NNHC - a Neural Network to Hardware Compiler

Sascha Schmalhofer*, Yasmine Abu-Haeyeh* and Lars Hedrich*
* *Institute for Computer Science, Goethe University Frankfurt, Germany*
schmalhofer/abu-haeyeh/hedrich@em.cs.uni-frankfurt.de

***Abstract*—In this paper we present a compiler which can translate trained neural networks into an analog hardware. The compiler creates schematics, layouts and symbols and stores them in a state-of-the-art design database. The layouts for these complex neural network circuits are fully automatic generated based on basic block cells using specialized placing methods depending on the layer and an own routing mechanism based on a command language to enable the needed routing of very large busses. Additionally, the compiler allows to simulate the generated hardware with transient and Monte-Carlo simulations and to evaluate the simulator outputs to provide quick, accurate and detailed information about the quality of the generated analog classification circuit. We show the feasibility of the approach and the layout quality on 10 examples.**

***Index Terms*—analog, neural networks, generation, power-efficiency, neural network compiler**

I. Introduction

A lot of AI-applications demand for edge devices with low power consumption and special AI tasks, like medical diagnosis, detection of analog signal events or image recognition. Analog neural networks (ANNs) are possible candidates for realizing low power inference engines. An overview of hardware implementations with corresponding figures of merit are presented in [1]. In this paper we want to concentrate on the electrical and physical implementation methodology for large, low power, pure analog, sequential ANNs. These kind of circuits do not reuse some analog hardware controlled by software and memory. In contrary they are doing the inference task with pure analog hardware having some advantages in power consumption and a lower vulnerability to attacks. However, due to their size, their schematic and layout can not be constructed manually. Hence we present here a sophisticated generator to construct these networks.

A. Related Work

There are different analog implementations of inference engines existing. An analog ReLU implementation is presented in [2]. These kind of circuits may be used as basis block cells. Independent of what basis cell is used, in all of these approaches, the block has to be combined to an ANN. In most cases, small sub-blocks consisting of several inputs and weights are constructed, which are digitally excited and finally digitally stored after some A/D converter. A repetitive use like in in-memory computing, explained in the overview paper [3] is more easy to construct as small blocks can be designed and copied.

Another approach is to remove every digital control and memory circuit to have a pure analog implementation. In [4] a version with at most 14 manually designed neurons is realized on a chip. However if the neuron number becomes large, the placement and routing is not trivial, as we will see later. The solution are analog generator approaches. [5, 6] constructed small OP and OTA-designs. Other approaches are concentrating on special devices like DACs or ADCs [7] with thousands of elements but in principle a very regular design.

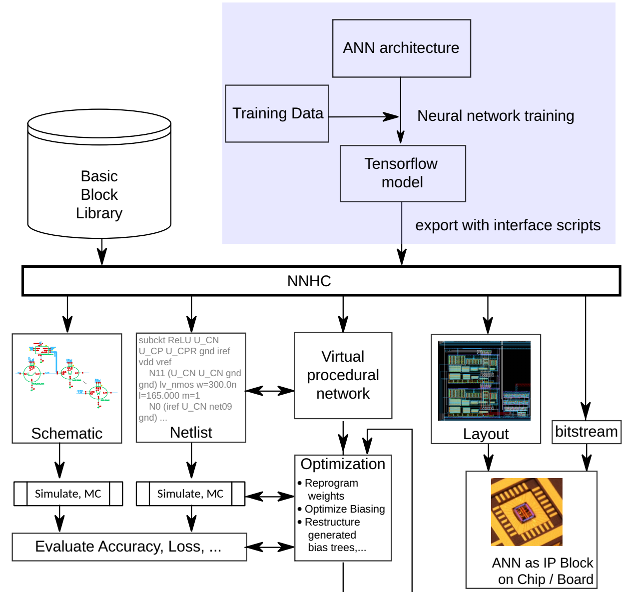

Fig. 1: Workflow of generation of analog neural net (ANN). The Tensorflow generation is shaded in blue. The basic block library consists of weight and bias cells stored in a Cadence database (DFWII). The schematic and netlist is generated in order to simulate, measure and debug the ANN. The blocks, structure and weights may be optimized, as an optimization loop can always use the same overall properties like accuracy. Finally a layout is generated which in combination with a programming bitstream can realize the ANN as an IP-block.

In our case we have different layout schemes for different layers and a lot (up to thousands) of parallel signals to handle and guide through the layout, such that a simple floorplan and template driven placer and router will not succeed. Additionally we have to deal with mutiple signals to be fed into the circuit, weights, input signals, and output signals as well as bias voltages/currents.

Our contribution in this paper is a Neural Network to Hardware Compiler (NNHC) based on the experiences with the generator from [8] and [9]. It combines the idea of a debugging framework for the complex evaluation problem presented in [10] with a procedural driven schematic, netlist and layout generator. The last one solves for the first time the complex layout generation of ANNs for hundreds of weights in a fully automatic way.

II. Preliminaries

A. Block Library

There is a number of basis cells in a Cadence-Virtuoso library to generate the neural network circuit. Those cells are

designed to calculate the application of activation functions like ReLU and Tanh, multiplication and sum by power aware analog operations. The cells are currently designed in a 130nm CMOS process. The whole library is designed and optimized to realize a power-efficient inference engine. The ReLU circuit and Weight cells are based on current mirrors [9]. The sum operation is implemented by a summation of currents. All cells are designed for a current range between $-200nA$ and $200nA$ to keep the MOS transistors in the subthreshold region. Tensorflow values x_{TF} can be translated to currents i_A in the analog domain (and vice versa) with the following equation:

$$i_A = x_{TF} \cdot 50nA \quad (1)$$

B. Metacircuit library

We needed a class library to describe circuits and layouts in Common Lisp, so we implemented METACIRCUIT. Circuits in this library can have one schematic and multiple layouts. The library is connected to the Cadence Design Framework with scripts in it's programming language SKILL [11]. METACIRCUIT enables to hierarchically describe circuits as schematic and layout. Symbols are generated automatically out of the circuit description. There are methods for importing and exporting schematics and layouts from Cadence Virtuoso with SKILL scripts. The library has its own SPICE-parser and its own netlister. It is possible to start Cadence Spectre simulations and read simulation results. There is a special turtle graphic inspired wiring language (see below).

C. Neural-Networks library

We implemented NEURAL-NETWORKS as a Common Lisp library for neural networks. It is an absolute minimal Tensorflow replacement. There are no training algorithms at all in this library, because training is done with the Tensorflow library. But it implements data structures for neural networks and their most important layer types. It can read and write its own file format for neural networks and it can read a neural network out of Tensorflow's *.keras files. There are datastructures and algorithms to hold and calculate the most relevant measures for classificators in general.

III. NEURAL NETWORK HARDWARE COMPILER (NNHC)

An overview of the functionality of the NNHC framework is given in Fig. 1. It is written in Common Lisp and depends on the block library and the programming libraries METACIRCUIT and NEURAL-NETWORKS, which were described in section II.

Schematics and layouts for all circuits are generated together hierarchical in one method. The relative positions of the subcell instances are mostly the same for schematic and layout. The wiring of the schematic is completely done by placing and naming net stubs. On top level, the neural network is hierarchically expanded into NN-layer cells. For each NN-layer type a dedicated translation method exists. They are mostly general, but implement different placement and network structures depending on the layer type. For example, a 2D-convolutional layer can not easily be mapped into a planar graph for the schematic and layout. A hierarchical graph consisting of the 2D-convolutional filters arranged in a 2D-matrix structure is used. On the other hand a dense layer may be arranged in a simple 2D-matrix structure.

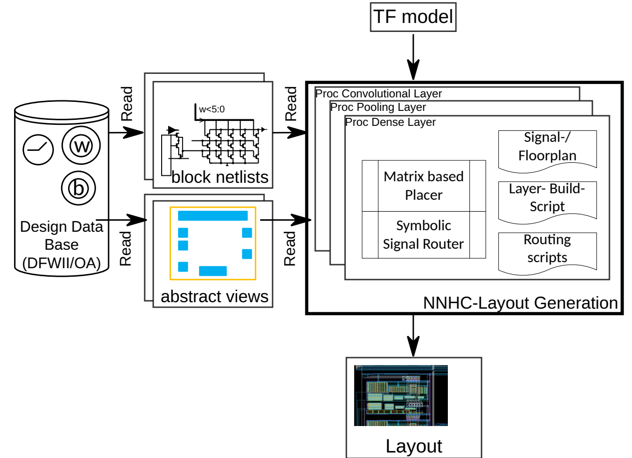

Fig. 2: Layout generator for ANN networks. It is based on a frontend to the DFWII-database to extract basic circuits and layouts. It consists of a generator script for each layer type using an appropriate symbolic floorplan resulting in a matrix like generation of the placement. The router is semi symbolic. It has a input language to perform a kind of symbolic step to connect to from the abstract view extracted pins.

For the schematic and the layout, a large number of different wire types have to be realized:

- The input and output signals of layers. Due to the used transistor-level architecture, a connection between an activation function and the weights uses 3 wires.
- The analog bias voltages and currents of the analog cells.
- The programming bits (6 bit or 8 bit) have to be provided to the weight cells and bias cells, respectively. Depending on the layer type, the underlying distribution graph looks different. For large convolutional layers the signals have to be buffered by an automatic buffer insertion process.
- The programming cells are connected as one large shift register for programming purposes.
- Each cell needs power supply and reference voltage signals.

In the most complex layout, the 2D-convolutional layer, 5 routing layers are needed to realize the whole wiring. As we are talking about large buses (e.g. 3*input size), an automatic wiring wire by wire as in a standard cell design flow is not possible.

The drawing of wires for the layout is automatically done accompanying the placement of wire-related cells in the schematic/layout. The wiring-language of the METACIRCUIT system is used here. It uses the following input data:

- A list of the form $(x\ y\ l)$ with a start position x, y and a wiring layer l to start the wire from, or a list of the form $(i\ p)$ with a layoutinstance i and a pin name p as a starting point. The starting layer is the layer of the pin material in the latter case.
- A starting width w for the wire. This can be adapted later during drawing of the wire.
- A list of arbitrary many command and argument pairs a_1 c_1 ... a_k c_k describing the wiring in the following way.

The command c_i and argument a_i mean the following:

- $c_i \in \{east, west, north, south\}$ means to move a_i many units in the direction of the name of the command.
- $c_i = width$ sets a new thickness for the next wire elements.
- $c_i = up$ or $c_i = down$ places a via stack at the current position.

- $c_i = horizontal$ or $c_i = vertical$ draw a wire with the current thickness in a horizontal first or vertical first manner to an absolute position $a_i = (x\ y)$ or to a pin position $a_i = (i\ p)$ given by an instance i and a pin p.
- $c_i = pin$ places a pin at the current position.

After the translation process, the hierarchical neural network consisting of cells for each layer can be stored in the DFII database as a schematic and a layout repectively. Additionally a testbench is generated. It is used for transient or Monte-Carlo simulations. The input patterns of the Tensorflow test set are translated into current sources. A direct transient and Monte-Carlo evaluation can be performed using the Spectre circuit simulator. As the netlists are directly generated from the NNHC-framework, a quick design exploration or an optimization over parameters or different architectures is possible.

IV. Results

We have set up a number of examples for neural networks. See Table I for an overview.

xornet consists of two dense layers with ReLU activation functions and calculates the result of a xor operation out of two floating point values.

iris has established as a classic dataset for machine learning applications [12]. The main idea is to train a neural network to distinguish species of iris.

The *circles* network with two inputs and one output detects whether a point is in a circle or not [12].

peakdetector is a neural network, which can detect wavelets (peaks) within sine-signals and a random noise floor. This transient signal is sampled to a vector of 11 input values. The output should be true for a peak in the transient signal.

The network examples *mnist1d*, *mnist* and *mnist_red* all operate on the handwritten number recognition problem [13] but differ in architecture and resolution of the input image (see Table I for details).

The example *fashion_mnist1d* and *fashion_mnist* are images from fashion products [14].

We also tested the compiler on a neural network which can detect guitar sounds out of audio samples (*guitar* example). A subset of the Nsynth dataset [15] is used for the training of this example. This leads to an achitecture with 400 input-neurons, six hidden layers and one output-layer.

Network	pat.	inp. dim.	outp. dim.	arch.
xornet	4	2	1	FC(2),FC(1)
iris	75	4	3	FC(8),FC(3)
circles	50	2	1	FC(20),FC(1)
peakdetector	100	11	1	FC(18),FC(1)
mnist1d	100	196	10	FC(10),FC(10)
fash._mnist1d	50	196	10	FC(10),FC(10)
guitar	50	400 x 1	1	CV(8),PL(6),2xCV(8), 2xPL(6),FL(20),FC(1)
mnist	100	14 x 14 x 1	10	CV(3x3),FL(144),FC(10)
fashion_mnist	50	14 x 14 x 1	10	CV(2x2),FL(169),FC(10)
mnist_red	100	14 x 14 x 1	10	CV(7x7),FL(16),FC(10)

TABLE I: Number of patterns in testset, input dimensions, output dimensions and the architecture for the example networks. FC: fully connected layer (with number of neurons in parenthesis), CV: convolutional layer (kernel size), FL: flatten layer(output size), PL:Pooling Layer(pooling size).

A. Transient Simulation

The direct call of transient simulation is a quick evaluation method using the generated testbench. There is no need for generating the schematic in Virtuoso before calling the transient simulation, the internal datastructure which describes the circuit can be translated directly into a Spectre netlist. The results are evaluated and returned as a classificator datastructure including the main properties like accuracy and loss, in our case mean squared error (see Table II). In nearly all cases the accuracy after training (Tensorflow) can not be reached with the analog NN due to nonlinearities, inaccuracies, leakage and other effects on transistor level. However the drop in accuracy is quite low with respect to the low energy used for a pattern inference (EPP). Only the guitar example has a bad hardware accuracy (HWACC) due to low signal to noise ratio at the input.

Network	TFACC	TFMSE	HWACC	HWMSE	EPP
xornet	100.0%	2.96E-7	100.0%	1.986E-3	9.1nJ
iris	96.0%	5.06E-2	94.7%	5.477E-2	25.3nJ
circles	96.0%	4.93E-2	80.0%	1.578E-1	56.8nJ
peakdetector	79.0%	1.56E-1	77.0%	1.74E-1	57.3nJ
mnist1d	87.0%	2.46E-2	78.0%	3.57E-2	182.3nJ
fash.mnist1d	70.0%	4.94E-2	68.0%	5.665E-2	228.1nJ
guitar	98.0%	2.07E+0	56.0%	8.637E+1	1403.3nJ
mnist	92.0%	2.23E-2	88.0%	3.585E-2	299.3nJ
fashion_mnist	76.0%	5.29E-2	74.0%	5.839E-2	471.0nJ
mnist_red	89.0%	3.16E-2	87.0%	3.503E-2	142.6nJ

TABLE II: Tensorflow Accuracy (TFACC), Hardware Accuracy (HWACC), Tensorflow Mean Squared Error (TFMSE), Hardware Mean Squared Error (HWMSE) and Energy Consumption per pattern (EPP) for a number of selected example networks

B. Monte-Carlo Simulation

In the presence of process and mismatch variations of the underlying manufacturing process, the NNHC framework can start Monte-Carlo simulations and evaluate the results, e.g. determine a yield depending on acceptance functions.

Fig. 3 shows a histogram of the accuracy of the *mnist_red* example. Besides the small tail with low accuracies, the NN is relatively robust versus process and mismatch variations. For example, with a limit of 84% accuracy a yield of more than 75% is reached.

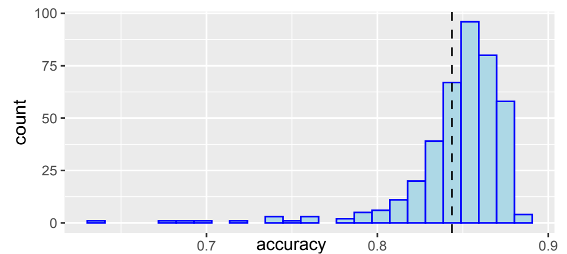

Fig. 3: Histogram of the accuracies of 400 Monte Carlo instances (mismatch and process variations) of *mnist_red* network.

C. Generated Layouts

For all examples we generate the layout for all layers in a fully automatic way as described in this paper. The results are shown in Table III. A_t is the active area of transistors estimated with W, L from the schematic and is a lower bound for the needed area. It is summed up over all layers and all subblocks. A_r is the recursive calculated real area of the generated layout

Network	n_w	n_b	n_t	$A_t[mm^2]$	$A_r[mm^2]$	o
xornet	7	8	2321	0.0065	0.0163	152%
iris	59	26	14209	0.0337	0.0774	130%
circles	61	44	19379	0.0554	0.3187	475%
peakdetector	217	49	53292	0.1436	0.9919	591%
mnist1d	2070	236	393005	0.7529	1.5484	106%
fash.mnist1d	2070	236	393005	0.7529	1.5484	106%
guitar	4589	1077	235737	1.6510	*	*
mnist	2746	360	309567	0.7714	1.6457	113%
fashion_mnist	2376	554	350039	0.8802	1.8869	114%
mnist_red	954	205	66361	0.2438	0.5785	137%

TABLE III: Number of weight cells n_w, number of bias cells n_b, number of transistors n_t, estimated area A_t (transistor area), real area A_r and the layout overhead $o = \frac{A_r - A_t}{A_t}$ of the network examples. There is no A_r for guitar example because there is currently no layout generated for onedimensional convolutional layers.

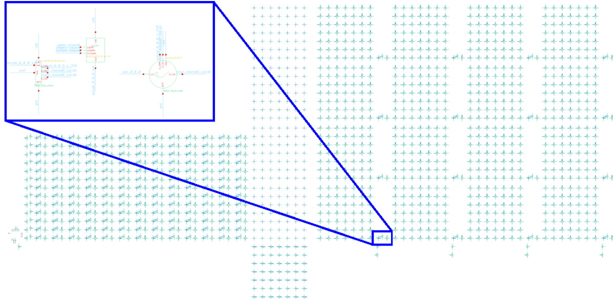

Fig. 4: Generated schematic of a twodimensional convolutional layer. This example is taken from *mnist_red* network. It is the first layer in this example. The result of this layer is a 4 by 4 matrix, which can be seen in the right part of the image.

for the examples. It is defined as the area of the bounding box of the cell if the cell has a bounding box and the sum of the areas of the layoutinstances if the cell has no bounding box. It represents the real needed chip area in the used 130nm Technology.

All examples have roughly an overhead of 120% over the transistor area, which is a good value, because this overhead is nearly reached if source and drain areas are taken into account. That means, that the area overhead due to routing areas seems to be very small.

As an example we show the results of the generation of *mnist_red* in detail: In Fig. 4 the generated schematic of the 2-D convolutional layer of that example is shown. The corresponding layout is shown in Fig. 5 being the most complex generated layout for one layer of our examples. An example for a generated layout for the fully connected layer of the same example is shown in Fig. 6.

V. Conclusion

In this paper we presented a new compiler for translation of neural networks into an analog hardware circuit including layout. Based on exchangeable basic blocks, it consists of a set of generators being able to generate complex layouts with a lot of wide buses. The generator needs about 120 % overhead over the pure active transistor area which is extremely compact. The method can be easily applied to more complex NNs. A migration to other technologies is possible by exchanging the underlying basic block library as the generators try to use the layout data in symbolic form by extracting bounding boxes and pin positions from the blocks. For future work we intend to extend the method to more optimization techniques like automatic yield improvement.

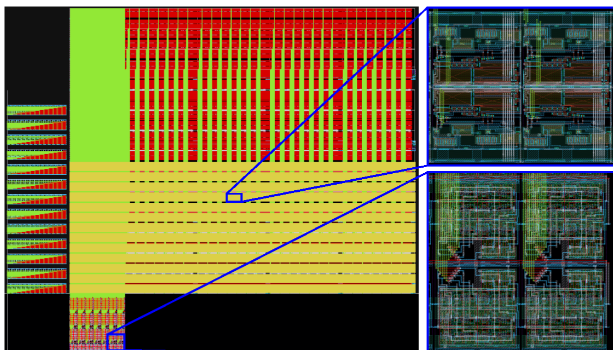

Fig. 5: Generated layout of the first layer of *mnist_red*. This is a two-dimensional convolutional layer. The schematic of this layer can be seen in Fig. 4.

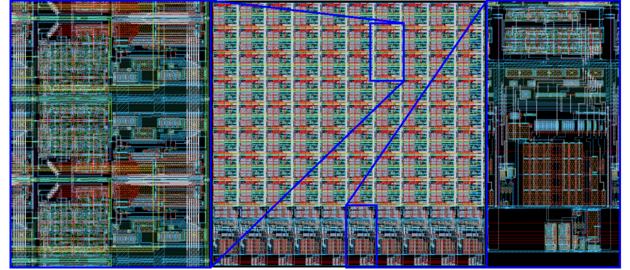

Fig. 6: Generated layout of a dense layer for *mnist_red* network. The dense matrix (6-bit programming cells and multiplication cells at every weight position in the matrix) can be found on the top of the image. Under this dense matrix, there are programming cells for the bias and the bias cells itself. On the bottom of the picture are the cells for bias currents and the activation cells.

References

[1] K. Guo, W. Li *et al.* "neural network accelerator comparison". [Online]. Available: https://nicsefc.ee.tsinghua.edu.cn/projects/neural-network-accelerator.html

[2] Y. Huang, Z. Yang, J. Zhu, and T. T. Ye, "Analog circuit implementation of neurons with multiply-accumulate and ReLU functions," in *Proceedings of the 2020 on Great Lakes Symposium on VLSI*, 2020.

[3] J. Sun, P. Houshmand, and M. Verhelst, "Analog or digital in-memory computing? benchmarking through quantitative modeling," in *2023 IEEE/ACM International Conference on Computer Aided Design*, 2023.

[4] J. Binas, D. Neil, G. Indiveri, S.-C. Liu, and M. Pfeiffer, "Precise deep neural network computation on imprecise low-power analog hardware," *arXiv: Computer Science/Neural and Evolutionary Computing*, vol. 1606, no. 1606.07786, 2016.

[5] A. Graupner, R. Jancke, and R. Wittmann, "Generator based approach for analog circuit and layout design and optimization," *Design, Automation Test in Europe*, 2011.

[6] B. Prautsch, U. Hatnik, U. Eichler, and J. Lienig, "Template-driven analog layout generators for improved technology independence," in *16th GMM/ITG-Symposium ANALOG*. VDE, 2018.

[7] B. Prautsch, U. Eichler *et al.*, "Iip framework: A tool for reuse-centric analog circuit design," in *International Conference Simulation Methods and Applications to Circuit Design (SMACD)*. IEEE, 2016.

[8] F. Aul, N. Katsaouni, M. H. Schulz, and L. Hedrich, "Synthesis of Power-Efficient Analog Neural Networks for Signal Processing," in *17. ITG/GMM-Symposium Analog*, 2020.

[9] F. Aul, N. Katsaouni, L. Krischker, S. Schmalhofer, M. H. Schulz, and L. Hedrich, "Schematic generation of programmable analog neural networks for signal proccessing," in *SMACD*, 2021.

[10] S. Schmalhofer, M. Moeller, N. Katsaouni, M. H. Schulz, and L. Hedrich, "Debugging low power analog neural networks for edge computing," in *Design, Automation & Test in Europe Conference*, 2023.

[11] Cadence Design Framework, "www.cadence.com."

[12] F. Pedregosa, G. Varoquaux *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, 2011.

[13] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, 2012.

[14] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms," 2017, https://github.com/zalandoresearch/fashion-mnist. [Online]. Available: http://arxiv.org/abs/1708.07747

[15] J. Engel, C. Resnick *et al.*, "Neural audio synthesis of musical notes with wavenet autoencoders," *arXiv:1704.01279*, 2017.