networks in FPGA Implementation. In 2016 International Conference on Field-Programmable Technology (FPT). 2016.
[22] Mannhee Cho and Youngmin Kim. FPGA-Based Convolutional Neural Network Accelerator with Resource-Optimized Approximate Multiply-Accumulate Unit. In MDPI Electronics. 2021.

Authors Profile

El Hadrami Cheikh Tourad, El Hadrami Cheikh Tourad is a Ph. D candidate at École Mohammedia d'Ingénieurs, Mohammed V University in Rabat, Morocco with engineer's degree in Computer Science from École Supérieure Polytechnique de Nouakchott ,Mauritania in 2016. His researches are in the fields of deep learning, neural networks optimization methods, and neural networks hardware accelerators. Further details in his ORCID page: https://orcid.org/0000-0001-6370-8039 .

Eleuldj Mohsine, Eleuldj Mohsine received his Ph.D. in 1989 in université de Montréal Canada and then joined as professor the computer science department in École Mohammedia d'Ingénieurs, Mohammed V University in Rabat, Morocco. His research interests include parallel processing and hardware implementations.