High-Throughput SAT Sampling

Arash Ardakani
University of California, Berkeley
arash.ardakani@berkeley.edu

Minwoo Kang
University of California, Berkeley
minwoo_kang@berkeley.edu

Kevin He
University of California, Berkeley
kevinjhe@berkeley.edu

Qijing Huang
NVIDIA
jennyhuang@nvidia.com

John Wawrzynek
University of California, Berkeley
johnw@berkeley.edu

***Abstract*—In this work, we present a novel technique for GPU-accelerated Boolean satisfiability (SAT) sampling. Unlike conventional sampling algorithms that directly operate on conjunctive normal form (CNF), our method transforms the logical constraints of SAT problems by factoring their CNF representations into simplified multi-level, multi-output Boolean functions. It then leverages gradient-based optimization to guide the search for a diverse set of valid solutions. Our method operates directly on the circuit structure of refactored SAT instances, reinterpreting the SAT problem as a supervised multi-output regression task. This differentiable technique enables independent bit-wise operations on each tensor element, allowing parallel execution of learning processes. As a result, we achieve GPU-accelerated sampling with significant runtime improvements ranging from $33.6\times$ to $523.6\times$ over state-of-the-art heuristic samplers. We demonstrate the superior performance of our sampling method through an extensive evaluation on 60 instances from a public domain benchmark suite utilized in previous studies.**

***Index Terms*—Boolean Satisfiability, Gradient Descent, Multi-level Circuits, Verification, and Testing.**

I. Introduction

High-throughput SAT samplers play a crucial role in advancing the state of the art in software and hardware verification methodologies [1]. Generating a set of random solutions to logical constraints is critical in the verification, testing, and synthesis. In software verification, SAT samplers enable efficient exploration of diverse execution paths, addressing the scalability challenges inherent in symbolic execution [2]–[17]. In hardware verification, they support the generation of varied input patterns, ensuring a rigorous and effective verification process [18]–[21].

The SAT sampling process begins by formulating the logical constraints of the target application into conjunctive normal form (CNF) [22]. CNF is the specific format required by most SAT samplers, where the logical formula is expressed as a conjunction of clauses, with each clause being a disjunction of literals. The complexity of the logical constraints in the target application can result in a CNF that is not always concise. Nevertheless, CNF remains the preferred format due to the strong performance of SAT samplers and solvers. The complexity of the CNF can, however, affect the efficiency of these solvers.

SAT solvers employ various strategies to find a satisfying assignment for the variables in the CNF. Modern SAT solvers [23]–[25] often use the conflict-driven clause learning (CDCL) algorithm [26], [27], that relies heavily on heuristics such as conflict-driven backtracking and clause learning. These heuristics effectively guide the CDCL algorithm in finding a satisfying assignment. Due to the sequential nature of these heuristics, that involve branching and backtracking, the latest SAT solvers are typically executed on CPUs. Consequently, state-of-the-art SAT samplers, which incorporate SAT solvers within their algorithms, also rely on a sequential process and are optimized for CPU execution.

Generating multiple satisfying solutions to the SAT problem is a good match to GPU computing, provided that a sampling method is available that performs consistent, data-parallel computations. To address this opportunity, we propose a transformation algorithm that converts logical constraints encoded as a CNF into a simplified multi-level, multi-output Boolean function while maintaining the original logical constraints. This transformation significantly reduces the number of bit-wise operations and thus lowers the complexity of the sampling process. We then formulate the resulting simplified multi-level, multi-output Boolean function as a multi-output regression task, where each logic gate is represented probabilistically, enabling the use of gradient descent (GD) to learn diverse solutions. This approach enables the parallel generation of independent SAT problem solutions, allowing for effective GPU acceleration. We demonstrate the superior performance of our sampling method across 60 instances from a publicly available benchmark suite [28] used in previous studies. The code of our sampler is available at https://github.com/arashardakani/High-Throughput-SAT-Sampler.

II. Preliminaries

A. SAT Sampling

SAT sampling involves drawing solutions from the solution space defined by a set of logical constraints expressed in CNF. In SAT sampling applications, Boolean expressions are typically represented in higher-level logical formats before being converted into CNF [22], [29]. These formats include propositional logic with operators like AND, OR, NOT, implications, and equivalences, as well as more complex structures such as if-then-else conditions, arithmetic expressions, and bit-level operations. In hardware verification, Boolean expressions can take the form of circuit representations, such as And-Inverter Graphs (AIGs) or Binary Decision Diagrams (BDDs). In cryptographic contexts, Algebraic Normal Form (ANF) is sometimes used. These representations are transformed into CNF through logical simplifications, flattening complex structures, and applying techniques like Tseitin transformation [30]. This transformation preserves the satisfiability of the original formula while introducing auxiliary variables when needed. The conversion to CNF provides SAT solvers with a standardized problem representation that retains the essential constraints of the original problem.

A CNF consists of a conjunction of clauses (i.e., an AND of multiple clauses), where each clause consists of a disjunction of literals (i.e., an OR of literals). Literals refer to Boolean variables or their negations. In SAT solving, the goal is to determine if there exists an assignment of binary values to the variables in a given CNF, representing a Boolean expression, such that all clauses evaluate to 1. SAT sampling adds a probabilistic layer to this process. Instead of seeking just one solution for

satisfiable instances, the aim is to produce multiple solutions or samples from the complete set of possible solutions. Generating samples from SAT instances plays a crucial role in design verification, testing, and synthesis, with significant applications in constrained-random verification (CRV) [18].

A common method for SAT sampling involves using SAT solvers with built-in sampling functionality. These solvers are designed not only to determine the satisfiability of a Boolean formula but also to extract solutions from the solution space. Efficient SAT solving techniques include backtracking algorithms like the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [31], stochastic local search methods such as WalkSAT [32], and CDCL algorithms [26], [27]. In recent years, several approaches have been developed for SAT sampling, including randomized algorithms, Markov chain Monte Carlo (MCMC) techniques, and heuristic-based sampling methods [1], [33]–[36]. These methods typically explore the solution space iteratively, selecting candidate solutions based on predefined criteria and stochastically deciding whether to accept or reject them.

SAT solvers and samplers have been optimized over decades to efficiently handle problems in CNF. CNF is well-suited for SAT-solving algorithms like DPLL and CDCL. These algorithms take advantage of CNF's structure to systematically explore possible truth assignments, detect conflicts early, and prune the search space efficiently. By focusing on individual clauses, which define specific constraints on solutions, the use of CNF enables solvers to address highly complex problems in a manageable way.

B. Multi-Output Regression Task

A multi-output regression task, in the context of example generation, refers to the process of creating data points (or "examples") that serve as input-output pairs for a given model [37]. The main goal is to generate inputs such that they satisfy multiple specific output constraints simultaneously. Various methods, such as linear regression, neural networks or probabilistic representations of the relationships between inputs and outputs, can be used to construct such a model. The inputs to the model are adjusted in an iterative manner to minimize the difference between the predicted and actual output values. One common way to measure this difference is by using metrics like mean squared error (MSE) or ℓ_2 loss.

III. Methodology

In this section, we present a transformation algorithm paired with an optimization method to generate valid and diverse solutions to SAT problems. Our algorithm transforms a CNF from its flat, two-level structure into a more streamlined, multi-level, multi-output Boolean function, reducing both operations and logical constraints. We then apply gradient-based optimization to efficiently learn valid solutions using the simplified multi-level, multi-level function.

As discussed in Section II-A, Boolean expressions are typically represented in more abstract logical formats before being converted to CNF. In other words, the sub-expressions (i.e., sub-clauses) in a CNF often result from the transformation of single logical operators or the combination of multiple operators. This transformation increases the size of the CNF. Therefore, a sampler that could operate directly on the logical operators represented by the sub-expressions in a CNF would be advantageous. Motivated by this, we introduce our transformation algorithm, which converts CNF into an equisatisfiable multi-level, multi-output Boolean function, and demonstrate how this function can be used to generate various valid and distinct solutions using gradient-based optimization.

A. Transformation Algorithm

Let us first review the clauses, known as the CNF signature [38], which represent primary logical operators as a result of the Tseitin transformation. The CNF structure of an inverter with the input x and the output f, i.e., $f(x) = \neg x$, is given by

$$(f \vee x) \wedge (\neg f \vee \neg x), \tag{1}$$

where $\vee$ and $\wedge$ denote logical OR and AND operators, respectively. The clauses representing the logical OR operator with n inputs and an output f, i.e., $f(x_1, x_2, \ldots, x_n) = \bigvee_{i=1}^{n} x_i$, can be expressed as

$$\left(\neg f \vee \bigvee_{i=1}^{n} x_i\right) \wedge \left(\bigwedge_{i=1}^{n} (f \wedge \neg x_i)\right). \tag{2}$$

The clauses representing the logical AND operator with n inputs and an output f, i.e., $f(x_1, x_2, \ldots, x_n) = \bigwedge_{i=1}^{n} x_i$, are similar to those of the logical OR operator with its input and output variables inverted, i.e.,

$$\left(f \vee \bigvee_{i=1}^{n} \neg x_i\right) \wedge \left(\bigwedge_{i=1}^{n} (\neg f \wedge x_i)\right). \tag{3}$$

The CNF structure of the logical NAND and NOR operators can be derived in a similar manner by inverting the output variable in the clauses associated with the logical AND and OR operators, respectively. The CNF signature of the logical XOR operator with n inputs and an output f is is given by

$$\begin{aligned}&(\neg f \vee \mathrm{XOR}(x_1, \ldots, x_n)) \wedge (f \vee \neg \mathrm{XOR}(x_1, \ldots, x_n))\\ =&\neg \mathrm{XOR}(x_1, \ldots, x_n, f) = \mathrm{XNOR}(x_1, \ldots, x_n, f).\end{aligned} \tag{4}$$

Similarly, the CNF structure for the logical XNOR operation can be described as $\mathrm{XOR}(x_1, \ldots, x_n, f)$.

While deriving logical operators from the CNF signatures described above is straightforward through pattern matching, this method is limited to identifying clauses linked to primary logical operators. It does not handle sub-expressions that may involve more complex Boolean expressions constructed from these operators. For example, the clauses

$$\begin{aligned}&(\neg x_4 \vee \neg x_{107} \vee x_5) \wedge\\ &(\neg x_4 \vee x_{107} \vee \neg x_5) \wedge\\ &(x_4 \vee \neg x_{108} \vee x_5) \wedge\\ &(x_4 \vee x_{108} \vee \neg x_5)\end{aligned} \tag{5}$$

from the '75-10-1-q' CNF instance correspond to the function $x_5(x_4, x_{107}, x_{108}) = (x_{107} \wedge x_4) \vee (x_{108} \wedge \neg x_4)$, which cannot be identified using pattern matching alone, as it is impractical to store all possible Boolean patterns. This underscores the need for a general transformation algorithm capable of identifying the Boolean sub-expressions and constraints represented by the clauses.

Before introducing our algorithm, let us establish some key definitions. The primary objective of the algorithm is to transform a given CNF into an acyclic combinational structure while preserving all logical constraints in the original CNF. Variables that correspond to the inputs and outputs of this circuit are referred to

as primary input variables and primary output variables, respectively. Variables representing the intermediate signals within the circuit are called intermediate variables. With these definitions in place, once a variable is identified as a primary input or intermediate variable, it cannot be redefined as an output variable in subsequent Boolean expressions, due to the acyclic nature of the circuit.

The main challenge in this transformation is first to identify sub-clauses corresponding to a Boolean expression and then to derive that expression. To achieve this, we begin by reading the first clause in the CNF. For each variable in the clause, if it has not already been defined as a primary input or intermediate variable, we treat it as a potential output of the clause or clauses read so far. We then derive its Boolean expression based on the clause or clauses (read so far) containing this variable in its negated form, and similarly derive its complementary expression from the clauses where the variable appears in its true form. If the resulting Boolean expressions are indeed complements of each other, we have successfully identified the Boolean expression for the current set of clauses. If not, we proceed to the next clause and repeat the process until the corresponding Boolean expression is found. Once the Boolean expression is identified, we designate its output variable as an intermediate variable. Additionally, if any input variables in the derived Boolean expression have not yet been classified as intermediate variables, they are now recognized as primary input variables. In case the resulting Boolean expression is identified to be a constant Boolean function with its output being a constant value of either 0 or 1, its output variable is recognized as a primary output variable. The obtained Boolean expression is simplified before adoption in the final circuit structure. For all Boolean manipulations, such as simplification and complement checking, we utilize the symbolic Boolean algebra system SymPy [39], a Python library for symbolic mathematics. It is worth noting that the resulting multi-level, multi-output Boolean function from our transformation can be further optimized by leveraging other techniques [40]–[42], for reducing the complexity of multi-level logic circuits, potentially leading to even more compact Boolean functions. Our transformation process is summarize in Algorithm 1.

Algorithm 1 Pseudo Code of our Transformation Method

Input: A list of clauses C
Output: List of primary outputs PO, primary inputs PI, intermediate variables IV, and Boolearn expressions BE
SC = [] {List of sub-clauses}
for $i = 1$ to length(C) **do**
  Append $C[i]$ to SC
  for each item v in SC **do**
    if $v \notin PI$ and $v \notin IV$ **then**
      $f \leftarrow$ FindBooleanExpression(v, SC)
      $g \leftarrow$ FindBooleanExpression$(\neg v, SC)$
      if $f = \neg g$ **then**
        Append Simplify(f) to BE
        if $f = True$ or $f = False$ **then**
          Append v to PO
        else
          Append v to IV
        end if
        for each item w in SC **do**
          if $w \notin IV$ and $w \neq v$ **then**
            Append w to PI
          end if
        end for
        SC = []
        break
      end if
    end if
  end for
end for
Return PO, PI, IV, BE

To clarify the transformation process, let us revisit the clauses in in Eq. (5). Treating x_5 as a potential output variable, we derive its Boolean expression from the clauses where x_5 appears in its negated form (i.e., $(\neg x_4 \vee x_{107} \vee \neg x_5) \wedge (x_4 \vee x_{108} \vee \neg x_5)$). Since $\neg x_5 = 0$ in these clauses, it becomes non-contributory, resulting in the Boolean expression $x_5(x_4, x_{107}, x_{108}) = (x_{107} \wedge x_4) \vee (x_{108} \wedge \neg x_4)$. This is because we are determining the Boolean expression for $x_5 = 1$, and thus the remaining clauses containing x_5 in its true form are already satisfied and do not contribute to the expression.

Similarly, we derive the complementary Boolean expression using the clauses where x_5 appears in its true form (i.e., $(\neg x_4 \vee \neg x_{107} \vee x_5) \wedge (x_4 \vee \neg x_{108} \vee x_5)$). In this case, $x_5 = 0$ rendering it non-contributory and resulting in $\neg x_5(x_4, x_{107}, x_{108}) = (\neg x_{107} \wedge x_4) \vee (\neg x_{108} \wedge \neg x_4)$. Since these two expressions are complementary, it confirms the validity of the Boolean expression with the specified input and output variables. It is worth mentioning that repeating this process for other variables as potential output variables does not yield complementary expressions, thereby invalidating the assumption for those cases.

During the transformation process, some sub-clauses may be under-specified, making them difficult to translate directly into a Boolean expression. A simple example is an OR function $x_3(x_1, x_2) = x_1 \vee x_2$, where the output is constrained to 1. The full description of the associated clauses would be $(\neg x_3 \vee x_1 \vee x_2) \wedge (x_3 \vee \neg x_1) \wedge (x_3 \vee \neg x_2) \wedge x_3$. However, since x_3 is equal to 1 in this case, these clauses can be simplified to $(x_1 \vee x_2)$. In this simplified form, where the output variable is not explicitly specified, we cannot directly extract the function with its output constrained to 1. To handle such cases, if the current clauses do not share variables with subsequent clauses, we simplify these clauses to a Boolean expression as the result of the transformation. The variables of these clauses are then treated as input variables to the simplified Boolean expression, and an auxiliary variable is assigned to the output, which is constrained to 1.

After transforming all the clauses, we integrate the resulting Boolean sub-expressions to construct a multi-level, multi-output Boolean function that is equisatisfiable with the original CNF. This function contains two types of paths: *constrained paths* and *unconstrained paths*. Constrained paths are those that run from primary inputs to primary outputs. The inputs along these paths must be adjusted to satisfy the explicit constraints applied to the primary output variables. Unconstrained paths originate from primary inputs and terminate at intermediate variables. Since they are not explicitly constrained to any values, any random initialization of their primary inputs will yield satisfying solutions for the variables on these paths. Fig. 1(a) and 1(b) present an example of a CNF and its corresponding circuit structure, produced through

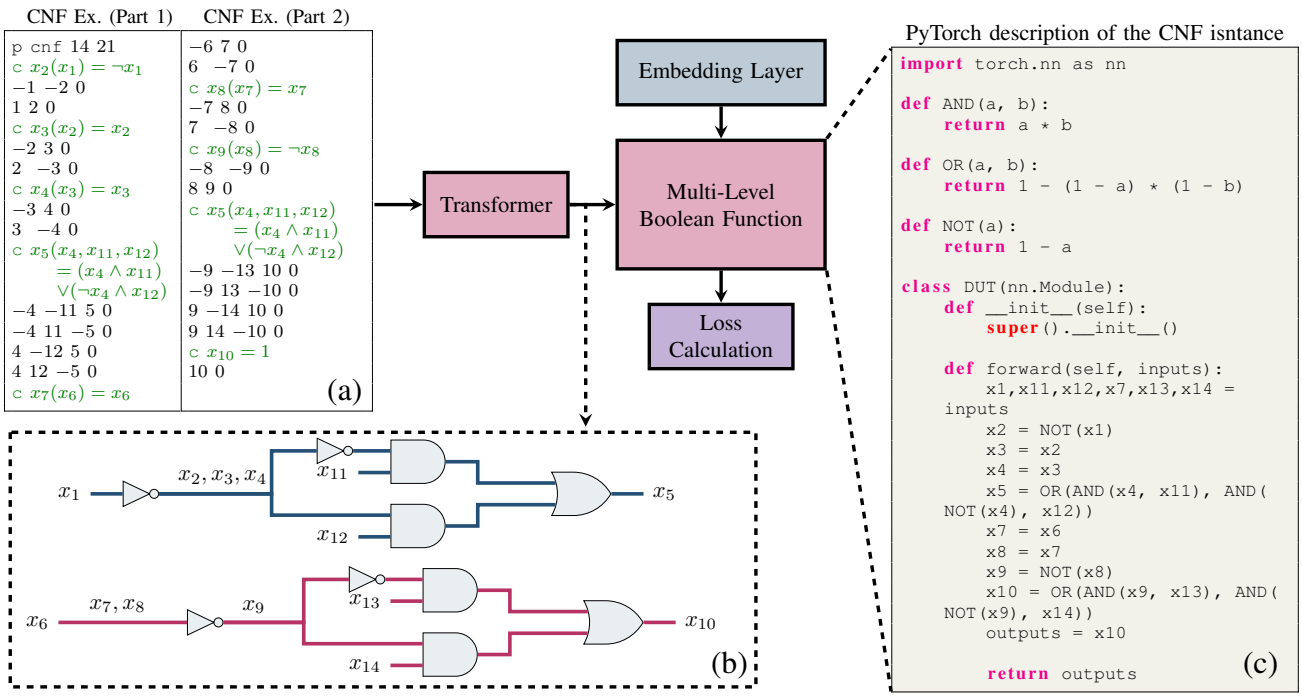

Figure 1: The overall workflow of our sampling approach is illustrated including (a) a CNF example with comments highlighted in green, (b) its simplified multi-level, multi-output Boolean function in a circuit form for illustrative purposes with unconstrained and constrained paths highlighted in blue and red, respectively, and (c) its corresponding PyTorch description.

our transformation method. The figures highlight two types of paths: unconstrained paths in blue and constrained paths in red. In this example, any random assignment to the primary input variables x_1, x_{11}, and x_{12} will satisfy the sub-clauses associated to the unconstrained path. Conversely, the primary input variables for the constrained path, namely x_6, x_{13}, and x_{14}, must be carefully selected such that x_{10} is forced to be 1.

With the definitions in place, the SAT solving/sampling problem now becomes a task of finding the primary inputs in the constrained paths to the multi-level, multi-output Boolean function resulting from the CNF transformation. Once the values of the primary input variables are determined to satisfy the constraints on the primary output variables, the intermediate variables can be computed using the corresponding Boolean operators in the function (see Fig. 1(c)).

To solve for the primary input variables in the constrained paths, we use GD optimization. Specifically, we reformulate the solving/sampling problem into a multi-output regression task, where the objective is to learn the primary input variables based on explicit constraints on the primary outputs and a differentiable model that maps the inputs to the outputs. While the multi-level, multi-output Boolean function fulfills the mapping requirements, it is not differentiable in its discrete form. To address this, we employ a probabilistic representation of logical operators such as AND, OR, NOT, XOR, and XNOR, enabling differentiability in the model as shown in Table I. Such a representation allows us to relax the discrete operations into continuous ones.

With the probabilistic model derived from replacing the discrete logical operators of the multi-level, multi-output Boolean function with their probabilistic counterparts, we can now learn satisfying primary input variables given the constraints on the output variables. To find satisfying solutions, we define the primary input variables a matrix $\mathbf{V} \in \mathbb{R}^{b \times n}$, where n indicates the number of primary input variables and b denotes the batch size.

Table I: The probabilistic representation of logical operators.

Operator	Input Variable	Output Variable	Derivative w.r.t Input
NOT	P_1	$P_y = \overline{P_1} = 1 - P_1$	$\frac{\partial P_y}{\partial P_1} = -1$
AND	P_1, P_2	$P_y = P_1\ P_2$	$\frac{\partial P_y}{\partial P_1} = P_2$, $\frac{\partial P_y}{\partial P_2} = P_1$
OR	P_1, P_2	$P_y = 1 - \overline{P_1}\ \overline{P_2}$	$\frac{\partial P_y}{\partial P_1} = \overline{P_2}$, $\frac{\partial P_y}{\partial P_2} = \overline{P_1}$
XOR	P_1, P_2	$P_y = \overline{P_1}\ P_2 + P_1\ \overline{P_2}$	$\frac{\partial P_y}{\partial P_1} = 1 - 2P_2$, $\frac{\partial P_y}{\partial P_2} = 1 - 2P_1$
XNOR	P_1, P_2	$P_y = P_1\ P_2 + \overline{P_1}\ \overline{P_2}$	$\frac{\partial P_y}{\partial P_1} = 2P_2 - 1$, $\frac{\partial P_y}{\partial P_2} = 2P_1 - 1$

Since our continuous multi-level, multi-output Boolean function takes real values between 0 and 1 in a form of probability, we convert the primary input variables into probabilities using the sigmoid function $\sigma(\cdot)$, such that:

$$\mathbf{P} = \sigma(\mathbf{V}), \tag{6}$$

where $\mathbf{P} \in [0, 1]^{b \times n}$ represents the input probabilities fed into the model. We refer to this conversion process as a form of continuous embedding, as it transforms the input space into a continuous probability space, allowing the model to interpret the inputs probabilistically. The model's primary outputs are then computed as:

$$\mathbf{Y} = \mathcal{F}(\mathbf{P}), \tag{7}$$

where $\mathcal{F} : [0, 1]^{b \times n} \rightarrow [0, 1]^{b \times m}$ represents the probabilistic multi-level, multi-output Boolean function. The matrix $\mathbf{Y} \in [0, 1]^{b \times m}$ holds the m primary output variables across the b data batches.

To optimize the inputs, we define an ℓ_2-loss function $\mathcal{L}$ that measures the difference between the computed outputs $\mathbf{Y}$ and the target output matrix $\mathbf{T} \in 0, 1^{b \times m}$:

$$\mathcal{L} = \sum_{b,m} ||\mathbf{Y} - \mathbf{T}||_2^2 . \tag{8}$$

Table II: The runtime performance of our sampling method against UNIGEN3, CMSGEN and DIFFSAMPLER in terms of unique solution throughput, where each sampler is tasked to generate a minimum of 1000 solutions within a timeout (TO) of 2 hours.

SAT Instance	# Primary Inputs	# Primary Outputs	# Variables (CNF)	# Clauses (CNF)	Throughput (# Unique Solutions per Second) **This work** (Speedup)	UNIGEN3	CMSGEN	DIFFSAMPLER
or-50-10-7-UC-10	50	4	100	254	**5,974,780.8** (79.6×)	64.7	36,693.5	75,040.1
or-60-20-10-UC-10	60	5	120	305	**4,777,137.7** (86.0×)	81.7	33,987.0	55,521.3
or-70-5-5-UC-10	69	7	140	357	**2,468,613.4** (77.8×)	94.5	31,732.4	16,035.1
or-100-20-8-UC-10	98	10	200	510	**1,707,142.3** (51.6×)	43.4	22,951.7	33,175.3
75-10-1-q	83	1	452	443	**478,723.0** (42.0×)	1.6	11,281.8	156.1
75-10-10-q	79	1	456	439	**2,075,175.0** (197.1×)	1.6	10,527.4	251.8
90-10-1-q	51	1	432	411	**2,809,981.5** (251.7×)	1.0	11,162.5	227.9
90-10-10-q	31	1	428	391	**3,567,035.2** (326.9×)	1.4	10,913.0	57.9
s15850a_3_2	600	3	10,908	24,476	**20,267.1** (47.1×)	0.4	430.4	TO
s15850a_7_4	600	7	10,926	24,552	**14,930.5** (34.1×)	0.5	437.9	TO
s15850a_15_7	600	15	10,995	24,836	**14,177.1** (33.6×)	0.5	422.2	TO
Prod-8	293	2	14,952	74,702	**994.9** (523.6×)	1.9	0.2	TO
Prod-20	677	2	37,320	186,734	**139.1** (347.8×)	0.4	TO	TO
Prod-32	1061	2	59,688	298,766	**96.0** (480×)	0.2	TO	TO

By minimizing this loss function through GD, the primary input variables (i.e., $\mathbf{V}$) are iteratively updated. Upon convergence, the b solutions are determined by converting the soft input values (i.e., $\mathbf{V}$) into hard binary values (i.e., $\widetilde{\mathbf{V}} \in 0,1^{b\times n}$).

In gradient descent optimization, the objective is to compute the derivative of the loss function with respect to each primary input variable associated with the constrained paths. This can be achieved using the chain rule, leveraging the derivatives of the logical operators presented in Table I. For instance, in the constrained path highlighted in red in Fig. 1(b), the derivative of the loss function with respect to the primary input variable x_{13} can be expressed as follows:

$$\frac{\partial \mathcal{L}}{\partial x_{13}} = \frac{\partial \mathcal{L}}{\partial x_{10}} \cdot \frac{\partial x_{10}}{\partial x_{13}} = 2 \cdot (x_{10} - 1) \cdot (1 - x_9 \cdot x_{14}) \cdot (1 - x_9). \quad (9)$$

For simplicity, we will exclude the embedding process from our calculations in this example. With the computed derivative, the value of x_{13} is updated using the following equation:

$$x_{13} = x_{13} - \gamma \cdot \frac{\partial \mathcal{L}}{\partial x_{13}}, \quad (10)$$

where γ represents the learning rate.

The overall workflow is illustrated in Fig. 1. Our method integrates a parser that describes the probabilistic multi-level, multi-output Boolean function in PyTorch. Since each solution is processed and learned independent from others, our approach is highly parallelizable and benefits from GPU acceleration, enables fast and scalable sampling by processing multiple data batches simultaneously.

IV. EXPERIMENTAL RESULTS

In this section, we demonstrate the effectiveness of our sampling technique. To accomplish this, we developed a prototype of our approach using PyTorch, an open-source machine learning library that integrates Torch's powerful GPU-optimized backend with a Python-friendly interface. For a thorough assessment, we evaluated 60 problem instances of varying sizes from a public domain benchmark suite. The experiments were performed on a system featuring an Intel Xeon E5-2698 processor running at 2.2 GHz and 8 NVIDIA V100 GPUs, each with 32 GB of memory. We present the runtime performance of our method in terms of throughput, defined as the number of unique and valid solutions generated per second, using a single NVIDIA V100 GPU. GD was employed as the optimizer during the training phase, with the learning rate set to 10 and the number of iterations to 5. We varied the batch size between 100 and 1,000,000, based on the specific instances tested.

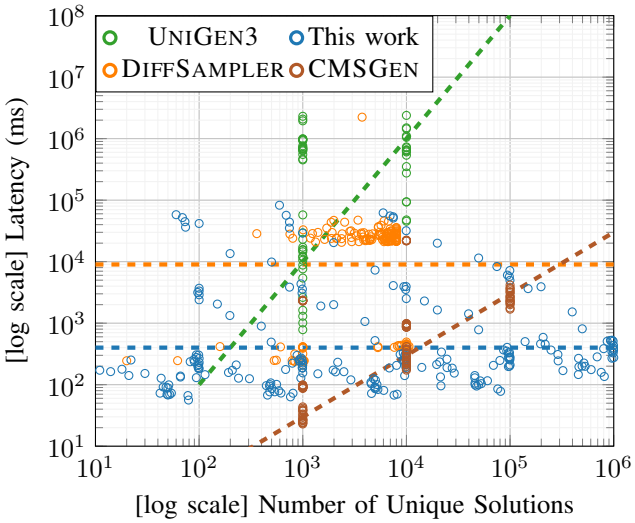

Figure 2: A log-log plot showing the runtime in milliseconds versus the number of unique satisfying solutions across all 60 instances from the sampling benchmark. The dotted lines represent the performance trends for each sampler.

We compare the performance of our sampling method with leading SAT sampling methods, specifically UNIGEN3 [35], CMSGEN [36], and DIFFSAMPLER [37]. These samplers operate directly on the CNF of SAT instances, whereas our method handles the simplified multi-level, multi-output Boolean expressions derived from transforming their logical constraints. Both UNIGEN3 and CMSGEN are highly optimized implementations written in C++, while DIFFSAMPLER is a Python-based, GPU-accelerated SAT sampler built using the high-performance JAX library. UNIGEN3 and CMSGEN were tested on a server-grade Intel Xeon Gold 6134 CPU with 3.2 GHz clock rate and 1TB of RAM. DIFFSAMPLER was run on a system featuring an Intel Xeon E5-2698 processor at 2.2 GHz and 8 NVIDIA V100 GPUs, each equipped with 32 GB of memory.

A. Runtime Performance

Table II presents the sampling performance of our method in terms of throughput for a representative subset of 14 instances from the sampling benchmark. Throughput is defined as the

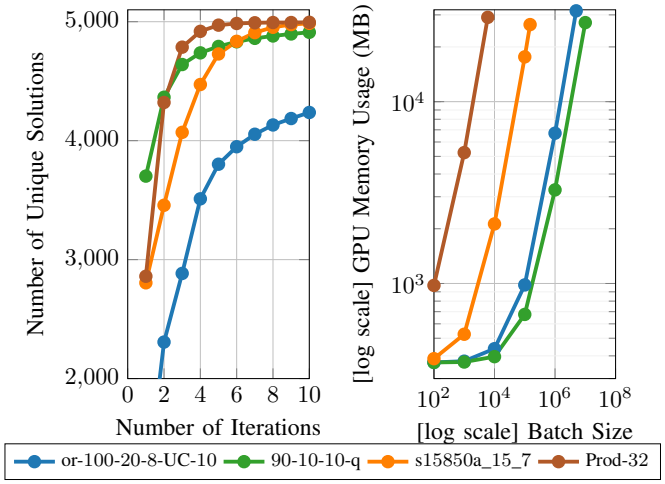

Figure 3: (Left) Learning curve showing the number of unique satisfying solutions over iterations. (Right) Log-log plot of GPU memory usage (MB), measured by "nvidia-smi", across different batch sizes for a subset of 4 CNF instances.

number of unique solutions generated per second. Each sampler is required to produce at least $1,000$ solutions within a maximum runtime of 2 hours. The table shows the best throughput results obtained from each sampler. Our method consistently outperforms state-of-the-art samplers in unique solution throughput, achieving speedups ranging from $33.6\times$ to $523.6\times$, depending on the under-test SAT instances. This substantial performance improvement is due to two key factors. First, many logical constraints in the CNF representation are satisfied during the transformation, where sub-clauses are converted into simplified Boolean sub-expressions. This transforms the SAT sampling task into solving constrained paths in a simplified, multi-level, multi-output Boolean expression, significantly reducing the number of logical operations. Second, by framing the sampling problem as a learning task, where the computation of each sample is independent, GPU acceleration can be leveraged to further enhance runtime performance.

Fig. 2 demonstrates how the runtime performance of each sampler varies as the number of unique solutions increases. A critical aspect of this comparison is the overall efficiency of our sampling method relative to UNIGEN3, CMSGEN, and DIFFSAMPLER. This is particularly evident when sampling larger quantities of solutions, where the runtime shows only a slight increase as the solution count grows.

B. Learning Dynamics

We analyze the learning dynamics of our sampling method, focusing on hyper-parameters such as iterations and batch size. Fig. 3 shows the progress in generating unique solutions over 10 iterations, where the number of unique solutions increases with more iterations. Increasing the batch size improves runtime performance by leveraging GPU parallelism, but at the cost of higher memory usage. The GPU memory demand, as shown in Fig. 3, grows with both the complexity of the Boolean function derived from the CNF transformation and the batch size. In scenarios requiring a high number of unique samples with limited GPU memory, the practical solutions are to either run more iterations, reducing throughput, or use GPUs with larger memory.

C. Ablation Study

In this section, we analyze the extent of GPU acceleration by comparing the runtime performance of our sampler between the GPU and CPU, as presented in Fig. 4. The data shows that GPU execution results in an average speedup of $6.8\times$ over CPU execution. Additionally, we provide the rate of reduction in the number of bit-wise operations due to the transformation, measured as the number of operations in the CNF divided by the number of operations in the resulting multi-level, multi-output Boolean function in terms of 2-input gate equivalents in Fig. 4, demonstrating an average reduction of $4.2\times$. Finally, we present the transformation time required to convert CNF into a multi-level, multi-output Boolean function in Fig. 4. This conversion time is comparable to that of conversion time of SAT applications represented in higher-level logical formats into CNF. Given the superior runtime performance of our sampler, we suggest that SAT applications in high-level logical formats could be directly transformed into a multi-level, multi-output Boolean function.

V. RELATED WORK

Numerous SAT formula sampling methods have been explored in prior research. For example, UNIGEN3 provides approximate uniformity guarantees [43], while both CMSGEN and QUICKSAMPLER [1] emphasize sampling efficiency. Other studies have also examined the use of data-parallel hardware for SAT solving, primarily focusing on parallelizing CDCL and various heuristic-based algorithms [44], [45]. Some recent efforts, such as MATSAT [46] and NEUROSAT [47], have attempted to frame SAT instances as constrained numerical optimization problems. However, these methods have struggled to demonstrate the effectiveness of GPU-accelerated formula sampling on large and diverse standard benchmarks, typically focusing on small, random instances. Recently, a new differentiable sampling technique called DIFFSAMPLER was proposed in [37], enabling GPU-accelerated SAT sampling on standard benchmarks and achieving competitive runtime performance compared to UNIGEN3 and CMSGEN. DEMOTIC is another GPU-accelerated, differentiable sampler specifically designed for CircuitSAT problems in CRV. It operates directly on circuit structures described in hardware description languages such as Verilog [48].

There have been only a few attempts in the literature that focus solely on extracting circuit structure from CNF descriptions [38], [49], primarily to recover information lost during the Tseitin transformation. These approaches typically rely on pattern matching based on predefined gate structures. In contrast, our transformation method is more general, capable of restoring any combination of logical gates from CNF, and, more importantly, it enables high-throughput SAT sampling using GPUs.

VI. CONCLUSION

In this paper, we have introduced a novel GPU-accelerated approach for SAT sampling that significantly outperforms traditional methods. By transforming CNF representations into simplified multi-level, multi-output Boolean functions and employing gradient-based optimization, our method reinterprets SAT as a supervised multi-output regression task. This enables parallel, bit-wise operations across tensor elements, leading to substantial runtime improvements. With speedups ranging from $33.6\times$ to $523.6\times$ compared to existing heuristic samplers, our extensive evaluation on 60 benchmark instances demonstrates the effectiveness and efficiency of this new technique for SAT

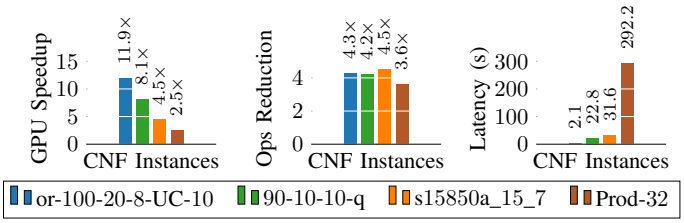

Figure 4: Comparison of GPU speedup over CPU (left), reduction rate of bit-wise operations in 2-input gate equivalents (middle), and transformation time from CNF to a simplified multi-level, multi-output Boolean function (right) for a subset of 4 instances.

sampling. This performance gain is primarily attributed to the GPU's acceleration over CPU execution and the reduction in the number of logic operations resulting from our transformation.

References

[1] R. Dutra *et al.*, "Efficient sampling of sat solutions for testing," in *Proc. of the International Conference on Software Engineering*, 2018.

[2] L. A. Clarke, "A program testing system," in *Proceedings of the 1976 Annual Conference*, ser. ACM '76. New York, NY, USA: Association for Computing Machinery, 1976, p. 488–491.

[3] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, p. 385–394, jul 1976.

[4] T. Avgerinos *et al.*, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, 2014, p. 1083–1094.

[5] S. Anand *et al.*, "Heap cloning: Enabling dynamic symbolic execution of java programs," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, 2011, pp. 33–42.

[6] ——, "Jpf-se: a symbolic execution extension to java pathfinder," in *Proceedings of the 13th Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. TACAS'07, 2007, p. 134–138.

[7] S. Artzi *et al.*, "Finding bugs in dynamic web applications," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08, 2008, p. 261–272.

[8] J. Burnim *et al.*, "Heuristics for scalable dynamic test generation," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 443–446.

[9] C. Cadar *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[10] V. Chipounov *et al.*, "The s2e platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, feb 2012.

[11] P. Godefroid *et al.*, "Dart: directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, 2005, p. 213–223.

[12] K. Jayaraman *et al.*, "jfuzz: A concolic whitebox fuzzer for java," in *NASA Formal Methods*, 2009.

[13] H. Yoshida *et al.*, "Klover: Automatic test generation for c and c++ programs, using symbolic execution," *IEEE Software*, vol. 34, no. 5, pp. 30–37, 2017.

[14] C. S. Pundefinedsundefinedreanu *et al.*, "Combining unit-level symbolic execution and system-level concrete execution for testing nasa software," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08, 2008, p. 15–26.

[15] P. Saxena *et al.*, "A symbolic execution framework for javascript," in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 513–528.

[16] D. Song *et al.*, "Bitblaze: A new approach to computer security via binary analysis," in *Information Systems Security*, R. Sekar *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–25.

[17] N. Tillmann *et al.*, "Pex–white box test generation for .net," in *Tests and Proofs*. Springer Berlin Heidelberg, 2008, pp. 134–153.

[18] N. Kitchen *et al.*, "Stimulus generation for constrained random simulation," in *2007 IEEE/ACM International Conference on Computer-Aided Design*, 2007, pp. 258–265.

[19] Y. Zhao *et al.*, "Random stimulus generation with self-tuning," in *2009 13th International Conference on Computer Supported Cooperative Work in Design*, 2009, pp. 62–65.

[20] R. Naveh *et al.*, "Beyond feasibility: Cp usage in constrained-random functional hardware verification," in *Principles and Practice of Constraint Programming*, C. Schulte, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 823–831.

[21] Y. Naveh *et al.*, "Constraint-based random stimuli generation for hardware verification," in *Proceedings of the 18th Conference on Innovative Applications of Artificial Intelligence - Volume 2*, ser. IAAI'06. AAAI Press, 2006, p. 1720–1727.

[22] *Handbook of Satisfiability*, ser. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, 2009.

[23] N. Eén *et al.*, "An extensible sat-solver." in *SAT*, ser. Lecture Notes in Computer Science, vol. 2919. Springer, 2003, pp. 502–518.

[24] M. Moskewicz *et al.*, "Chaff: engineering an efficient sat solver," in *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, 2001, pp. 530–535.

[25] G. Audemard *et al.*, "On the glucose sat solver," *International Journal on Artificial Intelligence Tools*, vol. 27, no. 01, p. 1840001, 2018.

[26] J. Marques Silva *et al.*, "Grasp-a new search algorithm for satisfiability," in *Proceedings of International Conference on Computer Aided Design*, 1996, pp. 220–227.

[27] J. Marques-Silva *et al.*, *Chapter 4: Conflict-driven clause learning SAT solvers*, ser. Frontiers in Artificial Intelligence and Applications. IOS Press BV, 2021, pp. 133–182, publisher Copyright: © 2021 The authors and IOS Press. All rights reserved.

[28] K. S. Meel, "Model counting and uniform sampling instances," May 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3793090

[29] C. Barrett, "Decision procedures: An algorithmic point of view," *Journal of Automated Reasoning*, vol. 51, no. 4, pp. 453–456, Dec. 2013.

[30] G. S. Tseitin, "On the complexity of derivation in propositional calculus," *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pp. 466–483, 1983.

[31] M. Davis *et al.*, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, p. 394–397, jul 1962.

[32] B. Selman *et al.*, "Local search strategies for satisfiability testing." *Cliques, coloring, and satisfiability*, vol. 26, pp. 521–532, 1993.

[33] R. Impagliazzo *et al.*, "Does Looking Inside a Circuit Help?" in *42nd International Symposium on Mathematical Foundations of Computer Science (MFCS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 83, 2017, pp. 1:1–1:13.

[34] N. Kitchen *et al.*, "A markov chain monte carlo sampler for mixed boolean/integer constraints," in *Computer Aided Verification: 21st International Conference, CAV 2009, Grenoble, France, June 26-July 2, 2009. Proceedings 21*. Springer, 2009, pp. 446–461.

[35] M. Soos *et al.*, "Tinted, detached, and lazy cnf-xor solving and its applications to counting and sampling," in *Proceedings of International Conference on Computer-Aided Verification (CAV)*, 2020.

[36] P. Golia *et al.*, "Designing samplers is easy: The boon of testers," in *Proc. of Formal Methods in Computer-Aided Design (FMCAD)*, 2021.

[37] A. Ardakani *et al.*, "Late breaking results: Differential and massively parallel sampling of sat formulas," in *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*, 2024.

[38] J. A. Roy *et al.*, "Restoring circuit structure from sat instances," in *proceedings of international workshop on Logic and synthesis*. Citeseer, 2004, pp. 663–678.

[39] A. Meurer *et al.*, "Sympy: symbolic computing in python," *PeerJ Computer Science*, vol. 3, p. e103, Jan. 2017. [Online]. Available: https://doi.org/10.7717/peerj-cs.103

[40] R. Brayton *et al.*, "Abc: An academic industrial-strength verification tool," in *Computer Aided Verification*, T. Touili *et al.*, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 24–40.

[41] A. Mishchenko *et al.*, "Dag-aware aig rewriting a fresh look at combinational logic synthesis," in *Proceedings of the 43rd Annual Design Automation Conference*, ser. DAC '06, 2006, p. 532–535.

[42] ——, "Scalable don't-care-based logic optimization and resynthesis," *ACM Trans. Reconfigurable Technol. Syst.*, dec 2011.

[43] Y. Pote *et al.*, "On scalable testing of samplers," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

[44] C. Costa, "Parallelization of sat algorithms on gpus," Technical report, INESC-ID, Technical University of Lisbon, Tech. Rep., 2013.

[45] M. Osama *et al.*, "Sat solving with gpu accelerated inprocessing," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2021, pp. 133–151.

[46] T. Sato *et al.*, "Matsat: a matrix-based differentiable sat solver," *arXiv preprint arXiv:2108.06481*, 2021.

[47] S. Amizadeh *et al.*, "Learning to solve circuit-sat: An unsupervised differentiable approach," in *International Conference on Learning Representations*, 2018.

[48] A. Ardakani *et al.*, "DEMOTIC: A differentiable sampler for multi-level digital circuits," in *Proceedings of the 30th Asia and South Pacific Design Automation Conference (ASP-DAC 2025)*, 2025.

[49] Z. Fu *et al.*, "Extracting logic circuit structure from conjunctive normal form descriptions," *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID'07)*, pp. 37–42, 2007.