Accelerating Quantum Circuit Simulation with Symbolic Execution and Loop Summarization

Tian-Fu Chen
d11k42001@ntu.edu.tw
Grad. School of Advanced Technology
National Taiwan University
Taipei, Taiwan

Yu-Fang Chen
gulu0724@gmail.com
Institute of Information Science
Academia Sinica
Taipei, Taiwan

Jie-Hong Roland Jiang
jhjiang@ntu.edu.tw
Grad. Inst. of Electronics Engineering
National Taiwan University
Taipei, Taiwan

Sára Jobranová
xjobra01@stud.fit.vutbr.cz
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic

Ondřej Lengál
lengal@fit.vutbr.cz
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic

ABSTRACT

Quantum circuit simulation is the basic tool for reasoning over quantum programs. Despite the tremendous advance in the simulator technology in the recent years, the performance of simulators is still unsatisfactory on non-trivial circuits, which slows down the development of new quantum systems. In this work, we develop a loop summarizing simulator based on multi-terminal binary decision diagrams (MTBDDs) with efficiently customized quantum gate operations. The simulator is capable of automatic loop summarization using symbolic execution, which saves repetitive computation for circuits with iterative structures. Experimental results show the simulator outperforms state-of-the-art simulators on some standard circuits, such as Grover's algorithm, by several orders of magnitude.

ACM Reference Format:
Tian-Fu Chen, Yu-Fang Chen, Jie-Hong Roland Jiang, Sára Jobranová, and Ondřej Lengál. 2024. Accelerating Quantum Circuit Simulation with Symbolic Execution and Loop Summarization . In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '24), October 27–31, 2024, New York, NY, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3676536.3676711

1 INTRODUCTION

The development of quantum computers started in 1980s with the promise to solve problems challenging for classical computers. Later, quantum algorithms more efficient than their best classical counterparts for certain problems started appearing, such as Shor's algorithm for integer factoring [27] or Grover's algorithm for search in an unstructured database [19]. With multiple major players investing into quantum and the consistent improvement of the hardware, it seems that quantum computers will occupy a prominent role in the future. The development of quantum algorithms is an extremely challenging task so adequate computer-aided support is needed for debugging and reasoning over quantum programs.

Debugging quantum programs is primarily done through *simulation*, which is considerably more challenging in the quantum world as compared to the classical world. This is because, in the quantum world, we need to keep track of a potentially exponentially sized *quantum state* that assigns *every* classical state a complex *amplitude* instead of keeping track of a *single* evolving classical program state.

Simulators of quantum programs have advanced tremendously in recent years, moving from the basic vector- and matrix-based representation [26] into representations based on decision diagrams [25, 28, 30, 32, 33, 36], graphical languages [14], or model counting [24]. Despite this advance, simulating quantum circuits of a moderate size is still considered infeasible. Therefore, faster simulators are needed to provide quantum developers with basic means to observe behaviour of quantum programs.

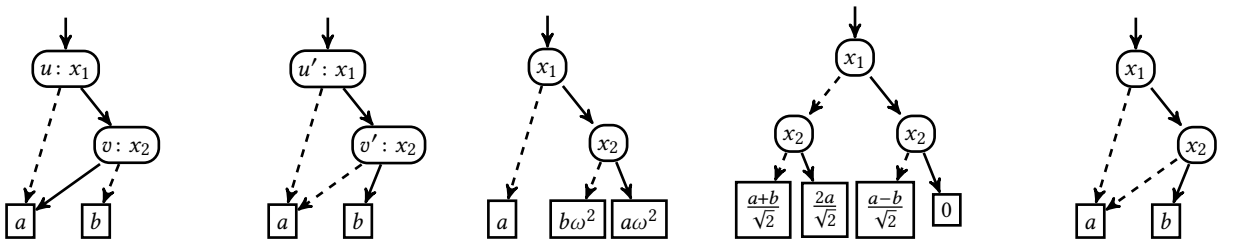
In this paper, we focus on accelerating the simulation of quantum circuits that contain repetition of some sub-structure. Some notable examples of such circuits include *Grover's search* [19], *period finding* [23], and *quantum counting* [8]. Current standards for describing quantum circuits, such as the OpenQASM 3.0 format [15], allow describing such repeated sub-structures compactly using loops or hierarchical gate definitions.

Our method for accelerating simulation involves computing a *symbolic summary* of a sequence of quantum gates that occur repeatedly, such as a loop body or the definition of a hierarchical gate. This summary is computed with respect to a particular quantum state and can be reused to execute the sequence of quantum gates from any state that shares the same high-level structure, i.e., computational bases with the same amplitudes in the first state will also have the same amplitudes in the second state, though these amplitude values may differ from those in the first state. We derive these summaries using *symbolic execution*, which is similar to standard quantum simulation but instead computes symbolic terms that remember the arithmetic operations to be performed, rather than computing the results of arithmetic operations over numbers.

Moreover, similarly to [30], we represent quantum states algebraically for exact simulation without numerical precision loss, which is crucial in tasks such as equivalence checking [34]. Unlike [30], which works only for concrete value simulation, ours

ICCAD '24, October 27–31, 2024, New York, NY, USA

ACM ISBN 979-8-4007-1077-3/24/10
https://doi.org/10.1145/3676536.3676711



(a) MTBDD M_q for q **(b) Applying X_2 to q** **(c) Applying S_1 to q** **(d) Applying H_1 to q** **(e) Applying CNOT_2^1 to q**

Figure 1: Examples of applying quantum gates on MTBDD-based representation of the state $q = a\,|00\rangle + a\,|01\rangle + b\,|10\rangle + a\,|11\rangle$.

allows symbolic simulation thanks to the use of *multi-terminal binary decision diagrams* (MTBDDs) [4, 9, 16]. We customize MTBDD procedures for efficient quantum gate execution instead of using only standard MTBDD functions *Apply* and *Restrict* as usual.

Our experimental evaluation shows that our proposed approach can significantly speed up simulation for some well-established quantum circuits. This allows us to tackle circuits of sizes that were previously considered infeasible.

2 PRELIMINARIES

$\mathbb{B} = \{0, 1\}$ denotes the Booleans. We fix a set $\mathbb{X} = \{x_1, \ldots, x_n\}$ of Boolean variables with an order $x_1 < x_2 < \cdots < x_n$; we use $\vec{x}$ to denote $(x_1, \ldots, x_n)$. Given an arbitrary set $S \neq \emptyset$, a *pseudo-Boolean* function is a function $f\colon \mathbb{B}^n \to S$; f is a *Boolean* function if $S = \mathbb{B}$. ω denotes the complex number $e^{\frac{i\pi}{4}}$, i.e., the unit vector that makes an angle of 45° with the positive real axis in the complex plane.

2.1 Decision diagrams

Given an arbitrary nonempty set S with finitely representable elements, a *multi-terminal binary decision diagram* (MTBDD) [4, 9, 16] is a graph $G = (N, T, low, high, root, var)$ where N is the set of *internal nodes*, $T \subseteq S$ is the set of *leaf nodes* ($T \cap N = \emptyset$, $T \neq \emptyset$), $low, high\colon N \to (N \cup T)$ are the *low*- and *high*-successor edges, $root \in N \cup T$ is the *root* node, and $var\colon N \to \mathbb{X}$ is the node-variable mapping, with the following three restrictions:

(i) (connectivity) every node from $N \cup T$ is reachable from *root* over some sequence of *low* and *high* edges,
(ii) (order) for every $u, v \in N$, if $low(u) = v$ or $high(u) = v$, then $var(u) < var(v)$, and
(iii) (reducedness) there is no node $u \in N$ s.t. $low(u) = high(u)$.

Each node $v \in N \cup T$ represents a pseudo-Boolean function $[\![v]\!]$ defined inductively as follows: (1) if $v \in T$, then $[\![v]\!](\vec{x}) = v$, and (2) if $v \in N$ and $var(v) = x_i$, then

$$[\![v]\!](\vec{x}) = \begin{cases} [\![low(v)]\!](\vec{x}) & \text{if } x_i = 0 \text{ and} \\ [\![high(v)]\!](\vec{x}) & \text{if } x_i = 1. \end{cases}$$

Moreover, we impose the following additional restriction on G:

(iv) (canonicity) there are no two nodes $u \neq v$ such that $[\![u]\!] = [\![v]\!]$.

G then represents the function $[\![G]\!]$ defined as $[\![root]\!]$. We abuse notation and confuse a function with the MTBDD representing it and use a node r to denote the MTBDD rooted in r and vice versa.

We will use the following standard MTBDD operations. The `apply`(f_1, f_2, op_2) operation is used to combine two MTBDDs f_1 and f_2 through a binary operation $op_2\colon S \times S \to S$ performed on the corresponding leaf notes, obtaining the MTBDD representing the pseudo-Boolean function $\{\vec{x} \mapsto op_2(f_1(\vec{x}), f_2(\vec{x})) \mid \vec{x} \in \mathbb{B}^n\}$. The `monadic_apply`$(f, op)$ operation updates the leaves of the MTBDD f with a unary operation $op_1\colon S \to S$, obtaining the MTBDD representing the pseudo-Boolean function $\{\vec{x} \mapsto op_1(f(\vec{x})) \mid \vec{x} \in \mathbb{B}^n\}$. We often use lambda expression for defining $op_{1/2}$. Additionally, MTBDDs provide the `spawn`(l, h, x) function that works as follows: (i) if $l = h$, then the result is l,otherwise (ii) the result is the unique node n such that $low(n) = l$, $high(n) = h$, and $var(n) = x$.

2.2 Quantum Computing Fundamentals

Quantum computers are programmed through *quantum gates*, which update the global *quantum state*. A *quantum circuit* is a sequence of gates, combined with programming constructs like *loops* or *hierarchical gate definitions* that allow a more concise presentation [15].

Quantum states: In a traditional computer system with n bits, a state is represented by n Booleans. In the quantum world, such states are called *computational basis states*. E.g., in a system with three bits labeled x_1, x_2, and x_3, the computational basis state $|011\rangle$ indicates that the value of x_1 is 0 and the values of x_2 and x_3 are 1.

In a quantum system, an n-qubit *quantum state* is a probabilistic distribution over n-bit basis states, denoted either as a column vector $(a_0, \ldots, a_{2^n-1})^T$ (given here as a transposed row vector) or as a formal sum $\sum_{j \in \{0,1\}^n} a_j \cdot |j\rangle$, where $a_0, a_1, \ldots, a_{2^n-1} \in \mathbb{C}$ are *complex amplitudes* satisfying the property that $\sum_{j \in \{0,1\}^n} |a_j|^2 = 1$. Intuitively, $|a_j|^2$ is the probability that when we measure the quantum state in the computational basis, we obtain the classical state $|j\rangle$; these probabilities must sum up to 1 for all basis states. We can view a quantum state as a function mapping each basis state in $\mathbb{B}^n$ to a complex amplitude and represent them using MTBDDs; cf. Figure 1a for an MTBDD M_q representing the state $q = a\,|00\rangle + a\,|01\rangle + b\,|10\rangle + a\,|11\rangle$ (for some $a, b \in \mathbb{C}$ s.t. $a \neq b$ and $3|a|^2 + |b|^2 = 1$).

Quantum gates: Two main types of quantum gates are being used: *single-qubit gates* and *controlled gates*. We support all commonly used gates except the arbitrary rotation single-qubit gate due to the use a precise complex number representation (cf. Sec. 5).

Single-qubit gates. In general, a single-qubit gate is presented as a *unitary complex matrix*. We directly support the following gates:

$$\mathrm{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad \mathrm{Y} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad \mathrm{Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

$$\mathrm{S} = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \quad \mathrm{T} = \begin{pmatrix} 1 & 0 \\ 0 & \omega \end{pmatrix}, \quad \mathrm{H} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix},$$

$$\mathrm{R_X}\left(\frac{\pi}{2}\right) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -i \\ -i & 1 \end{pmatrix}, \quad \mathrm{R_Y}\left(\frac{\pi}{2}\right) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}.$$

Algorithm 1: Execution of a single-qubit gate U_t

Input: MTBDD $M_q = (N, T, low, high, root, var)$, target qubit x_t, single qubit gate U
Output: MTBDD representing $U_t(M_q)$

```
1  return recurse(root);
2  Function recurse(node)
3      l ← low(node); h ← high(node); x_i ← var(node);
4      if i < t then
5          l_new ← recurse(l); h_new ← recurse(h);
6          return spawn(l_new, h_new, x_i);
7      else // i ≥ t or a leaf
8          if i = t then l' ← l; h' ← h ;
9          else l' ← h' ← node;
10         if U = X then return spawn(h', l', x_t) ;
11         if U ∈ {T, S, Z} then
12             if U = T then c ← ω;
13             if U = S then c ← ω^2;
14             if U = Z then c ← −1;
15             h_new ←monadic_apply(h', λx(c · x));
16             return spawn(l', h_new, x_t);
17         if U = Y then
18             l_new ←monadic_apply(h', λx(−ω^2 · x));
19             h_new ←monadic_apply(l', λx(ω^2 · x));
20             return spawn(l_new, h_new, x_t);
21         if U = H then
22             l_new ←apply(l', h', λx, y(1/√2 · (x + y)));
23             h_new ←apply(l', h', λx, y(1/√2 · (x − y)));
24             return spawn(l_new, h_new, x_t);
25         if U = R_X(π/2) then
26             l_new ←apply(l', h', λx, y(1/√2 · (x − ω^2 · y)));
27             h_new ←apply(l', h', λx, y(1/√2 · (y − ω^2 · x)));
28             return spawn(l_new, h_new, x_t);
29         if U = R_Y(π/2) then
30             l_new ←apply(l', h', λx, y(1/√2 · (x − y)));
31             h_new ←apply(l', h', λx, y(1/√2 · (x + y)));
32             return spawn(l_new, h_new, x_t);
```

For a single-qubit gate U, we often use a subscript to denote the qubit that it is applied to, e.g., U_i means we apply U to qubit x_i.

The X gate is the quantum "*negation*" gate. Applying gate X to a single-qubit state $\begin{pmatrix} l \\ h \end{pmatrix}$ produces the state $X \cdot \begin{pmatrix} l \\ h \end{pmatrix} = \begin{pmatrix} h \\ l \end{pmatrix}$. In the case of an MTBDD-based representation of $\begin{pmatrix} l \\ h \end{pmatrix}$, which would have a root node with the *low*-successor $l \in T$ and *high*-successor $h \in T$, this would effectively mean swapping the *low* and *high* successors of the root. For the general case, applying X_i to a quantum state's MTBDD swaps the high and low-successor edges of all nodes at level i. See Figure 1b for an example of applying X_2 to the MTBDD M_q introduced above (the edges leaving v' got swapped).

Algorithm 2: Execution of a controlled gate CU_t^c

Input: MTBDD $M = (N, T, low, high, root, var)$, control qubit x_c, target qubit x_t, single qubit gate U
Output: MTBDD representing $CU_t^c(M_q)$

```
1  M_l ← recurse(root, L);
2  M_h ← recurse(U_t(M_q), H);
3  return apply(M_l, M_h, λx, y(x + y));
4  Function recurse(node, dir)
5      l ← low(node); h ← high(node); x_i ← var(node);
6      if i < c then
7          l_new ← recurse(l, dir); h_new ← recurse(h, dir);
8          return spawn(l_new, h_new, x_i);
9      else // i ≥ c or a leaf
10         if i = c then l' ← l; h' ← h ;
11         else l' ← h' ← node;
12         if dir = L then return spawn(l', 0, x_c) ;
13         else return spawn(0, h', x_c) ;
```

Behaviours of Z, S, and T gates are similar to each other. In particular, applying the gates to $\begin{pmatrix} l \\ h \end{pmatrix}$ produces the states $Z \cdot \begin{pmatrix} l \\ h \end{pmatrix} = \begin{pmatrix} l \\ -h \end{pmatrix}$, $S \cdot \begin{pmatrix} l \\ h \end{pmatrix} = \begin{pmatrix} l \\ i \cdot h \end{pmatrix}$, and $T \cdot \begin{pmatrix} l \\ h \end{pmatrix} = \begin{pmatrix} l \\ \omega \cdot h \end{pmatrix}$, which multiply the $|1\rangle$-position with -1, i, and ω, respectively. Similarly, applying Z, S, and T to a quantum state's MTBDD multiplies all leaves in the *high*-subtrees of all nodes at level i with -1, i, and ω, respectively (cf. Figure 1c for an example of applying S_1 to M_q).

The last group of single-qubit gates we mention includes H (the *Hadamard* gate), $R_X(\frac{\pi}{2})$, and $R_Y(\frac{\pi}{2})$. These gates are more challening for implementation, since they fuse the amplitudes of the two basis states to form a new state. Taking H as an example, it updates the state $\begin{pmatrix} l \\ h \end{pmatrix}$ to the state $H \cdot \begin{pmatrix} l \\ h \end{pmatrix} = \frac{1}{\sqrt{2}} \cdot \begin{pmatrix} l+h \\ l-h \end{pmatrix}$. See Figure 1d for the result of applying H_1 to M_q. We refer the readers to Sec. 3 for the corresponding MTBDD constructions.

Controlled gates. A controlled gate CU uses another quantum gate U as its parameter. We often use CU_t^c to denote applying the controlled-gate with control qubit x_c and target qubit x_t. The effect of the controlled-U gate is that the gate U_t is applied only when the control qubit x_c has the value 1. For example, the controlled-X gate $CNOT_2^1$ has the control qubit x_1 and would apply X_2 when x_1 is valued 1. See Figure 1e for an example of applying $CNOT_2^1$ to M_q.

3 ALGORITHM FOR QUANTUM GATES

Single-qubit gates. In Algorithm 1, we present our procedure for applying *single-qubit gates* to an MTBDD $M_q = (N, T, low, high, root, var)$ at the target qubit x_t. The procedure performs the operations on M_q directly, as opposed to the standard approach (used, e.g., in SliQSim [30]), which uses only the standard (MT)BDD interface (in particular, functions *Apply* and *Restrict*).

The algorithm is as a modification of a standard `monadic_apply`. In particular, it performs a depth-first search (Line 5) until it reaches an x_t node, then it performs the semantic of the gate on the successors. The semantic differs for the particular gate, and was already briefly discussed in Sec. 2.2. We, however, need to be careful about *"don't care"* edges, i.e., edges that skip some variable in the MTBDD

(such as the *low* edge from u in Figure 1a). In such a situation, we need to stop the recursion and perform the gate operation by materializing the missing node (with the same *low* and *high*, cf. Line 9. E.g., when applying X_2 to the state q in Figure 1a, we have $l' = h' = a$ when handling the *low*-successor of u. Calling `spawn`(a, a, x_2) will just return the a leaf. On the other hand, $high(u')$ will be set to `spawn`$(high(v), low(v), x_2)$ = `spawn`$(a, b, x_2) = v'$.

To apply T, S, and Z gates, we use `monadic_apply` to multiply the leaf nodes of *high*-successors of the nodes labelled by x_t with ω, ω^2, and -1, respectively. When applying S_1, one step would be computing `monadic_apply`$(v, \lambda x(\omega^2 \cdot x))$ and connecting the result to *high* of the new root via the `spawn` function (Figure 1c). Meanwhile, the Y gate does for each node at level i the following: (1) it multiplies the *high* with $-\omega^2$ and sets it as the new *low*, and (2) it multiplies the *low* with ω^2 and sets it as the new *high*.

For each node at level i, applying the H, $R_X\left(\frac{\pi}{2}\right)$, or $R_Y\left(\frac{\pi}{2}\right)$ gates merges the *high* and *low*-successors using the `apply` function, creating new *high* and *low*-successors according to the gate's behaviour. In the case of the H gate, the new *low*-successor is `apply`$(l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x + y)))$ and the new *high*-successor is `apply`$(l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x - y)))$. When applying H_1 to the state q, we have $h' = v$ and $l' = a$. Fusing the two via `apply`$(l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x + y)))$ gives us the *low*-successor of the root in Figure 1d and via `apply`$(l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x - y)))$ gives us the *high*-successor of the root.

Controlled gates. Our procedure for applying *controlled*-U *gates* to M_q at the control qubit x_c for some quantum gate U is presented in Algorithm 2. The procedure involves three steps. First, in M_l, we will store a copy of M_q modified such that every base with $x_c = 1$ has amplitude 0 (Line 1). Second, we compute an MTBDD $U_t(M)$ using some of Algorithms 1 and 2 (depending on U, which can again be a controlled gate) and modify it such that every base with $x_c = 0$ has amplitude 0 (Line 2). Finally, both MTBDDs are summed up using the `apply` function (Line 3), which will, effectively, combine the two MTBDDs together (one operand of the + is always 0). Note that the *Toffoli* gate can be obtained by using the CNOT gate for U. A specialized more efficient version of the algorithm for phase gates (e.g., Z, S, T) can be used (omitted here due to space constraints).

Memoization. In order to avoid redundant computation, calls to the `recurse` functions in Algorithms 1 and 2 should be memoized.

Concrete execution and symbolic execution. Our gate operations work for both concrete and symbolic amplitude values. When leaf values are concrete, e.g., when $x = \frac{1}{2}$ and $y = \frac{1}{4}$, the function $\lambda x, y(\frac{1}{\sqrt{2}} \cdot (x + y)))$ will compute the value $\frac{1}{\sqrt{2}} \cdot (\frac{1}{2} + \frac{1}{4}) = \frac{3}{4\sqrt{2}}$. When leaf values are symbolic, e.g., , when $x = x_0$ and $y = y_0$, the same function will compute the symbolic term $\frac{1}{\sqrt{2}} \cdot (x_0 + y_0))$.

4 LOOP SUMMARIZATION

Our main contribution is an optimization that targets algorithms with loops[1], such as various *amplitude amplification* algorithms [7], with the most famous one being Grover's unstructured search [19]. The optimization is particularly effective in the case that the number of distinct amplitudes is small (which is the case for amplitude amplification algorithms, where there are typically only a limited number of different amplitudes at the beginning of a loop body, e.g., high amplitude, low amplitude, and zero).

[1]W.l.o.g., in the basic version of the optimization presented here, we assume the loop bodies are unitaries, i.e., do not contain measurements, and that they are not nested.

Algorithm 3: Loop summarization

Input: An MTBDD M_q, a loop body C
Output: An MTBDD M_α over $\mathbb{S}$ and a mapping $\tau\colon \mathbb{S} \to \mathbb{T}_\mathbb{S}$

1 $\alpha \leftarrow \emptyset$ (type $\alpha\colon \mathbb{C} \rightharpoonup \mathbb{S}$); `// init abstraction`
2 $M_\alpha^{refined} \leftarrow$ `monadic_apply`$(M_q,$ `abstract`$[\alpha])$;
3 **repeat**
4 $M_\alpha \leftarrow M_\alpha^{refined}$;
5 $M'_\alpha \leftarrow C_S(M_\alpha)$;
6 $\tau \leftarrow \emptyset$ (type $\tau\colon \mathbb{S} \rightharpoonup \mathbb{T}_\mathbb{S}$); `// update`
7 $\sigma \leftarrow \emptyset$ (type $\sigma\colon \mathbb{S} \rightharpoonup \mathbb{S}$); `// refinement subst`
8 $M_\alpha^{refined} \leftarrow$ `apply`$(M_\alpha, M'_\alpha,$ `refine`$[\tau, \sigma, \alpha])$;
9 **until** $M_\alpha = M_\alpha^{refined}$;
10 **return** (M_α, τ);

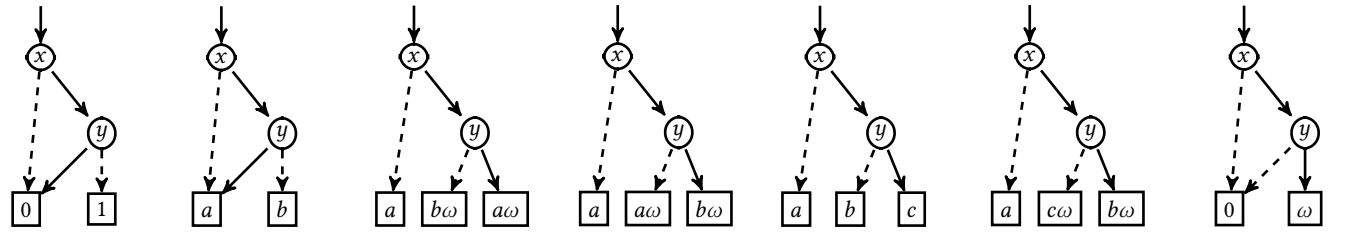
11 **Function** `abstract(val)`
12 **Data:** $\alpha\colon \mathbb{C} \rightharpoonup \mathbb{S}$
13 **if** α(val) $= \bot$ **then**
14 let $s_{new} \in \mathbb{S} \setminus \mathrm{rng}(\alpha)$ be a fresh symbolic var.;
15 $\alpha \leftarrow \alpha \cup \{\mathsf{val} \mapsto s_{new}\}$;
16 **return** α(val);

17 **Function** `refine(lhs, rhs)`
18 **Data:** $\tau\colon \mathbb{S} \rightharpoonup \mathbb{T}_\mathbb{S}, \sigma\colon \mathbb{S} \rightharpoonup \mathbb{S}, \alpha\colon \mathbb{C} \rightharpoonup \mathbb{S}$
19 **if** τ(lhs) $= \bot$ **then**
20 $\tau \leftarrow \tau \cup \{\mathsf{lhs} \mapsto \mathsf{rhs}\}$;
21 **else if** $\not\models \tau(\mathsf{lhs}) = \mathsf{rhs}$ **then**
22 **if** σ(lhs) $= \bot$ **then**
23 let $s_{new} \in \mathbb{S} \setminus \mathrm{rng}(\alpha)$ be a fresh symbolic var.;
24 $\sigma \leftarrow \sigma \cup \{\mathsf{lhs} \mapsto s_{new}\}$;
25 **return** σ(lhs);
26 **return** lhs;

Intuitively, the optimization works as follows. Consider a circuit with the following loop (in the OpenQASM 3.0 format [15]):

```
for int i in [1:K] { C; }
```

where C is the unitary for the loop body composed of standard gates and K is a constant. When a simulation of the circuit arrives to the loop with a quantum state q represented by MTBDD M_q, it will first create an MTBDD M_α with leaves containing symbolic variables (from a set $\mathbb{S}$, an infinite set of symbolic names). Then, it will run circuit C of the loop body with M_α as its input, with operations being done symbolically, i.e., instead of numbers, the leaves of the resulting MTBDD M'_α contain terms over $\mathbb{S}$; we denote the set of terms over $\mathbb{S}$ as $\mathbb{T}_\mathbb{S}$. M'_α contains information about how each of the computational bases needs to be updated. The information in M'_α is, however, fine-tuned for M_q, which can make the representation quite compact. This fine-tuning is done in the initial step called *abstraction*, when symbolic variables are being introduced—we start by introducing one symbolic variable for every distinct leaf value in M_q. The assumption is that computational bases with the same value will behave similarly. This does not need to hold, so after M'_α is



(a) Concrete state M_q **(b) Initial abstraction M_{α_1} of M_q** **(c) After applying T_x to M_{α_1}** **(d) End of first iteration M'_{α_1}** **(e) Refined abstraction M_{α_2} of M_q** **(f) End of second iteration M'_{α_2}** **(g) Result of M_q after M'_{α_2}**

Figure 2: An example run of Algorithm 3 on the circuit in Figure 3.

computed, we check it by observing whether bases mapping to the same symbolic variable in M_α also map to the same update in M'_α. If not, we introduce more symbolic variables (for the differing bases) and run the algorithm again, until the condition holds.

The formal algorithm is given in Algorithm 3. In the algorithm, we use the following formal notation: $f[p_1, \ldots, p_k]$ denotes the closure of function f with parameters $p_1, \ldots, p_k$ assigned to the variables in the **Data** declaration of f (passed by reference). Given a (partial) function f of the type $f\colon X \rightharpoonup Y$, we use $\mathrm{rng}(f)$ to denote the *range* of f, i.e., the set $\{y \in Y \mid \exists x \in X\colon f(x) = y\}$. Moreover, given an $x \in X$, if there is no $(x, y) \in f$, we write $f(x) = \bot$.

Example 1. We first demonstrate a run of the algorithm on the example circuit in Figure 3. The circuit starts in state q with $x = 1$ and $y = 0$. Then, it performs K executions of the loop body C. In each execution of the loop body, first, the T gate is applied to x, performing the multiplication of its $|1\rangle$ amplitude by ω and then CNOT of y controlled by x is performed. Therefore, the resulting state after K executions is $K\omega\,|11\rangle$ if K is odd and $K\omega\,|10\rangle$ if K is even. The run of Algorithm 3 on the circuit is shown in Figure 2.

M_q is in Figure 2a. In Figure 2b, we can see the initial abstraction M_{α_1} of M_q after Line 2; in this case, $\alpha_1 = \{0 \mapsto a, 1 \mapsto b\}$ for symbolic variables a and b. Then, we run (Line 5) the loop body with M_{α_1}, obtaining first the tree in Figure 2c (after T^x) and then the tree M'_{α_1} in Figure 2d (after CNOT^x_y). Then, when we call $\texttt{apply}(M_{\alpha_1}, M'_{\alpha_1}, \texttt{refine}[\tau_1, \sigma_1, \alpha_1])$ at Line 8, we realize that the inital abstraction α_1 was too coarse (going from left to right, we will construct $\tau_1 = \{a \mapsto a, b \mapsto a\omega\}$ for bases $|00\rangle$, $|01\rangle$, and $|10\rangle$; then, when processing $|11\rangle$, which would give us $a \mapsto b\omega$, which is in conflict with $a \mapsto a$, we will introduce a new symbolic variable c for the base $|11\rangle$ and obtain a new abstraction M_{α_2} (cf. Figure 2e). Then, in the second iteration of the refinement loop, we will run the loop body on M_{α_2} (cf. Figure 2f), obtaining M'_{α_2}. Running $\texttt{apply}(M_{\alpha_2}, M'_{\alpha_2}, \texttt{refine}[\tau_2, \sigma_2, \alpha_2])$ will not find any inconsistency this time ($\tau_2 = \{a \mapsto a, b \mapsto c\omega, c \mapsto b\omega\}$), so we can terminate the refinement. Applying M'_{α_2} on M_q with τ_2 once, we obtain the tree in Figure 2g. □

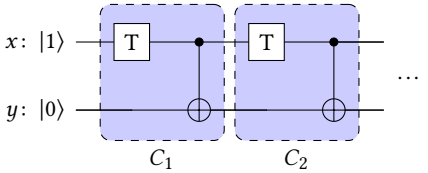


Figure 3: An example circuit for loop summarization

Formally, the algorithm computes a *summary* for a sequence of gates C w.r.t. a quantum state q (represented by an MTBDD M_q). The summary is a pair (M_α, τ) where M_α is a stable abstraction of M_q (w.r.t. C) and τ denotes how the symbolic variables should be updated during one loop iteration, computed as follows. On Line 2, we perform the initial abstraction of M_q, obtaining an MTBDD $M_\alpha^{\mathit{refined}}$ with one symbolic variable from $\mathbb{S}$ (the set of symbolic variables) for every amplitude occurring in M_q (the mapping is remembered in α). Then, we execute the sequence of gates C over M_q obtaining M'_q, where the resulting amplitudes are represented by symbolic terms over $\mathbb{S}$ (Line 5). On Line 8, we collect into τ the information about how the symbolic variables were updated and check whether all bases mapping to the same symbolic variable are updated in the same way—if not (on Line 19, we emphasize that we do not just check the *identity* of the two symbolic terms but, instead, check their *semantic equivalence*), we refine the abstraction (by introducing new symbolic variables for bases that have a different update) and try again. When we reach the fixpoint, we return the resulting abstracted MTBDD M_α together with the updates τ.

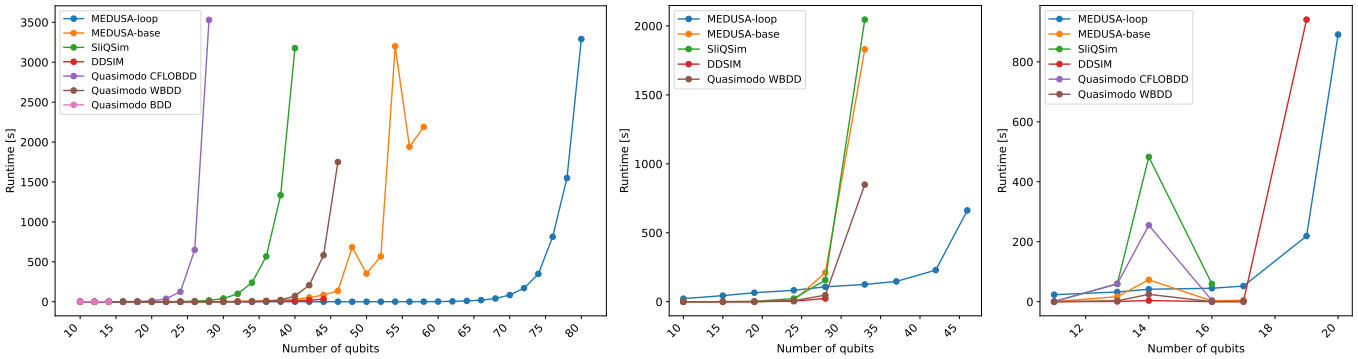
5 IMPLEMENTATION

We implemented the proposed techniques in a prototype called Medusa [22]. Medusa is written in C and uses Sylvan [31] for handling MTBDDs and the GNU GMP library [1] for handling integers of arbitrary length. We use two configurations of Medusa: with (Medusa_{loop}) and without (Medusa_{base}) loop summarization.

To achieve accuracy, we represent complex numbers algebraically as proposed in [37] and first realized in [30] (used also later in [11, 12]). The algebraic representation is given by the form

$$\left(\frac{1}{\sqrt{2}}\right)^k (a + b\omega + c\,\omega^2 + d\,\omega^3), \tag{1}$$

where a, b, c, d, and k are integers. A complex number is then represented by a five-tuple (a, b, c, d, k). Although it only represents a countable subset of $\mathbb{C}$, it can approximate any complex number up to a specified precision and suffices to support a set of quantum gates for universal quantum computation. The algebraic representation also allows for efficient encoding of some operations. For example, because $\omega^4 = -1$, the multiplication of (a, b, c, d, k) by ω can be carried out by a simple right circular shift of the first four entries and then taking the opposite number for the first entry, namely $(-d, a, b, c, k)$, which represents the complex number $\left(\frac{1}{\sqrt{2}}\right)^k (-d + a\omega + b\omega^2 + c\,\omega^3)$.



(a) Grover's search (b) Period finding (c) Quantum counting

Figure 4: Runtimes of the simulators on the Loops benchmark.

6 EXPERIMENTAL RESULTS

Simulators. We compared the performance of Medusa against the following state-of-the-art quantum circuit simulators: SliQSim [30], Quasimodo [28], DDSIM [38] (v1.21.0), and Quokka# [24]. For Quasimodo, which contains 3 different backends (BDD, WBDD, and CFLOBDD), we use Quas[B] to denote the version that uses backend B (we note that its WBDD backend uses a decision diagram package from DDSIM). To the best of our knowledge, only Medusa and SliQSim perform *accurate* simulation (using algebraic encoding of complex numbers) while the other tools use floating-point numbers (with possible numerical errors). The importance of accurate simulation has been demonstrated in applications such as quantum circuit equivalence checking [34]. All experiments were conducted on a server with two Intel Xeon X5650 (2.67 GHz) CPUs, 32 GiB of RAM running Debian GNU/Linux 12, with the timeout of 60 min.

Benchmarks. We performed experiments on the following two benchmark sets of quantum circuits in OpenQASM:

- Loops: This benchmark set contains circuits containing loops with fixed numbers of iterations. The particular circuits are implementations of Grover's search algorithm [19] (with a single solution), quantum counting [8], and period finding [23], the last two without the final inverse quantum Fourier transform (QFT)[2]. For quantum counting and period finding, we created several families of circuits with increasing size, denoted as $\langle FR\rangle\_\langle SR\rangle\_\langle MT\rangle$, where FR denotes the number of qubits in the *first register*, SR denotes the number of qubits in the *second register* (cf. [8]), and MT denotes the number of randomly generated multi-control Toffoli gates in the oracle. We always set $SR = \left\lfloor \frac{FR}{2} \right\rfloor$ and $MT \in \{5, 10, 15\}$. We unfolded the loops for tools that did not support them.
- StraightLine: This benchmark set contains circuits without loops implementing Bernstein-Vazirani's algorithm [6] (from 2 to 100 qubits $\rightsquigarrow$ 99 circuits), multi-control Toffoli gates (from 6 to 198 qubits with a step of 2 $\rightsquigarrow$ 97 circuits), benchmarks from the toolkit Feynman [3] (43 circuits), multi-oracle version of Grover's search (without loops; 9 circuits; MOG) from [2], randomly generated circuits from [2] (97 circuits), RevLib benchmarks [35] (80 circuits), and modifications of certain RevLib benchmarks from [30] (16 circuits) denoted as RevLib-H (these were obtained by inserting an H gate at each unassigned input).

The experiments measured the time it took for the final quantum state to be obtained in the given representation except Quokka#, where we measured the time to obtain the probability of the first qubit being zero (Quokka# does not compute representations of quantum states). The benchmarks did not contain measurements. A reproduction package for the experiments is available at [10].

Research questions. We were interested in the following two key research questions related to the proposed approach.

RQ1 What is the impact of loop summarization on the performance of quantum simulators?

RQ2 How does the MTBDD-based representation with custom gate operations compare to other simulators?

RQ1: Loop Summarization

For answering the first research question, which is the main target of this paper, we ran the simulators on the Loops benchmark set. The results can be seen in Figure 4 (for period finding and quantum counting, we show results for the families of circuits with oracle composed of 5 random multi-control Toffoli gates). Moreover, in Table 1, we give selected concrete results (we included for every simulator the largest circuit in the family where it finished). Quokka# is not included since it did not finish on any of the circuits. We also encountered some issues when running Quas[CFLOBDD] (internal error) and Quas[BDD] (incorrect implementation of the multi-control Toffoli `mcx` gate), which are labelled as ERR.

We first focus on comparing the performance of Medusa$_{loop}$ and Medusa$_{base}$, which differ only in loop summarization. The results show that in all three algorithms, Medusa$_{loop}$ scales much better than Medusa$_{base}$—it manages to simulate circuits of a size (the number of gates) one to three orders of magnitude larger. According to the results, the amount of necessary computation is significantly decreased, so we believe we can expect a similar behaviour if loop

[2]We did not include the inverse QFT because it requires rotations by $\frac{\pi}{2^n}$ for arbitrary n, which are not supported by our prototype, since it uses the algebraic encoding of complex numbers from Sec. 5. Note that this is not a conceptual limitation; one could solve it precisely by, e.g., dynamically refining the algebraic encoding to use finer base rotation than $\frac{\pi}{4}$, in particular $\frac{\pi}{2^n}$, or, not preserving accuracy, one could convert the algebraic encoding into floating-point numbers and continue with them. We wish to develop such solutions in our future work.

Table 1: Results for the Loops benchmark set (for every family, we include circuits which were the largest ones that some of the simulators managed to simulate before timeout). The columns "#q" and "#G" denote the number of qubits and gates (after loop unrolling) respectively. Times are given in seconds ("0" denotes a time <0.5 s), memory in MiB. TO denotes a timeout, ERR denotes an error, num denotes the fastest time, and num denotes the fastest *accurate* simulator (Medusa or SliQSim).

				Medusa$_{loop}$		Medusa$_{base}$		SliQSim		DDSIM		Quas[CFLOBDD]		Quas[WBDD]		Quas[BDD]	
	circuit	#q	#G	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem
Grover	7	14	480	0	99	0	37	0	12	0	30	0	463	0	444	1	445
	11	22	3,337	0	122	0	42	1	12	0	34	37	774	0	450	TO	TO
	14	28	12,115	0	145	1	56	17	13	1	50	3,530	9,532	0	470	TO	TO
	20	40	140,721	0	187	32	387	3,176	25	12	118	TO	TO	73	769	TO	TO
	22	44	310,367	0	196	85	1,088	TO	TO	32	254	TO	TO	583	1,083	TO	TO
	23	46	461,646	0	200	136	1,735	TO	TO	TO	TO	TO	TO	1,750	1,708	TO	TO
	29	58	4,676,916	2	214	2,190	10,032	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	40	80	292,359,936	3,290	251	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
Period Finding	16_08_05	24	1,507,322	83	600	8	24	23	130	4	1,235	ERR	ERR	7	449	ERR	ERR
	19_09_15	28	39,321,545	109	2,154	247	32	587	3,002	178	31,144	ERR	ERR	198	452	ERR	ERR
	22_11_05	33	146,800,628	125	922	1,830	38	2,046	10,293	TO	TO	ERR	ERR	849	454	ERR	ERR
	22_11_15	33	448,790,444	128	1,662	3,020	27	TO	TO	TO	TO	ERR	ERR	2,650	454	ERR	ERR
	31_15_15	46	277,025,390,495	673	1,973	TO	TO	TO	TO	TO	TO	ERR	ERR	TO	TO	ERR	ERR
Counting	10_05_05	16	40,937	45	2,115	3	83	60	15	0	42	4	459	0	446	ERR	ERR
	11_05_05	17	81,898	52	2,116	5	109	TO	TO	0	65	TO	TO	0	447	ERR	ERR
	12_06_15	19	376,760	250	7,691	TO	TO	TO	TO	1,280	294	TO	TO	TO	TO	ERR	ERR
	13_06_15	20	753,593	919	9,502	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	ERR	ERR

summarization is implemented for other representations. Therefore, the answer to RQ1 is that *the impact of loop summarization is profound for the performance of the simulator on circuits with loops.*

Let us also compare the performance with the other simulators in this benchmark set. We can see that in the case of Grover's algorithm (Figure 4a), Medusa$_{loop}$ managed to verify instances of a size far beyond the capabilities of any other simulator, in particular 80 qubits. The second best-performing simulator was Medusa$_{base}$, which scaled up to 58 qubits, followed by Quas[WBDD] (46 qubits), DDSIM (44 qubits), and SliQSim (40 qubits). The situation is similar for period finding (Figure 4b), where Medusa$_{loop}$ can scale up to 46 qubits, while the second best ones, Quas[WBDD] and Medusa$_{base}$, can scale only to 33 qubits. Let us note the size of the largest period finding circuit that Medusa$_{loop}$ managed to simulate in 12 minutes: over 277 billion gates. To the best of our knowledge, no existing quantum simulator is able to scale up to circuits of this size. Similar situation repeats for quantum counting, Medusa$_{loop}$ can, again, scale up to circuits of complexities that no other simulator could handle (although, due to the complexity of the circuits, it does not perform so well on smaller-sized circuits).

RQ2: MTBDD-Based Simulator

To answer the second research question, in addition to the results from the Loops benchmark set, we also evaluated the performance of simulators on the StraightLine benchmark (these circuits did not use loops, so we do not include Medusa$_{loop}$, since it would be the same as Medusa$_{base}$). Due to space limitations, we present only selected results. We chose circuits that took over one second to finish for three better-performing tools Medusa$_{base}$, SliQSim, and DDSIM. However, RevLib-H circuits were challenging for most tools, except for SliQSim which solved 13 cases. Both Medusa$_{base}$ and DDSIM solved 5 cases in RevLib-H. SliQSim splits amplitude values into bits and uses multiple BDDs to store a quantum state, resulting in better compression in this benchmark. Instead of showing a large table filled with TO, we show only the 5 solved cases in RevLib-H and refer readers to [30] for a more extensive comparison of SliQSim and DDSIM. Note that some tools had issues on some of the benchmarks due to unsupported gates.

The results show that Medusa$_{base}$ is competitive to other simulators and in many cases, especially for the challenging benchmarks from Feynman, is the best available accurate simulator. For the Loops benchmark, as mentioned previously, Medusa$_{base}$ is performing well also compared to other simulators: it is the best one on Grover and performs well also on the other two (it beats SliQSim, the only other accurate simulator). To conclude, the answer to RQ2 is that *the MTBDD-based representation with custom gate operations is competitive to other simulators, often complementary to SliQSim.*

7 RELATED WORK

DDSIM [38] is a quantum circuit simulator based on *quantum multiple-valued decision diagrams* (QMDDs) [25], which support representation and multiplication of state vectors and operator matrices. In [21], a QMDD variant, called *tensor decision diagrams* (TDDs), is proposed to allow tensor-network-like quantum circuit simulation. The TDD performance is comparable to DDSIM [21].

SliQSim [30] exploits the standard *reduced ordered binary decision diagrams* (ROBDDs) [9] to represent quantum states exactly with an algebraic number system and achieves precise quantum operations through Boolean formula manipulation. Note that similarly to Medusa, the supported quantum gate set of SliQSim, though universal, is restricted to those algebraically representable.

The paper [12] proposes verification of quantum circuits using *tree automata* to model their pre- and post-conditions. This method helps create an automatic verification framework that checks the correctness of the quantum circuit against a user-specified specification. Tree automata, similarly to decision diagrams, can efficiently represent identical subtrees using the same structure. Furthermore, they can use non-deterministic choice to represent multiple states in the same structure. We took inspiration from their extension to symbolic amplitudes in [11] to develop our symbolic execution.

Table 2: Selection of results for the StraightLine benchmark. The columns "#q" and "#G" denote the number of qubits and gates respectively. Times are given in seconds ("0.00" denotes a time <0.01 s), memory in MiB. TO denotes a timeout, ERR denotes an error, num denotes the fastest time, and num denotes the fastest *accurate* simulator (Medusa or SliQSim). We do not mark Quokka# as the fastest because it does not compute the quantum state representation.

				Medusa$_{base}$		SliQSim		DDSIM		Quas[CFLOBDD]		Quas[WBDD]		Quas[BDD]		Quokka#	
	circuit	#q	#G	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem
Feynman	gf2^{32}_mult	96	3,322	0.26	39	1.34	12	0.10	70	0.72	459	0.11	501	0.91	449	0.86	45
	gf2^{64}_mult	192	12,731	1.82	65	17.11	19	0.74	126	2.76	463	0.68	600	4.35	461	3.56	148
	gf2^{128}_mult	384	50,043	20.40	231	264.81	37	5.28	234	10.70	477	4.76	1,158	27.40	498	15.39	570
	gf2^{256}_mult	768	198,395	163.00	1,634	TO	TO	41.21	538	42.50	531	38.50	4,988	231.00	632	71.28	2,324
	hwb8	12	6,446	0.16	38	3.69	12	0.03	33	1.04	460	0.03	443	1.09	443	TO	TO
	hwb10	16	31,764	0.79	50	84.20	15	0.21	38	4.74	465	0.22	447	1.70	445	TO	TO
	hwb11	15	87,789	2.64	103	660.92	22	0.49	70	12.70	474	0.51	448	1.59	448	TO	TO
	hwb12	20	171,482	5.80	204	2,568.02	34	1.13	132	26.90	509	1.35	455	6.48	457	3,193.78	1,069
MOG	10	30	2,433	0.20	41	1.25	12	0.07	34	9.50	594	0.05	456	TO	TO	62.68	40
	11	33	3,746	0.36	44	3.12	12	0.12	42	52.00	905	0.08	462	TO	TO	167.00	56
Random	85	85	255	0.99	51	0.46	14	2.11	63	ERR	ERR	0.10	485	ERR	ERR	0.03	12
	86	86	258	15.30	213	0.47	14	2.24	72	ERR	ERR	3.25	553	ERR	ERR	0.07	12
	89	89	267	9.48	105	0.67	14	0.72	65	ERR	ERR	0.59	491	ERR	ERR	0.06	12
	93	93	279	1.68	61	0.32	13	0.18	67	ERR	ERR	0.10	493	ERR	ERR	0.04	12
	94	94	282	79.60	337	0.77	17	4.45	76	ERR	ERR	74.30	521	ERR	ERR	0.07	12
	97	97	291	5.70	117	0.42	13	1.46	77	ERR	ERR	0.42	524	ERR	ERR	0.03	12
	99	99	297	9.58	173	0.38	12	2.61	78	ERR	ERR	0.67	525	ERR	ERR	0.08	12
RevLib	apex5_290	1,025	2,909	1.75	61	0.37	43	1.02	535	0.30	466	1.33	1,214	4.16	516	2.10	72
	cps_292	923	2,763	1.19	57	0.20	30	1.25	484	0.24	464	1.09	1,035	2.99	527	1.38	59
	frg2_297	1,219	3,724	2.32	93	0.49	48	1.51	633	0.36	468	1.90	1,307	6.32	497	2.15	84
	seq_314	1,617	5,990	4.96	97	1.35	108	4.11	834	0.62	476	3.71	1,775	13.90	536	3.65	124
RevLib-H	add64_184	193	385	0.19	203	0.02	13	0.09	117	ERR	ERR	0.07	545	ERR	ERR	ERR	ERR
	cpu_register_32_405	328	890	0.46	213	0.08	14	0.41	194	ERR	ERR	0.70	668	ERR	ERR	ERR	ERR
	e64-bdd_295	195	452	1.98	238	2.48	13	2.00	126	0.65	476	0.54	613	ERR	ERR	ERR	ERR
	ex5p_296	206	655	7.61	283	12.02	21	3.56	132	ERR	ERR	1.15	691	ERR	ERR	ERR	ERR
	hwb9_304	170	708	33.00	662	13.50	20	12.16	114	ERR	ERR	4.90	1,105	ERR	ERR	ERR	ERR

SymQV [5] encodes quantum circuit verification problems into SMT with the theory of real numbers, using variables in trigonometric functions, which might lose precision in corner cases. Their approach requires 2^n variables to encode a n-qubit circuit in the worst case. A polynomial SMT encoding of quantum circuits was introduced in [13], where an extension of array theory, named *the theory of cartesian arrays (CaAL)*, was proposed and used to encode quantum gates. Both methods are effective only for small circuits.

Quasimodo [28] is a simulation tool with multiple backends, including BDDs, weighted BDDs (using the backend of DDSIM), and *context-free language ordered binary decision diagrams* (CFLOBDDs) [29], which combine BDDs with pushdown automata.

Hong *et al.* [20] proposed *symbolic TDDs* (symTDDs) for symbolically executing and representing quantum circuits and quantum states. However, in quantum circuit simulation, parameters are typically predetermined, making this approach useful mainly for parameterized quantum circuit equivalence checking.

Quokka# [24] extended the standard stabilizer formalism [17] to present a general pure state using its stabilizers. The representation circumvents complex numbers and only requires manipulating weights in real (possibly negative) numbers for the supported quantum gate operations. Thereby, quantum circuit simulation can be encoded into a weighted model counting problem. Quokka# only supports Clifford+T and rotation gates (which is, however, universal). Experimental results show the advantages of Quokka# on certain benchmarks such as quantum Fourier transform (QFT).

Although Clifford circuits should be efficiently simulatable according to the Gottesman–Knill theorem [18], simulating them in decision diagrams may suffer from exponential growth in size. To overcome this problem, Vinkhuijzen *et al.* [32, 33] proposed the *local invertible map decision diagrams* (LIMDDs), a data structure based on QMDDs that further merges nodes that are equivalent up to a *local invertible map* (LIM). LIMDDs successfully combine decision diagrams and the stabilizer formalism, and they efficiently overcome the challenge of exponential growth in decision diagrams on Clifford circuits. The authors of [32, 33] demonstrated that LIMDDs are more scalable in simulating QFT circuits than QMDDs.

8 CONCLUSION

We presented a technique for accelerating the simulation of quantum circuits with loops by computing the loops' summaries using symbolic execution. The experiments show that this technique enables the simulation of quantum circuits previously believed to be infeasible. In the future, we wish to further develop the loop summarization by integrating it with other data structures. Moreover, we wish to look at the problem of automatically generalizing a computed summary into a *closed form* (such as the description "$K\omega\,|11\rangle$ if K is odd and $K\omega\,|10\rangle$ if K is even" from Example 1), and use the technique also in the verification framework of [12].

ACKNOWLEDGMENTS

This work was supported by the Czech Ministry of Education, Youth and Sports ERC.CZ project LL1908, the Czech Science Foundation project 23-07565S, the FIT BUT internal project FIT-S-23-8151, the National Science and Technology Council of Taiwan under grant 113-2119-M-002-024, and the NTU Center of Data Intelligence: Technologies, Applications, and Systems under grant NTU-113L900903.

REFERENCES

[1] 2022. GMP: The GNU Multiple Precision Arithmetic Library. https://gmplib.org/

[2] 2024. The AutoQ repository. https://github.com/alan23273850/AutoQ/

[3] Matthew Amy. 2018. Towards Large-scale Functional Verification of Universal Quantum Circuits. In *Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018 (EPTCS)*, Peter Selinger and Giulio Chiribella (Eds.), Vol. 287. 1–21. https://doi.org/10.4204/EPTCS.287.1

[4] R. Iris Bahar, Erica A. Frohm, Charles M. Gaona, et al. 1997. Algebraic Decision Diagrams and Their Applications. *FMSD* 10, 2/3 (1997), 171–206. https://doi.org/10.1023/A:1008699807402

[5] Fabian Bauer-Marquart, Stefan Leue, and Christian Schilling. 2023. symQV: Automated Symbolic Verification of Quantum Programs. In *Formal Methods - 25th International Symposium, FM 2023, Lübeck, Germany, March 6-10, 2023, Proceedings (Lecture Notes in Computer Science)*, Marsha Chechik, Joost-Pieter Katoen, and Martin Leucker (Eds.), Vol. 14000. Springer, 181–198. https://doi.org/10.1007/978-3-031-27481-7_12

[6] Ethan Bernstein and Umesh V. Vazirani. 1993. Quantum complexity theory. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*, S. Rao Kosaraju, David S. Johnson, and Alok Aggarwal (Eds.). ACM, 11–20. https://doi.org/10.1145/167088.167097

[7] Gilles Brassard, Peter Høyer, Michele Mosca, and Alain Tapp. 2002. Quantum amplitude amplification and estimation. In *Quantum computation and information (Washington, DC, 2000)*. Contemp. Math., Vol. 305. Amer. Math. Soc., Providence, RI, 53–74. https://doi.org/10.1090/conm/305/05215

[8] Gilles Brassard, Peter Høyer, and Alain Tapp. 1998. Quantum Counting. In *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings (Lecture Notes in Computer Science)*, Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel (Eds.), Vol. 1443. Springer, 820–831. https://doi.org/10.1007/BFB0055105

[9] Randal E. Bryant. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers* 35, 8 (1986), 677–691. https://doi.org/10.1109/TC.1986.1676819

[10] Tian-Fu Chen, Yu-Fang Chen, Jie-Hong Jiang, Sára Jobranová, and Ondřej Lengál. 2024. Artifact for the ICCAD'24 paper "Accelerating Quantum Circuit Simulation with Symbolic Execution and Loop Summarization". https://doi.org/10.5281/zenodo.13243595

[11] Yu-Fang Chen, Kai-Min Chung, Ondrej Lengál, Jyun-Ao Lin, and Wei-Lun Tsai. 2023. AutoQ: An Automata-Based Quantum Circuit Verifier. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III (Lecture Notes in Computer Science)*, Constantin Enea and Akash Lal (Eds.), Vol. 13966. Springer, 139–153. https://doi.org/10.1007/978-3-031-37709-9_7

[12] Yu-Fang Chen, Kai-Min Chung, Ondřej Lengál, Jyun-Ao Lin, Wei-Lun Tsai, and Di-De Yen. 2023. An Automata-Based Framework for Verification and Bug Hunting in Quantum Circuits. *Proc. ACM Program. Lang.* 7, PLDI, Article 156 (jun 2023), 26 pages. https://doi.org/10.1145/3591270

[13] Yu-Fang Chen, Philipp Rümmer, and Wei-Lun Tsai. 2023. A Theory of Cartesian Arrays (with Applications in Quantum Circuit Verification). In *International Conference on Automated Deduction*. Springer, 170–189.

[14] Bob Coecke and Ross Duncan. 2008. Interacting Quantum Observables. In *Automata, Languages and Programming, 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II - Track B: Logic, Semantics, and Theory of Programming & Track C: Security and Cryptography Foundations (Lecture Notes in Computer Science)*, Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz (Eds.), Vol. 5126. Springer, 298–310. https://doi.org/10.1007/978-3-540-70583-3_25

[15] Andrew Cross, Ali Javadi-Abhari, Thomas Alexander, Niel De Beaudrap, Lev S. Bishop, Steven Heidel, Colm A. Ryan, Prasahnt Sivarajah, John Smolin, Jay M. Gambetta, and Blake R. Johnson. 2022. OpenQASM 3: A Broader and Deeper Quantum Assembly Language. *ACM Transactions on Quantum Computing* 3, 3, Article 12 (sep 2022), 50 pages. https://doi.org/10.1145/3505636

[16] Masahiro Fujita, Patrick C. McGeer, and Jerry Chih-Yuan Yang. 1997. Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. *Formal Methods Syst. Des.* 10, 2/3 (1997), 149–169. https://doi.org/10.1023/A:1008647823331

[17] Daniel Gottesman. 1997. *Stabilizer codes and quantum error correction*. Ph.D. Dissertation. California Institute of Technology.

[18] Daniel Gottesman. 1998. The Heisenberg representation of quantum computers. *arXiv preprint quant-ph/9807006* (1998).

[19] Lov K. Grover. 1996. A Fast Quantum Mechanical Algorithm for Database Search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*, Gary L. Miller (Ed.). ACM, 212–219. https://doi.org/10.1145/237814.237866

[20] X. Hong, W. Huang, W. Chien, Y. Feng, M. Hsieh, S. Li, C. Yeh, and M. Ying. 2023. Decision Diagrams for Symbolic Verification of Quantum Circuits. In *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. IEEE Computer Society, Los Alamitos, CA, USA, 970–977. https://doi.org/10.1109/QCE57702.2023.00111

[21] Xin Hong, Xiangzhen Zhou, Sanjiang Li, Yuan Feng, and Mingsheng Ying. 2022. A Tensor Network based Decision Diagram for Representation of Quantum Circuits. *ACM Trans. Des. Autom. Electron. Syst.* 27, 6, Article 60 (jun 2022), 30 pages. https://doi.org/10.1145/3514355

[22] Sára Jobranová. 2024. The Medusa repository. https://github.com/s-jobra/MEDUSA/

[23] Alexei Y. Kitaev. 1996. Quantum measurements and the Abelian Stabilizer Problem. *Electron. Colloquium Comput. Complex.* TR96-003 (1996). ECCC:TR96-003 https://eccc.weizmann.ac.il/eccc-reports/1996/TR96-003/index.html

[24] Jingyi Mei, Marcello Bonsangue, and Alfons Laarman. 2024. Simulating Quantum Circuits by Model Counting. In *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, Canada, July 22-27, 2024, Proceedings, Part III (Lecture Notes in Computer Science)*, Arie Gurfinkel and Vijay Ganesh (Eds.), Vol. 14683. Springer, 555–578.

[25] Philipp Niemann, Robert Wille, D. Michael Miller, Mitchell A. Thornton, and Rolf Drechsler. 2016. QMDDs: Efficient Quantum Function Representation and Manipulation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 35, 1 (2016), 86–99. https://doi.org/10.1109/TCAD.2015.2459034

[26] Hans De Raedt, Fengping Jin, Dennis Willsch, Madita Nocon, Naoki Yoshioka, Nobuyasu Ito, Shengjun Yuan, and Kristel Michielsen. 2019. Massively parallel quantum computer simulator, eleven years later. *Comput. Phys. Commun.* 237 (2019), 47–61. https://doi.org/10.1016/J.CPC.2018.11.005

[27] Peter W. Shor. 1994. Algorithms for Quantum Computation: Discrete Logarithms and Factoring. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*. IEEE Computer Society, 124–134. https://doi.org/10.1109/SFCS.1994.365700

[28] Meghana Sistla, Swarat Chaudhuri, and Thomas W. Reps. 2023. Symbolic Quantum Simulation with Quasimodo. In *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III (Lecture Notes in Computer Science)*, Constantin Enea and Akash Lal (Eds.), Vol. 13966. Springer, 213–225. https://doi.org/10.1007/978-3-031-37709-9_11

[29] Meghana Aparna Sistla, Swarat Chaudhuri, and Thomas Reps. 2023. CFLOBDDs: Context-free-language ordered binary decision diagrams. *ACM Transactions on Programming Languages and Systems* (2023).

[30] Yuan-Hung Tsai, Jie-Hong R. Jiang, and Chiao-Shan Jhang. 2021. Bit-Slicing the Hilbert Space: Scaling Up Accurate Quantum Circuit Simulation. In *58th ACM/IEEE Design Automation Conference, DAC 2021, San Francisco, CA, USA, December 5-9, 2021*. IEEE, 439–444. https://doi.org/10.1109/DAC18074.2021.9586191

[31] Tom van Dijk and Jaco van de Pol. 2017. Sylvan: multi-core framework for decision diagrams. *Int. J. Softw. Tools Technol. Transf.* 19, 6 (2017), 675–696. https://doi.org/10.1007/S10009-016-0433-2

[32] Lieuwe Vinkhuijzen, Tim Coopmans, David Elkouss, Vedran Dunjko, and Alfons Laarman. 2023. LIMDD: A Decision Diagram for Simulation of Quantum Computing Including Stabilizer States. *Quantum* 7 (2023), 1108. https://doi.org/10.22331/Q-2023-09-11-1108

[33] Lieuwe Vinkhuijzen, Thomas Grurl, Stefan Hillmich, Sebastiaan Brand, Robert Wille, and Alfons Laarman. 2023. Efficient Implementation of LIMDDs for Quantum Circuit Simulation. In *Model Checking Software - 29th International Symposium, SPIN 2023, Paris, France, April 26-27, 2023, Proceedings (Lecture Notes in Computer Science)*, Georgiana Caltais and Christian Schilling (Eds.), Vol. 13872. Springer, 3–21. https://doi.org/10.1007/978-3-031-32157-3_1

[34] Chun-Yu Wei, Yuan-Hung Tsai, Chiao-Shan Jhang, and Jie-Hong R. Jiang. 2022. Accurate BDD-based unitary operator manipulation for scalable and robust quantum circuit verification. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC '22)*. Association for Computing Machinery, New York, NY, USA, 523–528. https://doi.org/10.1145/3489517.3530481

[35] Robert Wille, Daniel Große, Lisa Teuber, Gerhard W. Dueck, and Rolf Drechsler. 2008. RevLib: An Online Resource for Reversible Functions and Reversible Circuits. In *38th IEEE International Symposium on Multiple-Valued Logic (ISMVL 2008), 22-23 May 2008, Dallas, Texas, USA*. IEEE Computer Society, 220–225. https://doi.org/10.1109/ISMVL.2008.43

[36] Alwin Zulehner, Stefan Hillmich, and Robert Wille. 2019. How to Efficiently Handle Complex Values? Implementing Decision Diagrams for Quantum Computing. In *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019*, David Z. Pan (Ed.). ACM, 1–7. https://doi.org/10.1109/ICCAD45719.2019.8942057

[37] Alwin Zulehner, Philipp Niemann, Rolf Drechsler, and Robert Wille. 2019. Accuracy and Compactness in Decision Diagrams for Quantum Computation. In *2019 Design, Automation and Test in Europe Conference (DATE)*. 280–283. https://doi.org/10.23919/DATE.2019.8715040

[38] Alwin Zulehner and Robert Wille. 2019. Advanced Simulation of Quantum Computations. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 38, 5 (2019), 848–859. https://doi.org/10.1109/TCAD.2018.2834427