Automatic Transpiler that Efficiently Converts Digital Circuits to a Neural Network Representation

Devdhar Patel
Manning College of Information and Computer Science
University of Massachusetts
Amherst, USA
devdharpatel@cs.umass.edu

Ignacio Gavier
Manning College of Information and Computer Science
University of Massachusetts
Amherst, USA
igavier@umass.edu

Joshua Russell
Manning College of Information and Computer Science
University of Massachusetts
Amherst, USA
jgrussell@umass.edu

Andrew Malinsky
Manning College of Information and Computer Science
University of Massachusetts
Amherst, USA
amalinsky@umass.edu

Edward Rietman
Manning College of Information and Computer Science
University of Massachusetts
Amherst, USA
erietman@umass.edu

Hava Siegelmann
Manning College of Information and Computer Science
University of Massachusetts
Amherst, USA
hava@umass.edu

2022 International Joint Conference on Neural Networks (IJCNN) | 978-1-7281-8671-9/22/$31.00 ©2022 IEEE | DOI: 10.1109/IJCNN55064.2022.9892305

Abstract*—Digital circuits are the basic structure of most of today's electronic devices. Simulation plays a critical role in the iterative development of such circuits as a consequence of the time and financial expenses that accompany fabrication. There are existing tools that achieve simulation by modeling circuits with Hardware Description Languages (HDL). However, with recent advances in neural networks (NN) and hardware accelerators for NN simulation, a niche of digital circuit simulation via NNs has opened up. Here, we introduce C2NN (Circuit to Neural Network), a novel method that converts (or *transpiles*) any digital circuit expressed in a HDL into a NN for simulation. The conversion to a NN representation not only affords the benefits of parallelization, the use of GPUs for simulation, and optimizations such as pruning, but it also provides a methodology for achieving equivalent digital circuit computation in neuromorphic hardware. We describe the transpilation process of C2NN and verify its correctness on small- and large-scale digital circuits. We also found that the simulation time of the transpiled circuits is competitive with one of the fastest digital circuit simulators.

***Index Terms*—Digital Circuits, HDL, Transpilers, Neural Network Representation, Neuromorphic Hardware**

I. Introduction

The conversion of digital circuits described at the level of HDLs to NNs for simulation has received limited attention [1], with existing work focusing on learning program execution with neural models [2], [3], optimization of program simulation through the use of compilation and learning [4], program acceleration with approximate functions [5], and constructing networks that implement simple digital circuits [6]. Our contribution is a transpiler that converts digital circuits expressed in a HDL into a NN representation, with behaviour equivalent to that of the original circuit.

NNs offer a universal computational substrate for the implementation of functions [7]. The artificial intelligence community has developed a spectrum of methods for implementing such functions in NNs, ranging from learning-based approaches that utilize data and optimization techniques [8] to algorithmic approaches that rely on known functional structure [9].

In recent years, advances in computation paired with an abundance of available data has favored the learning-end of the spectrum, resulting in significant developments in deep learning. Learning has proven useful for a wide-range of problems that deal with functions of unknown or complex structure, such as image classification, machine translation and playing video games. However, the collection of ever-larger datasets, longer training times and unsustainable energy usage are what we sacrifice for eliciting functions through learning [10], [11].

The utilization of functional structure has played an important role in the deep learning revolution through the introduction of various inductive biases [12]. This additional structure reduces the degree of the aforementioned drawbacks of learning by increasing sample efficiency and in turn reducing the number of training examples, time and energy required to learn a function of interest. Implementing functions from known computational structure alone has not been widely pursued since theoretical work on the Turing computability of NNs [13] motivated the idea of simulating machine computation in NNs. This inspired a few research projects throughout the 1990s and early 2000s which lead to a theoretical language [14] and neural compilers that were able to algorithmically construct neural networks that would simulate the execution of programs originally expressed in programming [15], [16] and high-level [17] languages.

In this work, we revisit the notion of using computational structure alone to construct NNs for simulating digital circuits. We propose C2NN, a transpiler that converts a circuit expressed in the CHISEL HDL into Python code specifying a NN under the PyTorch [18] framework. Our transpiler extends the work of [15], [16], [17] to HDLs and modern developments in NN software, differing from [6] in that we utilize constructed primitive networks to transpile arbitrary, small- and large-scale circuits, while also differing from much of the contemporary work in terms of NN architecture, our approach of using structure over learning [1], and in extracting such structure from a HDL as opposed to from programs written in terms of machine instructions [4]. The conversion of digital circuits to NNs allows for the utilization of parallel computation, modern tensor hardware, and further optimizations for digital circuit simulation. Moreover, the mapping from a functional representation native to stored-program computers into one that is akin to the representations native in neuromorphic computers provides a strong first step towards a methodology for expressing the vast developments in digital circuits in neuromorphic hardware. In the following sections of the paper we provide a background on digital circuit representation, outline the methodology used for transpiling circuits into NNs, detail optimizations afforded by an NN circuit representation, provide experimental results against one of the fastest SOTA digital circuit simulators, and finally discuss the implications of this work along with our future plans.

II. Background

A *Boolean Function* $f : \{0,1\}^n \rightarrow \{0,1\}$ takes n binary arguments and outputs a binary value. Due to the binary nature of Boolean Functions, the number of possible functions is finite for a given n. Considering n input values, there are 2^n possible input values, which can result in values of 0 or 1 in the output. Therefore, there are 2^{2^n} possible Boolean Functions for a given n.

Boolean Functions can be divided into two groups according to their condition of being linearly separable. We say that a Boolean Function f with n input variables $x_1, \dots, x_n$ is *Linearly Separable* if $\exists w_1, \dots, w_n, b \in \mathbb{R}$ such that $w_1x_1 + \cdots + w_nx_n \geq b \Leftrightarrow f(x_1, \dots, x_n) = 1$. This is the case of `and`, `or`, `not`, `majority`; while it is not the case of `xor`. Linearly Separable Boolean Functions are also known as *threshold functions* and have been widely studied in the field of discrete mathematics [19], [20], in the area of electronics [21] and in the area of machine learning [22], [23]. In the latter, it is useful to recognize the Linearly Separable Boolean Functions because they can be represented with a single-layer perceptron [24] —that is, one single neuron— with n inputs:

$$f(x_1, \dots, x_n) = \mathcal{H}\left(\sum_{i=1}^{n} w_i x_i - b\right), \qquad (1)$$

where $\mathcal{H}(\cdot)$, the Heaviside function, is the threshold function or non-linearity used in the perceptron.

As n grows, the cardinal of the set of Linearly Separable Boolean Functions becomes smaller and smaller compared to the cardinal of Boolean Functions. Therefore, finding an equivalent perceptron representing the Boolean Function becomes meaningless due to its high computational cost. However, it is known that any Boolean Function can be represented with a two-layer perceptron [25].

While there is no algorithm to find the equivalent two-layer perceptron with the *minimum* number of neurons in the hidden layer, a two-layer perceptron can be created by taking advantage of the postulate that any Boolean Function can be represented as an `or` of `and`s by using the truth table (often referred to as Sum of Products or SOP). Then, since `and` and `or` are Linearly Separable Boolean Functions, if each neuron in the hidden layer represents one of the `and`s (i.e., the neuron is activated when the Boolean Function is true), the resulting two-layer perceptron is equivalent to the Boolean Function. With this method, however, the size of the hidden layer grows in average exponentially with n. We take a different approach in this work by constructing modular NNs based on atomic Boolean Functions.

A. Digital circuits

A *Vectorial Boolean Function* is a function $F : \{0,1\}^n \rightarrow \{0,1\}^m$ that takes n binary arguments and outputs $m > 1$ binary values. Digital Circuits can be represented as Vectorial Boolean Functions when they have multiple output bits. In digital electronics, the digital circuits are designed with more complexity, generally including clocks and registers. Digital circuits guided by a clock signal are called *synchronous circuits*, while *asynchronous circuits* can be designed to follow transaction protocols with "ready" signals or simply without any communication protocol.

B. Hardware Simulation

Hardware description languages (HDLs) used to describe digital hardware at the register transfer level (RTL) typically means either Verilog or VHDL. In the last ten years, a new HDL called Constructing Hardware in a Scala Embedded Language (CHISEL) has emerged [26]. As the CHISEL language is based on Scala, it has inherited the object-oriented and functional program aspects of Scala which allow more abstract programming techniques to be applied to hardware design. CHISEL is not a language that directly translates to RTL but through compiling produces an intermediary representation called Flexible Internal Representation for RTL (FIRRTL) which may then be used to produce more traditional RTL, Verilog.

CHISEL also provides a test harness for simulating the behavior of RTL designs. Non-commercial tools that provide similar simulation capabilities for Verilog include Icarus Verilog and Verilator. All of these elaborate the HDL to a form that may be simulated. Both Icarus Verilog and Verilator create and compile an executable that is run at the command line. CHISEL produces FIRRTL which is simulated in the testing framework of Scala using the CHISEL testing harness. The selection of which simulation tool depends a lot on the size of the hardware design and the type of tests. For example, Verilator may take longer to elaborate the design than Icarus Verilog but the resulting simulation executable runs significantly faster. This difference between these two tools is expected as Verilator performs non-event driven optimizations during elaboration. This indicates that Icarus Verilog may be more appropriate for hardware descriptions that may be easily partitioned into smaller, independent testable modules while Verilator may be more appropriate for larger hardware descriptions meant to be tested as a whole system.

CHISEL has been proven to be one of the fastest among the SOTA frameworks for simulating digital electronic circuits being as fast as handwritten Verilog, and overperforming by dozens of times PyMTL, MyHDL, PyRTL and Migen [27]. It was found that CHISEL performed almost as well as hand-optimized Verilog [28] when addressing Database Management Systems acceleration, and faster than other High Level Synthesis languages, such as Altera OpenCL or LegUp.

C. Abstract Syntax Trees (AST)

ASTs are structures widely used in compilers to represent the structure of program code [29]. They often serve as an intermediate representation of the code to analyze and help correct errors.

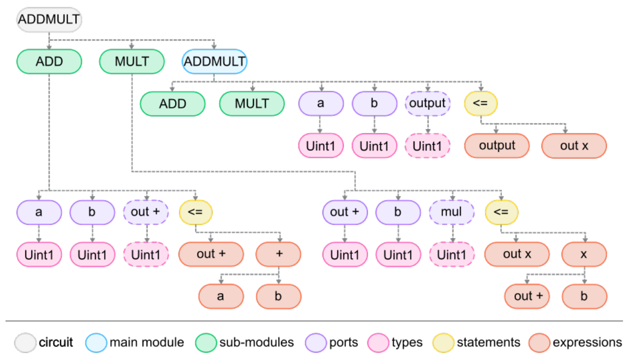


Fig. 1. AST example of the function $f(a,b) = (a+b)b$ or `addmult`. The circuit and main module share the name, while the sub-modules `add` and `mult` are instantiated in the main module. Dashed lines indicate the elements are outputs.

The underlying internal data structure of a FIRRTL is an AST. This type of hierarchical structure is a tree of nodes, where a node can contain child nodes. Fig. 1 illustrates this structure for a circuit that computes $f(a,b) = (a+b)b$, which we call `addmult`. The root node for any FIRRTL tree is the Circuit. Only one Circuit node is defined, and it contains at least one Module, the main-module, which shares its name with the Circuit, `addmult`. In our case, since the circuit includes an addition and a multiplication, they can be computed by other modules, `add` and `mult`. In FIRRTL ASTs, there is always only one main module, so we refer to other non-main modules as sub-modules. The sub-modules `add` and `mult` are instantiated in the main-module and are used for operations in it. Not shown in this example, but sub-modules can be instantiated in other sub-modules, not only in the main-module. Each Module contains Ports, which are its inputs and outputs. For example, there are two inputs to `mult`, one is $out+$ (also the output of `add` module) and the other one is b. The connections between ports of different modules and its operations are stated in the body of each module that instantiates them. These are the Statements of the modules, which describe all the operations and connections taking place within that Module. In the example, only one statement of the type *connection* ($<=$) is defined in each module, but other different statements (like *conditional*, *register*, etc.) can be defined, even more than one, and the order *does* matter.

III. Methodology

To convert representations of digital circuits written in FIRRTL, we implemented C2NN with two main stages in which different transpilers are in charge of interpreting and generating files, reaching a final representation in NN written in Python using the PyTorch library. This pipeline is summarized in Fig. 2. In the following subsections we describe each of the instances that compose the algorithm.

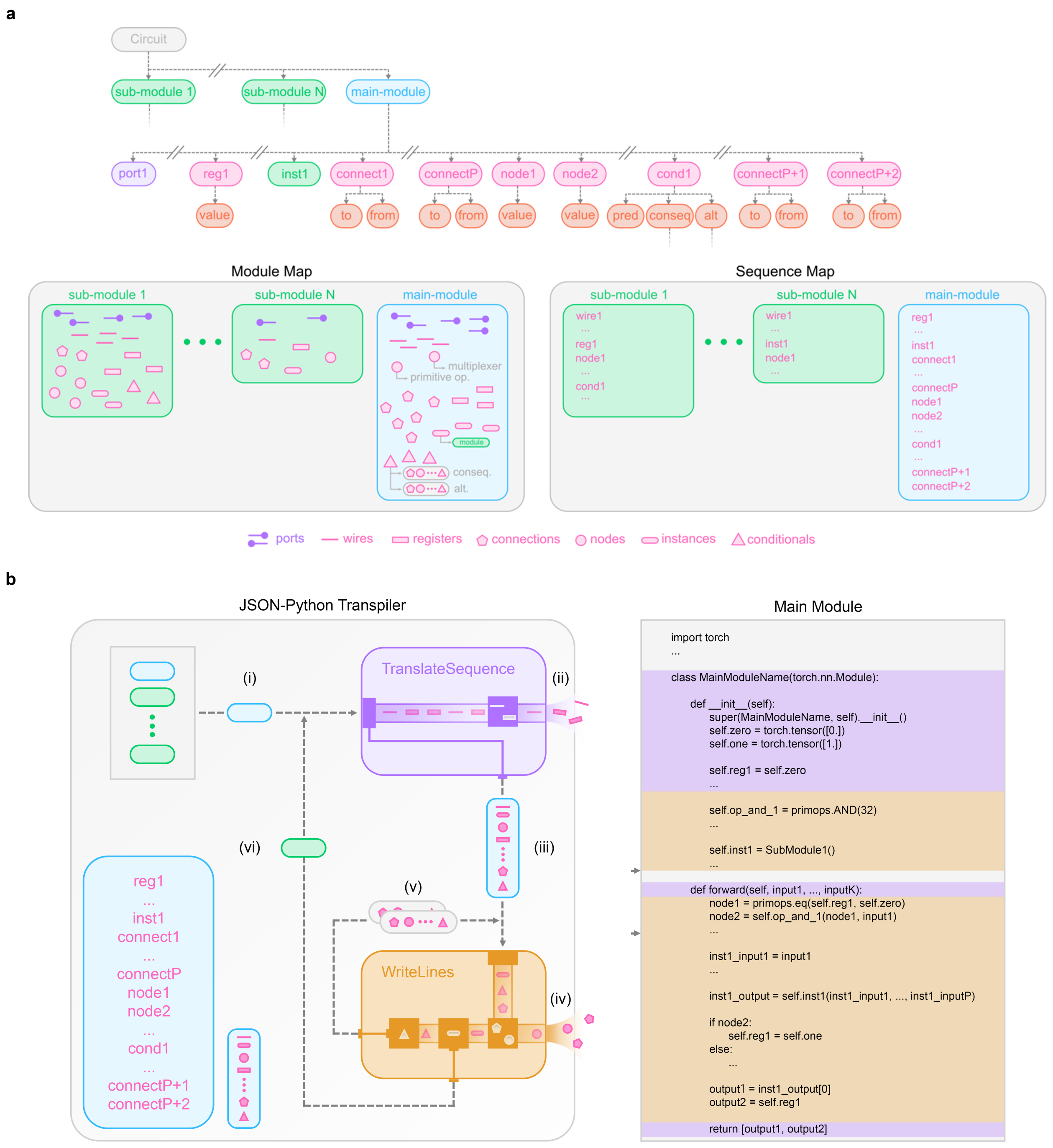
A. Scala-JSON Transpiler

The first stage of converting digital circuits to NN is performed by the Scala-JSON transpiler. The functionality of this stage is to convert a sequential representation of a digital circuit to an unambiguous and interpretable computational structure, so it can then be converted to a NN structure.

The combination of inputs, outputs, operations, and connections help define a component's overall behavior. In order to extract this information, the FIRRTL AST can be traversed recursively by visiting each node. The Scala-JSON transpiler takes in a FIRRTL file, interprets the AST of the given circuit, and constructs a simplified mapping of the circuit. There are two main structures that are generated by the Scala-JSON transpiler, the Module Map and the Sequence Map. Here we briefly explain the purpose and execution of each, making references to Panel (a) of Fig. 2.

The Module Map contains in the description of each module all its components that are declared in the AST. On the one hand, all the ports, separated into inputs and outputs; on the other hand, the conditionals, connections, multiplexers, nodes, logical operators, prints, registers, wires and instances of other modules. In the Module Map, the order in which the components are described *does not* matter, as it mainly serves as a reference in relation to the Sequence Map. The Sequence Map contains the ordered list of operations performed in each module to preserve the logical and sequential relationship of the variables used in each of them, as specified in the original FIRRTL file. These two maps together unambiguously represent the AST, and thus the circuit.

Finally, the Scala-JSON transpiler converts the combination of the Module Map and Sequence Map into JSON format. JSON is a data format that stores an object of key-value pairs. This type of data structure is highly interchangeable across programming languages, which makes it easier to transfer and read the circuit information from Scala to Python. The JSON is then used by the JSON-Python transpiler to create the NN.



```
import torch
...
class MainModuleName(torch.nn.Module):

    def __init__(self):
        super(MainModuleName, self).__init__()
        self.zero = torch.tensor([0.])
        self.one = torch.tensor([1.])

        self.reg1 = self.zero
        ...

        self.op_and_1 = primops.AND(32)
        ...

        self.inst1 = SubModule1()
        ...

    def forward(self, input1, ..., inputK):
        node1 = primops.eq(self.reg1, self.zero)
        node2 = self.op_and_1(node1, input1)
        ...

        inst1_input1 = input1
        ...

        inst1_output = self.inst1(inst1_input1, ..., inst1_inputP)

        if node2:
            self.reg1 = self.one
        else:
            ...

        output1 = inst1_output[0]
        output2 = self.reg1

        return [output1, output2]
```

Fig. 2. General architecture of C2NN. (a) Scala-JSON transpiler: converts the AST generated by CHISEL in the FIRRTL file into a JSON structure containing the Module Map and the Sequence Map. (b) JSON-Python transpiler: converts the JSON structure using TranslateSequence and WriteLines into Python code.

B. JSON-Python Transpiler

The JSON-Python transpiler is the second and final stage of the conversion from digital circuit to NN, taking the JSON representation output by the Scala-JSON transpiler and producing Python code specifying a NN under the PyTorch framework. Here we briefly explain its execution, making references to Panel (b) of Fig. 2.

Two key functions are responsible for turning the JSON representation into Python NN code, TranslateSequence and WriteLines. TranslateSequence is called with a module identifier (i) (starting with the main module's) and writes the module's class declaration, initialization method (with class variables for wire and register components), and forward method definition (with module input arguments) (ii); referencing the module map and sequence map for required details.

WriteLines is called by TranslateSequence after the above class structure has been written, and is passed the ordered list of statements within the module's sequence map to iterate over (iii). Connection components are written as assignments and node components are written as function calls with assignments inside the forward method (iv). A pointer is kept at the top of the forward method for "repeat nodes"; nodes with results that are reused by other nodes. Each node implements either a primitive operation or multiplexer operation, and therefore the class of the operation being performed is also written as an instantiation inside the class's initialization method. Conditional components result in recursive calls to WriteLines, with *consequence*, and subsequently *alternative*, component lists of the conditional being passed as input (v). Instance components result in a recursive call to TranslateSequence, if they have not already been compiled, with the instance's module identifier being passed as input (vi). Once the transpilation of an instance's module is complete, it is saved to a separate file of transpiled networks and added to a library such that it does not need to be transpiled again. Instances are written inside the forward method of the current module as function calls to the forward method of the instance's module class, with an assignment for the output. The instance's module class is also written as an instantiation inside the current module's class initialization method.

Once WriteLines has completed iterating over all of the module's components, execution of the transpiler will return to the TranslateSequence method, which writes the return statement of the class forward method to complete the transpilation of the module's NN Python class. If the module is the main module of the circuit, transpilation of the circuit will be complete. The Python class of the main module is saved to a unique file, and its forward method can be called with a circuit state to simulate the circuit's computation using a NN.

C. Primitive Operations

In the previous sections we have described how the digital circuit representation is converted into a neural network architecture through the division of modules and components. However, we have not specified how C2NN converts the primitive operations (`add`, `sub`, `and`, `or`, etc.) to be compatible in a neural network. Primitive operations are important in defining the internal behavior between components of a circuit.

In the FIRRTL language, these operations are defined and then used in a digital circuit simulator to compute the relationship between the components. In a NN, however, they must be predefined, either by the PyTorch library or by the transpiler. It is necessary to ensure that when transpiling a given operation, its behavior in the NN matches its behavior in a digital circuit. For example, the operation $a = \texttt{add}([1],[1])$ in the digital circuit results in $a = [10]$, but in a NN, using the predefined addition operation in PyTorch, $a = [2]$. And in some cases, primitive operations used in FIRRTL have not been explicitly defined within the PyTorch framework. Therefore, predefining all the functions that are used within FIRRTL is necessary to achieve an exact match of the behavior of the digital circuit with the NN.

Of all the primitive operations defined in FIRRTL, we implemented the most common as neural networks with the help of built-in functions of Python and PyTorch. The `add` operation (and similarly the `sub` operation) was implemented following the truth table of a full-adder for each bit (from the least significant to the most significant). The number of series operations increases linearly with the bit-width since the adder requires the carry bit to be passed from one bit to another. However, we parallelized the neural networks to replicate this behavior in a constant number of layers for any arbitrary bit-width. The bottleneck we solved is that each full-adder required to provide a carry term to the next bit. For example, for a 2-bit adder:

$$\begin{aligned} c_3 = \quad & a_2 \cdot b_2 + (a_2 \oplus b_2) \cdot \Big(a_1 \cdot b_1 + (a_1 \oplus b_1) \cdot \\ & \cdot \big(a_0 \cdot b_0 + (a_0 \oplus b_0) \cdot c_{\text{in}} \big) \Big) \end{aligned} \tag{2}$$

where $\cdot$ represents `and`, $\oplus$ represents `xor` and $+$ represents `or`. c_0, c_1 and c_2 can also be calculated using a similar method. Furthermore, this result can be extended to any arbitrary number of bits. Since `xor` can be calculated using 2-layer neural network and `and` can be calculated in a single neuron, we can compute all the carry bits in 4 layers (independent of the bit-width). The output of the adder can then be calculated in 2 more layers. Thus, we can calculate any arbitrary bit-width networks in 6 layers.

D. Optimizations

Sections III-A and III-B, together with III-C describe the basic operation of C2NN, that converts a digital circuit described in FIRRTL to a NN written in Python using PyTorch. These parts ensure that the outputs of the NN exactly match the outputs of the digital circuit. However, the NN generated in Python can be exploited in different ways to make computing the outputs more efficient.

First, many of the components generated in the digital circuit representation are intermediate variables that are used to store information that may or may not be relevant to computing the final outputs of the circuit. This is the case of *wires* in the FIRRTL files, which are generated by default for all the registers. For the transpiled NNs, some of these wire assignments do not contribute to computing the final output and take up additional time in the forward pass. To avoid wasteful computation, our JSON-Python transpiler trims all the wire components in the forward method of transpiled circuits that do not contribute to computing the final output.

PyTorch uses the `autograd` library which computes the gradients in each call of the forward method of the NN. This additional computation can also be avoided since the networks that we use to represent the circuits are completely feedforward NNs that do not require computing gradients for parameter updates.

IV. Results

With the methodology described in the previous section, we can take any digital circuit that is written in FIRRTL and convert it to a NN. In this section, we provide the experimental results of digital circuit simulation using NNs. Simulating the computation of digital circuits involves performing a function of the circuit state at each clock cycle. Thus, we evaluated the transpiled NNs and CHISEL simulator on sequences of circuit states, ensuring correct behaviour and measuring simulation time. For all the circuits, the generation of the corresponding NN was less than a minute.

A. Simplified Arithmetic Logic Unit (ALU)

As a first instance for testing C2NN, we chose an Arithmetic Logic Unit (ALU). An ALU is the fundamental block of the majority of computing circuits (like CPUs or GPUs). It is able to perform different operations, like addition, subtraction, multiplication, division, and, or, etc. This circuit receives two binary inputs of the same bit width and a binary signal that allows to select the operation to be performed. The result is a binary vector of the same bit width as the input vectors. We tested a simplified version of an ALU, which is able to perform only the operations of addition and subtraction. We found that the outputs of the NN matched exactly the outputs of the CHISEL simulator. The simulation times are summarized in Table I.

TABLE I
SIMULATION SPEED COMPARISON BETWEEN THE CHISEL SIMULATOR AND TRANSPILED NN FOR THE TWO ALU OPERATIONS `add` AND `sub`. WHERE w IS THE BIT WIDTH OF THE ARGUMENTS AND SPEEDUP IS THE RATIO BETWEEN THE CHISEL AND NN SIMULATION TIMES (I.E., $\frac{CHISEL}{NN}$). THE ALU OPERATIONS WERE BENCHMARKED ON 100 EXAMPLES ON A CPU.

Circuit		**Simulation Speed (ms)** CHISEL	NN	Speedup
ALU `add`	$w = 4$	73.4	33.243	2.2x
	$w = 8$	85.9	32.854	2.6x
	$w = 16$	136.5	32.469	4.2x
	$w = 32$	251.4	34.387	7.3
	$w = 64$	498.2	33.348	14.9x
ALU `sub`	$w = 4$	54.3	37.135	1.4x
	$w = 8$	71.8	37.019	1.9x
	$w = 16$	102.5	37.175	2.8x
	$w = 32$	161.8	37.755	4.3x
	$w = 64$	349.2	38.311	9.1x

An important aspect of these results is that the time it takes for the CHISEL simulator to run grows exponentially with bit width w, while for NN, it is approximately constant, as can be seen in Fig. 3. This is due to exponential increase of the number of components simulated in CHISEL as a function of w. CHISEL (like most HDL simulators) computes circuits at the logic gate level, while with NN these operations are computed in parallel, yielding a significant advantage for large w.

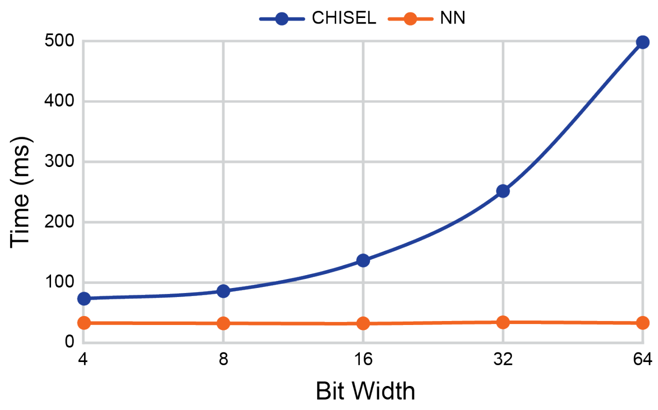


Fig. 3. ALU `add` operation simulation speeds over various bit widths for the CHISEL simulator and NN executed on a CPU. In the CHISEL simulation, the number of series operations increases linearly with bit width, unlike the NN implementation which remains fixed in depth and increases layer width for increased bit widths. Benchmarked on 100 examples.

B. Specialized Circuits

Going further with the simulation of digital circuits with C2NN, we chose circuits that are made up of a significantly greater number of components than an ALU, and are designed to perform more specialized operations. Digital circuits are often designed to interface with different types of integrated circuits, like monitoring a CPU. In such cases, the digital circuits are designed *ad hoc* so that they can perform the desired tasks. This is the case for a family of circuits designed to interface and monitor RISC-V [30], an open standard Instruction Set Architecture that has gained popularity in recent years. The circuits that we analyze in this work are an exporter and a pipeline, which are described below. Both can be found in our public repository[1].

1) Exporter: The Exporter is the module that interfaces directly with the RISC-V core. It is used to take monitored signals in a serial way (one per clock cycle) from the core. Those signals are then split in parallel signals. Therefore, the Exporter works as a clock frequency reducer while sending the signals to the next module. The number of serial signals that are serially read and split is a power of two, 2^n, where n is a predefined natural number. When the number of read signals reaches 2^n, a ready signal is triggered indicating that the output is ready.

2) Pipeline: The Pipeline is responsible for monitoring specific patterns in the signals received from the previous module. The patterns are looked for in the metadata of the signals, which consists of four instructions. The Pipeline takes as a valid pattern those where instructions appear in an increasing order, and generates a *valid* signal if that is the case. The Pipeline also generates a warning signal if the instructions fall outside a predefined range.

The Pipeline is divided into four stages: stabilize, interest, compare, fault report. The stabilize stage connects to the monitoring points received from the previous module. The

[1]https://github.com/ignaciogavier/C2NN_circuits

stabilize stage is primarily to make sure all of the monitoring points are captured and passed to the next stage as a coherent state in a single clock cycle. The interest stage sifts through the states from the main RISC-V pipeline to determine if the pattern of states is "of interest" as explained above, and then passes the relevant information to the next stage. The compare stage serves to make additional decisions and manipulations with the information extracted from the interest stage. The report stage captures the faults and sends that information out to the next module.

TABLE II
SIMULATION SPEED COMPARISON BETWEEN THE CHISEL SIMULATOR AND TRANSPILED NN FOR EXPORTER AND PIPELINE. WHERE n IS THE EXPONENT OF THE NUMBER OF MONITORED SIGNALS, *Test*'S REPRESENT VARIOUS TEST SEQUENCES OF CIRCUITS STATES, AND SPEEDUP IS THE RATIO BETWEEN THE CHISEL AND NN SIMULATION TIMES (I.E., $\frac{CHISEL}{NN}$). EXPORTER WAS BENCHMARKED ON TESTS CONSISTING OF 27 CIRCUIT STATES AND PIPELINE WAS BENCHMARKED ON TESTS CONSISTING OF 472, 471, 467, AND 470 CIRCUIT STATES.

Circuit		**Simulation Speed (ms)** CHISEL	NN	Speedup
Exporter	$n = 1$	46.279	33.623	1.4x
	$n = 2$	103.191	36.548	2.9x
	$n = 3$	92.888	37.282	2.5x
Pipeline	*Test 1*	1278.078	1237.941	1.0x
	Test 2	1409.153	1242.674	1.1x
	Test 3	1576.842	1224.373	1.3x
	Test 4	1269.558	1239.686	1.0x

The outputs of the transpiled NNs for Exporter and Pipeline matched exactly with the outputs obtained in the CHISEL simulator for hundreds of inputs. We also found that the simulation time of NNs is comparable to the CHISEL simulator, as indicated in Table II. These results demonstrate that NN simulation of digital circuits is possible, and it can achieve simulation speeds of the same order of magnitude than SOTA digital circuit simulators without the loss of operation accuracy.

V. CONCLUSION

We introduced a novel method for the conversion of any digital circuit expressed in the CHISEL HDL into Python code specifying a NN under the PyTorch framework, verifying correctness for small- and large-scale circuits. In addition, we found that the simulation time of such circuits is competitive with one of the fastest digital circuit simulators CHISEL. We plan to complete the implementation of the less common primitive operations of the FIRRTL language as NNs for completeness, and use the C2NN transpiler to simulate larger, composite and more complex digital circuits. We believe that there are many optimizations to explore given the NN representation of computation that can be incorporated into the compiler. Finally, we will consider other, first-principles-based approaches to transpilation that treat digital circuits as Boolean functions.

REFERENCES

[1] Z. Chen, M. Raginsky, and E. Rosenbaum, "Verilog-a compatible recurrent neural network model for transient circuit simulation," in *2017 IEEE 26th Conference on Electrical Performance of Electronic Packaging and Systems (EPEPS)*. IEEE, 2017, pp. 1–3.

[2] A. Graves, G. Wayne, and I. Danihelka, "Neural turing machines," *arXiv preprint arXiv:1410.5401*, 2014.

[3] S. Reed and N. De Freitas, "Neural programmer-interpreters," *arXiv preprint arXiv:1511.06279*, 2015.

[4] R. R. Bunel, A. Desmaison, P. K. Mudigonda, P. Kohli, and P. Torr, "Adaptive neural compilation," *Advances in Neural Information Processing Systems*, vol. 29, pp. 1444–1452, 2016.

[5] H. Esmaeilzadeh, A. Sampson, L. Ceze, and D. Burger, "Neural acceleration for general-purpose approximate programs," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2012, pp. 449–460.

[6] D. Sharma, "Neural network simulation of digital circuits," *International Journal of Computer Applications*, vol. 79, no. 6, 2013.

[7] K. Hornik, M. Stinchcombe, and H. White, "Multilayer feedforward networks are universal approximators," *Neural networks*, vol. 2, no. 5, pp. 359–366, 1989.

[8] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[9] C. W. Omlin and C. L. Giles, "Constructing deterministic finite-state automata in recurrent neural networks," *Journal of the ACM (JACM)*, vol. 43, no. 6, pp. 937–972, 1996.

[10] E. Strubell, A. Ganesh, and A. McCallum, "Energy and policy considerations for deep learning in nlp," *arXiv preprint arXiv:1906.02243*, 2019.

[11] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark *et al.*, "Learning transferable visual models from natural language supervision," *arXiv preprint arXiv:2103.00020*, 2021.

[12] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner *et al.*, "Relational inductive biases, deep learning, and graph networks," *arXiv preprint arXiv:1806.01261*, 2018.

[13] H. T. Siegelmann and E. D. Sontag, "Turing computability with neural nets," *Applied Mathematics Letters*, vol. 4, no. 6, pp. 77–80, 1991.

[14] H. T. Siegelmann, "Neural programming language," in *AAAI*, 1994, pp. 877–882.

[15] F. Gruau, J.-Y. Ratajszczak, and G. Wiber, "A neural compiler," *Theoretical Computer Science*, vol. 141, no. 1-2, pp. 1–52, 1995.

[16] J. P. Neto, H. T. Siegelmann, and J. F. Costa, "Symbolic processing in neural networks," *Journal of the Brazilian Computer Society*, vol. 8, pp. 58–70, 2003.

[17] P. J. Carreira, M. A. Rosa, J. P. Neto, and J. F. Costa, "Building a neural computer," 1998.

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *Advances in neural information processing systems*, vol. 32, pp. 8026–8037, 2019.

[19] S. Muroga, T. Tsuboi, and C. R. Baugh, "Enumeration of threshold functions of eight variables," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 818–825, 1970.

[20] N. Gruzling, "Linear separability of the vertices of an n-dimensional hypercube," Ph.D. dissertation, University of Northern British Columbia, 2007.

[21] M. H. Hassoun *et al.*, *Fundamentals of artificial neural networks*. MIT press, 1995.

[22] C. Mingard, J. Skalse, G. Valle-Pérez, D. Martínez-Rubio, V. Mikulik, and A. A. Louis, "Neural networks are a priori biased towards boolean functions with low entropy," *arXiv preprint arXiv:1909.11522*, 2019.

[23] C. Cortes and V. Vapnik, "Support-vector networks," *Machine learning*, vol. 20, no. 3, pp. 273–297, 1995.

[24] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review*, vol. 65, no. 6, p. 386, 1958.

[25] I. Parberry, M. R. Garey, and A. Meyer, *Circuit complexity and neural networks*. MIT press, 1994.

[26] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, “Chisel: constructing hardware in a scala embedded language,” in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1212–1221.

[27] S. Jiang, B. Ilbeyi, and C. Batten, “Mamba: closing the performance gap in productive hardware development frameworks,” in *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 2018, pp. 1–6.

[28] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Luján, “An empirical evaluation of high-level synthesis languages and tools for database acceleration,” in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2014, pp. 1–8.

[29] G. Fischer, J. Lusiardi, and J. W. von Gudenberg, “Abstract syntax trees-and their role in model driven software development,” in *International Conference on Software Engineering Advances (ICSEA 2007)*. IEEE, 2007, pp. 38–38.

[30] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, “The risc-v instruction set manual, volume i: Base user-level isa,” *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, vol. 116, 2011.