A Design Flow Framework for Fully-Connected Neural Networks Rapid Prototyping

Nikolaos Zompakis
n.zompakis@microlab.ntua.gr
MicroLab-ECE-NTUA
Athens, Greece

Dimitrios Anagnostos
anagnostos.d@microlab.ntua.gr
MicroLab-ECE-NTUA, Greece
Katholieke Univ. Leuven, Belgium

Konstantina Koliogeorgi
konstantina@microlab.ntua.gr
MicroLab-ECE-NTUA
Athens, Greece

Georgios Zervakis
zervakis@microlab.ntua.gr
MicroLab-ECE-NTUA
Athens, Greece

Kostas Siozios
ksiop@auth.gr
Aristotle University
Thessaloniki, Greece

ABSTRACT

The current work deploys a framework for rapid prototyping of Fully-Connected Neural Networks (FCNs). The scope is to provide an automatic design flow that generates a template-based VHDL code considering the accuracy, the resource utilization and the design complexity. More precisely, the deployed tool incorporates hardware optimizations in the implementation of the multiplications, the activation function and the definition of the fixed-point types providing user-defined configurations thought a GUI. The FCNs of two applications (Alexnet and Lenet) were implemented to evaluate our approach. The results seem promising and prove the design flexibility of our framework generating optimized code that exceeds the 10K lines for each hardware instance within a few hours, while preserving low levels of latency that does not exceed 400 cycles for our applications.

KEYWORDS

fpga, neural networks, fully-connected, accuracy, rapid prototyping, DNN, vhdl, bit precision,

ACM Reference Format:
Nikolaos Zompakis, Dimitrios Anagnostos, Konstantina Koliogeorgi, Georgios Zervakis, and Kostas Siozios. 2019. A Design Flow Framework for Fully-Connected Neural Networks Rapid Prototyping. In *COINS '19: COINS'19: International Conference on Omni-layer Intelligent systems, MAY 05–07, 2019, Crete, GR.* ACM, New York, NY, USA, 6 pages.

1 INTRODUCTION

Deep Neural Networks (DNNs) have penetrated in everyday life from web search engines to computer vision applications. The key feature is that they deal with incorrect and incomplete data without programming. DNNs capture the desirable property, exploiting representative training sets. One of the main challenge is the huge inputs. Even a low resolution image processing DNN algorithm of just one-megapixel has an input dimension of three billions parameters, considering the three RGB channels. The fully connected neural networks (FCNs), imitating the human brain, require high computational and memory resources for training. Respectively, the convolutional neural networks (CoNNs) overcome this issue applying a parameter sharing approach [5] splitting the problem in sequential filtering stages/layers that decrease the dimension complexity. However, CoNNs's face optimization difficulties to parallelize in hardware designs.

More precisely, CoNNs consist of three main layer types 1) the convolutional layer (CoL), 2) the pooling layer (PL) and 3) the fully connected layer (FCL). While CoLs are typically execution time dominant, in real conditions due to the memory data accesses lead on substantial execution stalls, influencing the performance bottlenecks. Such stalls in case of FCLs can reach up to 30% of the total stall time[18] at image processing applications like AlexNet and VGG16, representing just 5-10% of the total execution time. These stalls occurs due to on-chip (L1, L2 cache) and off-chip memory misses. Thus, while GPUs have a clear benefit in accelerating CoLs, they face critical latency issues due to memory misses. FPGAs overcome these issues due to the different architecture and due to the memory locality close to the processing elements. Another issue that enforce FPGA as a choice is the prohibitive GPU power consumption (>100W) for embedded applications. FPGAs also implement more efficiently non-linear complex operation due to the existence of extended number of DSPs. Moreover, FPGAs are ideal platforms for rapid prototyping ASIC-oriented designs.

2 RELATED WORK

The above considerations motivate the DNNs implementation in FPGAs. The majority of the literature are high level synthesis (HLS) approaches [3] that focus on the CoLs optimization [5] due to the uniform functionality of the matrix multiplications that simplifies the hardware implementation process. A few works follow a more customized approaches [10],[17] that concentrate on an optimized application mapping. More sophisticated template-based approaches [12] provide more fine grain optimization but literature lacks of FCLs customized solutions with automatic design flows. Most of the existing works incorporate the FCL [4] as a special CoL or exploit system generator library blocks [14],[11] without considering their special characteristics, partially scarifying the design efficiency. Typical implementations that highlight this impact are accessible in[9]. More precisely, the existence or not of FCLs despite their theoretical limited execution time can decrease the achieved performance in frames-per-second (fps) by a factor of 12 at AlexNet and up to 50% at VGG-16 [9]. The FCL influences the CoNNs accuracy so in many cases it has to be included.

COINS'19, May 5–7, 2019, Crete, Gr

The scope of the current work is to deploy a rapid-prototyping framework for FCLs on FPGAs, considering latency and resource utilization in trade-off with achieved accuracy. The rest of the paper is organized as follows: Chapter II outlines the contribution, Chapter III describes the methodology steps, Chapter IV presents the deployed Framework tool and the incorporated optimization, Chapter V presents the experimental results and the Chapter VI includes the final conclusions.

3 CONTRIBUTION OVERVIEW

Traditional programming languages such as C/C++ as hardware description languages lack the capability to express hardware timing explicitly. Moreover, HLS synthesizers [3] infer generic hardware descriptions that influence negatively the final performance. A hardware description language, like VHDL,[12] provides a closer supervision of the hardware design at the final netlist. In this direction, we have developed a hand-optimized template-based framework that allows the automatic generation of optimized VHDL code, directly from software training tools. For our training needs, we exploit the Matlab neural toolbox [16]. After training, all the required VHDL code libraries and simulation files, are automatically generated, ready for implementation. The required time depends on the complexity of the targeted neural architecture (number of nodes), the utilized workstation and the required accuracy in bit precision. Outlining the main breakthroughs of our framework: 1) it optimizes the hardware implementation of the multiplications improving performance, 2) it adjusts the data types definition saving in resources, 3) it optimizes the activation function hardware implementation improving accuracy and 4) it generate automatically the required HDL source files.

The FCNs deployment is a complex task considering several parameters (the number of layers, the number of nodes, the activation function etc) that effect the final performance. No reliable ways exist for a optimal FCN configuration for a given dataset. Software modeling tools provide flexibility in design exploration and verification. The challenge for the current work is to create a software/hardware co-design for FCN rapid-prototyping in FPGAs, deploying a framework that instantiates the high level design exploration in optimized VHDL. The goal is to provide an efficient approach, in terms of 1) FCN parameterization flexibility (number of neurons, activation function, training method etc), 2) performance, 3) resource utilization and 4) accuracy. Compared with the most relevant existing works Table 3 outlines the supported specs in respect with 1) the automation in FCN topology exploration, 2) the bit precision adjustment, 3) the optimization of the multiplications, 4) the variable resizing based on the value fluctuation, 5) the optimization of the activation function and 6) the optimal resource mapping.

Table 1: State of the Art Comparison

	Topology Eplor.	Bit Prec.	Mult. Opt.	Var. resizing	Activ. Opt.	Res. Map
Work#1 [11]	☑	☑	-	-	-	-
Work#2 [14]	-	☑	☑	-	☑	-
Work#3 [4]	-	☑	☑	-	☑	-
Work#4 [17]	-	☑	☑	-	-	☑
Work#5 [12]	-	☑	☑	-	☑	
Proposed	☑	☑	☑	☑	☑	-

4 METHODOLOGY

The current section deploys the proposed methodology. Our goal is to maximize the FCN generalization, achieving high prediction/accuracy rates. Our dataset is fragmented in three sets for training, validation and testing. We exploit the training sets as learning examples for our network, the validation sets as fine-tune parameters to prevent an over-fitting, and the testing sets to evaluate the accuracy The methodology steps (see Fig 2) are described in detail below.

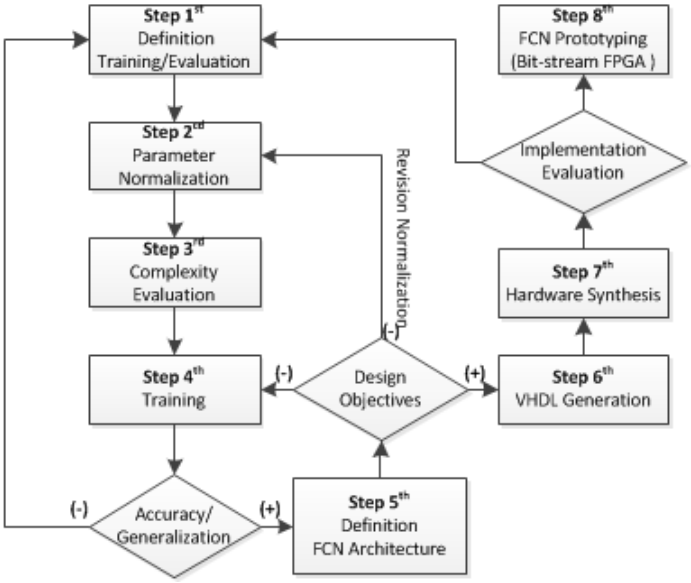


Figure 1: Methodology

4.1 Data pre-processing

Importing the training and the evaluation data set **(Step1)**, an input normalization follows **(Step2)**. FCNs training convergence is usually faster if the average of the training input is close to zero. Considering for example that all the inputs are simultaneously positive or negative, the respecting weights will be updated by an amount proportional to δx (where δ is the (scalar) error at that node and x is the input vector) and these weights will decrease or increase simultaneously. Thus, if a weight has to change direction does extreme zigzagging which is inefficient and converge very slowly. Instantiating many networks, we consider a normalization of the input values in the range [-1.25 1.25] as the most efficient.

A heuristic algorithm proposes a recommended number of nodes based on a data set complexity analysis (due to limited space the heuristic will published in future work). The recommended nodes should provide enough storage and generalization capacity for the network. However, this suggestion does not always provide a fixed optimal solution. The designer can rely on the tool recommendation and to adjust the nodes number based on the training results **(Step4)**. If training leads to many errors, extra nodes have to be added. Respectively, no errors lead to node pruning. A trial and error procedure **(Step5)** will specify the optimal FCN architecture for the final implementation. Each FCN consists multiple node layers where each node is interconnected with the nodes of the next

and previous layer in a dense network. Increasing layers also increases the learning capacity, but it also increases the hardware overhead. At CoNNs, FCLs rarely exceed the three layers. Based on this assumption, we give priority at our Framework (Section IV) to the node scalability, supporting for now up to three layers, without excluding the possibility with a design effort to be added extra layers.

4.2 FCN Trainning

In the training phase **(Step4)** back-propagation algorithms fit well to the targeted FCN feed-forward topology. After a combined trial-and-error procedure along with search in literature, we have concluded that the most efficient algorithm, in terms of efficiency and speed is LevenbergâĂŞMarquadt algorithm [7], which blends the steepest descent method and the GaussâĂŞNewton algorithm [1]. Fortunately, it inherits the speed advantage of the GaussâĂŞNewton algorithm and the stability of the steepest descent method. A study highlights the advantages of the specific algorithm [8], however it is noticed that its usability is limited to networks with up to a few hundred hidden nodes due to significant required memory resources. We use conjugate gradient methods as alternatives for large networks.

4.3 Building of the VHDL Code

After training is finished, a template-based hardware code generation is triggered **(Step6)**. This process includes three critical optimizations: 1) a FCN data types optimization by re-sizing the involved variables, 2) a multiplier optimization and 3) a activation function optimization. All the aforementioned improvements are explained at the following section. The goal is to achieve an optimal trade-off that maximizes the benefits in terms of prediction rate/accuracy and implementation cost. After training, the designer can import the derived VHDL and simulation files to the corresponding FPGA EDA tool.

5 FRAMEWORK TOOL

A Scripting Framework (see Fig 2) like Matlab APP, supports the aforementioned methodology, decreasing the design and the implementation overhead. A GUI configures the FCN through user-defined fields. We outline the most critical of them: 1) the number of the nodes - defines the learning capacity, 2) the preferable bit precision - influences the added hardware logic 3) activation function - two options one high precision calculation and another less based on the added hardware overhead, 4) multiplication style - defines the hardware multiplications implementation, which will be explained later and 5) number of training instances - number of the compared instances to define the best training. The rest of this section provides extensive analysis of the main hardware modules that utilize the aforementioned fields.

5.1 FCN Top Module

The FCN Top module implements the structure and the central control of the entire design. It instantiates the nodes and distributes tasks on them. A Top FSM arbitrates the whole execution (see Fig 3). Except from the 'Enable' and 'Reset' signals, it includes special signals to trigger the node module function and to apply a selective activation of a nodes subset. While the last option is not exploited in the current study it is usefully for dynamic nodes pruning at

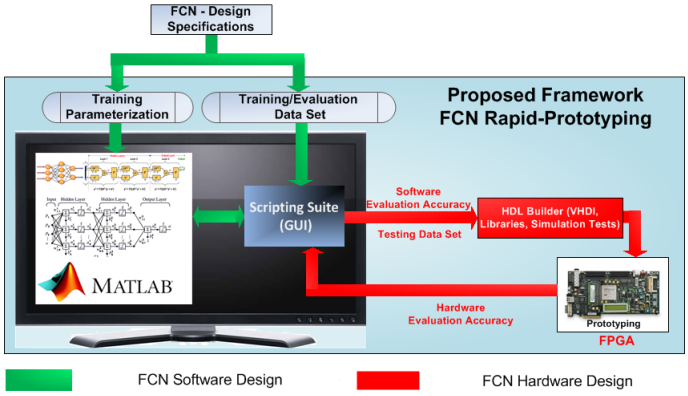
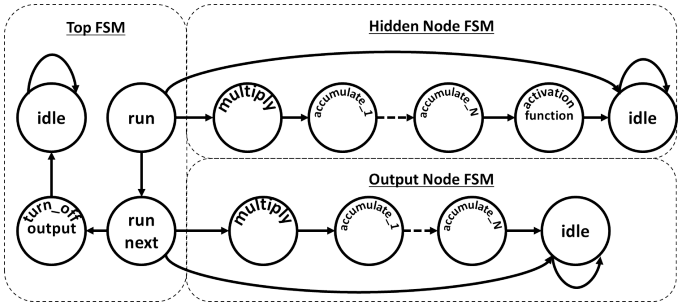


Figure 2: Proposed Framework

Figure 3: Hidden Layer FSM



future works. A main control process behaves like large AND gate that combines the 'flag' signals from all the nodes of each layer and verifies if all the nodes are enabled in order to proceed to the next computation stage. Figure 3 outlines the FSM function at the two last node layers (last hidden and output layer). 'Run' state sets the enable signals of the hidden nodes and orders their respective FSMs to start computations, whereas âĂIJrun_nextâĂİ state switches off hidden nodes while switching on output nodes. The same functionality is applied between each node layer.

5.2 Library Module

Library module consists of a package of customized types, variables, component declarations and functions that are utilized as global objects/constants to the entire design and facilitates the design scalability. Thus, any design modification or extension can be easily Incorporated thought this module, gathering all the necessary interfaces. Library also holds the auto generated types that define the possible node FSMs states in the entire design, providing a good visibility of the expected latency in clock cycles. It also holds the node data variables including weights and bias that represents the stored âĂIJknowledgeâĂİ. The input, output and intermediate values of each nodes are also stored. Considering the inputs of the FCN can be of different lengths, it would be a waste of valuable resources to use the same fixed-types. As mentioned in the methodology section, training is more efficient when the FCN input is normalized in the range [-1.25 +1.25]. Respectively this value restriction is propagated to the other interconnected layer nodes, decreasing

also the expected value range and the respective necessary bits. We will show at the node module how this improves the multiplication latency. So we use a customized fixed-point type (explained below) for each variable exploring the maximum values individually, given exactly the amount of the required bits. While this process is hard to be implemented for each node especially for large FCNs, the framework does this process automatically, generating the adjusted fixed-points in VHDL code. This can improve dramatically the design effort considering the potential scaling of the nodes in the network providing a fine-grain optimization that HLS tools is hard to support.

Library module also declares the estimation accuracy based on the user definitions. Considering that decimal numbers are inevitable in FCN and the float pointing numbers are computationally expensive for hardware implementations, fixed-points are preferred. However, EDA tools do not support fixed-point by default. For compatibility and optimization reasons, we have built our own libraries. A flexible type of signed fixedâĂŞpoint type (fixedX) for fast and resource efficient implementations where two constants specify the length of integer and fraction part. We have incorporated some extra choices to improve the design trade-offs:

(1) In the implementation of the rounding routine two options exist: round and truncate. Rounding provides more accurate results with added resources (it ensures fourth decimal place rounding) while truncating is less expensive (user defines the bit precision) but it should have adequate bits to not lose critical information due to truncation.
(2) Overflowing routine also offers two options: Saturate and wrap. Saturation is more accurate routine, but in terms of hardware consumes important resources, so in most cases it is preferred the wrap option.

5.3 Activation Function Module

The activation function is the non-linear part of the FCN influencing the training sensitivity and the achieved accuracy. After experimentation, we have concluded that sigmoid function (logsig) provides more accurate training results for the same number of hidden nodes, compared to a) hyperbolic tangent function (tansig), b) combinations of tansig and logsig and c) RelU function. Relu while being widely acceptable especially at convolutional DNNs, carries some disadvantages for the training process. One of its issue is that the negative values become zero, restricting the training ability to trace and fit properly these values influencing the accuracy. Relu is also a linear function with no upper bound, which is convenient for implementation in GPUs but a disadvantage for FPGA implementations for given limited word sizes. On the other side, the sigmoid function is a non-linear function with high computational needs that are addressed more efficiently exploiting the special FPGA DSP components. The improved accuracy in activation function is translated to less required FCN nodes for the same detection results which means less hardware resources. We rely on this effect to further improve the hardware implementation of the sigmoid applying some heuristic modifications that are analyzed below.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

To instantiate efficiently the Logistic sigmoid function (see EQ.1) in hardware, we have to deal with the time consuming division and exponential operations. Our approach relies on [15] taking advantage of two basic attributes of the function:

- f(x) is practically 0 if x ≤-8, and practically 1, if x≥8.
- It is a symmetrical function, f(-x)=1-f(x)

Heuristic 1 performs the implementation of the comparisons between -8 and 8 instead utilizing comparing signs (leq , geq) that require hardware expensive substractors. More precisely, we utilize a more simplified logic as follows.

- Input bits, except the 3 LSBs, are checked through AND/OR operations to quickly decide if the value does not reside in the [-8,8] range, in which case depending on the sign 0 or 1 is returned.
- If the value is in the [-8,8] range, then 3 bits per integer/fraction part are preserved and the value is calculated through a fast AND-OR tree, as described in [15].

Algorithm 1: Sigmoid Activation Function

```
ToZero = Input(MSB) ∧ Input(MSB-1) ...∧... Input(3))
ToOne = Input(MSB) ∨ Input(MSB-1) ...∨ ...Input(3))
if ToZero then
    Output = 0
else if ToOne then
    Output = 1
else
    if Input(MSB) = 0 then
        Keep 3 bits from decimal
        Use LUT to output logsig(Input)
    else
        Keep 3 bits from decimal
        Use LUT to output 1-logsig(-Input)
    end if
end if
```

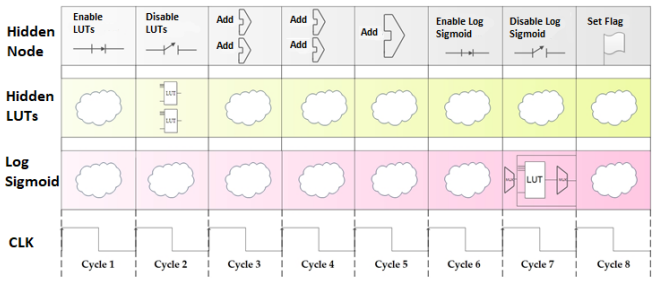
5.4 Node Module

Nodes are the FCN processing units consisting a activation function module and a local FSM. All nodes share the same code. The node output is the result of the log_sigmoid function with a range between [0, 1]. For the binary expression of the node output a 8-bit fixed point is used, 7 of which is dedicated to the decimal part. The weights and biases of the training are the structural elements of the FCN that are declared in the library module and are exploited in the node modules. When a node is enabled, parallel multiplications are triggered. Each input from the previous layer nodes is multiplied by its corresponding weight. The multiplication outcomes are accumulated and pass to the activation function to generate the node output. Equation 2 expresses this accumulation, where F is the activation function.

$$output = F(\sum(weights \times input + bias)) \tag{2}$$

A number of operands which is equal to the number of inputs from the previous layer need to be summed, and the sum will be used as input to the log_sigmoid function in order to provide the final output of the node. The scheme of additions is of critical importance, because if we choose adders with many operands this

Figure 4: Hidden node pipeline stages



will eventually be the bottleneck of our system and will have negative effect in timing performance. Eventually, we choose to add operands in pairs of two, thus the formed adder tree will have a depth of ceil(log2(N_inputs)). Fig.4 shows an example of the described adder tree, which in this case handles 6 inputs, so the depth of the tree will be ceil(log2(6)) = 3, that means that 3 cycles are required for the final sum.

A remarkable number of multipliers is required, even for mediumâĂŞsized networks. For example a given network of 8 Input Nodes, 100 Hidden Nodes and 10 Output Nodes would consist of 10x100 + 100x8 =1800 multipliers. FPGAs needs to consume valuable resources to efficiently supports such estimations, but apart from the resource restrictions, multiplications have impact at the design latency, since require significantly extended number of clock cycles. However, some conditions allow a more efficient approach of the nodes' multiplications. The first is that the training weights (the first multiplications operand) remains stable. The second is that for small fluctuation of the node inputs (the second operand), the multiplications can be simplified even more by replacing multipliers with Look-Up-Tables (LUTs). As mentioned, the FCN input data are normalized in the range [-1.25, 1.25]. Knowing the weights and the bias, the input maximum value range can be also pre-estimated as the nodes are interconnected and the normalized values are propagated in the network.

Considering that the node output is next-layer node input, having a limited range [0, 1] due to the logsig function result and after experimentation, we decide that seven bits for the fraction part provides the best trade-off between accuracy and implementation overhead. Thus, a 7-bits fraction defines 128 discrete values for the each node input defining a set of potential combinations based also on the user's desired bit precision in the intermediate operations (multiplications and accumulations). Thus, the discrete logsig input values and the weights consist a combination of multiplications predefined in LUTS that can totally replace multipliers. The key is that this is implemented transparent to the designer saving in logic units (DSPs), decreasing the required clock cycles. Framework provides the option to the designer to choose between to fully implemented multiplications in LUTs (activation function and weight multiplications) or to partial excluding the weights multiplication at the intermediate node layer(s) with the utilization of conventional multipliers.

6 EXPERIMENTAL RESULTS

The deployed methodology was applied utilizing the aforementioned framework, with two representative (CoNNs) datasets. The first CoNN network is the LeNet-5 trained on the MNIST database of handwritten digits [6], while the second one is AlexNet trained on a subset of the caltech256 dataset [2]. Both examples attempt input image classification based on feature extraction through convolution layers on a first stage, followed by a selection through dense FCLs. Both cases include a FCN with a number of parameters comparable to the convolutional part, that heavily affects the total latency [9],[18]. The main objective is to build the FCNs : 1) pruning nodes at the intermediate hidden layer (the input and output layers are stable for compatibility with the rest CoNN) 2) evaluating the generated VHDL code in a real platform modifying the accuracy in bit precision and 3) evaluating the hardware deployment time.

Considering that AlexNet has two orders of magnitude more classes than LeNet (1000 compared to 10) and due to limited computation resources for retraining AlexNet's FCN, we reduced the examined AlexNet classes at 40 (the intersection between caltech256 and ImageNet datasets). Our intention in case of AlexNet is not to compete with the original AlexNet topology but to examine our framework's scalability and to study the implementation impact modifying the accuracy . The fully AlexNet implementation will be part of future work utilizing hardware training accelerators.

For this work, we utilize the pretrained models from Caffe Model Zoo [13] for the convolutional networks to extract the inputs to the FCLs, then utilizing the described framework for the creation of the FCLs. Training, testing and validation sets are segmented per the suggested recommendations for caltech256 to 40-10-50, whereas MNIST offers separate datasets.

Tables 2, 3 outline the results for both cases. Software accuracy is estimated after FCN training keeping the best among 10 trainings and represents the theoretical bottleneck for the hardware instantiation. The objective is to examine the trade-offs between bit precision, latency, resource utilization and accuracy in hardware. For the scope of our measurements, we utilize the Zynq Ultra Scale (xqku095) platform of Xilinx and the Vivado EDA tool. Analyzing our results we conclude that: 1) a 7-bit precision seems to be enough to achieve a close to the maximum accuracy, 2) the bit precision does not seems to remarkably influence the resource utilization and the latency for a step of one bit in case of up to 200 #hidden nodes, 3) latency remains low just few hundred cycles even for quite complex FCNs (1000 #hiden nodes), 4) the code extraction time for cases up to 200 hidden nodes does not exceed the 1 hour while for big designs (1000 #nodes Alexnet) it is up to 8-9 hours in a server (2x Intel(R) Xeon(R) CPU E5-2658A v3 @ 2.20GHz 12 Cores - 24 Threads 128Gb Ram) at Matlab environment and 5) our templated-based source code exceed the 10K -50k lines for each instance that corresponds to hundred man hours for deployment by hand.

7 CONCLUSION

In the context of the current study, we deployed a methodology flow for FCN rapid prototyping. The target is to provide an automatic Framework that generates optimized VHDL code considering the accuracy, the latency, the resource utilization and the design complexity. Our tools incorporate optimizations in several phases applying customized solutions concentrating mainly at the optimal hardware implementation of the calculation operations. Two FCN application prove Framework flexibility, generating optimized code that exceeds the 10K lines for each instance within a few hours.

Table 2: Lenet - FCN Experimental Results

(a) # Hidden Nodes 40 - Software Accuracy 32%, Freq. 250 MHz

Bit Prec.	LUTS	DSPs	Hard. Accuracy	Latency (cycles)
5	108483	5	30.7%	72
6	109096	5	31.2%	72
7	109986	5	31.8%	72
8	110456	5	32%	72

(b) # Hidden Nodes 60 - Software Accuracy 58.4%, Freq. 250 MHz

Bit Prec.	LUTS	DSPs	Hard. Accuracy	Latency(cycles)
5	156398	6	36.2%	96
6	156403	6	42.7%	98
7	156453	8	47.9%	98
8	157393	8	58.1%	98

(c) # Hidden Nodes 80 - Software Accuracy 89.7%, Freq. 250 MHz

Bit Prec.	LUTS	DSPs	Hard. Accuracy	Latency (cycles))
5	169829	6	76.4%	110
6	175213	8	78.5%	110
7	175497	7	81.1%	113
8	176593	8	81.7%	112

(d) # Hidden Nodes 100 - Software Accuracy 98.2%, Freq. 250 MHz

Bit Prec.	LUTS	DSPs	Hard. Accuracy	Latency (cycles)
5	182344	8	91%	121
6	198781	11	93%	123
7	199111	14	97.1%	124
8	208781	13	97.8%	124

Table 3: Alexnet - FCN Experimental Results

(a) #Hidden Nodes 200 - Software Accuracy 32,7%, Freq. 180 MHz

Bit Prec.	LUTS	DSPs	Hard. Accuracy	Latency (cycles)
5	217668	115	21,1%	179
6	233119	124	26,4%	185
7	256993	128	30,9%	178
8	257891	143	32,4%	183

(b) #Hidden Nodes 400 - Software Accuracy 54,3%, Freq. 180 MHz

Bit Prec.	LUTS	DSPs	Hard. Accuracy	Latency (cycles)
5	269834	89	39,3%	283
6	289345	91	48,4%	284
7	309819	96	52,1%	292
8	324891	93	53,8%	298

(c) #Hidden Nodes 800 - Software Accuracy 76%, Freq. 180 MHz

Bit Prec.	LUTS	DSPs	Hard. Accuracy	Latency (cycles)
5	316366	97	68,4%	326
6	339738	105	72,9%	323
7	346671	102	75,1%	345
8	369643	107	75,3%	361

(d) #Hidden Nodes 1000 - Software Accuracy 84.6%, Freq. 180 MHz

Bit Prec.	LUTS	DSPs	Hard. Accuracy	Latency (cycles)
5	367541	181	80%	375
6	369632	186	82.7%	370
7	379505	182	83.89%	387
8	387591	190	83.9%	390

ACKNOWLEDGMENT

This research is co-financed by Greece and the European Union (European Social Fund- ESF) through the Operational Program "Human Resources Development, Education and Lifelong Learning 2014-2020" in the context of the project "*Automated methodology for production and execution of data-centric multi-level approximate equivalent applications for heterogeneous computing platforms*" (MIS 5005377).

REFERENCES

[1] Wei-Sheng Chin, Bo-Wen Yuan, Meng-Yuan Yang, and Chih-Jen Lin. 2018. An efficient alternating newton method for learning factorization machines. *ACM Transactions on Intelligent Systems and Technology (TIST)* 9, 6 (2018), 72.

[2] Gregory Griffin, Alex Holub, and Pietro Perona. 2007. Caltech-256 object category dataset. (2007).

[3] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. 2017. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 152–159.

[4] Muhammad K Hamdan and Diane T Rover. 2017. VHDL generator for a high performance convolutional neural network FPGA-based accelerator. In *ReConFigurable Computing and FPGAs (ReConFig), 2017 International Conference on*. IEEE, 1–6.

[5] Andrej Karpathy, George Toderici, Sanketh Shetty, Thomas Leung, Rahul Sukthankar, and Li Fei-Fei. 2014. Large-scale video classification with convolutional neural networks. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 1725–1732.

[6] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* 86, 11 (Nov 1998), 2278–2324. https://doi.org/10.1109/5.726791

[7] Chen Lv, Yang Xing, Junzhi Zhang, Xiaoxiang Na, Yutong Li, Teng Liu, Dongpu Cao, and Fei-Yue Wang. 2018. Levenberg–Marquardt Backpropagation Training of Multilayer Neural Networks for State Estimation of a Safety-Critical Cyber-Physical System. *IEEE Transactions on Industrial Informatics* 14, 8 (2018), 3436–3446.

[8] Chen Lv, Yang Xing, Junzhi Zhang, Xiaoxiang Na, Yutong Li, Teng Liu, Dongpu Cao, and Fei-Yue Wang. 2018. Levenberg–Marquardt Backpropagation Training of Multilayer Neural Networks for State Estimation of a Safety-Critical Cyber-Physical System. *IEEE Transactions on Industrial Informatics* 14, 8 (2018), 3436–3446.

[9] Ingo LÃijtkebohle. 2017. Xilinx CHai w/ 1024DSP @ 250/500MHz. https://github.com/Xilinx/CHaiDNN/. [Online; accessed 06-Dec-2018].

[10] Yufei Ma, Yu Cao, Sarma Vrudhula, and Jae-sun Seo. 2017. An automatic RTL compiler for high-throughput FPGA implementation of diverse deep convolutional neural networks. In *Field Programmable Logic and Applications (FPL), 2017 27th International Conference on*. IEEE, 1–8.

[11] Leonardo Reis, Luis Aguiar, Darío Baptista, and Fernando Morgado-Dias. 2014. A software tool for automatic generation of neural hardware. *Neuron* 1, 1 (2014), 229–235.

[12] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 17.

[13] Marcel Simon, Erik Rodner, and Joachim Denzler. 2016. ImageNet pre-trained models with batch normalization. *arXiv preprint arXiv:1612.01452* (2016).

[14] Alin Tisan and Jeannette Chin. 2016. An end-user platform for FPGA-based design and rapid prototyping of feedforward artificial neural networks with on-chip backpropagation learning. *IEEE Transactions on Industrial Informatics* 12, 3 (2016), 1124–1133.

[15] MT Tommiska. 2003. Efficient digital implementation of the sigmoid function for reprogrammable logic. *IEE Proceedings-Computers and Digital Techniques* 150, 6 (2003), 403–411.

[16] Andrea Vedaldi and Karel Lenc. 2015. Matconvnet: Convolutional neural networks for matlab. In *Proceedings of the 23rd ACM international conference on Multimedia*. ACM, 689–692.

[17] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design*. ACM, 56.

[18] Hengyu Zhao, Colin Weinshenker, Mohamed Ibrahim, Adwait Jog, and Jishen Zhao. 2017. Layer-wise Performance Bottleneck Analysis of Deep Neural Networks. *The 1st International Workshop on Architectures for Intelligent Machine* (2017).