FPGA Code Generator for High-Level Neural Network Descriptions

Matheus Woeffel, Gabriel Freytag, Philippe O. A. Navaux
Federal University of Rio Grande do Sul (UFRGS)– Porto Alegre, RS – Brazil
matheuswoeffel@gmail.com, {gfreytag,navaux}@inf.ufrgs.br

Abstract

FPGAs are known for their inherent architecture parallelism and their low power consumption. Despite advantages for specific application ranges, this architecture has a more laborious workflow that can inhibit the implementation and enjoyment of all architectural features. High-level synthesis alternatives target a large number of applications and therefore, while representing easier and faster implementation alternatives, often do not retrieve the full benefits of the architecture. Hence, this work presents the proposal and current state of the implementation of a tool that automatically converts high-level descriptions of neural networks to synthesizable VHDL for FPGAs. With a narrower range of applications, it is possible to combine customization, ease of implementation, and full use of the concepts inherent to the problem as well as the characteristics of the architecture itself.

1. Introduction

Research in Neural Networks (NN) in recent years has advanced in such a way that today, it is possible to use neural networks in a large number of problems [9] [1]. Some of the most common applications of NN are natural language processing, image recognition, and generic approximation algorithms. However, to compute these problems, a large amount of computational power and consequently, energy is required.

Neural Networks are usually run in High-Performance Computing systems with a large number of CPU and GPU cores, reducing the amount of time required by large applications [10]. Despite the reduced running time, the amount of energy consumed by theses architectures is quite high. As an alternative to reduce energy consumption, Field Programmable Gate Array (FPGA) architectures are gaining prominence due to its intrinsic parallelism and low energy consumption. [7]

Although being highly parallel and power-efficient, a significant drawback of FPGA architectures is the programming complexity. As in these architectures, the hardware needs to be configured to do the desired computations with extremely low-level hardware descriptions. Some of the most common programming languages for FPGAs are Verilog, VHDL, and RTL. In this way, developing applications for FPGAs involves specific knowledge of architecture itself as well as a significant amount of time.

Thus, the objective of this work is to present the proposal and the current state of the implementation of an automated Neural Network code generator for FPGAs based on a high-level description. With this tool, it will be possible to convert textual descriptions of generic neural networks into synthesizable VHDL code that can be then executed by FPGAs. This tool will be open-source to turn it easier to use by the community and also easy to modify in the future.

2. Background

Neural networks are a computational model inspired by the way processing is performed in the central nervous system of animals. Its basic unit is the neuron, which is a module responsible for making the scalar product of an input vector by a corresponding weight vector and then summing it with an offset and applying it to an activation function. The activation function can be any mathematical function, but usually derivable and soft functions such as *sigmoid*, *tanh* and *ReLU* are chosen. The neurons are then replicated in parallel to form a layer, and their concatenation is performed to form the neural network itself. This concatenation is performed by adopting the output of each neuron from an anterior layer as the input of the current layer.

Recently, this computational model has gained emphasis in the literature, due to the possibility of automatically training the network by *machine learning* algorithms. The use of such model usually involves two distinct phases: *classification/inference* and *training*. In the *classification/inference* the neural network is used to process the input, while in the *training* phase *machine learning* algorithms are used to find the weights and bias of the neurons to optimize the functionality implemented by the neural network.

In FPGAs, with rare exceptions, neural networks are implemented considering only the inference/classification phase, being the training phase usually performed in GPUs or heterogeneous architectures [14]. The present tool currently only considers the classification phase, but as future work it would be possible to evaluate alternatives for implementing the training phase also targeting the FPGAs architectures.

3. Related Work

In [7] the authors implemented a fixed point deep neural network for recognizing handwritten digits in a Xilinx FPGA. After comparing the results with GPU and CPU implementations, they found that the FGPA implementation consumed significantly less energy compared to the CPU and GPU implementations.

Given the difficulty of deploying varied applications using only hardware description languages, several manufacturers provide high-level synthesis tools to enable software application deployment and automatic conversion to synthesizable code for FPGAs. Due to the generality of the targeted applications of such tools, it is expected that the full benefit of the architecture will not be fully extracted. There are several tools in the literature that aims at enabling high level conversion using different interfaces whose target applications are specifically neural networks (like fpgaConvNet [12], DeepBurning [14], Angel-Eye [4], ALAMO [6], Haddoc2, DnnWeaver [8], and Caffeine [15]) and each of them adopt a different architectural model to implement Neural Networks in FPGAs using different interfaces and targeting different boards [13].

In [3], beside reporting the difficulty in its implementations, which are based on hardware description languages, the authors proposed an automated framework for mapping neural networks on FPGA using RTL-HLS hybrid templates. Compared to the present work, we aim to use RTL exclusively, without using other High-Level Synthesis tools.

Our work aims to target deep neural networks in general, and as many boards as possible, not being limited by manufacturers libraries or synthesis tools. Moreover, our tool will be open source and customizable to turn it easier to incorporate the user needs and ideas. Due to the direct mapping of the problem domain modules and the implementation modules, it is easy to switch between different solutions.

4. Implementation

The language selected for development was Python, due to the large number of functions that support string manipulation, which was very useful when generating the VHDL files describing the neural network. The implementation was divided in a modular way, so that the development was more easily tested, and as later justified, customizable. The following modules were considered:

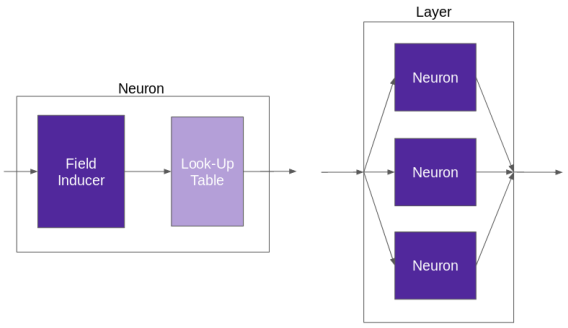

Figure 1. Neuron and Layer implementation.

- Field Inducer - Module responsible for calculating the scalar product of the input vector multiplied by the weight vector, added with the bias.
- Activation Function - Module that implements an Activation Function, which is applied to the field inducer output.
- Neuron - Module that joins the field applicator to the activation function. applicator, responsible for the synchronization and control logic of the two modules above.
- Layer - Module that gathers the neurons that define it and assumes the responsibility of mapping the inputs and weights to the respective neurons that it encompasses.
- Network - Module that brings together the layers and takes responsibility for synchronization and control between them.

In Figure 1 we show the implementation of the neurons and the layers using the five modules previously described. With the objective of facilitating posterior changes and generality of the neural networks implemented, all the model concepts defined above were directly mapped to computational modules chosen to solve the problem. Any changes in the strategy of implementation concerned with some module it is then easily mapped and modified.

From the definition of the modules to be implemented, the following design alternatives were determined for each one: considering the linear computation of the neurons, the use of parallel multipliers together with an accumulator and a state machine or the use of multipliers in parallel with tree adders were considered as alternatives. Considering the application of the activation function were found as alternatives the use of Look-up tables, approximation by *sigmoid-allipi* [11] and other linear piece-wise approximations [2]. Some alternatives for the architecture of the layers are the

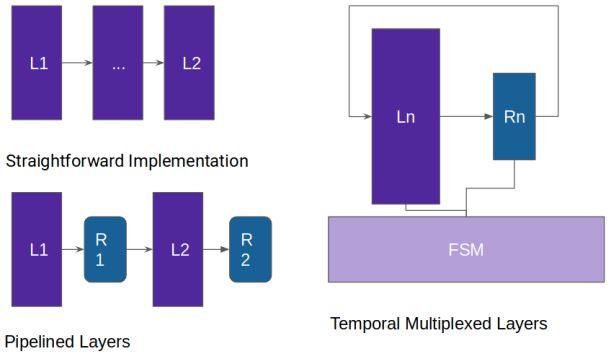

Figure 2. Layer architecture alternatives.

integration of cascaded layers with and without pipeline, as well as temporal multiplexing of a single layer together with a FSM [5]. The difference of the three alternatives can be observed in Figure 2.

Each module was represented as a class of the developed software, responsible for implementing the necessary functions of each module and the respective generation of the VHDL code that implements it. In order to assemble common responsibility for VHDL syntax pertaining to all entities used in the project, a class called VHDL entity was created, which brings together the default structure of all files as well as the instantiating syntax functions common to different components.

One of the reasons to choose modular design was the possibility of fast customization. Given the relative independence of modules, it is easy to change their implementation independently and evaluate different strategies, also supporting the goal of facilitating future code modifications for various purposes and the process of evaluating different strategies.

To measure the consumed area of each implementation alternative, the numbers of FFs, DSPs, RAMs, BRAMs, ALUTs will be collected from reports generated by the Quartus Pro tool after the synthesis process. To assess the consequences in terms of performance and energy, the neural network classification phase will be performed using the Arria X FPGA to extract execution time, instant energy consumption and power-delay product. This will allow a futher refinement of the tool and a future comparison with other deployment alternatives, such as the high-level Intel HLS compiler[1].

1 Intel HLS - https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html

Area Usage	
Resource Name	Resource Usage
ALM	404
LABs	50
Logic Registers	171
DSPs Blocks	6

Table 1. Arria 10 FPGA resource usage.

5. Initial Results

Considering the different design alternatives specified before, the present work implements the neurons using parallel multipliers in conjunction with a finite state machine and adder, layer architecture using pipeline and activation function implemented from look-up tables.

A problem encountered during the implementation was the need to keep the tool compatible with different models from the same manufacturer or even from different manufacturers, which made it difficult to use specific libraries normally provided in conjunction with each company's synthesis software. To work around this problem, only IEEE-standardized VHDL libraries were used to avoid potential compatibility issues and keeping the tool as generic as possible.

The neural network tested so far consisted of 2 layers with 2 neurons in the first layer and 1 neuron in the last layer. The network implements a simple XOR function of 2 inputs and was trained in software using the genann[2] library, which was also used for testing and debug during the implementation phase of the tool. The results for area usage are summarized in Table 1.

Due to the usage of look-up tables for the activation function, the data width used was of only 8 bits, 4 for each fractional and integer portion. To test the tool with more realistic networks will be necessary to implement approximation of the *sigmoid* or other activation functions, because of the exponential nature characteristic to look up tables.

6. Conclusions and Future Work

In this paper, we present the proposal and current state of the implementation of an automated FPGA code generator for high-level neural network descriptions. We show the various obstacles and alternatives encountered during the process of design and implementation of the tool to automatically convert high-level descriptions of neural networks to synthesizable VHDL code.

One of the major obstacles was the absence of standard libraries between different vendors, which made us implement the tool from a very low level of abstraction. Another

2 Genann - https://github.com/codeplea/genann

difficulty was the process of debugging and testing,that was laborious and error-prone due to the RTL exclusive implementation.

Given the current temporary results it is difficult to drawn symbolic conclusions and infer considerations about the project alternatives chosen. To evaluate the results and drawn more deep conclusions will be necessary to test more realistic networks and compare the results with other options presented in the literature. The aspects that will be evaluated include area usage, execution time and mainly energy consumption, using the same metrics mentioned in 4.

Regarding interface issues, future work would include the development of a caffe[3]-like scripts parser to provide an interface well known for the implementation of neural networks applications. Using an already established interface will facilitate the use and future researches regarding the tool here presented.

References

[1] U. R. Acharya, H. Fujita, S. L. Oh, Y. Hagiwara, J. H. Tan, and M. Adam. Application of deep convolutional neural network for automated detection of myocardial infarction using ecg signals. *Information Sciences*, 415:190–198, 2017.

[2] K. Basterretxea, J. Tarela, and I. Del Campo. Digital design of sigmoid approximator for artificial neural networks. *Electronics Letters*, 38(1):35–37, 2002.

[3] 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)*, pages 152–159. IEEE, 2017.

[4] K. Guo, L. Sui, J. Qiu, J. Yu, J. Wang, S. Yao, S. Han, Y. Wang, and H. Yang. Angel-eye: A complete design flow for mapping cnn onto embedded fpga. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(1):35–47, 2017.

[5] S. Himavathi, D. Anitha, and A. Muthuramalingam. Feedforward neural network implementation in fpga using layer multiplexing for effective resource utilization. *IEEE Transactions on Neural Networks*, 18(3):880–888, 2007.

[6] Y. Ma, N. Suda, Y. Cao, S. Vrudhula, and J.-s. Seo. Alamo: Fpga acceleration of deep learning algorithms with a modularized rtl compiler. *Integration*, 62:14–23, 2018.

[7] J. Park and W. Sung. Fpga based implementation of deep neural networks using on-chip memory only. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 1011–1015. IEEE, 2016.

[8] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh. From high-level deep neural models to fpgas. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, page 17. IEEE Press, 2016.

[9] B. Shi, X. Bai, and C. Yao. An end-to-end trainable neural network for image-based sequence recognition and its application to scene text recognition. *IEEE transactions on pattern analysis and machine intelligence*, 39(11):2298–2304, 2016.

[10] D. Strigl, K. Kofler, and S. Podlipnig. Performance and scalability of gpu-based convolutional neural networks. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 317–324. IEEE, 2010.

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

[12] S. I. Venieris and C.-S. 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)*, pages 40–47. IEEE, 2016.

[13] S. I. Venieris, A. Kouris, and C.-S. Bouganis. Toolflows for mapping convolutional neural networks on fpgas: A survey and future directions. *ACM Comput. Surv.*, 51(3):56:1–56:39, June 2018.

[14] Y. Wang, J. Xu, Y. Han, H. Li, and X. Li. Deepburning: automatic generation of fpga-based learning accelerators for the neural network family. In *Proceedings of the 53rd Annual Design Automation Conference*, page 110. ACM, 2016.

[15] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong. Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2018.

3 Caffe - https://caffe.berkeleyvision.org/