Evaluation of the Design Exploration of a Binarized Neural Network for FPGA using HDLRuby

Lovic Gauthier*, Yōhei Ishikawa*, Takuma Matsuno*

aNational Institute of Technology, Ariake College, Higashi-Hagio 150, Omuta, Fukuoka 836-8585 JAPAN

*Corresponding Author: lovic@ariake-nct.ac.jp

Abstract

This paper presents an example of design exploration using the hardware description language HDLRuby of a binarized neural network (BNN) targeting a FPGA. It took 21 hours for describing and testing down to the FPGA mapping a BNN circuit that is customized directly from the result of offline software training. The description in HDLRuby has a total of 714 lines of code for any possible BNN architecture, while the same hardware reimplemented in Verilog HDL code requires 1107 lines of code for a single BNN architecture. Mapped on a FPGA, the BNN with an optimal structure of one inner layer with 256 neurons succeeded in computing one result per cycle, for a power consumption lower than 6.5W.

Keywords: Hardware Description Language, Design Exploration, FPGA, Neural Network.

1. Introduction

HDLRuby(1-2) is a hardware description language (HDL) based on Ruby(3) aiming at increasing the productivity of hardware (HW) design. Fig. 1. Illustrates the design flow with this language: HDLRuby is processed by the HDLRuby compiler for simulation, or generation of Verilog HDL(4) (Verilog) or VHDL(5) synthesizable Register Transfer Level (RTL) code. This resulting code can then be synthesized or simulated by conventional RTL design tools.

In previous research we evaluated the productivity of HDLRuby for several cases(12-13). In this paper we focus on the metaprogramming capability of HDLRuby and evaluate its benefits for the design exploration of HW As target application we chose to implement a binarized neural network(6) (BNN) with offline software (SW) training and online HW execution on FPGA.

BNN are implementations of neural networks for binary-oriented computing systems where floating-point computations are replaced by binary ones: XNOR operations are used in place of multiplications, bit count operations are used in place of summations, and the sign bit is used in place of the activation function. While the execution of such neural networks (NN) on FPGA is highly efficient, its online binary training is an open problem. Instead, offline floating-point-based training methods are usually preferred (6). BNN present two advantages as targets for evaluating HW design exploration with HDLRuby: first they are promising architectures that could make FPGA implementations of NN more efficient than GPU ones, and second, the offline training is a good opportunity to evaluate the easiness of integrating SW with HW using this language.

2. Related Works

Several research works have been done for decades for improving the productivity of HW design since RTL synthesis(7). We can mention the early works in behavioral
synthesis\textsuperscript{(8)}, or the more recent high-level synthesis\textsuperscript{(9)} that synthesizes HW from SW code. Unfortunately, the synthesis results of these approaches are still weak compared to RTL synthesis. More advanced HW description languages like System Verilog\textsuperscript{(10)} and recent instance of VHDL\textsuperscript{(11)} have also emerged, but the support of their features remains limited.

We presented HDLRuby in a previous paper\textsuperscript{(2)} and made preliminary evaluations\textsuperscript{(12-13)} as well as proposed new design constructs for this language\textsuperscript{(13-14)}. These works showed promising results but the design exploration of large circuits, and the comparison with handwritten Verilog using its generic constructs remained to be done. This is the primary goal of this paper.

The natural targets for implementing neural networks are GPU for their high efficiency in multiplication-heavy floating-point computations. Yet, they are power-hungry and hard to scale for embedded systems. Being even more parallel, and consuming far less power, FPGA may be good and hard to scale for embedded systems. Being even more floating-point computations. Yet, they are power-hungry are GPU for their high efficiency in multiplication-heavy floating-point computation is low compared to GPU, and implementing large structures is more complex in HW than is SW\textsuperscript{(15)}. For addressing the issue about floating-point computations, recent works propose to binarize the neuron computations\textsuperscript{(6,16-17)}. In this paper we propose to implement such an architecture using the HDLRuby language.

3. Implementation of the BNN

3.1 The architecture of the BNN

BNN are neural networks (NN) modified for binary computation where the only possible input and output values of a neuron are -1 and +1. Specifically, with a typical NN, each neuron computes the following function where \(X=(x_0,..., x_{N-1})\) is the input vector \(W=(w_0,..., w_{N-1})\) is the weights vector, \(b\) is the bias and \(g\) is the activation function:

\[
a(X) = g\left(\sum_{i=0}^{N-1} w_i x_i + b\right) \quad (1)
\]

The neurons of a BNN compute a similar function, but the only possible values of each element of vectors \(X\) and \(W\) are -1 and +1, the bias is an integer, and the activation function \(g\) is the sign function.

BNN are especially good for FPGA implementations because of the cost of the floating-point \(WX\) vector product. In case of BNN, each multiplication of this vector product is reduced to one of the following operations: \(-1 \times -1 = 1\), \(-1 \times 1 = -1\), \(1 \times -1 = -1\), and \(1 \times 1 = 1\). When implemented in hardware, the -1 values are represented by \(0\), so that these operations can be implemented by XNOR functions. Then the sum of equation (1) can then be implemented by the count of 1 in the bit vector resulting of the XNOR operation subtracted by \(N/2\) for compensating the conversion of -1 to 0. This last subtraction can be merged into the bias so that the resulting function is the following, where \(\bigcirc\) is the bitwise XNOR, \(pco\) is the function counting the 1 bits of a vector (also called popcount), and \([N-I]\) is the access to the most significant bit, which corresponds to the sign function:

\[
a(X) = \frac{\text{pco}(W \bigcirc X) + b}{[N-I]} \quad (2)
\]

Fig. 2 illustrates the HW implementation of such a BNN neuron. In the figure, \(\Sigma\) represents the popcount circuit, \(\bigcirc\) represents the adder circuit and the grey boxes are constants.

When targeting a FPGA implementation, the binarized XNOR of two bit-vectors and the sign functions are trivial. However, several implementations can be considered for the popcount function. The most straightforward implementation is the successive addition of each bit value, but its performance may be low. A possibly more efficient implementation consists in first using \(2^M\) entries lookup tables (LUT) for obtaining in parallel the number of 1 in each M-bit portion of the vector then to sum the outputs of these LUT. However, what will be the best implementation is dependent on the target devices and the used synthesis and technology mapping tools. For this paper, the type of implementation as well as the size of the LUT are deduced from one generic parameter of the design.

 Compared to other BNN implementations\textsuperscript{(6,16-17)}, an originality of the proposed designed is the use of constants for weights and biases instead of coefficients stored into a memory. The rational for this choice is that it is more efficient and that the target is a FPGA so that the whole circuit can be changed after production as easily as the content of a coefficient memory.
The global architecture of the BNN is a pipeline where each stage is a layer of binarized neurons as shown in Fig. 3. Each neuron of a layer sends its result to a pipeline register that serves as input vector for the neurons of the next layer (if there is one). The last pipeline register serves as output for the circuit. With such an architecture, the BNN can produce one result per cycle independently of the number of layers it contains.

3.2 The offline training step

The HDLRuby description of the circuit has been designed so that its structure can be automatically generated from the result of the offline training step. We used a Ruby-based neural network SW library called FastNeurons(18) for this step. The training method consists of using a floating-point version of the network using the sign function as activation function and rounding the weights to -1 or +1 when performing the $WX$ product. The training results are saved into an open format (json) containing the weights and biases of the neurons. The method has been selected because it is simple enough to be implemented with a readily available NN library, although a very high accuracy of the training result is unlikely compared to more advanced algorithms(6).

Several BNN architectures have been trained for recognizing handwritten digits using the MINST(19) dataset. To match this dataset, the input of each BNN were vectors of 784 elements and the output vectors of 10 elements. Fig. 4. gives the accuracy of a BNN depending on the number of neurons in the inner layer. As seen in the figure, the increase of accuracy stalls around 80% with 256 neurons in the inner layer.

3.3 The description in HDLRuby of the BNN

All the possible BNN architectures are produced from a single HDLRuby description taking as only parameters the training results file name and the optional size of the LUT used in popcount. The remaining parameters, including the actual structure of the BNN are automatically inferred from them. The code is available at: https://github.com/civol/HDR_BNN

The description comprises the following modules.

**bact**: it is the module implementing the addition of the bias and activation function. Its code is given Fig. 5, lines 1-9 as an illustration of HDLRuby code. In the figure, 'typ' is a parameter giving type of the input, 'x' is the input value, 'b' is the input bias, 'z' is the output and 'y' is an inner signal of signed type one bit larger than 'x'. 'typ.width' is the construct that gives the bit width of type 'typ', '[−1]' is the operator selecting the most significant bit, '[u,v]' is the concatenation operator and '−' is the not operator.

**popcount**: it the module computing the number of 1 in a bit vector. Its full code is omitted here, but it uses the generic function popcountLUT given lines 11-19 of Fig. 5. for generating the initial sub sums. This function generates LUT computing the sum of the 'n' bits of input vector 'ivec'. When 'n' is 1, no LUT is generated and 'ivec' is returned as is (lines 12-13). Otherwise, a LUT is generated whose output size is the bit width of parameter 'n' (line 15) and whose content is given by table 'tbl' of $2^n$.

![Fig. 3. The pipeline structure of the BNN.](image)

![Fig. 4. Accuracy of the BNN after training.](image)

![Fig. 5. HDLRuby code samples of BNN (1).](image)
elements (line 16). The compile-time generation of each entry of this table is performed by calling the Ruby function 'popcount_' (its code is omitted for sake of conciseness). Line 17 finally returns the access to the LUT.

**bnmac**: it is the module that computes the BNN version of the WX product in a neuron. Its inputs are the input vector X of the neuron and the constant weights vector W. It performs the XNOR between its inputs and include an instance of the popcount module for counting the number of bits of the result.

**bneuron**: it is the module that describes a binarized neuron. Its input is the input vector of the neuron X, its output in the result of the neuron computation. This module simply describes the interconnections between an instance of the bnmac module and an instance of the bact module.

**bdense**: it is the module that describes a fully interconnected layer of neurons. Its input is the input vector of the layer, i.e., the outputs of the previous layer or the input of the BNN, its outputs is the vector of all the outputs of its neurons. Its code is given Fig. 6 lines 1-14 as an illustration. Its generic parameters are 'typ' that gives the data type of the input vector, 'width' that gives the number of bits summed by a LUT in the popcount module (4 by default) and 'ws_b' that is a Ruby structure containing all the weight and bias constants of its neurons. Each neuron of the layer and corresponding output is generated directly from 'ws_b'. Lines 5-8 generates the outputs, named respectively 'a0', 'a1' and so on. In this code, the ':"#{...}"' construct is used for generating names where the three-dot part is some Ruby code, and 'num.times' construct is for iterating. The access to these output signals is stored in a Ruby array 'as' for easier reference. The neurons are instantiated similarly lines 10-14.

**bnpipe**: it is the module that describes a pipelined BNN. Its inputs are the clock and reset signals, and the input vector. Its output is the vector holding the results of the last layer of neurons. Its code is given Fig. 6. lines 16-56. Its generic parameters are like bdense with 'bnn' in place of 'ws_b' for describing all the layers. Each layer, each pipeline register and each inner connection are generated directly from 'bnn'. Lines 23-31 and lines 32-36 generate the pipeline registers 'reg' and the inner connection signals 'pt' in a similar fashion the outputs of bdense were generated. For the registers, the last one is declared as output of the module. Lines 38-45 instantiate the layers like the neurons of bdense were, using variable 'cur_typ' for matching the layers input and outputs.

Finally lines 47-56 describe the process that handles the pipeline on each rising edge of the clock 'clk', with 'rst' as reset signal. In this process the 'regs.each' construct iterates on each register of array 'regs'.

**bnpipe_chip**: it the module that generates and wraps a BNN as well as the additional verification circuit for ensuring the BNN works properly after being mapped onto the target FPGA. It also includes a Ruby program for reading the offline training result description file (in json) and providing an easy access to its content. The code of these classes is counted as HDLRuby code.
4. Evaluation

4.1 Evaluation of the design effort

It took 21 hours for designing in HDLRuby the generic BNN circuit from scratch (i.e., without any predetermined architecture nor algorithm) down to the FPGA mapping. This time includes the tests of each module and of the whole BNN. The total number of lines of code (LOC), including the benchmarks, is 714.

The BNN has also been reimplemented directly in Verilog using the HDLRuby code as template. The re-implementations of modules bact, bmac and bneuron are near identical to the HDLRuby ones, with the difference that the generic parameters are all explicitly declared. By contrast it appeared more practical to implement one popcount module per LUT size. The bdense module has been implemented using a generate statement for instantiating a generic number of neurons. Fig. 7. gives a sample of this generation code. In the figure, IWIDTH, CWIDTH and OWIDTH are parameters giving respectively the input size, the number of neurons and the popcount result size, while VECWS and BIASES are the parameters containing the relevant weights and biases. While this code has some similarities with the HDLRuby one, it was necessary to write new code with different names for one bdense module as well one bneuron module per layer to avoid conflicts between their generated instances. This constraint may be due to some limitation of the used synthesis tool (Vivado). In addition, there were no native way to generate the constants holding the weights and biases (i.e., without using a memory construct). Therefore, they are declared as parameters whose values are written from the results of the training algorithm. Finally, since each layer has a different size and must be described in its own file, the global module bnn_pipe must be rewritten from scratch for each new BNN structure.

With this reimplementation, both versions of the code can be compared in term of number of lines of code (LOC) and their variations related to the structure of the BNN. For a comparison as fair as possible, the code in both HDLRuby and Verilog has been stripped of all the comments, and an identical convention for spacing and carriage returns has been used. For a BNN with a single inner layer with 8 neurons, the LOC including all the benchmarks is 714 for HDLRuby and 1107 for Verilog. When the benchmark code is removed, the LOC falls respectively to 372 and 591. The LOC of for any BNN structure is fixed for HDLRuby but varies for Verilog. Fig. 8. shows such variations: on the left a graph gives the LOC for a BNN depending on the number of neurons of the inner layers. The linear increase for the Verilog code is due to the number of parameter definition for the weights and biases. The graph on the right of the

![Graph](image_url)

Fig. 7. Generic Verilog code for instantiating neurons.

Fig. 8. Comparison of LOC for HDLRuby and Verilog descriptions of several BNN structures.
figure gives the LOC depending on the number of inner layers assuming they all include 8 neurons. For this second graph, the weight and bias parameters definitions of the Verilog version and the parameter generation code of the HDLRuby one have been removed for estimating the pure hardware code requirement. As seen in this second graph, additional code is required for each new layer for the Verilog version.

4.2 Hardware performance

The designed BNN have been mapped onto a KINTEX UltraSCALE FPGA using the Vivado toolset. Two optimization objectives have been used: default, and area optimized high. Speed optimization is also a possible objective, but for all the tested BNN architectures, the design was able to run at full clock speed so that such an optimization was not relevant. Due to the pipeline structure of the designed BNN and the fact that the FPGA could operate at full speed, the resulting circuits were able to process 100 million pictures per second.

Fig. 9. gives the area in LUT usage with default and area optimization objects for BNN of various structures with three configurations for the popcount component: without LUT-based sub sums, with 4-input LUT sub sums, and with 6-input LUT sub sums. Fig. 10. gives the power consumption for the same BNN architectures and optimization objectives. As it can be seen in the figures, the area and power consumption are approximately linear with the number of neurons. More surprising is the effect of the configuration of the popcount module: when maximum optimization is used, it appeared that 6-input LUT sub sum is better than no LUT which is again better than 4-input LUT sub sum. If we assume that the Vivado toolset did not change the structure of the modules and

![Fig. 9. Area of BNN with different popcount algorithms.](image1)

![Fig. 10. Power consumption of BNN with different popcount algorithms.](image2)
knowing that the target FPGA is mainly composed of 4-input and 6-input LUT, it would have been expected that both LUT sub sum popcounts are better than no LUT, but this is not the case. If we assume that the toolset does modify the structure of the modules, it would have been expected to have identical results for the three configurations. Such an outcome shows that even with recent synthesis tools, optimizations are still sub-optimal heuristics whose results are hard to predict.

As final exploration step, the best architecture for both accuracy and power can be identified by using a graph as the one given Fig. 11. For this graph, the best popcount (i.e., 6-input LUT) and the best synthesis optimization target (i.e., area optimized high) are selected for generating BNN circuits with 8 to 384 neurons in the inner layer. Their power consumption and accuracy are then plot in a graph. As seen in the graph, a good compromise may be for a power consumption of about 6W that corresponds to 256 neurons in the inner layer (6.22W).

5. Discussion

As shown in section 4.2, in term of pure number of lines of code for an BNN with a given structure HDLRuby require less lines of code than Verilog. However, the real advantage of HDLRuby comes when several BNN architectures must be explored, as it is often the case with real life designs. With the HDLRuby version of the BNN, any new BNN can be produced by just providing the relevant offline training result file. Also, controlling the implementation detail of the popcount module require just modifying one single parameter when compiling HW code. By contrast, the Verilog version require to rewrite some code for each new BNN structure and to provide the weights and bias in a Verilog format. Though, two arguments can be made about such a comparison. First, instead of Verilog, more advanced languages like System Verilog or SystemC could have been used. However, we did try to reimplement some parts in System Verilog, but without many differences in term of generative power compared to Verilog. Moreover, the support of System Verilog by design tools is still unclear so that it is difficult to know which construct can be used. By contrast the Verilog code obtained after compiling HDLRuby is basic RTL code supported by a wide range of tools. The other argument is that it is possible to write scripts for generating Verilog code and perform a design exploration as well as with HDLRuby. This is true, however, such an ad-hoc approach lacks standardization, whereas this can be natively done with HDLRuby. Another difficulty with Verilog was the debugging of the generate statements because of lack of information about what exactly is produced. With HDLRuby, the reflection paradigm and the integration of Ruby allows to display at compile time the state and structure of any element of the design.

The surprising variability on the design area and power consumption depending on description details is an important factor when deciding which environment (language) to use for designing a circuit: if a high-level design approach does not allow to easily describe low RTL-level variations, the result may become sub-optimal. On the other hand, a low-level language like Verilog that requires strong efforts to describe variations of implementations also limits the feasible range of exploration for optimal design. By contrast, HDLRuby allows an easier design of synthesizable RTL code with multiple variations through generic parameters that makes it possible to explore a wider range of unpredictable results to find an optimal solution.

With the current version of HDLRuby there remain a drawback. While the compiler is operational, it is still in development and its algorithms are not all optimized for speed. Consequently, a few of them require more than linear time to complete with consequence that largest designs sometimes have a compile time of hours (about two hours was required for 256 neurons in an inner layer with a 2.3GHz laptop computer). While compile time for the largest design was not an issue because the BNN have been fully tested with quick-to-compile small structures, we do plan to improve the compiler using dynamic programming.

6. Conclusions

In this paper we presented the design exploration of a BNN for FPGA using the HDLRuby language as well as an
evaluation of both the required coding effort and the efficiency of the resulting HW. The design was quick, and the resulting code is short and supports various BNN structures. By contrast, the Verilog reimplementation code size is dependent on the number of neurons and layers of the BNN even when using generate statements. But the best advantage of using HDLRuby was its tight integration with Ruby so that no effort was required for directly using the offline training results. After synthesis, the resulting HW was fast enough to process 100 million images per second from the MNIST dataset while the power consumption remains inferior to 6.5W for the best accuracy. Interestingly, the area and power consumption fluctuated in an unpredictable way depending on small implementation details. This encourages to go on with RTL description of circuits, while using a productivity-efficient description paradigms, like the ones proposed by HDLRuby, for easing the exploration of the various implementation details that may influence the result.

As future work, we plan to go on optimizing the compiler for reducing its processing time for large designs. We also consider replacing the training method used in this paper by a more accurate one like the approaches mentioned in the related works\cite{6,15-17}.

Acknowledgment

This research is founded by the Japanese Grants-in-Aid for Scientific Research (KAKENHI) No. 18K11284.

References

(2) Lovic Gauthier, and Yōhei Ishikawa : “HDLRuby, a new High Productivity Hardware Description Language”, Proc. of the 5th International Conference on Business and Industrial Research (ICBIR), pp. 215-220, 2018
(12) Gauthier Lovic, and Yōhei Ishikawa. "Evaluation of the HDLRuby Hardware Description Language by implementing an 8-bit RISC Processor", 2020
(15) Joshua Wolff. "Implementing Binary Neural Networks.", Thesis --University of Washington, 2020