2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3503-3766-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPS54959.2023.00067

Neural Network Compiler for Parallel High-Throughput Simulation of Digital Circuits

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

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

***Abstract*—Register Transfer Level (RTL) simulation and verification of Digital Circuits are extremely important and costly tasks in the Integrated Circuits industry. While some simulators have incorporated the exploitation of parallelism in the structure of Digital Circuits to run on multi-core CPUs, the maximum throughput they achieve quickly reaches a plateau, as described by Amdahl's Law. Recent research from Nvidia has obtained much higher throughput in simulations using GPUs, highlighting the potential of these devices for Digital Circuit simulation. However, they were required to incorporate sophisticated algorithms to support GPU simulation. In addition, the unbalanced structure of real-life Digital Circuits provides difficulties for processing on multi-threaded devices. In this paper, we present a Digital Circuit compiler that utilizes Neural Networks to exploit the various parallelisms in RTL simulation, making use of PyTorch, a widely-used Neural Network framework that facilitate their simulation on GPUs. By using properties of Boolean Functions, we developed a novel algorithm that converts any Digital Circuit into a Neural Network, and optimization techniques that help in pushing the thread computational capability to the limit. The results show three orders of magnitude higher throughput than Verilator RTL simulator, an improvement of one order of magnitude compared to the state-of-the-art GPU techniques from Nvidia. We believe that the use of Neural Networks not only provides a significant improvement in simulation and verification tasks in the Integrated Circuits industry, but also opens a line of research for simulators at the logic and physical gate level.**

***Index Terms*—Integrated Circuits, Design and Verification of Digital Circuits, RTL Synthesis and Simulation, Neural Network Representation, Parallel Computation, GPU**

We would like to thank Frank Vetesi from Lockheed Martin for feedback and helpful discussions, and Thomas Buckley from University of Massachusetts Amherst for contributions in the experiments. This work was supported by Defense Advanced Research Projects Agency Grant, DARPA/MTO HR0011-21-9-0049.

I. Introduction

Since the dawn of Computer Science in the mid-20th century, Digital Circuits (DCs) have been at the heart of research in theory of computation and complexity [1]. Their practical significance has led to the massive industrialization of Integrated Circuits (ICs) [2], electronic elements composed of a large number of transistors that implement the logic gates of a DC. Present in all digital electronic devices that perform computation, ICs have become a cornerstone of contemporary society.

Simulation and verification at the Register Transfer Level (RTL) is a significant part of the IC development cycle, potentially taking days or weeks for extremely large circuits [3]. There is a vast variety of commercial and open-source software developments that handle these tasks [4]. In general, the process consists of reading and parsing the circuit written in a Hardware Description Language (HDL), constructing a netlist of the components and their connections, and transforming it into a language appropriate for the hardware where the simulation and verification will take place. Although these tools have improved over time, they have problems carrying out the simulation and verification of large ICs because they lack techniques and algorithms to efficiently exploit the various natural parallelisms underlying these tasks.

In recent years, however, there has been an effort to exploit *structural parallelism* (across logic gates) by using multi-thread computation. Such is the case of Verilator [5], an open-source software that is widely used in industry and academia. Verilator uses the netlist of the DC to convert it from Verilog, a HDL, into the C++ programming language. This conversion allows simulations to be run in a multi-threaded fashion, making use of modern multi-core CPUs for improved

1530-2075/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPS54959.2023.00067

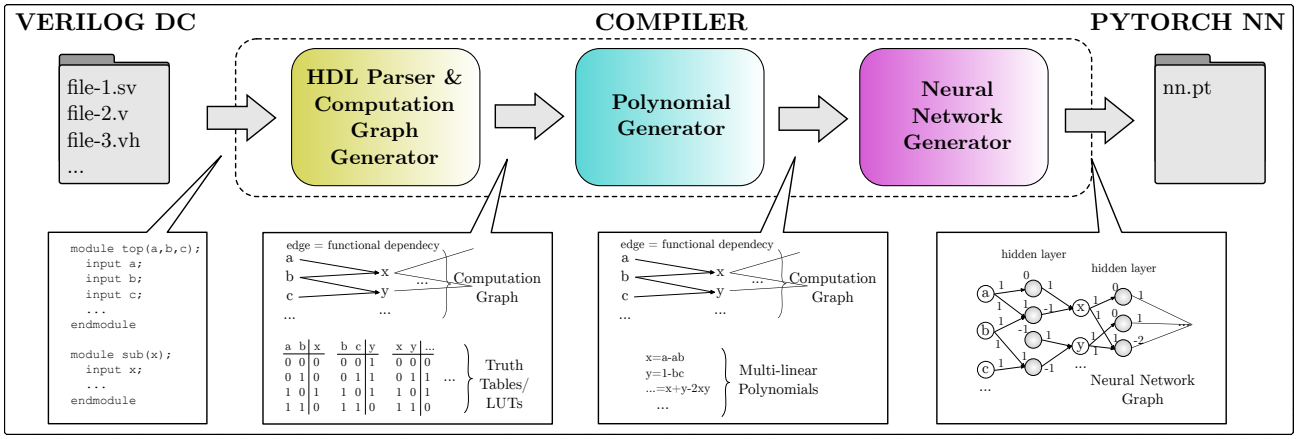

Fig. 1. Modules of the compiler for Digital Circuits. The first module reads a set of Verilog files and converts them into a Computation Graph with truth tables; the second module converts the truth tables into a multi-linear polynomial representation; and the third module converts the Computation Graph and the polynomials into a Neural Network representation. The Neural Network is stored using the PyTorch library.

simulation speeds. Although there are other simulators that perform faster than the single-thread Verilator, when it comes to large ICs it is one of the fastest for it uses multi-threading to exploit structural parallelism. Other simulators, such as ESSENT [6], benefit from the sparsity of events happening in a DC to skip unnecessary computations (event-driven simulation). Researchers have also explored exploiting the *stimulus parallelism* (across clock cycles) using GPUs [7] [8] [9] [10]. This approach has shown very promising results but solutions need sophisticated algorithms that allow efficient CPU-GPU data transfer, GPU load balancing, and GPU task scheduling.

In this paper, we propose a novel Neural Network (NN) based method for RTL simulation and verification of DCs that exploits both structural and stimulus parallelism. NNs are artificial intelligence models that utilize many types of parallel operations, making their simulation efficient on hardware specialized for these operations, such as GPUs and TPUs. It is expected that the simulation speed of NNs will continue to improve not only with these devices but also with the advancement of neuromorphic hardware [11]. In addition, the Machine Learning software industry has developed several open-source frameworks (PyTorch [12], TensorFlow [13], DGL.ai [14], etc.) that considerably facilitate simulation on GPUs, allowing any person with basic programming skills to be able to run a NN efficiently.

Up until now, NNs have been used to approximately simulate DCs using black-box training, where computational accuracy is sacrificed for simulation speed [15] [16]. NNs have also been used for exact computation by transpiling (source to source) the structure of the DC's Abstract Syntax Tree into a Pytorch NN [17]. However, the solution fails in optimizing the structures and modules, and hence did not lead to significant speed-up. In this work we adopt the use of optimal NNs for exact computation, based on the IC netlist, without sacrificing any accuracy, and in such a way that speeds up simulation by several orders of magnitude. Thus, our compiler is the first to take advantage of the benefits of a constantly updated and optimized GPU simulation library, PyTorch, for efficient RTL simulation using a computationally equivalent NN.

The main contributions of this work are:

1) Open-source compiler for Digital Circuits written in Verilog for RTL simulation on multi-thread devices using PyTorch Neural Networks[1].
2) Algorithm to find the equivalent two-layer Neural Network of any Boolean Function.
3) Optimization techniques of the Neural Networks to accelerate the RTL simulation on GPUs.
4) Exploitation of sparsity features in the Neural Networks to handle large Digital Circuits.

In Section II, we review HDLs and RTL simulation techniques, as well as theoretical concepts of DCs and their representation. In Section III, we detail the idea of using NNs to represent DCs and their suitability for simulation using parallel computing. We also describe the automatic compiler developed in this work and the optimizations implemented in order to handle large and complex DCs. In Section IV, we report experimental results of our compiler and compare its performance with the Verilator simulator and state-of-the-art techniques. Finally, in Section V, we summarize the contributions and propose possible improvements of our compiler.

II. Background

A. HDLs and RTL simulation techniques

Hardware Description Languages (HDLs) have been developed to encode the structure of a circuit into a computer-readable file. Nowadays there are several HDLs, although historically VHDL or Verilog have always been used. The former is the oldest, but Verilog has gained ground in recent decades and has become the most popular. In recent years, HDLs based on higher level programming languages have

[1]https://github.com/ignaciogavier/C2NN

also become popular, supporting functionalities such as object-oriented and functional programming. Such is the case of Chisel [18], based on Scala, and PyRTL [19], based on Python. Although these HDLs are easier to use for circuit creation, as they are easier for a human to understand, they end up sacrificing simulation speed because the DC presents sub-optimal solutions when converted to a lower level language for simulation [20]. This is why Verilog and VHDL, being lower level HDLs, achieve simulation speeds orders of magnitude better than any higher level HDL. Consequently, Verilog and VHDL are still the most widely used in industry and academia. In this work, we compile ICs written in Verilog, but this should not be viewed as a constraint since our pipeline has a modular structure, and it suffices to replace the HDL parser module to adapt it to an alternative language.

Verilog has had several standards throughout its history, incorporating more and more functionality. With these updates, HDL readers and parsers have had to adapt to new additions in the standards. Verilator is one of the open-source traditional simulators that is compatible with all versions of Verilog. It is also one of the simulators that has exploited the structural parallelism of DCs for simulation on multi-core devices. Verilator separates the circuit design into tasks and generates task pipes to be executed in different threads, with possible interactions between them. This has allowed Verilator to simulate large circuits on 24-core machines with speed-ups of up to $\times 10$ compared to its single-threaded implementation [21]. This speed-up cannot be further increased due to Amdahl's Law [22], where there is a bottleneck generated by inter-thread communication. In general, any simulator that attempts to parallelize for multi-core CPUs will suffer from the same problem due to accessing and writing to memory. Moreover, in traditional RTL simulation, the different verification benchmarks for ICs have to be processed one after the other, and there is no commercial simulator that exploits stimulus parallelism. Consequently, the verification time ends up being proportional to the number of benchmarks to be tested.

In the last decade, there have been several attempts at increasing RTL simulation speeds by improving parallelization techniques. For example, in 2011 Qian & Deng [23] obtained a speed-up of up to $\times 50$ compared to the commercial single-threaded simulator, ModelSim SE. The authors developed a compiler that converts Verilog source code into code interpretable by a GPU. This idea was taken up ten years later to develop RTLFlow [7], research done in collaboration with Nvidia, which transpiles Verilog code into CUDA code for simulation on Nvidia GPUs. RTLFlow obtained a speed-up of up to $\times 40$ compared to Verilator. Another recent work published by Nvidia [3] has shown opportunities to accelerate RTL simulation on parallel processing hardware platforms such as GPUs. The paper demonstrated how Machine Learning libraries such as PyTorch or DGL.ai, specialized in GPU processing, can be used to obtain significant speed-ups with respect to Verilator. Its demonstration serves as support for the implementation of the compiler developed in this work. The novelty of our approach lies in it being the first to use optimized and exact NNs within the PyTorch framework for RTL simulation on GPUs. Thus, our compiler benefits from all the optimizations PyTorch provides for efficient computation on multi-threaded devices.

B. Boolean Functions and Digital Circuits

In this work, we have developed a DC to NN compiler that is based on different theoretical concepts of Boolean Functions (BFs) that are necessary to understand the compilation process.

A BF is a function of type $f : \{0,1\}^n \rightarrow \{0,1\}$, where n binary input elements are mapped to a binary output element. Due to the binary nature of the Boolean domain, the number of possible distinct BFs is finite for a given n. This number can be calculated by considering that there are 2^n possible inputs and 2 possible assignments to each one of the inputs, which results in 2^{2^n} possible BFs.

There are several ways to represent a BF. It can be represented through truth tables (also known as look-up tables or LUTs), diagrams (BDD, AIG), or propositional formulas (NNF, CNF, DNF, Zhegalkin polynomials [24], multi-linear polynomial [25]). The conversion from one form of representation to another is an NP-Complete problem [26], and the complexity depends exponentially on the number of inputs ($\mathcal{O}(2^n)$). In this work, for reasons that we will explain in the end of Section III, we adopt the multi-linear polynomial representation of BFs for its conversion to NNs.

The representation of BFs through multi-linear polynomials is widely used in the Analysis of Boolean Functions, since it gives rise to concepts of Fourier transform and derivations, such as the influence of an input or the stability of the circuit in the presence of noise. It is also known as the Hamiltonian representation of the BF [27], as it expresses its behavior against certain input. This representation expands the codomain of the BF to the set of real numbers. In this way, any BF can be expressed as [25]:

$$\begin{aligned} & f(x_1, \ldots, x_n) = \\ & = w + \sum_i w_i x_i + \sum_{i<j} w_{ij} x_i x_j + \sum_{i<j<k} w_{ijk} x_i x_j x_k + \ldots \\ & = w + \sum_i w_i h_i + \sum_{i<j} w_{ij} h_{ij} + \sum_{i<j<k} w_{ijk} h_{ijk} + \ldots \\ & = \sum_{S \subseteq \{1,\ldots,n\}} w_S h_S. \end{aligned} \tag{1}$$

In BFs, the sparsity and low-order properties of the Hamiltonian are linked to the complexity and sensitivity of the DCs, studied in the Analysis of Boolean Functions [25]. That is, the more complex and sensitive the DC is (for a fixed number of inputs), the less sparse the polynomial will be, and the higher the degree of the polynomial will be. It is hypothesized that in real-life systems, the type of Hamiltonians we care about evaluating take the form of polynomials which exhibit characteristics that simplify their expression, such as being sparse and low-order [27]. This is especially the case when we

are working with a BF since its polynomial is being evaluated on binary values, so any x_i^k can be reduced to x_i for $k \geq 1$,

DCs are nothing more than a generalization of the BF concept extended to multiple outputs. Thus, a DC is a function of type $f : \{0,1\}^n \rightarrow \{0,1\}^m$, where m is the number of outputs. Each output of the DC can be thought of as having its own multi-linear polynomial representation.

III. Methodology

In this section we describe the technique we used to represent DCs using NNs, through the representation of multi-linear polynomials shown in Section II. This idea serves as the basis for developing our neural compiler.

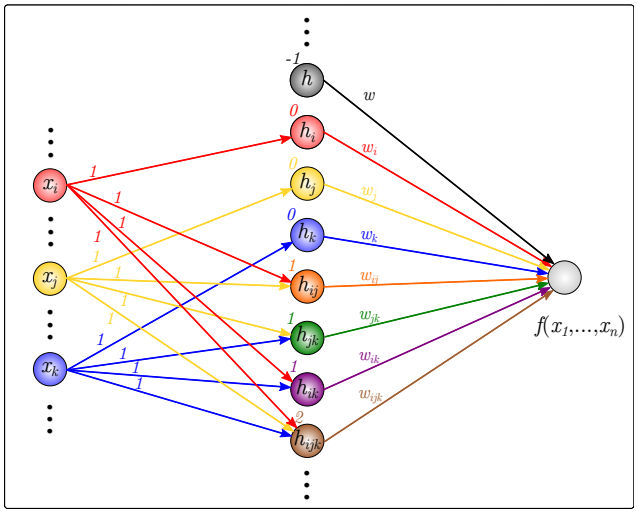

Fig. 2. Equivalent two-layer Neural Network representation of a generic Boolean Function through its multi-linear polynomial representation (contribution 2). Only three input variables are shown, x_i, x_j, x_k, but the structure can be generalized for an arbitrary number of inputs. The hidden neurons (center) are perceptrons computing AND of their inputs; that is $h_S = \bigcap_{s \in S} x_s$. The numbers at their left are the biases.

A. Neural Network representation of Digital Circuits

Any BF can be computed by using a two-layer NN [28]. There are some BF, those called *threshold functions*, that can be represented using a perceptron (single-layer NN, or NN with no hidden layers) [29]:

$$f(x_1, \ldots, x_n) = \Theta\left(\sum_i x_i w_i - b\right), \qquad (2)$$

where $\Theta(x) = 1$ if and only if $x > 0$, also known as the threshold function, w_i are the weights and b is the bias. This property is held by functions like AND, OR, NOT, MAJ, while it is not held by XOR, ADD, SUB, MOD. But in general, larger BFs are not threshold functions [28], and they require at least two layers for their computation.

While there is no algorithm to find the equivalent two-layer NN with the *minimum* number of neurons in the hidden layer, a two-layer NN can be created by noting in Equation 1 that h_S can be rewritten as:

$$h_S = \prod_{s \in S} x_s = \bigcap_{s \in S} x_s = \Theta\left(\sum_{s \in S} x_s - |S| + 1\right). \qquad (3)$$

It is easy to see that $h_S = 1$ if and only if $x_s = 1$ for all $s \in S$. This equivalence shows that AND is a threshold function because it can be expressed as indicated in Equation 2. Notice that the equivalent perceptron of h_S has weights equal to 1 and bias equal to $|S| - 1$. To obtain the NN that represents the BF, we can multiply the values of each h_S by the corresponding weight w_S following Equation 1, as shown in Figure 2. For the case of a DC, only the corresponding output neurons are to be added in the output layer, while the input and hidden neurons are the same for any BFs with the same inputs.

B. Compiler Description

Following the idea described in Section III-A, we created a compiler (contribution 1) that can interpret any DC and exploit its parallelisms by converting it into a NN representation using multi-linear polynomials. For large circuits, however, obtaining the multi-linear polynomial representation suffers from the exponential cost described in Section II-B. We solve this problem by splitting the DC into smaller BFs, whose size is limited by a hyperparameter of the compiler. This step impacts different factors of the equivalent NN, such as the number of layers, sparsity, and order of polynomials used in its construction. Naturally, all of these factors will affect the simulation speed.

The compiler consists of three main modules, as described in Figure 1 that each perform a modular function: HDL parsing and creation of computation graph, conversion to polynomials, and creation of NN. We describe each of them below.

1) HDL Parser and Creation of Computation Graph: In this work, we adopted the Verilog HDL. However, the modularity of the compiler permits the use of other HDLs by simply replacing the Verilog parser by the desired HDL parser.

The compiler starts by interpreting the DC described in different Verilog files. There are several tools available [4] that have been developed to parse code written in Verilog. However, not all of them have APIs or can be easily embedded in other tools. Among the most versatile and open-source tools, Yosys [30] is one of the most popular. Yosys allows the interpretation of DCs written in Verilog[2] (and other HDLs) and allows the circuit to be split[3] into smaller BFs —also known as look-up tables (LUTs)—, in the same way as it is done in hardware simulation.Indeed, this step in hardware synthesis is the prior step to simulation over an FPGA, composed of small programmable LUTs (typically 4, 6 or 8 inputs and 1 output), that receive mapping instructions from a synthesis file.

[2]Although Yosys does not have compatibility with the newest versions of Verilog, there are open-source tools, like sv2v [31], that translate to older versions of Verilog that are compatible with Yosys.

[3]This algorithm is performed by Berkeley ABC library [32], and it is a modification of FlowMap [33].

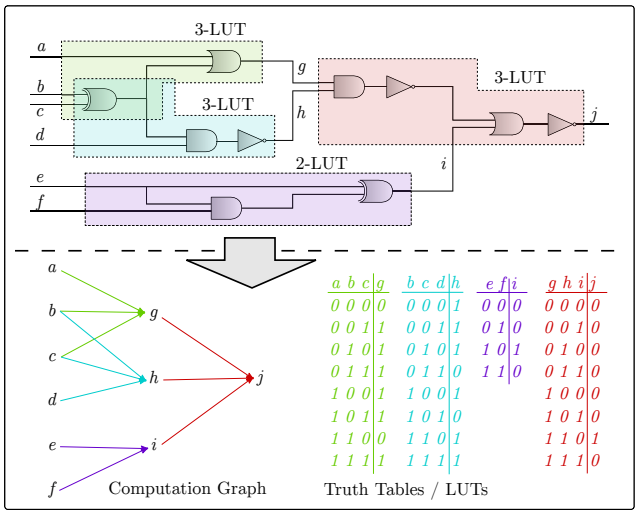

Fig. 3. Demonstration of splitting a Digital Circuit into LUTs (look-up tables) of size $L = 3$ and its conversion into Computational Graph with truth tables. In some Digital Circuits, it is possible that some LUTs are smaller than L (purple LUT), or some LUTs are overlapping (green and cyan LUTs). The truth tables are generated using a SAT solver.

We use this Yosys functionality to create BFs and convert each of them into a LUT with a maximum of L inputs and 1 output[4]. Once the DC is divided into LUTs, a computation graph is formed as shown in Figure 3, in which each node represents a binary signal within the DC, and each edge represents a functional dependency between the connected nodes. This graph is a Directed and Acyclic Graph (DAG), since its original graph (the one with logic gates) was a DAG, and we have simply 'grouped' nodes and edges to form more complex BFs. Another feature of this graph is that the number of predecessors of any node is at most L, since no LUT has more than L dependencies. This implies that the graph is sparse if the number of nodes is large, a feature that we exploit and will explain in Section III-D.

It is important to note that, the larger the value of L:

- the longer the LUTs ($\mathcal{O}(2^L)$)
- the fewer the LUTs
- the shallower the computation graph ($\mathcal{O}((\log_2(L))^{-1})$)

The first point is extremely important, as it is strongly linked to the NP-Complete nature of the DC synthesis problem described in Section II-B. If this limitation did not exist, a single LUT could be generated for each output (see Section III-A), which would depend on the input bits, and the simulation would consist of searching the LUTs for the given input and returning the corresponding outputs. However, the length of the LUT, and therefore the memory required to store it, grows exponentially. This is the reason why commercial FPGAs with L greater than 8 are not designed, as their size in silicon would also grow with this exponential trend.

The second point is the number of splits that we obtain and it is the number of nodes in the computation graph of the DC.

[4]From now on, *inputs* and *outputs* may also refer to those of any LUT.

It is clear that the more inputs we allow to each BF (larger L), the fewer number of BFs we obtain for the same DC. The third point originates from the second one, and it has a direct impact on the number of layers that the equivalent NN will have. Specifically, in a DC that is split into LUTs of size L with no overlap between them, the number of layers is proportional to $\mathcal{O}((\log_2(L))^{-1})$.

On the other hand, a small[5] L would imply more elements and greater depth in the computation graph. Greater depth, under assumption of parallel computation, implies more delay in the simulation time, which is undesirable for most purposes.

2) Conversion to Polynomials: Each of the generated LUTs of the DC is represented by a logical truth table with 2^L rows and L columns, and every truth table has its equivalent representation (CNF, DNF, multi-linear polynomial). To convert the truth table into a multi-linear polynomial (obtaining the coefficients in Equation 1), one could convert to Sum of Products (also known as DNF, or OR of ANDs) and, by using the polynomial expansions of OR and AND functions, obtain the corresponding polynomial. This process is highly expensive as the second step (DNF to polynomial) requires $\mathcal{O}(2^L 2^L) = \mathcal{O}(2^{2L})$ operations[6].

In the framework of this work, however, we developed a divide-and-conquer algorithm inspired by the FFT method to convert value-representation to coefficient-representation of polynomials [34]. The steps are described in Algorithm 1. This algorithm allows the conversion to multi-linear polynomial to be performed much faster than the DNF version mentioned above. The complexity of this algorithm can be obtained by using the Master Theorem for divide-and-conquer algorithms. Notice that the sub-problems have half the size of the main problem (2^L), and the merging step also has half the size of the main problem. Therefore, our algorithm requires at most $2^L L$ operations and saves a significant amount of time for the compilation.

Figure 4 shows a comparison of the two methods mentioned above. As L grows, both methods require more polynomial generation time, but our method has a noticeably weaker trend than the DNF method. It should be noted that polynomial generation is only part of the DC synthesis and must be done for each LUT generated. Therefore, any method that takes more than a few minutes to generate each polynomial is undesirable, since in industry there are compilation time constraints that must be met.

As seen in Section II-B, polynomials representing real-life systems have the property of being sparse and low order. We exploit the property of sparsity by using sparse tensors to represent and manipulate multi-linear polynomials. Also, notice that any polynomial has terms of, at most, order L, since it represents a BF with at most L inputs.

3) Creation of Neural Network: In Section III-A we have seen how a multi-linear polynomial of n variables can be

[5]The minimum value that L can take is 2, where the computation graph is known as an And-Inverter Graph (AIG), if AND and NOT gates are used.

[6]Consider that each of the 2^L clauses can have any of its L literals negated, so each clause is represented by a polynomial of 2^L terms.

Algorithm 1 Divide-and-conquer algorithm to convert a BF from its truth table representation (LUT) to its multi-linear polynomial representation.

input l-LUT $y \in \{0,1\}^{2^l}$ expressed as

$$y = [[\dots[[f(0,\dots,0), f(0,\dots,1)],$$
$$\dots,$$
$$[f(1,\dots,0), f(1,\dots,1)]]\dots]]$$

output Coefficients of the polynomial $w \in \mathbb{R}^{2^l}$ expressed as

$$w = [[\dots[[w_{\dots,0}, w_l],$$
$$\dots,$$
$$[w_{1,\dots,l-1}, w_{1,\dots,l-1,l}]]\dots]]$$

function LUTtoPoly(y)
  if $l = 0$ **then**
    return y `// Base case`
  else
    $w_{\text{left}} \leftarrow$ LUTtoPoly($y[0]$) `// 1st sub-problem`
    $w_{\text{right}} \leftarrow$ LUTtoPoly($y[1]$) `// 2nd sub-problem`
    return $[w_{\text{left}}, w_{\text{right}} - w_{\text{left}}]$ `// Merging`
  end if
end function

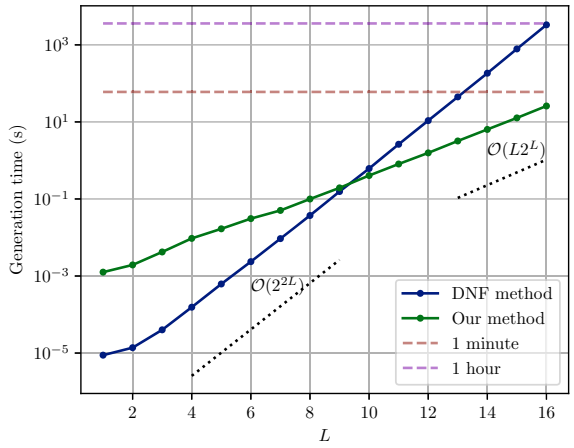

Fig. 4. Generation time (s) of a multi-linear polynomial representation of a Boolean Function from its truth table representation using the DNF method (blue) and our method (green) for different values of L (LUT size). The minute and hour are shown, as well as the trend of each method.

converted into a two-layer NN. Essentially, the hidden layer contains neurons where each one of them correspond to a particular polynomial term. That is, a hidden neuron represents the term of the polynomial that, given an input, the neuron is active if and only if the term is different to zero.

Thus, we can automatically convert any polynomial of the DC into two-layer NNs with the following features:

- The weight of the connection of an input with the hidden neuron will be 1 if the input is present in the term that the hidden neuron represents, or 0 if not. This hidden neuron requires a bias equal to the length of the term minus 1, and a threshold function, to compensate for the possible negative values.
- The weight of the connection of the hidden neuron with the output corresponds to the coefficient of the term that the neuron represents. The output neuron *does not* require any bias or threshold since the polynomial computation is exact; that is, the sum of the active terms multiplied with the coefficients will output either 0 or 1.

This process can be incorporated into the computation graph by replacing each edge (functional dependency) by intermediate (hidden) nodes. We call the resultant graph the NN graph. Notice that because we are adding hidden layers, the depth of the NN graph will be double[7] the depth of the computation graph. This phenomenon, under assumption of parallel computation, implies more steps of computation and thus more delay in the global simulation time. We will see how to sort this issue out in Section III-D by benefiting from the multi-linear polynomial representation of BFs (any other representation would impede such a solution).

Notice that the size of an intermediate (hidden) layer in the NN graph is strongly linked to the value of L, the size of the LUT used. This is because the number of terms in polynomials grows exponentially with L —in the worst case, a polynomial will have all possible monomials with non-zero coefficients, resulting in 2^L terms—. Therefore, as we create one hidden neuron per term in the polynomials, the size of the hidden layer also grows exponentially with L. In total, the number of hidden neurons added will be $\sum_i 2^{L_i}$, where L_i is the degree of the i-th polynomial in a given layer of the computation graph. This is an important constraint that should be considered when utilizing a particular GPU hardware, due to the limited memory that it may have.

C. Handling Complex Digital Circuits

In Section III-B1, we explained how to split a large DC into LUTs of size at most L. This algorithm is straightforward for a combinational module (such as an ALU): for each output, we traverse the network upstream and 'grab' as many logic gates as possible until we complete a group where the inputs to that group of gates are at most L.

Of course, modern DCs have flip-flops, multiple clocks, recurrent connections and modules. However, the algorithm can be robustly applied by following these considerations:

- Clock unification: inspired by circuit synthesis for FPGA simulation, clock unification can be applied in circuits with flip-flops, where all of them become D-type flip-flops referenced to a global clock, at the cost of adding some logic gates to the DC.
- Flip-flop cuts: Also inspired by FPGA synthesis, flip-flops can be 'cut' and their input signals forwarded to the DC's output signals set, and their output signals

[7]Except for the input and output layers.

added to the DC's input signals set. They are frequently called *pseudo-inputs* and *-outputs*. All that is needed is to add the recurring connection that connects the two and thus we abstract the time lag produced by this digital element. Note that with this solution we obtain a purely combinational circuit and therefore, if the DC is properly designed, the recurring connections that may be present are removed[8], resulting in a DAG.

Another step that helps to optimize the splitting algorithm is the unpacking of the modules in the circuit. While modules are represented as clusters in the computation graph that are disconnected from the rest of the nodes (except through their ports), it is possible that the LUT splitting algorithm is prevented from 'grabbing' logic gates that it could grab if module boundaries did not exist. Thus, before applying the LUT splitting algorithm, we unpack the sub-modules to obtain more flexibility and efficiency of the algorithm.

D. Depth Reduction in the Neural Network Graph

In Section III-B3, we explained how the architecture of the NN is constructed. That is, a succession of neural layers representing binary signals, with hidden neural layers in-between, representing polynomial terms. We have also clarified that the layers resulting from the output of a hidden layer do not require bias or threshold function because the computation of their polynomial is exact. This means that the function that represents that layer is a linear function (linear operator or matrix multiplication) and, therefore, can be combined with the next (nonlinear) layer to form a unique nonlinear layer. As shown in Figure 5, the combination consists of multiplying the weights of one layer with the weights of the next one. This procedure can be applied to all output layers except the last one (since no layer follows it). In general, for DCs of significant size (multiple layers), this results in a halving of the number of layers in the neural network, reducing the computational delay and thus, under the assumption of fully parallel computation, reducing the simulation time.

E. PyTorch Framework

After the DC has been read by the compiler, a NN graph is created. This abstract object, however, is not useful for simulation. That is, a library that implements NNs through, for example, matrix multiplication is required. For this work, we chose the PyTorch library, which implements NNs generated through matrix multiplication and threshold functions. PyTorch is a framework for running NNs on GPUs in a very straightforward way, making use of all the parallel computing power that these devices offer.

Converting the NN graph to PyTorch layers is relatively straightforward. We simply map the edges of the graph to values in the matrix of weights. The connected neurons' labels indicate the indices of the weight matrix, which are in a predefined order. On the other hand, the biases are taken one by one from the NN graph. As a result, we obtain a weight matrix and a bias vector for each pair of connected layers in the NN graph. After each set of weights-biases, we add a threshold layer to make the result binary.

[8]They are implicit in the recurrent connection created between output/input of the flip-flop cuts.

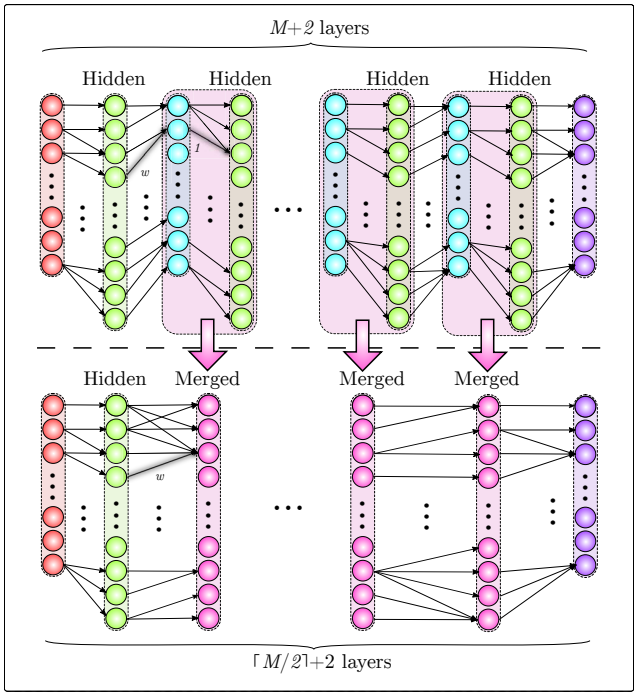

Fig. 5. Algorithm for halving the number of layers in the Neural Network graph thanks to the use of the multi-linear polynomial representation for Boolean Functions. Neurons corresponding to outputs of LUTs (cyan) are combined with the subsequent hidden layer (green) to form the merged layers (magenta). The process of merging is exemplified with the highlighted edges in the top (w and 1), being converted into a single weight in the bottom ($w \times 1 = w$).

Although the weights and biases of the layers can be represented with integers, and the variables at the input and output of each layer with Boolean values, the PyTorch library still has certain limitations for using these types of variables in the NN simulation. In particular, the only data types that are fully implemented for sparse layers (see Section III-F) are the 32-bit and 64-bit floating point types. Therefore, we represent all variables with 32-bit floating point; that is, both the weights and biases of the layers, as well as the inputs and outputs of the layers. It is important to note that, although the data type to represent the variables is changed, the values themselves do not change. That is, instead of $\{0, 1\}$ for Boolean variables we use $\{0.0, 1.0\}$; and instead of $\{..., -1, 0, 0, 1, ...\}$ for integer variables, we use $\{..., -1.0, 0.0, 0.0, 1.0, ...\}$. This one-to-one equivalence is important to preserve the original computation result of the NN.

F. Sparsity of Neural Networks

As mentioned in Section III-B1, the computation graph generated is sparse, a feature that is strongly intensified as the size of the circuit grows. The sparsity of the computation

graph implies that the weight matrices of the NN are also sparse. This results in two main advantages. First, the storage cost of a matrix decreases significantly as the sparsity of the matrix increases. This feature of sparse matrices is important when simulating the NN on GPU devices, since their memory is usually much smaller than that of a CPU. Secondly, the computational cost of matrix multiplication also decreases significantly as the sparsity of the matrix increases. This reduction is valid, however, to a certain extent, depending on the matrix multiplication algorithm used. For libraries such as CuSPARSE [35] (used by PyTorch on GPUs), the statement is true for sparsity $\gtrsim 98\%$ [36]. But this limit, as hypothesized for real-life systems [27], is well exceeded by all the circuits we investigate in this project.

IV. Results

Once the compilation process is complete, simulations can be run with the generated NN model in PyTorch. In this work we measured the throughput obtained for a DC as the product between the number of logic gates and the number of cycles per second simulated (gates*cycles/s). This unit of measurement is useful as it abstracts the simulation time from the size of the circuit and the number of clocks that the test benchmarks have. The number of cycles per second simulated takes into account the time it takes to transfer the binary inputs to the GPU and the time it takes for the binary outputs to be obtained. It does not include, however, the creation of the data, for it is a process that is done prior to the simulation and verification, like in any simulation setup.

The compiler and the NN simulations were run on a computer with RAM 125.8 GiB, processor Intel®Xeon(R) CPU E5-2687W v3 @ 3.10GHz x 20, and GPU GeForce GTX TITAN X/PCIe/SSE2.

A. Benchmarks on Multiple Digital Circuits

The DCs we selected for benchmarking are designs taken from industrial projects, and are commonly used modules in various applications such as communication, encryption, core interfaces, etc. These are Advanced Encryption System (AES), Secure Hash Algorithm 2 (SHA), Serial Peripheral Interface (SPI), Universal Asynchronous Receiver/Transmitter 16550 (UART), Direct Memory Access (DMA) and an *ad-hoc* processor designed to interface with RISC-V core.

Although we demonstrated the one-to-one computational equivalence between the DC and the generated NN in Section III, we performed a verification of the correctness of the NN outputs. That is, we made a direct comparison of the outputs obtained by the commercial DC simulator, Verilator, with the outputs obtained by the NN corresponding to that DC, when stimulated with the same inputs. For all cases, the NN outputs were identical to those of the commercial simulator, which corroborates that the compiled NN is computationally equivalent to the DC.

The first columns of Table I summarize the names of these circuits and their parameters that define their size and complexity. Also included is the throughput obtained with the Verilator simulator measured in g*c/s, or gates*clocks/s. Notice that the throughput given by Verilator is relatively constant compared to the variations of size in the DCs. This phenomenon indicates that Verilator has low parallel computation power because having a larger circuit produces small improvements in throughput.

The middle columns show the parameters and NN compilation results using different values of L (LUT size, or maximum polynomial size). In this section we use values of 3, 7 and 11 to analyze the performance of NNs with different L in general terms, but in section 6 we analyze in more detail the impact of this parameter on one of the circuits. In general, we can observe a larger Generation Time (compilation time) as we increase L. This is an important variable to be considered by the user when deciding what software to use. If the DC is not going to be run with multiple test benchmarks, which is commonly the case for small and simple circuits, then the compilation time is much more important than the simulation time, as it could be the bottleneck of the entire RTL simulation process. But if the DC requires multiple tests, which is commonly the case for large or complex circuits, the compilation time will be insignificant compared to the simulation time. In the case of Verilator, the compilation time is quite fast since it consists of a source-to-source language conversion (transpilation), so the bottleneck is determined by the simulation time. In our case, the compilation involves the generation of LUTs, whose lengths increase exponentially with the L parameter, and the compiler has been developed in the Python programming language. Therefore, the compilation time could be the bottleneck for small and simple circuits. As this work is focused on simulating large or complex circuits, the analyses presented are with respect to simulation and not compilation, although the latter is added for completeness.

The memory size of the NN (as well as the neurons' connections, for they are strongly correlated) holds the same behavior with L. This places an important constraint on the variable that depends on GPU memory limitations. The number of layers, however, decreases as we increase L. This indicates that the greater the value of L, the slower the NN simulation (for a constant number of stimuli). The Mean Sparsity, measured among all layers, is significantly high for all circuits analyzed, and all values of L used. This allows the NNs to be simulated on GPUs much more efficiently than their dense versions, as explained in Section III-F.

In the last columns, the throughput results of the NN simulation on GPUs are shown together with the speed-up obtained with respect to the Verilator simulator.Comparing with Verilator may not seem pertinent since RTLFlow has shown faster simulation speeds (up to $\times 40$ for industrial circuits). However, research focused on improving the speed of RTL simulation compare to commercial simulators given their reliability, detailed documentation, and stability to be run on different devices without errors. In this work, for all circuits analyzed, we obtain a throughput up to three orders of magnitude greater than the traditional simulator Verilator.

TABLE I

SUMMARY OF ALL THE DIGITAL CIRCUITS ANALYZED TO TEST THE COMPILER AND SIMULATION OF NEURAL NETWORKS. THE FIRST COLUMNS SHOW THE LINES OF CODE (IN VERILOG), THE NUMBER OF GATES (INCLUDING FLIP-FLOPS), AND THE THROUGHPUT USING VERILATOR MEASURED AS GATES*CYCLES/S. THE MIDDLE COLUMNS SHOW THE TIME OF NEURAL NETWORK GENERATION, THE MEMORY SIZE OF THE NEURAL NETWORK FILE AND THE NUMBER OF CONNECTIONS, AND THE NUMBER OF LAYERS AND THEIR MEAN SPARISTY, FOR DIFFERENT VALUES OF L (LUT SIZE). THE LAST COLUMNS SHOW THE THROUGHPUT OF THE NEURAL NETWORK SIMULATION ON A GPU AND THE SPEED-UP COMPARED TO VERILATOR.

	Verilog Simulation			**Neural Network Compilation**						**Neural Network Simulation**	
Circuit Name	**LoC**	**Gates**	**Verilator Throughput (g*c/s)**	L	**Generation Time (s)**	**Memory (MB)**	**Neurons' connections $\times 10^6$**	**Layers**	**Mean Sparsity**	**Throughput (g*c/s)**	**Speed-up ($\times$)**
AES	13625	56380	7.71E+08	3	226.67	8.19	0.45	22	0.99982	4.55E+11	589.86
				7	253.57	16.10	1.18	12	0.99975	8.41E+11	1090.17
				11	885.24	147.93	7.13	10	0.99943	8.87E+11	**1150.37**
SHA	48988	120897	9.56E+08	3	698.65	24.52	1.27	52	0.99989	4.47E+11	468.11
				7	1935.45	55.42	4.06	22	0.99979	9.38E+11	981.05
				11	3322.19	686.11	44.17	16	0.99944	1.07E+12	**1115.13**
SPI	8130	9519	2.03E+09	3	26.54	1.84	0.10	22	0.99935	8.19E+11	404.32
				7	30.35	2.98	0.28	11	0.99877	1.40E+12	**691.53**
				11	190.85	34.48	2.54	8	0.99726	2.15E+11	106.01
UART	9565	7320	1.44E+09	3	23.29	1.44	0.08	22	0.99899	4.63E+11	322.83
				7	34.44	3.77	0.26	11	0.99822	8.52E+11	593.56
				11	174.05	37.91	2.46	9	0.99451	1.21E+12	**840.31**
DMA	79591	372297	1.72E+09	3	7498.22	102.77	5.14	52	0.99998	6.45E+11	**374.94**
				7	4854.03	165.95	11.74	23	0.99996	2.93E+11	170.67
				11	22346.23	2755.36	236.81	16	0.99977	4.07E+11	237.23
RISC-V interface	28003	33568	5.63E+08	3	183.89	8.84	0.47	26	0.99990	2.48E+11	439.52
				7	188.79	12.08	0.81	12	0.99976	4.89E+11	868.10
				11	675.43	170.52	7.08	9	0.99890	6.43E+11	**1142.06**

B. Impact of LUT Size

As seen in Section III-B1, L (LUT size) plays an important role in the architecture of the NN and so in its simulation. That is, the number of layers of the NN approximately depends on $(\log_2(L))^{-1}$ and the number of neurons' connections in the NN depends on 2^L.

To illustrate these dependencies, we compiled the UART circuit with different values of L. We can see that these dependencies are fulfilled in Figure 6. The number of neurons' connections grows exponentially with L, while the number of layers decays logarithmically with L. We can also observe the simulation times in a single-stimulus simulation of the NNs generated, on GPU (parallel) and CPU (non-parallel). Due to the implementation of PyTorch NNs as matrix multiplication, it is natural that the simulation time on CPU depends directly on the number of connections. That is, the simulation time on CPU is proportional to the number of operations performed in matrix multiplication, which is proportional to the number of non-zero values of the weight matrices. The number of non-zero values is simply the number of connections of the NN graph. On the other hand, the GPU simulation time has a completely different behavior. It does not depend on the number of neurons' connections, but on the number of layers. The reason is that GPU devices can split the task of matrix multiplication and perform it in parallel. Therefore, the GPU simulation time is proportional to the number of matrix multiplications, which is exactly equal to the number of layers in the NN.

V. CONCLUSION

In this paper, we proposed a novel method to accelerate and simplify the RTL simulation of Digital Circuits on GPU devices, which have shown potential in this field and room for further research in recent years. For this purpose, we developed a compiler that performs a one-to-one conversion of Digital Circuits into Neural Networks, which are models that allow a straightforward simulation on GPUs by using libraries such as PyTorch. Within the compiler we utilized different Neural Network techniques, such as sparsity exploitation, which allows the suitability of simulation on GPUs, and reduction of layers, which halves the number of computations. The speed-up obtained by this method is up to $\times 1100$ compared to Verilator, one of the most used simulators at research and industrial level, and is much higher than the $\times 40$ speed-up reported by RTLFlow, a SOTA technique for RTL simulation on GPU.

Although our automatic compiler shows superior simulation efficiency, there are possible directions to improve these results. First, the Neural Network implementation through PyTorch suffers from a large overhead at simulation time. This is due to three main reasons:

- The software runs in Python while the Neural Network runs in C++. This language jump adds overhead to each forward pass of the Neural Network that can be avoided through simulation in C++.
- Due to the current state of the PyTorch library, we had to implement the Neural Network simulation using floating point numbers. However, in our solution, the variables

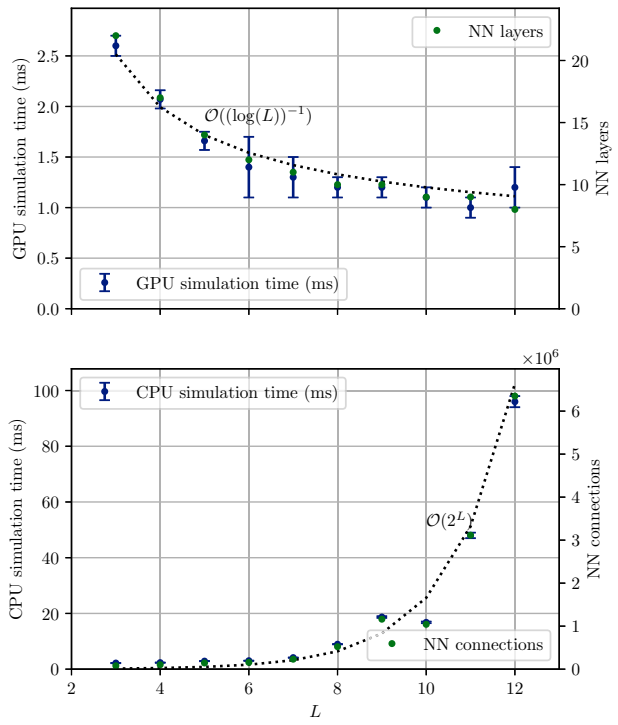

Fig. 6. GPU (parallel) and CPU (non-parallel) simulation time and number of NN layers and NN connections for different values of L (LUT size), showing their correlations for single-stimulus simulation of circuit UART. (*top*) Number of NN layers (green) and GPU simulation time (blue), together with an inverse logarithmic trend. (*bottom*) Number of NN connections (green) and CPU simulation time (blue), together with an exponential trend.

are binary and the parameters (weights and biases) of the Neural Network are integers. The matrix multiplication performance is significantly worse when using floating point numbers, which slows down the circuit simulation. This problem can be solved by creating GPU kernels specifically designed to work with integers and binaries.

- Due to its versatility to work with tensors, PyTorch adds overhead to the network forward pass by checking what type of floating number is being used, or if it belongs to the complex numbers.

Secondly, although LUT splits allow us to achieve great versatility to simulate any type of circuit, this process has the practical limitation that it will always be limited to the value of L used due to the exponential growth of the polynomials. This process, however, does not use any abstraction from the HDL code. For example, an AND circuit of a 9-bit vector would be split into four parts using $L = 3$, even though the polynomial of this function is extremely simple: it is the multiplication of the 9 input bits. By implementing polynomial libraries for known functions (ADD, SUB, MOD, etc.) the polynomial generation process can be significantly improved (the equivalent of increasing L) and the depth of the Neural Network further reduced.

We believe that these modifications can not only bring further improvements in Digital Circuit simulation throughput, but also open a research path for other types of simulation in the Integrated Circuits industry, like gate-level simulation or event-driven simulation, using Neural Network representations.

References

[1] H. Vollmer, *Introduction to circuit complexity: a uniform approach*. Springer Science & Business Media, 1999.

[2] K. Y. Kamal, "The silicon age: Trends in semiconductor devices industry." *Journal of Engineering Science & Technology Review*, vol. 15, no. 1, 2022.

[3] Y. Zhang, H. Ren, and B. Khailany, "Opportunities for rtl and gate level simulation using gpus," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–5.

[4] T. S. Tan and B. A. Rosdi, "Verilog hdl simulator technology: a survey," *Journal of Electronic Testing*, vol. 30, no. 3, pp. 255–269, 2014.

[5] W. Snyder, "Verilator and systemperl," in *North American SystemC Users' Group, Design Automation Conference*, 2004.

[6] S. Beamer and D. Donofrio, "Efficiently exploiting low activity factors to accelerate rtl simulation," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[7] D.-L. Lin, H. Ren, Y. Zhang, and T.-W. Huang, "From rtl to cuda: A gpu acceleration flow for rtl simulation with batch stimulus," in *ACM International Conference on Parallel Processing (ICPP)*, 2022.

[8] S. Holst, M. E. Imhof, and H.-J. Wunderlich, "High-throughput logic timing simulation on gpgpus," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 3, pp. 1–22, 2015.

[9] Y. Zhu, B. Wang, and Y. Deng, "Massively parallel logic simulation with gpus," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 16, no. 3, pp. 1–20, 2011.

[10] A. Sen, B. Aksanli, M. Bozkurt, and M. Mert, "Parallel cycle based logic simulation using graphics processing units," in *2010 Ninth International Symposium on Parallel and Distributed Computing*. IEEE, 2010, pp. 71–78.

[11] H. Johnston. (2022) Are neuromorphic systems the future of high-performance computing? Physics World Magazine. [Online]. Available: www.physicsworld.com/a/are-neuromorphic-systems-the-future-of-high-performance-computing

[12] 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, 2019.

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "{TensorFlow}: a system for {Large-Scale} machine learning," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 265–283.

[14] M. Wang, D. Zheng, Z. Ye, Q. Gan, M. Li, X. Song, J. Zhou, C. Ma, L. Yu, Y. Gai, T. Xiao, T. He, G. Karypis, J. Li, and Z. Zhang, "Deep graph library: A graph-centric, highly-performant package for graph neural networks," *arXiv preprint arXiv:1909.01315*, 2019.

[15] I. Scarabottolo, G. Ansaloni, G. A. Constantinides, L. Pozzi, and S. Reda, "Approximate logic synthesis: A survey," *Proceedings of the IEEE*, vol. 108, no. 12, pp. 2195–2213, 2020.

[16] M. Pradhan and B. B. Bhattacharya, "A survey of digital circuit testing in the light of machine learning," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 11, no. 1, p. e1360, 2021.

[17] D. Patel, I. Gavier, J. Russell, A. Malinsky, E. Rietman, and H. Siegelmann, "Automatic transpiler that efficiently converts digital circuits to a neural network representation," in *International Joint Conference on Artificial Intelligence*. IEEE, 2022.

[18] 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.

[19] J. Clow, G. Tzimpragos, D. Dangwal, S. Guo, J. McMahan, and T. Sherwood, "A pythonic approach for rapid hardware prototyping and instrumentation," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2017, pp. 1–7.

[20] 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.
[21] S. Wilson, "Verilator 4.0-open simulation goes multithreaded," in *The Open Source Digital Design Conference (ORConf)*, 2018.
[22] G. M. Amdahl, "Computer architecture and amdahl's law," *Computer*, vol. 46, no. 12, pp. 38–46, 2013.
[23] H. Qian and Y. Deng, "Accelerating rtl simulation with gpus," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2011, pp. 687–693.
[24] J. Gomda, "Transformation of the canonical disjunctive normal form of a boolean function to its zhegalkin-polynomial and back," *rn*, vol. 2, p. 1, 2001.
[25] R. O'Donnell, *Analysis of boolean functions*. Cambridge University Press, 2014.
[26] C. Umans, T. Villa, and A. L. Sangiovanni-Vincentelli, "Complexity of two-level logic minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1230–1246, 2006.
[27] H. W. Lin, M. Tegmark, and D. Rolnick, "Why does deep and cheap learning work so well?" *Journal of Statistical Physics*, vol. 168, no. 6, pp. 1223–1247, 2017.
[28] M. Anthony, "Connections between neural networks and boolean functions," *Boolean Methods and Models*, vol. 20, 2005.
[29] S.-T. Hu, "Threshold logic," in *Threshold Logic*. University of California Press, 1965.
[30] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.
[31] Z. Snow, "sv2v: Systemverilog to verilog," 2020.
[32] R. Brayton and A. Mishchenko, "Abc: An academic industrial-strength verification tool," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 24–40.
[33] J. Cong and Y. Ding, "Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based fpga designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 13, no. 1, pp. 1–12, 1994.
[34] R. T. Moenck, "Practical fast polynomial multiplication," in *Proceedings of the third ACM symposium on Symbolic and algebraic computation*, 1976, pp. 136–148.
[35] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi, "Cusparse library," in *GPU Technology Conference*, 2010.
[36] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse gpu kernels for deep learning," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.