BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

SIMULATION AND ANALYSIS OF QUANTUM CIRCUITS

SIMULACE A ANALÝZA KVANTOVÝCH OBVODŮ

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **SÁRA JOBRANOVÁ**
AUTOR PRÁCE

SUPERVISOR **Ing. ONDŘEJ LENGÁL, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

154537

Institut:	Department of Intelligent Systems (DITS)
Student:	**Jobranová Sára**
Programme:	Information Technology
Title:	**Simulation and Analysis of Quantum Circuits**
Category:	Formal Verification
Academic year:	2023/24

Assignment:

1. Study the theory of quantum computation.
2. Study techniques of simulation, analysis, and verification of quantum circuits.
3. Propose techniques for simulation or analysis of quantum circuits. Get inspired by existing approaches based on decision diagrams and tree automata.
4. Implement the proposed techniques in a tool.
5. Compare the tool to existing tools, focusing on the speed and other parameters (precision, etc.).
6. Evaluate the achieved results and discuss possibilities of further development.

Literature:

- M. A. Nielsen and I. L. Chuang, *Quantum computation and quantum information*, 10th anniversary ed. Cambridge ; New York: Cambridge University Press, 2010.
- Y.-H. Tsai, J.-H. R. Jiang, and C.-S. Jhang, “Bit-Slicing the Hilbert Space: Scaling Up Accurate Quantum Circuit Simulation,” in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, Dec. 2021, pp. 439–444. doi: 10.1109/DAC18074.2021.9586191.
- Y.-F. Chen, K.-M. Chung, O. Lengál, J.-A. Lin, W.-L. Tsai, and D.-D. Yen, “An Automata-Based Framework for Verification and Bug Hunting in Quantum Circuits,” *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, p. 156:1218-156:1243, erven 2023, doi: 10.1145/3591270.
- Y.-F. Chen, K.-M. Chung, O. Lengál, J.-A. Lin, and W.-L. Tsai, “AutoQ: An Automata-Based Quantum Circuit Verifier,” in *Computer Aided Verification*, C. Enea and A. Lal, Eds., in Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 139–153. doi: 10.1007/978-3-031-37709-9_7.
- M. Sistla, S. Chaudhuri, and T. Reps, “Symbolic Quantum Simulation with Quasimodo,” in *Computer Aided Verification*, C. Enea and A. Lal, Eds., in Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 213–225. doi: 10.1007/978-3-031-37709-9_11.

Requirements for the semestral defence:
First two items from the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Lengál Ondřej, Ing., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	1.11.2023
Submission deadline:	9.5.2024
Approval date:	6.11.2023

Abstract

Simulation of quantum circuits is a key tool for future advancements in the promising field of quantum computing. Due to the fact that this task is very computationally demanding, the performance of state-of-the-art simulators on more complex circuits is still far from satisfactory. In this thesis, we propose a new approach to simulate quantum circuits and present an implementation based on this approach. Our simulation technique allows for accurate simulation and is based on multi-terminal binary decision diagrams. We extended the usual process of a decision diagram-based simulation by symbolic execution of repeating structures in a quantum circuit (such as loops), where we compute the big-step semantics of this structure and do not re-evaluate the gates. We show that symbolic loop execution significantly accelerates the simulation and that the implemented tool is not only competitive with other state-of-the-art simulators, but also greatly outperforms the state of the art for many quantum circuits.

Abstrakt

Simulace kvantových obvodů je klíčovým nástrojem pro další výzkum v oblasti kvantové výpočetní techniky, která je velmi perspektivní. Jedná se však o velmi výpočetně náročný problém, a z tohoto důvodu jsou i u moderních nástrojů při simulaci komplexních obvodů z hlediska výkonu značné rezervy. V této práci představíme nový přístup k simulaci kvantových obvodů a nástroj implementovaný na základě tohoto přístupu. Tato technika umožňuje přesnou simulaci a je založena na multi-terminálních binárních rozhodovacích diagramech. Také rozšiřuje standardní proces simulace založené na rozhodovacích diagramech o symbolickou exekuci opakujících se struktur v kvantovém obvodě (např. smyček), kdy se spočítá sémantika jednoho opakování této struktury a neprovádíme tudíž opětovné vyhodnocování hradel. Ukázali jsme, že symbolické provádění smyček výrazně urychluje simulaci a že implementovaný nástroj je nejen konkurenceschopný s ostatními nejmodernějšími simulátory, ale také tyto simulátory pro mnoho kvantových obvodů značně překonává.

Keywords

Quantum computing, Simulation of quantum circuits, Multi-terminal binary decision diagrams, Symbolic execution

Klíčová slova

Kvantové výpočty, Simulace kvantových obvodů, Multi-terminální binární rozhodovací diagramy, Symbolická exekuce

Reference

JOBRANOVÁ, Sára. *Simulation and Analysis of Quantum Circuits.* Brno, 2024. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Ondřej Lengál, Ph.D.

Rozšířený abstrakt

Potenciál kvantových počítačů má řadu zajímavých důsledků nejen přímo v oblasti informačních technologií (např. pro kryptografii), ale i v mnoha dalších odvětvích jako je fyzika, chemie či finance. Především kvůli technickým problémům, které provází sestavení a provoz kvantového počítače, se jedná o oblast pomalého, avšak stále trvajícího výzkumu a pokroku. Z důvodu problematické dostupnosti kvantových počítačů (zejména z finančního hlediska) a omezeným možnostem pozorování stavu reálného kvantového systému (pouze pomocí nevratné operace měření qubitu) jsou nástroje umožňující efektivní simulaci kvantových obvodů na klasických počítačích pro další pokrok v oblasti kvantových výpočtů nezbytné. Tento úkol je však výpočetně velmi náročný kvůli poměru velikosti stavového prostoru klasického bitu ve srovnání s velikostí stavového prostoru qubitu, a proto i u nejmodernějších simulátorů není výkon pro netriviální obvody stále uspokojivý.

V této práci popisujeme nový přístup k simulaci kvantových obvodů na klasických počítačích a jeho implementaci v nástroji `MEDUSA`. Představená simulační technika je založena na multi-terminálních binárních rozhodovacích diagramech (MTBDD), kdy kvantový stav interpretujeme jako funkci a reprezentujeme ji pomocí MTBDD, a využívá již dříve představenou algebraickou reprezentaci komplexních čísel, díky čemuž `MEDUSA` provádí *přesnou* simulaci. Nejen že je přesná simulace klíčová např. pro řešení testu ekvivalence kvantových obvodů, ale zároveň se použitím přesné reprezentace komplexních čísel vyhneme potenciálním numerickým nestabilitám. Pro aplikaci kvantových hradel se používají speciální MTBDD procedury namísto použití pouze standardního rozhraní pro operace s MTBDD (pomocí procedur *Apply* a *Restrict*), jak je tomu obvyklé.

Jelikož pro reprezentaci kvantového stavu využíváme MTBDD, jsme schopni simulovat opakující se struktury (např. smyčky) v kvantovém obvodu pouze symbolicky. To je velmi výhodné, protože smyčky jsou často klíčovou součástí mnoha kvantových algoritmů. V praxi to znamená, že vypočítáme, jak se změní kvantový stav po jedné iteraci smyčky (toto reprezentujeme pomocí určitých algebraických výrazů nad proměnnými, které představují jednotlivé pravděpodobnostní amplitudy v původním kvantovém stavu) a následně spočítáme nový kvantový stav dle těchto algebraických výrazů a počtu iterací smyčky. To znamená, že díky této tzv. *symbolické exekuce* smyček není potřeba opakovaně vyhodnocovat hradla v těle smyčky.

Implementovaný nástroj `MEDUSA` jsme porovnali s několika nejmodernějšími simulátory, a to jak verzi se symbolickou exekucí smyček, tak verzi bez ní. Verze bez symbolické exekuce se ukázala jako kompetetivní s ostatními simulátory, a dokonce tyto další nástroje pro některé typy obvodů i značně překonala (zejména pokud vezmeme v potaz pouze přesné simulátory). Verze se symbolickou exekucí škálovala během simulace všech měřených obvodů (implementujících různé kvantové algoritmy obsahující smyčky) podstatně lépe než všechny ostatní nástroje, což ukazuje, že symbolická exekuce smyček vede k výraznému zrychlení simulace. Stejné chování lze očekávat i v případě rozšíření symbolické exekuce smyček pro další moderní simulátory založené na rozhodovacích diagramech.

Z hlediska další práce je prostor pro další optimalizace simulátoru, také je v plánu rozšířit současnou množinu podporovaných kvantových operací. Dále by bylo možné rozšířit funkcionalitu implementovaného nástroje pro účely verifikace kvantových obvodů.

Simulation and Analysis of Quantum Circuits

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Ing. Ondřej Lengál, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Sára Jobranová
May 14, 2024

Acknowledgements

I would like to thank my supervisor Ing. Ondřej Lengál, Ph.D. for his time, patience, and guidance throughout this work. I would also like to express my thanks to all of my loved ones for the immense support and encouragement they have given me while I have been working on this thesis.

Contents

Chapter 1

Introduction

Quantum computing is an intriguing field of computer science that leverages the principles of quantum mechanics to perform computations in ways that classical computers are unable to. The concept of quantum computing originated in the early 1980s, when physicist Richard Feynman introduced quantum computers as efficient means for simulating quantum mechanics in his lecture [14]. Soon followed the emergence of the first quantum algorithms and the interest in this field continued to grow in the 1990s [25] with the discovery of Shor's algorithm for factoring and Grover's search algorithm, as these breakthroughs demonstrated the potential of this novel paradigm for computation to efficiently solve certain problems that would be practically unsolvable on classical computers.

Yet, a lot of uncertainties are still present in this field. This is mainly due to the many technical challenges that arise when building and operating a quantum computer—it is only within recent years that technology has sufficiently advanced to allow for quantum computing (although this hardware still has a largely experimental character). Actually, a research from 2019 claims to have achieved this promise of an unmatched performance of quantum computers compared to classical computers in some specific tasks, or the so-called *quantum supremacy* [2]. There is an ongoing research on the practical usage of quantum computers in many different fields, such as physics [45], chemistry [23], and finance [19].

Still, the ability to simulate the behaviour of quantum circuits on classical computers is a vital tool for understanding the potential of quantum computers and for future research in this field for two reasons. The first one is the still ongoing unavailability of quantum computers, especially because of the price of building such a system. The second reason is that in a real system, we need to measure a qubit to make observations about its state, which leads to the collapse of the state of the qubit (this operation is irreversible). This means that it is not possible to directly examine the probability amplitudes of a real system, which is thus achievable only in simulation. It is, however, not an easy task to perform the needed calculations efficiently due to the ratio of the size of the state space of a classical bit in comparison with the size of the state space of a qubit.

Today, several simulators of quantum circuits exist, however, when discussing the current state of the art in terms of performance, there is still a lot of room for improvement when it comes to many complex circuits, and especially more complex circuits with a large number of qubits. For this reason, this work introduces a new tool for quantum circuit simulation. This simulator is based on Multi-Terminal Binary Decision Diagrams (MTBDDs) in combination with algebraic complex number representation for *accurately* representing the state of the simulated system. The implemented tool also supports *symbolic execution* of loops in the circuit, where one iteration of the given loop is expressed by a single symbolic operation. It is shown that this symbolic loop execution leads to a significant acceleration of the simulation. Furthermore, this work provides an experimental comparison with the current state of the art, showing not only that the implemented simulator keeps up with the current state-of-the-art tools, but also demonstrating superior performance of the implemented simulator for various quantum circuits (especially if we consider only accurate simulators). This is particularly profound when we consider quantum circuits that contain loops (so that we can take advantage of the symbolic execution of the loop), such as circuits that implement Grover's search algorithm, quantum counting, and period finding (the latter two without the inverse quantum Fourier transform).

First, we will go through the necessary theory regarding quantum computing and related key concepts and MTBDDs in Chapter 2. Then, in Chapter 3, this thesis covers the basic characteristics of current approaches to classical quantum circuit simulation, with a focus on providing an overview of the underlying data structures used in current state-of-the-art tools. We continue with the aforementioned key principles of the implemented simulator (MTBDDs whose leaves represent complex numbers in algebraic form) in Chapter 4. Finally, this thesis describes the architecture and further specifics of the implemented simulator in Chapter 5 and provides an experimental comparison of the performance of the implemented simulator with the state of the art in Chapter 6.

Chapter 2

Preliminaries

This chapter covers the basic concepts of quantum computing needed for this thesis, especially those regarding quantum circuits and their simulation. First, we present the properties of a single qubit, then of a multi-qubit system. Next, we will move on to quantum logic gates, quantum circuits, and then briefly to well-known quantum algorithms. Then this chapter also introduces binary decision diagrams (BDDs) and their generalised modification called multi-terminal binary decision diagrams (MTBDDs), as the implemented simulator is MTBDD-based.

2.1 Quantum Computing

Quantum computing leverages the principles of quantum mechanics to provide the potential for solving certain complex problems much more efficiently than with classical computing. This section uses information presented in [25]. The following notation overview draws from [8].

In this thesis, we use the usual notation, where $\mathbb{C}$ denotes the set of complex numbers, $\mathbb{Z}$ denotes the set of integers, and $\mathbb{N}$ denotes the set of natural numbers. We use $\mathbb{V}^n$ to represent the finite-dimensional vector space of dimension n ($n \in \mathbb{N}$), i.e., the set of all vectors over $\mathbb{V}$ of length n.

If not stated otherwise, all matrices and vectors in this work are assumed to be over $\mathbb{C}$. The *complex conjugate* of a complex number $z = a+bi$ is the complex number $\overline{z} = a-bi$ and the *transpose* of an $m \times n$ matrix $A = (a_{xy})$ is the $n \times m$ matrix A^T such that $A^T = (a_{yx})$. Then the *conjugate transpose* of a matrix $A = (a_{xy})$ is the matrix $A^\dagger = (\overline{a_{yx}})$. We denote the *identity* matrix of an arbitrary dimension as $\mathbb{I}$ and the *inverse* matrix of a matrix A is the matrix A^{-1} such that $A \cdot A^{-1} = \mathbb{I}$. A square matrix A is *unitary* if $A^{-1} = A^\dagger$.

Given a $m \times n$ matrix $A = (a_{xy})$ and a $k \times l$ matrix B, the *Kronecker product* $A \otimes B$ is the $mk \times nl$ matrix $A \otimes B = (a_{xy}B)$, e.g.,

$$\begin{pmatrix} 2+i & 0 \\ -i & 3 \end{pmatrix} \otimes \begin{pmatrix} 5 & 1 \\ i & 0 \end{pmatrix} = \begin{pmatrix} (2+i) \cdot \begin{pmatrix} 5 & 1 \\ i & 0 \end{pmatrix} & 0 \cdot \begin{pmatrix} 5 & 1 \\ i & 0 \end{pmatrix} \\ -i \cdot \begin{pmatrix} 5 & 1 \\ i & 0 \end{pmatrix} & 3 \cdot \begin{pmatrix} 5 & 1 \\ i & 0 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 10+5i & 2+i & 0 & 0 \\ -1+2i & 0 & 0 & 0 \\ -5i & -i & 15 & 3 \\ 1 & 0 & 3i & 0 \end{pmatrix}.$$

A *row* vector of length l is a $1 \times l$ matrix, a *column* vector of length l is a $l \times 1$ matrix. Given vectors u, v, the operation $u \otimes v$ denotes the tensor product of the two vectors, which is consistent with the previously defined Kronecker product operation. From now on, the *Dirac notation* (also known as the *bra-ket* notation) is used for vectors, since it is the standard notation used for quantum mechanics. An example of the relationship between the Dirac and column matrix notation of a vector can be seen here:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \; |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}.$$

We often denote $u \otimes v$ simply as $|uv\rangle$ and $|u^n\rangle$ represents the operation

$$\underbrace{|u\rangle \otimes |u\rangle \otimes \ldots \otimes |u\rangle}_{n \text{ times}}.$$

Also, in the following, the qubit with the lowest index, q_0, is the most significant (the topmost qubit in the circuit diagram).

2.1.1 Qubits

Just as bits are the foundation for classical computation, *qubits* (quantum bits) are the foundation for quantum computation. Further, just as classical bits are described by their state (either 0 or 1), qubits are also described by their *quantum state* (usually also referred to simply as their state). Equivalent to states 0 and 1 of a classical bit are states $|0\rangle$ and $|1\rangle$, respectively, of a qubit. States $|0\rangle$ and $|1\rangle$ are called the *computational basis states*—they form the orthonormal basis for the vector space $\mathbb{C}^2$ (the vector space of a qubit's state as will become apparent momentarily).

A qubit's state $|\psi\rangle$ can generally be in a linear combination called a *superposition* of the aforementioned computational basis states

$$|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle\,,$$

where $\alpha, \beta \in \mathbb{C}$ are the *probability amplitudes* for the respective basis states. A single qubit's state is therefore a *two-dimensional complex vector* and it is sometimes also called a *state vector*.

Apart from the computational basis states, the following states of a single qubit are also essential:

$$|+\rangle = \frac{1}{\sqrt{2}} \cdot (|0\rangle + |1\rangle),$$

$$|-\rangle = \frac{1}{\sqrt{2}} \cdot (|0\rangle - |1\rangle).$$

These states can be thought of as halfway between $|0\rangle$ and $|1\rangle$, and play a key role in many quantum algorithms.

Therefore, a qubit's state can be in a superposition of $|0\rangle$ and $|1\rangle$, so not necessarily purely only either $|0\rangle$ or $|1\rangle$, but rather somewhere on the continuum between $|0\rangle$ and $|1\rangle$. However, in a real system, it is impossible to precisely determine the state of the qubit, i.e., find out the values of the probability amplitudes α and β.

Qubit Measurement

We can only make observations about a given qubit's state when we *measure* the qubit. However, upon measurement, a qubit's state collapses into one particular basis state. Therefore, the only possible results of the measurement are the states $|0\rangle$ or $|1\rangle$ and the only thing that is determined by the initial unobservable state of a qubit is the probability with which we measure the result of either $|0\rangle$ or $|1\rangle$, as this is determined by the probability amplitudes α and β, respectively:

$$\mathcal{P}(|0\rangle) = |\alpha|^2,$$
$$\mathcal{P}(|1\rangle) = |\beta|^2.$$

A single qubit's state is therefore more precisely a two-dimensional *unit* complex vector, as it has to be a unit vector (a vector of magnitude 1) to satisfy the probability constraint

$$|\alpha|^2 + |\beta|^2 = 1,$$

i.e., the sum of the probabilities must be 1. So, if the result of our measurement is the state $|0\rangle$, then such a qubit now has a state with $\alpha = 1$, $\beta = 0$ and it is not possible to reverse its previous state (values of α, β). The new state's α and β reflect both the collapse of the quantum state and the probability constraint. Similarly, when the result of the measurement is the state $|1\rangle$, this qubit has a new state with $\alpha = 0$, $\beta = 1$.

Systems with Multiple Qubits

Generally an n-qubit system's state $|\psi'\rangle$ can be in a superposition of all the system's computational basis states

$$|\psi'\rangle = \sum_{i \in \{0,1\}^n} \alpha_i \cdot |i\rangle\,,$$

and therefore is generally a 2^n-dimensional unit complex vector, where, again, $\alpha_i \in \mathbb{C}$ are the probability amplitudes of the corresponding basis states. Similarly to a state of a single qubit, we call this vector the state vector of the given system. The basis states for an n-qubit system again form the orthonormal basis of the system's vector space, which is clearly $\mathbb{C}^{2^n}$—so if, for example, we have a system where $n = 2$, the computational basis states for this system are $|00\rangle$, $|01\rangle$, $|10\rangle$ and $|11\rangle$ (similarly to the possible states 00, 01, 10 and 11 of the classical 2-bit system).

Just as with a single qubit, even with a multi-qubit system, we cannot observe the state of this system directly, i.e., we cannot precisely determine the values of the individual probability amplitudes. The only thing that is observable about a multi-qubit system's state is the result of measuring a certain subset of the qubits of the given system. Similar to when we measure a qubit of a single-qubit system, if we measure the values of all n qubits of an n-qubit system, we get one of the computational basis states $|\{0,1\}^n\rangle$, where the probability of measuring this result is determined by the corresponding amplitude

$$\mathcal{P}(|i\rangle) = |\alpha_i|^2,$$

where $i \in \{0,1\}^n$. So, for example, if we have a 2-qubit system and we measure both the qubits, we get the result $|00\rangle$ with the probability $|\alpha_{00}|^2$, the result $|01\rangle$ with the probability $|\alpha_{01}|^2$ and so on.

However, as mentioned above, we can measure only a subset of these qubits. The probability with which a qubit's measured value will be $|0\rangle$ or $|1\rangle$ is determined by all the probability amplitudes of the given system corresponding to the computational basis states, where this qubit's value is 0 or 1, respectively. For example, the probability that qubit q_j will collapse into $|1\rangle$ is for an n-qubit system calculated as

$$\mathcal{P}(q_j = |1\rangle) = \sum_{i \in \{0,1\}^{n-1-j} \times \{1\} \times \{0,1\}^j} |\alpha_i|^2\,. \tag{2.1}$$

Naturally, when a qubit q_j collapses into $|1\rangle$ after measurement, all the probability amplitudes corresponding to contradicting basis states (where q_j is 0) are set to the value 0. Then, the remaining non-zero amplitudes must be normalized by multiplication with the coefficient

$$c = \frac{1}{\sqrt{\mathcal{P}(q_j = |1\rangle)}} \ , \tag{2.2}$$

in order to keep the sum of all the state's probability amplitudes as 1 (the probability in the denominator is generally the probability with which the measurement result occurred). Likewise, if the result of the measurement of the qubit q_j is $|0\rangle$, all amplitudes for basis states where q_j is 1 are zeroed and the amplitudes are re-normalized with $c = \frac{1}{\sqrt{\mathcal{P}(q_j=|0\rangle)}}$. Equations 2.1 and 2.2 can be easily modified using the same logic to measure an arbitrary subset of the system's qubits.

Quantum Entanglement

Quantum entanglement is a phenomenon fundamental to quantum mechanics in general and is crucial for fast quantum algorithms, quantum teleportation, and quantum error-correction. It is important to note that there is still an ongoing research on this subject, as a complete theory of quantum entanglement does not yet exist and a better understanding of this important aspect of quantum mechanics could result in novel and promising applications of quantum computing.

Quantum entanglement is an extraordinary behaviour that some quantum states can exhibit—we refer to these states as *EPR states* or *EPR pairs* and they are named after A. Einstein, B. Podolsky and N. Rosen, who first brought attention to the concept of quantum entanglement in their famous paper [13] in 1935. Specifically, if we talk about 2-qubit maximally entangled states, we may also refer to these states as *Bell states* (named after J. Bell who further investigated the EPR paradox and provided great insights on this subject). The interesting consequence of the properties of EPR states is that the measurement results of the individual qubits are correlated (this correlation has no equivalent in classical physics). This means, that after measuring one of the entangled qubits, the measurement of the other entangled qubits will always yield the same result (regardless of the physical distance between the qubits themselves). So, for example, if we have one of the Bell states

$$|\psi\rangle = \frac{1}{\sqrt{2}} \cdot (|00\rangle + |11\rangle), \tag{2.3}$$

and we want to measure for instance the first qubit q_0, then according to Equation 2.1 we have

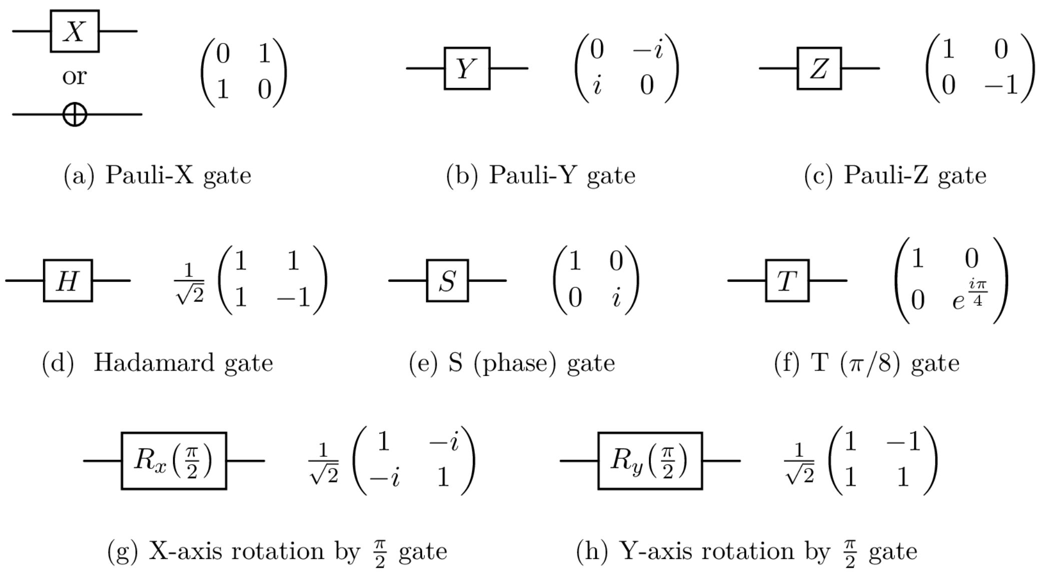
$$\mathcal{P}(q_0 = |0\rangle) = \mathcal{P}(q_0 = |1\rangle) = \frac{1}{2}.$$

However, when we obtain the result of this measurement, for example $q_0 = |0\rangle$, then when measuring the second qubit q_1, contrary to intuition we have

$$\mathcal{P}(q_1 = |0\rangle) = 1,$$

$$\mathcal{P}(q_1 = |1\rangle) = 0.$$

Not only that, but if we have an EPR state, some correlation between the measurement results still occurs even when we first perform additional operations on one of the qubits



$\begin{pmatrix}0 & 1\\ 1 & 0\end{pmatrix}$ (a) Pauli-X gate

$\begin{pmatrix}0 & -i\\ i & 0\end{pmatrix}$ (b) Pauli-Y gate

$\begin{pmatrix}1 & 0\\ 0 & -1\end{pmatrix}$ (c) Pauli-Z gate

$\frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$ (d) Hadamard gate

$\begin{pmatrix}1 & 0\\ 0 & i\end{pmatrix}$ (e) S (phase) gate

$\begin{pmatrix}1 & 0\\ 0 & e^{\frac{i\pi}{4}}\end{pmatrix}$ (f) T ($\pi/8$) gate

$R_x(\frac{\pi}{2})$ $\frac{1}{\sqrt{2}}\begin{pmatrix}1 & -i\\ -i & 1\end{pmatrix}$ (g) X-axis rotation by $\frac{\pi}{2}$ gate

$R_y(\frac{\pi}{2})$ $\frac{1}{\sqrt{2}}\begin{pmatrix}1 & -1\\ 1 & 1\end{pmatrix}$ (h) Y-axis rotation by $\frac{\pi}{2}$ gate

Figure 2.1: Examples of single-qubit gates and their matrix representation

(the specific correlation will depend on the operations and the initial state). Note also that entangled states cannot be written as a tensor product of states of the individual qubits, e.g., there are no states $|u\rangle$, $|v\rangle$ of a single qubit such that $|\psi\rangle = |u\rangle \otimes |v\rangle$, where $|\psi\rangle$ is the Bell state specified by Equation 2.3.

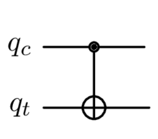
2.1.2 Quantum Gates

Quantum gates are means for altering a system's quantum state. And since the quantum state is a vector, it is quite natural to represent operations on quantum states as matrices. The only restriction on a matrix to represent a valid quantum gate is that it must be *unitary* (to maintain the principles of quantum mechanics). Therefore, all quantum gates are also reversible.

The simplest case of quantum gates are single-qubit gates, which can be represented as 2×2 matrices. Some examples are shown in Figure 2.1, the most important of these gates are the Pauli-X gate and the Hadamard gate. The X gate is an equivalent of a classical NOT gate. The Hadamard gate transforms the basis states $|0\rangle$ and $|1\rangle$ into equal superpositions and is a fundamental gate for many quantum algorithms.

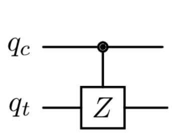
There are naturally also multi-qubit gates. Some examples can be seen in Figure 2.2, where q_c, q'_c are *control* qubits and q_t, q'_t are *target* qubits. The behaviour of multi-qubit gates is generally such that if all the control qubits are set to $|1\rangle$, the given operation is performed on the target qubits, else the target qubits' state is not altered. For example, CNOT and CCNOT may perform the X gate on the target qubit and Fredkin gate may swap the target qubits.

Then it is simple to update the system's quantum state, as the calculation is carried out as a matrix multiplication of the gate matrix with the system's state vector (example is shown in Figure 2.3). Sometimes, it may be necessary to modify the gate matrix so it has the right dimensions for the multiplication (e.g., when we apply X gate on one of the qubits in a 2-qubit system, as is depicted in Figure 2.4), since we must always apply the operation on the system's whole state vector due to a possible entanglement of qubits.



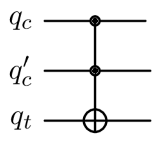
$$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&0&1\\0&0&1&0\end{pmatrix}$$

(a) Controlled-NOT (CNOT) gate



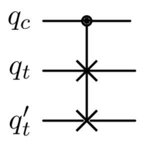
$$\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&-1\end{pmatrix}$$

(b) Controlled-Z (CZ) gate



$$\begin{pmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\\0&0&0&0&0&0&1&0\end{pmatrix}$$

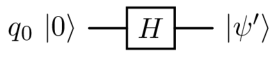
(c) Toffoli (CCNOT) gate



$$\begin{pmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\end{pmatrix}$$

(d) Fredkin (controlled-swap) gate

Figure 2.2: Examples of multi-qubit gates and their matrix representation



$$|\psi'\rangle = \tfrac{1}{\sqrt{2}}\begin{pmatrix}1&1\\1&-1\end{pmatrix}\cdot|0\rangle = \tfrac{1}{\sqrt{2}}\begin{pmatrix}1&1\\1&-1\end{pmatrix}\cdot\begin{pmatrix}1\\0\end{pmatrix} = \tfrac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix} = \tfrac{1}{\sqrt{2}}\cdot(|0\rangle+|1\rangle)$$

Figure 2.3: Example Hadamard gate application using the matrix representation

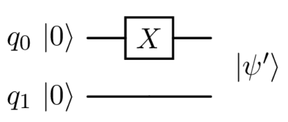
We achieve this by performing the tensor product of the properly sized identity matrix with the gate matrix.

It is also important to note that similarly as some sets of classical logical gates are *universal* (e.g., NAND), meaning they can express and thus compute any Boolean function, there are also universal quantum gate sets. Such a set is sufficient to create a quantum circuit approximating an arbitrary unitary operation with an arbitrary precision. Example of an universal set can be the Hadamard, CNOT, S, and T gate (the standard universal gate set, also called *Clifford + T* gate set) or the Hadamard, CNOT, S, and Toffoli gate.

2.1.3 Quantum Circuits

A *quantum circuit*, just like a classical circuit, consists of wires (which transfer information) and operations on qubits. The operations that alter the transferred information can be either quantum gates or qubit measurements. Every quantum circuit also has its input state, i.e., the initial state vector of the system. For an n-qubit circuit, this is usually $|0^n\rangle$ (or some other computational basis state).

However, there are also a few differences between quantum and classical circuits. Quantum circuits must be acyclic and also it is not possible to join the wires (performing logical



$$U = X \otimes \mathbb{I} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix}$$

$$|\psi'\rangle = U \cdot |00\rangle = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = |10\rangle$$

Figure 2.4: Pauli-X gate application in a 2-qubit system using the matrix representation (the gate matrix when applied to q_0 is denoted as U)

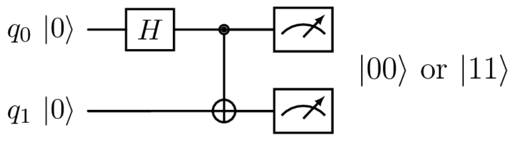


Figure 2.5: Example circuit that creates a Bell state with measurements at the end

OR on the inputs) as it is not reversible or split the wires (performing copies of the input), because it is not possible to duplicate qubits due to the no-cloning theorem [44, 11].

Figure 2.5 shows an example of a quantum circuit that has an initial state of $|00\rangle$ and it creates one of the Bell states $|q_0q_1\rangle = \frac{1}{\sqrt{2}} \cdot (|00\rangle + |11\rangle)$. Finally, both the qubits are measured and we get either the result $|00\rangle$ or $|11\rangle$. It is also worth noting that the wires in the quantum circuit (depicted as lines) are more abstract than the wires in classical circuit diagrams and may not strictly represent a physical wire—they simply visualize the flow of the qubits.

2.1.4 Quantum Algorithms

The main motivation behind the research of quantum computation is the potential to significantly surpass the limits of what is possible on classical computers for certain computational tasks. Currently, there are two main branches of algorithms that form this potential, *quantum search algorithms* and algorithms based on the *quantum Fourier transform*. The effectiveness of these algorithms is based on *interference* (interference of probability amplitudes can lead to the enhancement or suppression of specific measurement outcomes) and *quantum parallelism* (this concept allows processing of multiple possibilities simultaneously), both of which take advantage of quantum superposition.

The aim of *Grover's algorithm* [18], known as a quantum search algorithm, is to efficiently locate a specific item or solution within an unsorted search space by iteratively amplifying the probability of the correct solution. This algorithm offers quadratic speedup over the most efficient classical algorithms. Applications of quantum search include determining statistics of an unordered set or to speed up the solution of some NP problems.

To not be mistaken, the quantum Fourier transform (QFT) is analogous to the discrete Fourier transform and it is only more efficient when performed on a quantum state (it does not offer any speedup on classical data). However, thanks to the principles of QFT, we can perform *phase estimation* (estimate the phase of a unitary operator's eigenvalue). This is fundamental for the famous *Shor's algorithm*, providing an exponential speedup compared to classical algorithms for solving the problem of factorization and discrete logarithm. Note that both of these problems can be reduced to the problem of finding a period of a given periodic function—this can be effectively solved with the *period finding* [22] algorithm. Another example of an algorithm that utilizes phase estimation is *quantum counting* [4] (based on Grover's algorithm), which estimates the number of solutions to a computational problem in a quadratic speedup compared to classical counting algorithms.

Also worth mentioning is also the the *Bernstein-Vazirani algorithm* [3], which solves the following problem: given an oracle implementing a Boolean function $f(x) = x \cdot s$ ($f\colon \{0,1\}^n \to \{0,1\}$), find the hidden string s. Although this algorithm has little practical use, it provides a great demonstration of the capabilities of quantum computers (note that it does not fit neatly into either of the branches of quantum algorithms mentioned above).

2.2 Decision Diagrams

Decision diagrams are an important data structure used widely in formal verification, circuit design, artificial intelligence, and many more areas. There are many different types of decision diagrams, this section will however present only those necessary for the purposes of this thesis, namely Binary Decision Diagrams (BDDs) and Multi-Terminal Binary Decision Diagrams (MTBDDs). Note that this section draws, among other things, from [38]. In the following, we use $f|_{v_i=c}$ to denote a *restriction* of the function f (c is a constant value), i.e.,

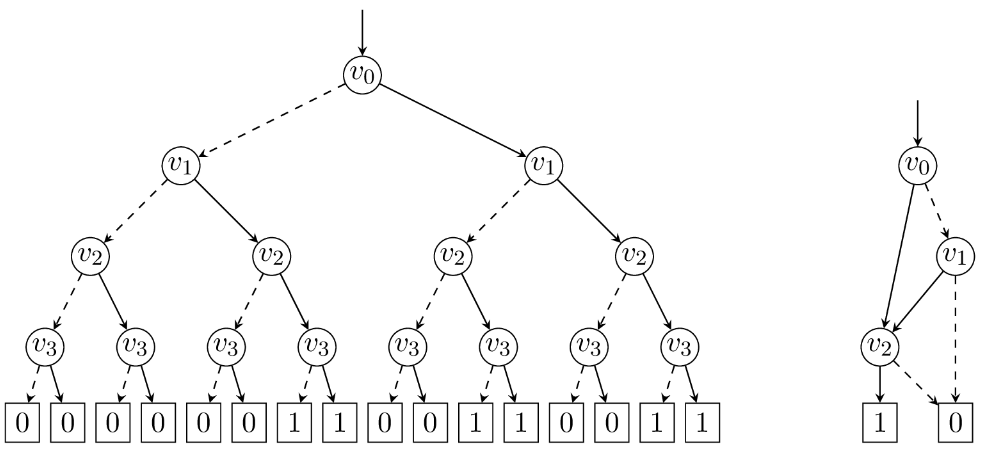
$$f|_{v_i=c}(v_0,\dots,v_i,\dots,v_{n-1}) = f(v_0,\dots,v_{i-1},c,v_{i+1},\dots,v_{n-1}).$$

2.2.1 Binary Decision Diagrams

A *Reduced Ordered Binary Decision Diagram* (ROBDD) is a data structure that can be efficiently used for encoding Boolean functions as was suggested by Bryant [5]. Commonly, ROBDDs are simply referred to as *Binary Decision Diagrams* (BDDs) and the same holds true throughout this work.

Let f be a Boolean function $f(v_0,\dots,v_{n-1})\colon \{0,1\}^n \to \{0,1\}$, where $\{v_0,\dots,v_{n-1}\} = V$, and let $\prec$ be a total ordering on V. Then a BDD representing f is a rooted directed acyclic graph (DAG) with two types of nodes—*nonterminal* (internal) nodes and *terminal* (leaf) nodes—satisfying the following properties:

- Each nonterminal node x corresponds to a single input variable v_i, $i \in \{0,\dots,n-1\}$. We denote this as $var(x) = v_i$.
- Each nonterminal node x has two child nodes $low(x)$ and $high(x)$, a *low* and a *high* successor, respectively. Suppose that $var(x) = v_i$. If $low(x)$ is a nonterminal node, then it holds that $var(low(x)) = v_j$ such that $v_i \prec v_j$. Likewise, if $high(x)$ is a nonterminal node, then it holds that $var(high(x)) = v_k$ such that $v_i \prec v_k$. It is worth noting that among other things, this ordering constraint ensures that the graph is acyclic.
- All terminal nodes x are of a value $val(x) \in \{0,1\}$ so that it holds $f_{root} = f$, where f_{root} is the Boolean function represented by the root node of the BDD. A general definition for a Boolean function f_x represented by a node x is as follows:
    1. If x is a terminal node, then $f_x = val(x)$.



(a) Binary decision tree (b) BDD

Figure 2.6: Different representations of the Boolean function $f(v_0, v_1, v_2, v_3)\colon (v_0 \vee v_1) \wedge v_2$, where $v_0 \prec v_1 \prec v_2 \prec v_3$

2. If x is a nonterminal node and $\mathit{var}(x) = v_i$, then $f_x = \overline{v_i} \cdot f_{\mathit{low}(x)} + v_i \cdot f_{\mathit{high}(x)}$.

I.e., $\mathit{val}(x)$ equals to the evaluation of f, where all the input variables v_i, for $i \in \{0, \ldots, n-1\}$ are assigned a truth value based on whether $\mathit{low}(y_i)$ or $\mathit{high}(y_i)$ is on the path to x (y_i is the node corresponding to v_i, i.e., $\mathit{var}(y_i) = v_i$).

- There is no node x such that $\mathit{low}(x) = \mathit{high}(x)$. Also, there are no two distinct nodes x_1, x_2 such that they represent the same Boolean function $f_{x_1} = f_{x_2}$.

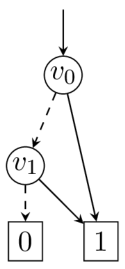
It is also important to mention that this form of representation of Boolean functions is canonical (w.r.t. $\prec$) thanks to the reduction property. Note that throughout this thesis, we often abuse notation and use a BDD and the Boolean function it represents interchangeably.

If we were to construct a BDD without removing any redundant isomorphism (i.e., not removing any unnecessary nodes with identical successors and not merging nodes that represent the same Boolean function), meaning we would construct a diagram representing a Boolean function f only based on the *Shannon's expansion*:

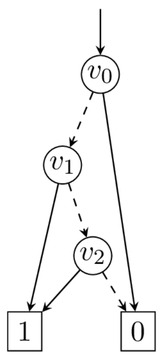
$$f(v_0, \ldots, v_i, \ldots, v_{n-1}) = \overline{v_i} \cdot f|_{v_i=0}(v_0, \ldots, v_{n-1}) + v_i \cdot f|_{v_i=1}(v_0, \ldots, v_{n-1}), \tag{2.4}$$

we would get a *complete binary decision tree.* An example of a BDD compared to a complete binary decision tree can be seen in Figure 2.6—even a simple Boolean function clearly benefits from the compact representation BDDs offer. The subgraph sharing does not only provide efficiency in the sheer size of the representation, but also for further performance of BDD algorithms, as the operation is performed on the subgraph only once. It can be seen that the graph does not contain any redundant nodes, e.g., representing the variable v_4, which does not affect the result of the function in any way. Throughout this work, we will depict BDDs and their variants in such a way that the nonterminal nodes are circular, the leaf nodes are rectangular, a dashed edge denotes a low successor and a solid edge denotes a high successor.

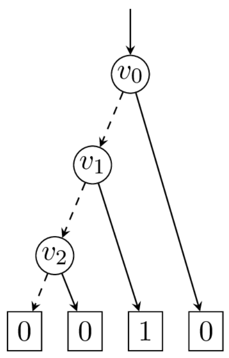
For the purposes of the following algorithms, we use $\mathit{root}(\beta)$ to denote the root node of a BDD β and $|\beta|$ to denote the number of all nodes (both terminal and nonterminal) of β. In addition to this, we use $\mathit{id}(x)$ to denote a unique identifier of the node x, where



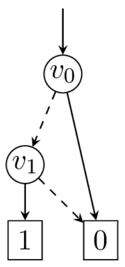
(a) BDD β_1 representing $f_1(v_0, v_1, v_2)\colon v_0 \vee v_1$, where $v_0 \prec v_1 \prec v_2$



(b) BDDs β_2 representing $f_2(v_0, v_1, v_2)\colon \overline{v_0} \wedge (v_1 \vee v_2)$, where $v_0 \prec v_1 \prec v_2$



(c) Result of $\mathit{ApplyFrom}(\mathit{root}(\beta_1), \mathit{root}(\beta_2), \wedge)$



(d) Final result of the *Apply* after reduction

Figure 2.7: Example of $\mathit{Apply}(\beta_1, \beta_2, \wedge)$, where $\wedge$ is the standard AND operation

$\mathit{id}(x) \in \{0, \dots, |\beta| - 1\}$. Further, let us define $\mathit{IsLeaf}(x)$ to be a function which returns *true* if the node x is a terminal node and *false* otherwise. Also we assume that if $\mathit{IsLeaf}(x) = \mathit{true}$ in a BDD β representing $f(v_0, \dots, v_{n-1})$, it holds that $\mathit{var}(x) = n$.

To perform operations on BDDs, a standard procedure *Apply* is used. Let $*_{op}$ be a binary operator and let β_1, β_2 be BDDs that represent Boolean functions $f_1(v_0, \dots, v_{n-1})$ and $f_2(v_0, \dots, v_{n-1})$, respectively. Result of $\mathit{Apply}(\beta_1, \beta_2, *_{op})$ is the BDD β representing the function $f_1 *_{op} f_2$. The semantics of this procedure is shown in Algorithm 1. It consists of two operations—a recursive function implementing the *Apply* itself followed by reducing the resulting BDD (i.e., removing redundant isomorphism in the DAG as described at the beginning of this section).

The basis of $\mathit{Apply}(\beta_1, \beta_2, *_{op})$ is formed by the following recursion:

$$f_1 *_{op} f_2 = \overline{v_i} \cdot (f_1|_{v_i=0} *_{op} f_2|_{v_i=0}) + v_i \cdot (f_1|_{v_i=1} *_{op} f_2|_{v_i=1}).$$

It is easy to see that the algorithm is based on Shannon's expansion of Boolean functions (see Equation 2.4). Note that the recursion shown in Algorithm 1 is only a naive implementation of *Apply*, where it is possible that *Apply* would be evaluated over certain sub-DAGs of β_1 and β_2 multiple times. In practice, the implementation of *Apply* therefore uses result caching, where we check if the *Apply* result over these subgraphs is already known before proceeding with a further recursion. The main idea is that the algorithm traverses both

BDDs simultaneously and performs the given operation $*_{op}$ on the corresponding leaf nodes of β_1 and β_2, while trying to unfold the structure of these diagrams as little as possible.

Because β_1 and β_2 can be reduced in different ways (e.g., not all nonterminal nodes from β_1 have to be in β_2) the resulting DAG is not necessarily a BDD and we have to reduce it. This operation is described closely in Algorithm 2, however, the main principle of the algorithm is that we sort all the BDD nodes into lists according to the variables they represent, and then we process these lists from the leaf nodes upwards, gradually removing redundant nodes. The time complexity of *Reduce* on BDD β is $O(|\beta| \cdot \log |\beta|)$ (it is influenced mainly by the time needed to sort the lists). The time complexity of the whole *Apply* operation if we use caching is $O(|\beta_1| \cdot |\beta_2|)$. An example of running the *Apply* procedure can be seen in Figure 2.7.

Intuitively, sometimes we may want to perform a unary operation on the BDD. In this case, we can just simplify the recursion in the binary *Apply* procedure so that it structurally traverses the specified BDD and if the currently processed node is a leaf, it performs the specified operation on it. Throughout this thesis, we refer to this operation as a *unary Apply*.

Algorithm 1: *Apply* procedure

```
Input: root nodes of BDDs $\beta_1$, $\beta_2$ representing Boolean functions $f_1(v_0, \dots, v_{n-1})$
       and $f_2(v_0, \dots, v_{n-1})$, respectively, where $\{v_0, \dots, v_{n-1}\}$ is ordered w.r.t. $\prec$,
       and a binary operator $*_{op}$
Output: root node of BDD $\beta$ representing $f_1 *_{op} f_2$
begin
    $root(\beta) \leftarrow$ ApplyFrom($root(\beta_1), root(\beta_2), *_{op}$);
    return Reduce($root(\beta)$);
end
Function ApplyFrom($x_1$, $x_2$: nodes, $*_{op}$: binary operator) : node is
    $x \leftarrow New$(node);
    if $IsLeaf(x_1)$ and $IsLeaf(x_2)$ then
        $val(x) \leftarrow val(x_1) *_{op} val(x_2)$;
    else
        if $var(x_1) = var(x_2)$ then
            $var(x) \leftarrow var(x_1)$;
            $low(x) \leftarrow$ ApplyFrom($low(x_1), low(x_2)$);
            $high(x) \leftarrow$ ApplyFrom($high(x_1), high(x_2)$);
        else if $var(x_1) \prec var(x_2)$ or $IsLeaf(x_2)$ then
            $var(x) \leftarrow var(x_1)$;
            $low(x) \leftarrow$ ApplyFrom($low(x_1), x_2$);
            $high(x) \leftarrow$ ApplyFrom($high(x_1), x_2$);
        else
            $var(x) \leftarrow var(x_2)$;
            $low(x) \leftarrow$ ApplyFrom($x_1, low(x_2)$);
            $high(x) \leftarrow$ ApplyFrom($x_1, high(x_2)$);
        end
    end
    return $x$;
end
```

Algorithm 2: Reduction of a BDD β representing function $f(v_0, \ldots, v_{n-1})$ [5]

```
Function Reduce($root$: node) : node is
    $reducedGraphNodes \leftarrow New(\text{array}[|\beta|])$;
    $nodeList \leftarrow New(\text{array}[n+1])$;
    put each node $x$ of the BDD on the list at $nodeList[var(x)]$;
    $nextId \leftarrow -1$;
    for $i \leftarrow n$ to 0 do
        $currentNodes \leftarrow \emptyset$;
        for each $x$ in $nodeList[i]$ do
            if $IsLeaf(x)$ then
                add <$key$, $x$> to $currentNodes$, $key = val(x)$;
            else if $id(low(x)) = id(high(x))$ then
                // Redundant node
                $id(x) \leftarrow id(low(x))$;
            else
                add <$key$, $x$> to $currentNodes$, $key = (id(low(x)), id(high(x)))$;
            end
        end
        sort elements of $currentNodes$ by $key$;
        $oldKey \leftarrow (-1; -1)$; // Initialize as an unmatchable key
        for each <$key$, $x$> in $currentNodes$ removed in order do
            if $key = oldkey$ then
                // Matches an existing node
                $id(x) \leftarrow nextId$;
            else
                // Unique node
                $nextId \leftarrow nextId + 1$;
                $id(x) \leftarrow nextId$;
                $reducedGraphNodes[nextId] \leftarrow x$;
                $low(x) \leftarrow reducedGraphNodes[id(low(x))]$;
                $high(x) \leftarrow reducedGraphNodes[id(high(x))]$;
                $oldKey \leftarrow key$;
            end
        end
    end
    return $reducedGraphNodes[id(root)]$;
end
```

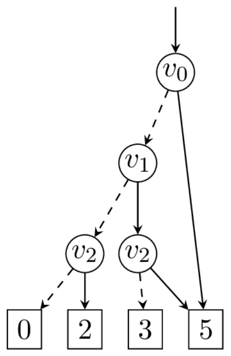


Figure 2.8: Example MTBDD for the function $f(v_0, v_1, v_2) = \begin{cases} 3v_1 + 2v_2 & \text{if } \overline{v_0}, \\ 5 & \text{otherwise,} \end{cases}$ where $v_0 \prec v_1 \prec v_2$ and $+$ in this case represents arithmetic addition

2.2.2 Multi-Terminal Binary Decision Diagrams

There are many modifications of BDDs, one of them being *Multi-Terminal Binary Decision Diagrams* (MTBDDs) [15]. MTBDDs are a generalised variant of BDDs, the only difference is that MTBDD's terminal nodes can have an arbitrary value, not only 0 or 1. Because of that, MTBDDs can represent any function $f(v_0, \ldots, v_{n-1})\colon \{0,1\}^n \to \mathbb{D}$, for any $\mathbb{D} \neq \varnothing$ with finitely representable elements. Thus, MTBDDs provide efficient representation of matrices, especially sparse matrices. Not only that, but the complexity of basic matrix operations performed on MTBDDs is not greater than the complexity of these operations over any standard matrix representation. As with BDDs, this representation is also canonical (w.r.t. the variable ordering). An example MTBDD is shown in Figure 2.8.

Given the little differences between BDDs and MTBDDs, one can easily extend the previously mentioned *Apply* algorithm (Algorithm 1) to MTBDDs as well. We simply let $*_{op}$ be a map $\mathbb{D}_1 \times \mathbb{D}_2 \to \mathbb{D}$, where $\mathbb{D}_1$, $\mathbb{D}_2$ are the value domains for MTBDDs μ_1 and μ_2, respectively, and $\mathbb{D} \neq \varnothing$. Just as with BDDs, we use caching and result reusing to great advantage in MTBDD *Apply.* Of course, the *unary Apply* procedure can be trivially extended to MTBDDs as well.

Chapter 3

Previous Work on Quantum Circuit Simulation

The aim of this chapter is to provide an overview of the state-of-the-art quantum circuit simulators and to delve into the main principles these tools are based on. The emphasis in this chapter is only on an overview of the data structures used, for more detailed information and comparison of these approaches one can refer to [36, 41, 32], from which this chapter draws. However, the main data structures used in the classical simulation of quantum circuits can be divided into the following categories: vectors and matrices, i.e. arrays (although not very practical), various variants of decision diagrams, tensor networks, and ZX-calculus. As with all data structures, there is a question of the ideal compromise between compactness of the representation and efficiency of manipulation operations.

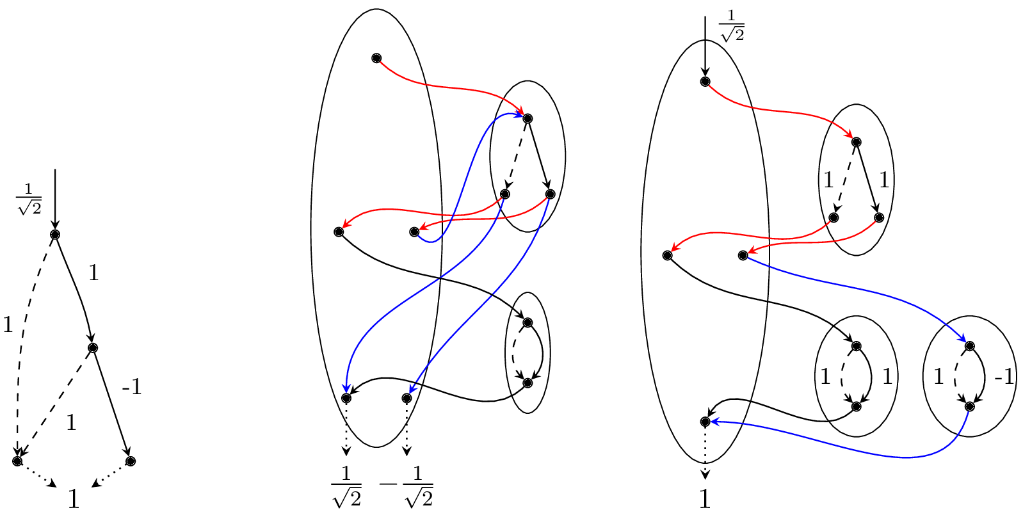
3.1 Vector-based Approach

As mentioned in Section 2.1.2, simulation based on vector representation of the quantum state and matrix representation of the applied gates comes quite naturally. However, this approach has some serious limitations in terms of scalability of the simulated circuits, as these structures (the state vector and gate matrices) grow exponentially in size w.r.t. the number of qubits in the circuit. Due to these memory requirements, this simulation method is practically unusable for even slightly larger circuits (with roughly 50 and more qubits) [28].

This approach is available as a native simulator backend option for the open-source `Qiskit` framework [43] developed by IBM. `Qiskit` provides a set of tools for both defining and simulating quantum circuits, as well as connecting to IBM's cloud services to run quantum circuits on real quantum hardware.

3.2 Decision Diagram-based Approaches

Approaches based on some variant of decision diagrams are very popular when simulating quantum circuits. These data structures provide a compact representation of quantum states and operations on them by taking advantage of existing redundancies. The main idea behind this representation is that we assume the individual system's basis states determine the path through which we traverse the diagram. Then by obtaining the leaf value for this path, we obtain the probability amplitude corresponding to this base vector.



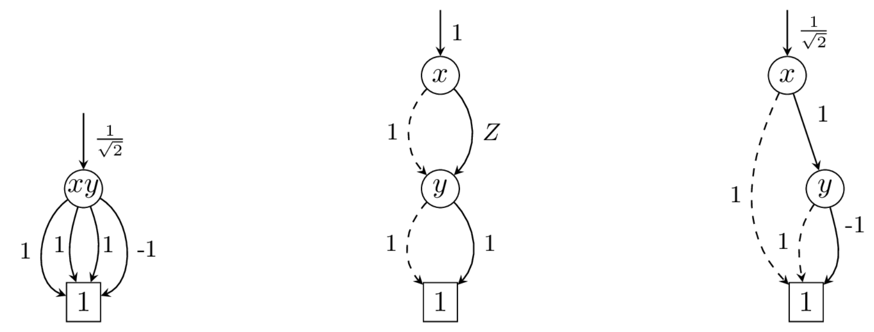
(a) WBDD (b) CFLOBDD (c) WCFLOBDD

Figure 3.1: Different representations of the Hadamard gate matrix [31] (see Figure 2.1d) that are used in `Quasimodo`. The first variable is the row variable, the second is the column variable, and exit edges are denoted as dotted arrows. These diagrams use the same notation as in the original paper, where leaf values are denoted by both a node and an exit edge and the nodes are not labeled with their variables (otherwise the diagrams would not be clear for CFLOBDDs and WCFLOBDDs, which use variable sharing). For better readability, some edges are coloured.

Classic BDDs are used, for example, in `SliQSim` [33]. This simulator also utilizes the algebraic complex number representation further introduced in Section 4.1. Then the system's quantum state is represented with a series of BDDs, where each BDD represents a single bit from one of the integers, which together form a single probability amplitude (calling this method *bit-slicing*).

However, modifications of BDDs are used more often than regular BDDs for quantum circuit simulation. These modifications include the previously mentioned MTBDDs, used, for example, by `Quasimodo` [30]. This tool can operate on several built-in data structures, including, among others, *Weighted Binary Decision Diagrams* (WBDDs). They differ from regular BDDs in that the edges in a WBDD are weighted, in this case with a complex value. Then, when evaluating a leaf value, all the weights on the path to this leaf node are multiplied with the leaf value.

Another variant on BDDs used in `Quasimodo` are *Context-Free-Language Ordered Binary Decision Diagrams* (CFLOBDDs) [29] and *Weighted Context-Free-Language Ordered Binary Decision Diagrams* (WCFLOBDDs) [31]. The difference between CFLOBDDs and BDDs is that CFLOBDDs allow certain procedure calls within their structure (by reusing *groupings*), which can offer additional compactness. This means that CFLOBDDs can reuse not only sub-DAGs like BDDs, but also the „middle of the DAG", which can result in structures that are (in the best case) exponentially more succinct than BDDs. WCFLOBDDs are CFLOBDDs where certain edges are weighted (specifically the entry edge and the edges of the innermost groupings). Also the terminal values must be binary. Examples of these representations used in `Quasimodo` can be seen in Figure 3.1. In CFLOBDDs and WCFLOBDDs, the ovals represent the individual variable sharing groupings.



(a) QMDD [26] (b) LIMDD [35] (c) TDD

Figure 3.2: Different decision diagram representations of the Hadamard gate matrix (see Figure 2.1d). In these figures, x denotes the row variable, and y denotes the column variable.

In the above-mentioned cases, quantum gates are represented as certain operations over the given decision diagrams. Another approach is to represent quantum operations using *Quantum Multiple-valued Decision Diagrams* (QMDDs) [26], which is relatively common in practice, for example in the `DDSIM` simulator [47]. The main idea is that QMDDs can succinctly represent gate matrices by repeatedly partitioning such matrix into 4 sub-matrices. Each nonterminal node in the QMDD therefore has 4 successors and the edges in this decision diagram are again weighted. Node merging in QMDDs happens up to constant complex factors (unlike with BDDs or MTBDDs, where two nodes representing functions f, g are only merged if $f = g$).

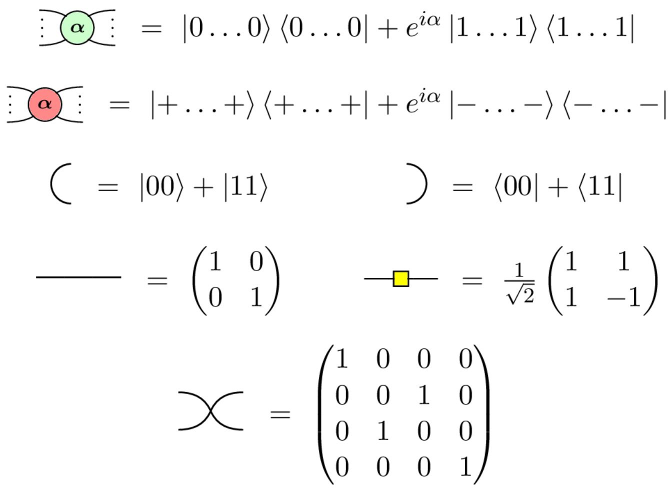
It is also worth mentioning the *Local Invertible Map Decision Diagrams* (LIMDDs) [35, 37], which are a generalization of QMDDs. LIMDDs can be more succinct than QMDDs due to a different node merging strategy allowing also merging of nodes that are equivalent up to tensor product of single-qubit Pauli gate operations. However, as usual, the price for this succinctness is a slowdown of manipulation operations, precisely a cubic factor overhead w.r.t. QMDDs.

Another interesting example are *Tensor Decision Diagrams* (TDDs) [20], which combine decision diagrams with tensor networks (more about them in the next section). This data structure extends the concept of BDDs and provides a way to efficiently represent and manipulate high-dimensional tensors. TDDs also use weighted edges and the individual nonterminal nodes in a TDD represent indices in the tensor network representing the given quantum circuit. Examples of these decision diagrams (QMDDs, LIMDDs and TDDs) can be seen in Figure 3.2.

It is also worth noting that most of the above mentioned tools (except `SliQSim`, which uses the algebraic representation of complex numbers described in more detail in Section 4.1 and is thus accurate) use floating point numbers for complex number representation, which can lead to numerical instabilities [46, 27]. Accurate simulation is also critical for tasks such as quantum circuit equivalence checking [39].

3.3 Tensor Networks-based, ZX-calculus-based, and Other Approaches

We can also take advantage of redundancies present in the vector-based simulation with *tensor networks* (TNs). In such a simulation, a quantum state and gates are represented by



$$= |0\ldots0\rangle\langle0\ldots0| + e^{i\alpha}|1\ldots1\rangle\langle1\ldots1|$$

$$= |+\ldots+\rangle\langle+\ldots+| + e^{i\alpha}|-\ldots-\rangle\langle-\ldots-|$$

$$= |00\rangle + |11\rangle \qquad = \langle00| + \langle11|$$

$$= \begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} \qquad = \frac{1}{\sqrt{2}}\begin{pmatrix}1 & 1\\ 1 & -1\end{pmatrix}$$

$$= \begin{pmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}$$

Figure 3.3: ZX-diagram examples and their semantics [40]

tensors, therefore we can view a quantum circuit as a tensor network. Then the simulation of a quantum circuit can be performed through *contraction* of its TN.

Special types of TNs are used to decompose high-dimensional tensors into lower-dimensional tensors, such as *Matrix Product States* (MPS). With MPS, a quantum state is represented as a product of matrices. This simulation method is used, e.g., as one of the available simulator backend options for the `Qiskit` framework. This method seems to be efficient when the amount of entanglement between qubits is limited [34].

Another formalism we can use to simulate quantum circuits is a graphical language called *ZX-calculus* [40], introduced by Coecke and Duncan in 2008 [9]. It consists of *ZX-diagrams*, schemes built from wires and two types of nodes (Z-spiders and X-spiders) similar to a classical quantum circuit diagrams, and a set of their rewrite (simplification) rules. Some ZX-diagrams and their semantics can be seen in Figure 3.3.

For the purposes of simulation, we would like to perform automated rewriting of ZX-diagrams and therefore we use more restricted ZX-diagrams called *graph-like* ZX-diagrams (e.g., they may only contain Z-spiders and not X-spiders). Then the simulation itself is done by further decomposition and simplification of this graph-like ZX-diagram. In this context, it is important to mention the `QuiZX` [21] simulator, which is based on the ZX-calculus.

Of course, this is not an exhaustive list of all the methods used to simulate quantum circuits, but only an overview of the most commonly used used ones. An example of a tool that takes a completely different approach is `Quokka#` [24]. This simulator encodes the quantum circuit into the Boolean model counting problem of a formula in the *conjunctive normal form* (CNF) and then solves these constraints by a weighted model counter.

Chapter 4

MTBDD-based Quantum Circuit Representation

This chapter discusses the fundamentals specific for this implementation of a quantum circuit simulator. These concepts are crucial in ensuring that the simulator can operate at the required speed with reasonable computing power, while maintaining accuracy. Before we get to the MTBDD-based simulation itself, it is necessary to introduce an algebraic representation of complex numbers, which is used for MTBDD's leaf node values. Then we take a closer look at how MTBDDs are used for the simulation itself, both for representing the quantum state of the system and performing the quantum gate operations on this quantum state.

4.1 Algebraic Representation of Complex Numbers

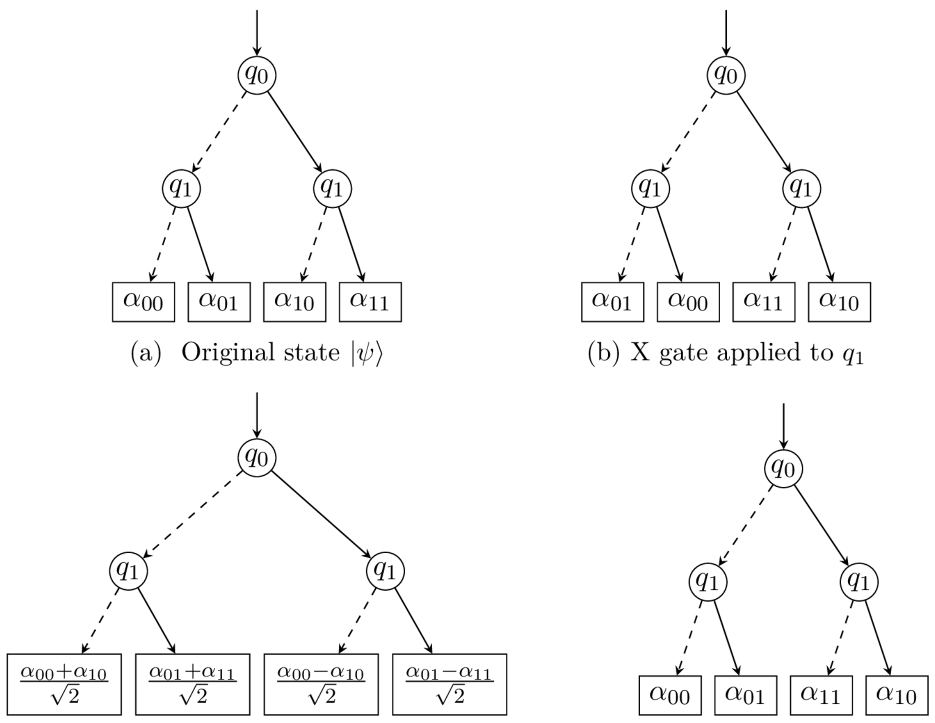
One of the problems quantum simulators face is the complex number representation as it has to be accurate yet it must not slow down the whole implementation. In this work, we utilize the algebraic representation presented and used in [33, 8, 7], which was originally proposed in [27].

The formula for exact algebraic representation is for $z \in \mathbb{C}$ as follows:

$$z = \left(\frac{1}{\sqrt{2}}\right)^k \cdot (a + b\omega + c\omega^2 + d\omega^3), \tag{4.1}$$

where $a, b, c, d, k \in \mathbb{Z}$ and $\omega = e^{\frac{i\pi}{4}}$ (it is the unit vector which makes an angle of 45° with the positive part of the real axis of the complex plane). Thus, a complex number can be represented as an integer 4-tuple (a, b, c, d) and the coefficient k for normalization. Not only is this encoding convenient by itself, it is also trivial to multiply a complex number in this form by ω or its power, as $\omega^4 = -1$ (one simply needs to rotate the four coefficients and negate some of them accordingly). This is very useful as many quantum gates multiply the state vector by some power of ω.

Even though this method does not cover the entire set of complex numbers ($\mathbb{C}$ is uncountable while $\mathbb{Z}^5$ is countable), the subset is sufficient for a quantum circuit simulation without loss of generality. Not only is it able to represent with perfect accuracy all operations that can be realized exactly by the universal Clifford + T gate set, but since this subset is a dense subset of $\mathbb{C}$, it allows to approximate any operation and quantum state with arbitrary precision.


q_0
q_1
α_{00} α_{01} α_{10} α_{11}
α_{01} α_{00} α_{11} α_{10}
$\frac{\alpha_{00}+\alpha_{10}}{\sqrt{2}}$ $\frac{\alpha_{01}+\alpha_{11}}{\sqrt{2}}$ $\frac{\alpha_{00}-\alpha_{10}}{\sqrt{2}}$ $\frac{\alpha_{01}-\alpha_{11}}{\sqrt{2}}$
α_{00} α_{01} α_{11} α_{10}

(a) Original state $|\psi\rangle$

(b) X gate applied to q_1

(c) Hadamard gate applied to q_0

(d) CNOT gate (the result is the same for both combinations of control and target qubits)

Figure 4.1: Example of MTBDD-based quantum state representation and its transformations caused by gate application (a dashed edge denotes the value $|0\rangle$, solid edge denotes the value $|1\rangle$)

4.2 Quantum Circuit Representation Using MTBDDs

As stated in Chapter 3.1, the classic representation of quantum state as a vector is not very convenient. Not only does the size of the system's state vector itself grow exponentially with the number of qubits, meaning it is of the length 2^n for an n-qubit system, but this also means that one needs $2^n \times 2^n$ matrices for representing quantum gates. Instead, the implemented simulator uses an MTBDD to represent the system's quantum state, which greatly reduces the memory requirements of the tool.

This representation is quite intuitive, as we will show in an example. Let us assume we have an arbitrary quantum circuit with two qubits, q_0 and q_1. We describe the state $|\psi\rangle$ of this system as

$$|\psi\rangle = \alpha_{00} \cdot |00\rangle + \alpha_{01} \cdot |01\rangle + \alpha_{10} \cdot |10\rangle + \alpha_{11} \cdot |11\rangle\,. \tag{4.2}$$

The MTBDD representation of this state $|\psi\rangle$ is shown in Figure 4.1a. The graph is read so that the low successor (dashed) represents 0 and the high successor (solid) represents 1 in the computational basis state and the leaf value is then the corresponding probability amplitude of this basis state. In other words, we view the system's state as a function $f\colon \{0,1\}^n \to \mathbb{C}$, where the evaluation of input variables corresponds to a computational basis state and the value of this function for the given input corresponds to the probability amplitude of this basis state. To clarify, in the implementation itself, complex probability amplitudes are represented algebraically according to Equation 4.1. This means that the leaves of the state's MTBDD actually contain an integer 4-tuple (a, b, c, d). The coefficient k is kept separate because we only modify this coefficient with gate operations; the gate operations always alter the k coefficients of the complex amplitudes of the whole system in the same way (they always have the same value throughout the whole MTBDD). This means that we cannot use the smallest possible k to make this representation unique [27]. However, this is not an issue from a practical point of view because the uniqueness is clearly achieved as the value of k is fixed for the whole MTBDD (i.e., it is not possible that different leaves representing the same value are present in this MTBDD). It is worth re-emphasizing that this representation is perfectly accurate for the currently supported gates (see Table 4.1).

Application of gates on this representation of the state vector consists of transforming the MTBDD according to the matrix representation of the gates—see Figure 4.1 for some examples. The simulation uses two approaches to apply gates on the state's MTBDD. In the first approach, the gate application is executed using the universal update formulae as a sequence of operations over the MTBDD using the standard *Apply* procedure. The second approach uses permutation-based update formulae, which execute the gate application in a single custom *Apply*. This results in a less computationally demanding operation, but the drawback is that this approach can only be used for single qubit gates.

4.2.1 Universal Update Formulae For Quantum Gates

This method of performing the gate operations using universal update formulae for the system's MTBDD was presented in [8]. The main idea of this approach is that a sequence of elementary operations is used to construct the resulting MTBDD. These operations are on the one hand classical arithmetic operations using the standard *Apply* procedure and also *projection* and *restriction*.

Let $T(b_0, ..., b_{n-1})\colon \{0,1\}^n \to \mathbb{Z}^4$ be the function that is represented by the system's MTBDD. Projections T_{q_t} and $T_{\overline{q_t}}$ are used to fix the value of the target qubit q_t to 1 or 0, respectively:

$$\begin{aligned} T_{q_t}(b_0, ..., b_t, ..., b_{n-1}) &= T(b_0, ..., 1, ..., b_{n-1}) \\ T_{\overline{q_t}}(b_0, ..., b_t, ..., b_{n-1}) &= T(b_0, ..., 0, ..., b_{n-1}). \end{aligned}$$

Restrictions B_{q_t}, $B_{\overline{q_t}}$ only output the value (or the complemented value, respectively) of the target qubit q_t in the given basis state, i.e.,

$$\begin{aligned} B_{q_t}(b_0, ..., b_t, ..., b_{n-1}) &= b_t \\ B_{\overline{q_t}}(b_0, ..., b_t, ..., b_{n-1}) &= \overline{b_t}. \end{aligned}$$

If we look at these operations directly from a DAG point of view, projections T_{q_t}, $T_{\overline{q_t}}$ set both target qubits successors to the same subgraph—the subgraph of the high successor in the case of T_{q_t} and the subgraph of the low successor in the case of $T_{\overline{q_t}}$. The restriction B_{q_t} consists of construction of a diagram that represents the function

$$f(b_t) = \begin{cases} 1 & \text{if } b_t, \\ 0 & \text{if } \overline{b_t}. \end{cases}$$

Similarly performing $B_{\overline{q_t}}$ constructs a diagram representing

$$f(b_t) = \begin{cases} 1 & \text{if } \overline{b_t}, \\ 0 & \text{if } b_t. \end{cases}$$

Table 4.1: Universal update formulae for MTBDD gate application [8] (target qubits are denoted as q_t and control qubits are denoted as $q_c, q_{c'}$, if the gate uses them)

Gate	Update formula
$\mathrm{X}[q_t]$	$B_{\overline{q_t}} \cdot T_{q_t} + B_{q_t} \cdot T_{\overline{q_t}}$
$\mathrm{Y}[q_t]$	$\omega^2 \cdot (B_{q_t} \cdot T_{\overline{q_t}} - B_{\overline{q_t}} \cdot T_{q_t})$
$\mathrm{Z}[q_t]$	$B_{\overline{q_t}} \cdot T - B_{q_t} \cdot T$
$\mathrm{H}[q_t]$	$\frac{1}{\sqrt{2}} \cdot (T_{\overline{q_t}} + B_{\overline{q_t}} \cdot T_{q_t} - B_{q_t} \cdot T)$
$\mathrm{S}[q_t]$	$B_{\overline{q_t}} \cdot T + \omega^2 \cdot B_{q_t} \cdot T$
$\mathrm{T}[q_t]$	$B_{\overline{q_t}} \cdot T + \omega \cdot B_{q_t} \cdot T$
$\mathrm{Rx}\left(\frac{\pi}{2}\right)[q_t]$	$\frac{1}{\sqrt{2}} \cdot (T - \omega^2 \cdot (B_{q_t} \cdot T_{\overline{q_t}} + B_{\overline{q_t}} \cdot T_{q_t}))$
$\mathrm{Ry}\left(\frac{\pi}{2}\right)[q_t]$	$\frac{1}{\sqrt{2}} \cdot (T_{\overline{q_t}} + B_{q_t} \cdot T - B_{\overline{q_t}} \cdot T_{q_t})$
$\mathrm{CNOT}[q_c, q_t]$	$B_{\overline{q_c}} \cdot T + B_{q_c} \cdot (B_{\overline{q_t}} \cdot T_{q_t} + B_{q_t} \cdot T_{\overline{q_t}})$
$\mathrm{CZ}[q_c, q_t]$	$B_{\overline{q_c}} \cdot T + B_{q_c} \cdot (B_{\overline{q_t}} \cdot T - B_{q_t} \cdot T)$
$\mathrm{Toffoli}[q_c, q_{c'}, q_t]$	$B_{\overline{q_c}} \cdot T + B_{q_c} \cdot (B_{\overline{q_{c'}}} \cdot T + B_{q_{c'}} \cdot (B_{\overline{q_t}} \cdot T_{q_t} + B_{q_t} \cdot T_{\overline{q_t}}))$

Using all the previously mentioned operations, we can represent the semantics of quantum gate application on the system's MTBDD using update formulae. All operations from now on use a shorthand notation, e.g., B_{q_t} instead of $B_{q_t}(b_0, ..., b_{n-1})$, since all the further shown operations are over $b_0, ..., b_{n-1}$. The update formulae for all the supported gates are shown in Table 4.1. As an example, consider the formula for the X gate. First, let us take a look at the semantics of this gate. Assume a two qubit quantum circuit with qubits q_0, q_1 whose state $|\psi\rangle$ is described by Equation 4.2. The operation $\mathrm{X}[q_0]$ could be then expressed as

$$|\psi\rangle = \alpha_{10} \cdot |00\rangle + \alpha_{11} \cdot |01\rangle + \alpha_{00} \cdot |10\rangle + \alpha_{01} \cdot |11\rangle \,. \tag{4.3}$$

The projections in the formula give us MTBDDs representing these states

$$T_{q_0} = \alpha_{10} \cdot |00\rangle + \alpha_{11} \cdot |01\rangle + \alpha_{10} \cdot |10\rangle + \alpha_{11} \cdot |11\rangle$$
$$T_{\overline{q_0}} = \alpha_{00} \cdot |00\rangle + \alpha_{01} \cdot |01\rangle + \alpha_{00} \cdot |10\rangle + \alpha_{01} \cdot |11\rangle \,.$$

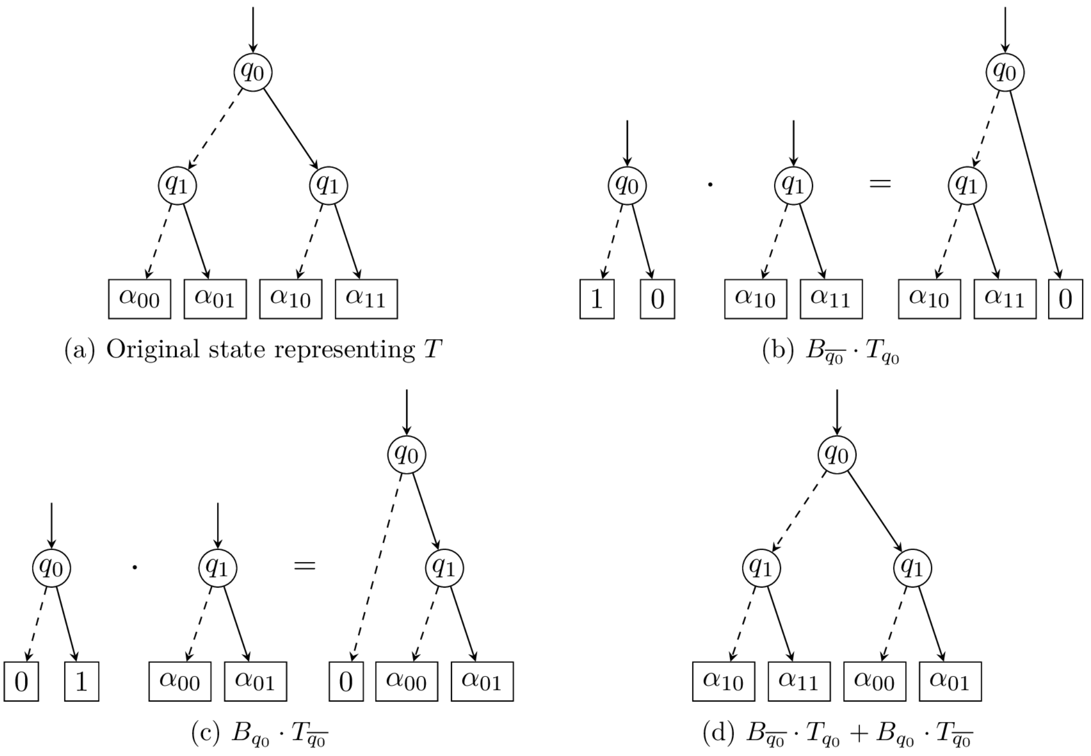
If we combine it with the restrictions we get

$$B_{\overline{q_0}} \cdot T_{q_0} = 1 \cdot \alpha_{10} \cdot |00\rangle + 1 \cdot \alpha_{11} \cdot |01\rangle + 0 \cdot \alpha_{10} \cdot |10\rangle + 0 \cdot \alpha_{11} \cdot |11\rangle = \alpha_{10} \cdot |00\rangle + \alpha_{11} \cdot |01\rangle$$
$$B_{q_0} \cdot T_{\overline{q_0}} = 0 \cdot \alpha_{00} \cdot |00\rangle + 0 \cdot \alpha_{01} \cdot |01\rangle + 1 \cdot \alpha_{00} \cdot |10\rangle + 1 \cdot \alpha_{01} \cdot |11\rangle = \alpha_{00} \cdot |10\rangle + \alpha_{01} \cdot |11\rangle .$$

and when we add those two intermediate results up, we get exactly the expression in Equation 4.3. This particular operation is illustrated using decision diagrams in Figure 4.2.

4.2.2 Permutation-based Update Formulae For Quantum Gates

Though the universal update formulae can be used for all the gates mentioned in Table 4.1, we use permutation-based update formulae instead for all the mentioned single-qubit gates (X, Y, Z, H, S, T, $\mathrm{Rx}\left(\frac{\pi}{2}\right)$, $\mathrm{Ry}\left(\frac{\pi}{2}\right)$) and the CZ gate. The advantage of this approach is that it allows us to execute the gate application in a single custom *Apply* rather than in a sequence of operations, which allows for a less computationally intensive workload. However, the drawback is that since MTBDDs are DAGs, once we process a node, we cannot return to it later during a single traversal of the MTBDD. However, if the control qubit has a greater



(a) Original state representing T (b) $B_{\overline{q_0}} \cdot T_{q_0}$ (c) $B_{q_0} \cdot T_{\overline{q_0}}$ (d) $B_{\overline{q_0}} \cdot T_{q_0} + B_{q_0} \cdot T_{\overline{q_0}}$

Figure 4.2: Execution of X$[q_0]$ using the universal update formula, expressed in terms of decision diagrams

index than the target qubit, i.e., the control qubit is less significant than the target qubit, we would have to perform this backward traversal to properly alter the graph. This means that this single-traversal approach cannot be generally extended to multi-qubit gates. However, there is one exception and that is the controlled phase gates, such as CZ gate. This is because these gates are symmetric in the sense that it does not matter which qubit is the control qubit and which is the target qubit—the rotation is performed only in the case they are both set to $|1\rangle$.

To illustrate this, we shall take a closer look at the main idea of this specialized *Apply* for single-qubit gates. This recursive algorithm gradually traverses the whole tree and if it encounters a node representing the target qubit (the qubit the gate is applied to), it performs the gate operation $\mathcal{G}$ on this node and returns the result. For example, if we consider the gate Z, $\mathcal{G}_Z$ keeps the low successor the same and multiplies the high successor with -1. Thus, this function does not necessarily reach all leaf nodes of the MTBDD, since it stops the recursion when it is clear that no more changes will occur in the subgraph. However, we have to consider the case when the node representing the target qubit q_t is not present in the graph as it is reduced. This is not a problem for the X gate, because if the node is missing, it means that its low and high successors are the same, which in turn means that if we swap these two successors, the result will be identical to the current subgraph. For this reason, the X gate actually uses a standard *unary Apply*, which does not address this case. However, all other mentioned single-qubit gates perform some arithmetic operation on at least one of the successors, which means we have to generate this target qubit node manually. For even better performance, this algorithm uses caching of all computed results (it stores the root node of the current subgraph, operation $\mathcal{G}$, q_t, and, of course, the result of this operation). This custom operation is described in more detail in Algorithm 3.

The algorithm for controlled phase gate application (presented in Algorithm 4) is conceptually the same as the algorithm for single-qubit gates. The only difference is that it also checks for the control qubit nodes and lets the recursion continue only for the high successor of these nodes (the low subtrees always remain the same). It must also make sure that the control qubit node is always present, and if not, create it manually.

Algorithm 3: Execution of a single-qubit gate operation $\mathcal{G}[q_t]$

```
Input: root node of MTBDD $\mu$ representing $T(b_0, \ldots, b_{n-1})\colon \{0,1\}^n \to \mathbb{Z}^4$, where
       $\{b_0, \ldots, b_{n-1}\}$ is ordered w.r.t. $\prec$, a unary gate operator $\mathcal{G}$, and an index
       of the target qubit $q_t$
Output: root node of MTBDD $\mu_{\mathcal{G}}$ representing $\mathcal{G}[q_t](\mu)$
1  begin
2      $root(\mu_{\mathcal{G}}) \leftarrow$ ApplyGateFrom($root(\mu), \mathcal{G}, q_t$);
3      return Reduce($root(\mu_{\mathcal{G}})$);
4  end
5  Function ApplyGateFrom($x$: node, $\mathcal{G}$: unary gate operator, $q_t$: integer) : node is
6      if is in cache then
7          return cached result;
8      end
9      $result \leftarrow x$;
       // All gate operations applied to zero are again zero, so return
          the result immediately
10     if $x$ is not a zero leaf then
11         if $IsLeaf(x)$ or $q_t \prec var(x)$ then
               // Missing target node, create it manually
12             $result \leftarrow New$(node);
13             $var(result) \leftarrow q_t$;
14             $low(result) \leftarrow low(x)$;
15             $high(result) \leftarrow high(x)$;
16         end
17         if $var(result) = q_t$ then
18             $result \leftarrow \mathcal{G}(result)$;
19         else
               // Recursion
20             $result \leftarrow New$(node);
21             $var(result) \leftarrow var(x)$;
22             $low(result) \leftarrow$ ApplyGateFrom($low(x), \mathcal{G}, q_t$);
23             $high(result) \leftarrow$ ApplyGateFrom($high(x), \mathcal{G}, q_t$);
24         end
25         Put $result$ into cache;
26     end
27     return $result$;
28 end
```

Algorithm 4: Execution of a controlled phase gate operation $\mathcal{CG}_\varphi[q_c, q_t]$

```
Input: root node of MTBDD $\mu$ representing $T(b_0, \ldots, b_{n-1})\colon \{0,1\}^n \to \mathbb{Z}^4$, where
       $\{b_0, \ldots, b_{n-1}\}$ is ordered w.r.t. $\prec$, a unary phase gate operator $\mathcal{G}_\varphi$, an
       index of the control qubit $q_c$, and an index of the target qubit $q_t$
Output: root node of MTBDD $\mu_\mathcal{G}$ representing $\mathcal{CG}_\varphi[q_c, q_t](\mu)$
1  begin
2      if $q_c > q_t$ then
3          $Swap(q_c, q_t)$;          // This can be done only for controlled phase
4      end
5      $root(\mu_\mathcal{G}) \leftarrow$ ApplyCPhGateFrom($root(\mu), \mathcal{G}_\varphi, q_c, q_t$);
6      return Reduce($root(\mu_\mathcal{G})$);
7  end
8  Function ApplyCPhGateFrom($x$: node, $\mathcal{G}_\varphi$: unary gate op., $q_c, q_t$: integer) : node is
9      if is in cache then
10         return cached result;
11     end
12     $result \leftarrow x$;
       // All gate operations applied to zero are again zero, so return
          the result immediately
13     if $x$ is not a zero leaf then
14         if $IsLeaf(x)$ or $q_c \prec var(x)$ then
               // Missing control node, create it manually
15             $result \leftarrow New(\text{node})$;
16             $var(result) \leftarrow q_c$;
17             $low(result) \leftarrow low(x)$;
18             $high(result) \leftarrow high(x)$;
19         end
20         if $var(result) = q_c$ then
21             $high(result) \leftarrow \mathcal{G}_\varphi[q_t](high(result))$;
22         else
               // Recursion
23             $result \leftarrow New(\text{node})$;
24             $var(result) \leftarrow var(x)$;
25             $low(result) \leftarrow$ ApplyCPhGateFrom($low(x), \mathcal{G}_\varphi, q_c, q_t$);
26             $high(result) \leftarrow$ ApplyCPhGateFrom($high(x), \mathcal{G}_\varphi, q_c, q_t$);
27         end
28         Put $result$ into cache;
29     end
30     return $result$;
31 end
```

Chapter 5

Implementation

This chapter focuses on the implemented quantum circuit simulator called MEDUSA. To get an idea of the basic components of the implementation, let us first look at the the underlying key concepts of the simulator as a whole. We will then go on to examine the two supported simulation modes: standard execution and symbolic execution, in more detail. Standard execution is the mode in which regular simulation and the process of qubit measurement takes place, while symbolic execution consists of converting a classical representation into a symbolic representation, followed by symbolic simulation, and a final evaluation of all symbolic variables to convert the symbolic representation back into the classical representation.

5.1 Architecture of the Simulator

The implemented simulator MEDUSA (***M****ulti-Terminal Binary* ***D****E****cision*** ***D****iagram-based Q****U****antum* ***S****imul****A****tor*) is written in C. The reason for this is the fact that C is a low-level programming language well known for its efficiency and high performance. This is useful because performance is absolutely critical for this type of tool, since it must perform a large number of computationally non-trivial computations. Not only that, but this also gives access to an extensive set of C libraries.

MEDUSA is built on top of the Sylvan [12] library. Sylvan is a parallel BDD library including, among others, MTBDDs and their operations. Even though not all features of Sylvan were utilized in the implementation—e.g., parallelism support is not very suitable for the problems that MEDUSA solves (also confirmed by a few experimental results)—it provides a convenient framework for custom MTBDDs and custom MTBDD operations.

Due to the character of the algebraic representation of complex numbers introduced in Section 4.1 and the fact that we store the coefficient k globally for the whole MTBDD (meaning we cannot reduce the coefficients used for this representation in any way), the values of the integers needed for this representation increase exponentially. For this reason, there is a need for arbitrary integer precision, which is achieved using the general arbitrary precision arithmetic library GMP [17].

To specify the quantum circuit to be simulated, MEDUSA accepts an OpenQASM (Open Quantum Assembly Language) [10] file as input. OpenQASM is a standard programming language for the description of quantum circuits and algorithms. The supported set of gates is identical to the gates for which their MTBDD update formulae are defined in Table 4.1.

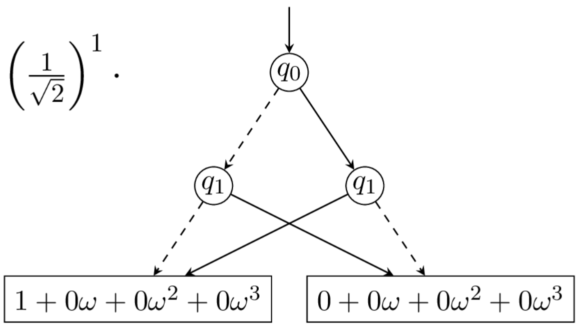

$\left(\frac{1}{\sqrt{2}}\right)^1 \cdot$
q_0
q_1
q_1
$1 + 0\omega + 0\omega^2 + 0\omega^3$
$0 + 0\omega + 0\omega^2 + 0\omega^3$

Figure 5.1: MTBDD representing the Bell state $|\psi\rangle = \frac{1}{\sqrt{2}} \cdot (|00\rangle + |11\rangle)$

The simulation itself proceeds in such a way that the parser module gradually processes the input OpenQASM file and applies individual gate operations to the state vector represented by the MTBDD accordingly. In the process, it can switch from the standard execution mode to the symbolic execution mode and back (more details on these modes are in the next two sections).

The result of the simulation are output in a DOT file, which contains a reduced MTBDD representing the end state of the simulated circuit. This MTBDD can then be plotted using a tool such as `Graphviz` [16]. Each leaf value is equal to a probability amplitude represented algebraically as described in Chapter 4. In the output leaf values, the complete coefficient 5-tuple (a, b, c, d, k) is shown in each individual leaf. This allows us to reduce these coefficients so that the representation of this complex number uses the smallest possible value of k, which in turn results in better readability. However, despite this coefficient reduction for each of the complex amplitudes, these coefficients can be quite large as a, b, c, d grow exponentially during the simulation. In this case, if some coefficient is exceptionally large, `MEDUSA` uses a symbolic variable instead of this large value in the DOT file and also outputs a text file with mapping of these symbolic variables to their actual values. It is possible to let `MEDUSA` directly calculate the probabilities of measuring the corresponding basis states from the probability amplitudes and output these probabilities as the MTBDD's leaf values in the DOT file instead. Another possible output of the simulation is, of course, the result of the qubit measurements, i.e., how many times each basis vector has been measured.

5.2 Standard Execution

The standard execution mode is the default simulation mode, where a regular simulation of the given circuit takes place. The main ideas of the used simulation technique, namely the representation of the simulated circuit's state vector with an MTBDD and the application of quantum gates on this MTBDD, were described in detail in Chapter 4. As mentioned in the previous section, `MEDUSA` uses the `Sylvan` library for the custom 4-tuple MTBDD and for custom operations on this MTBDD type. An example of this representation of the simulated system's state can be seen in Figure 5.1. Note that since `Sylvan` itself ensures the reduction property of the decision diagrams using hash tables and other internal checks, there is no need to explicitly call any implementation of the *Reduce* procedure anywhere during the simulation. The initial state of the simulation corresponds to all qubits in the circuit having the value $|0\rangle$.

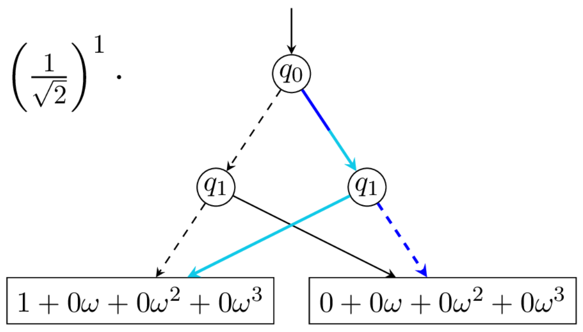
In addition to the already mentioned gates, MEDUSA also supports the operation of qubit measurement. More precisely, MEDUSA only supports the qubit measurement operation at the end of the circuit, however, this is not restrictive in any way as qubit measurements can always be moved from anywhere in the circuit to its end [25]. The main idea of the algorithm implementing this operation is as follows. First, we order the qubits to be measured according to the qubit ordering (since Sylvan does not support variable reordering in the decision diagrams, we have to start from the root and continue exactly according to the variable ordering of the MTBDD). We then sum up all the probabilities of the basis states, where the currently measured qubit is $|1\rangle$, while taking into account the results of the already measured qubits. Note that while calculating these intermediate probabilities, we switch from the algebraic representation of complex numbers to classical floating point representation in order to properly count for the reduced nodes skipped during the MTBDD traversal. However, the potential loss of accuracy is unavoidable at this stage as we need to represent the final probability of qubit being $|1\rangle$ with a floating point number anyway to generate the measurement result. Also the probability normalization factor is not necessarily algebraically representable in this way.

The operation of measuring the probability of a single qubit being $|1\rangle$ is implemented using a custom *unary Apply*. The probability normalization factor c (see Equation 2.2) caused by the previous measurements can be applied to this result at the very end of the probability calculation, because we can always pull a constant out of summation (see Equation 2.1). An example of the probability calculation needed for qubit measurement and the effects of qubit measurement on the subsequent probability calculations can be seen in Figure 5.2.

5.3 Symbolic Execution

MEDUSA also supports symbolic execution, which consists of simulating the circuit only symbolically. This allows us, for example, to compute the big-step semantics of loops in the quantum circuit. This in turn leads to a significant acceleration of the calculation for circuits that use loops with more than just a few iterations, because there is no need to reevaluate the individual gates in each iteration. We represent the modification of the system's MTBDD caused by a single loop iteration with symbolic update formulae. Then MEDUSA computes the end result by repeated (corresponding to the number of iterations) substitution of the symbolic variables in the symbolic update formulae with the actual values of probability amplitude coefficients. This is particularly useful because loops are often a key element of quantum algorithms, for example in algorithms that are based on amplitude amplification (Grover's search algorithm, quantum counting) or algorithms that use phase estimation (Shor's algorithm).

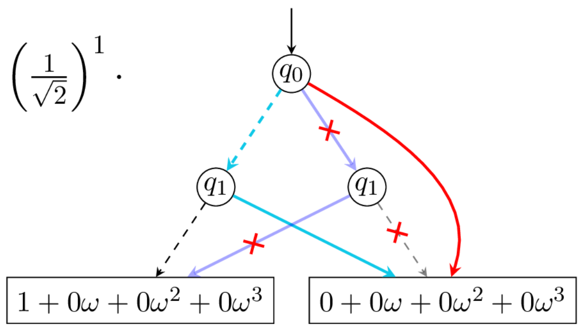
For simplicity, assume that our system's MTBDD represents some function $f\colon \{0,1\}^n \to \mathbb{C}$ (i.e., the leaves contain the complex number itself instead of the integer 4-tuple). Suppose MTBDD μ represents the quantum state just before the start of a loop in the quantum circuit. Let $\mathbb{S}$ be an infinite set of symbolic variable names and $\mathbb{T}_\mathbb{S}$ be the set of terms over $\mathbb{S}$. Symbolic execution uses *a pair of symbolic* MTBDDs: (i) an MTBDD μ_α with the variable mapping (this MTBDD represents the variable mapping partial function $\alpha\colon \mathbb{C} \rightharpoonup \mathbb{S}$, where the domain is the set of all leaf values of μ), and (ii) an MTBDD μ'_α with the symbolic values of these variables, which are expressions (terms) over $\mathbb{S}$ (i.e., μ'_α represents the partial function $\tau\colon \mathbb{S} \rightharpoonup \mathbb{T}_\mathbb{S}$, where the domain is the set of all leaf values of μ_α). Note that μ_α is specific to μ at the start of the loop. This means that the current im-



$\left(\frac{1}{\sqrt{2}}\right)^1 \cdot$
q_0
q_1
q_1
$1 + 0\omega + 0\omega^2 + 0\omega^3$
$0 + 0\omega + 0\omega^2 + 0\omega^3$

$$\mathcal{P}(q_0 = |1\rangle) = \sum_{i \in \{10, 11\}} |\alpha_i|^2 = |0|^2 + \left| \left(\frac{1}{\sqrt{2}}\right)^1 \cdot (1 + 0\omega + 0\omega^2 + 0\omega^3) \right|^2 = \frac{1}{2}$$

(a) Calculation of the probability $\mathcal{P}(q_0 = |1\rangle)$



$\left(\frac{1}{\sqrt{2}}\right)^1 \cdot$
q_0
q_1
q_1
$1 + 0\omega + 0\omega^2 + 0\omega^3$
$0 + 0\omega + 0\omega^2 + 0\omega^3$

$$\mathcal{P}(q_1 = |1\rangle) = \sum_{i \in \{01, 11\}} |c \cdot \alpha_i|^2 = c^2 \cdot \sum_{i \in \{01, 11\}} |\alpha_i|^2 = \left(\frac{1}{\sqrt{\frac{1}{2}}}\right)^2 \cdot \left(|0|^2 + |0|^2\right) = 2 \cdot 0 = 0$$

(b) Calculation of the probability $\mathcal{P}(q_1 = |1\rangle)$ after we measured q_0 and got the result $|0\rangle$ (c denotes the probability normalization coefficient)

Figure 5.2: Example of the measurement operation on one of the Bell states

plementation could not, for example, reuse the symbolic MTBDD pair used for simulating loop L on μ_1 for an input MTBDD μ_2 if $\mu_1 \neq \mu_2$ (unless μ_1 and μ_2 have the same structure and only differ in the leaf values). On the other hand, this allows us to take advantage of the compactness that MTBDDs offer.

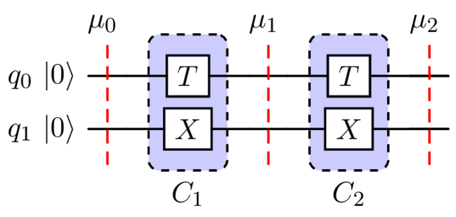
In order to keep the number of symbolic variables used as small as possible, in the *initial abstraction* we introduce variables only for the distinct leaf values of μ, because we expect basis states with the same valued probability amplitudes to transform equally during the loop execution. Of course, this may not always be true, which is why we check if there are any conflicting values for the same variable after simulating one iteration of the loop. If so, we introduce more variables and run the loop iteration again. If not, we evaluate the result and update μ accordingly. This evaluation is done by using τ to obtain the final values of the symbolic variables and then replacing the variables in μ_α with these values.

This *loop summarization* algorithm is described in more detail in Algorithm 5. Here, $rng(f)$ denotes the range of the function f, e.g., if $f\colon \mathbb{X} \to \mathbb{Y}$, then $rng(f) = \mathbb{Y}$. Also, $f(x) = \perp$ for $x \in \mathbb{X}$ denotes that there is no $y \in \mathbb{Y}$ such that $(x, y) \in f$. We denote the operation representing a single loop iteration as C (C_S represents the symbolic execution of this operation), and $f[p_0, \ldots, p_{k-1}]$ denotes the closure of function f with parameters $p_0, \ldots, p_{k-1}$ passed by reference to the variables specified in the **Data** declaration of f. Figure 5.3 shows an example of a run of this loop summarization algorithm. The trees for all MTBDDs are shown instead for easier demonstration. First, we perform the initial abstraction of the input MTBDD μ_0. However, after symbolically simulating the loop operation C with the initial abstraction, there are conflicting symbolic update values for the variable b. Therefore, we introduce an additional variable c. After performing the loop operation on this abstraction, no more changes are introduced with the *Refine* operation, and we can evaluate the result using the just obtained symbolic MTBDD pair and the input MTBDD μ_0. The symbolic simulation is run on μ'_α in the same way as the simulation is run on μ in the standard execution, i.e., using the gate operations defined in Section 4.2. Note that this version of loop summarization does not support nested loops and assumes that the loops do not contain qubit measurement operations, however, it could be extended to support both.

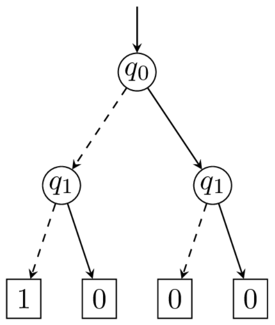
This algorithm can be trivially extended for the 4-tuple MTBDD that is used in the actual implementation. The only difference is, that in μ_α and μ'_α, there is also a 4-tuple of the corresponding values (4 symbolic variables for μ_α, 4 symbolic expressions over these variables). The symbolic expressions are implemented as singly linked lists, where each element consists of a signed integer coefficient and a symbolic variable. There is an implicit addition operation between the elements of this list. The only other operations with these symbolic expressions needed for the symbolic simulation are subtraction and multiplication by some constant, both of which are accomplished by manipulating the coefficients.

Algorithm 5: Loop summarization

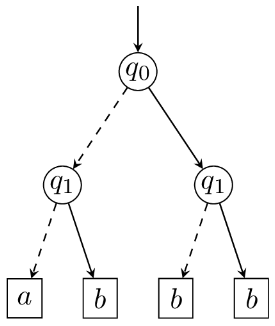
```
Input: An MTBDD $\mu$
Output: An MTBDD $\mu_\alpha$ over $\mathbb{S}$ and a mapping $\tau\colon \mathbb{S} \to \mathbb{T}_\mathbb{S}$
begin
    $\alpha \leftarrow \emptyset$ (type $\alpha\colon \mathbb{C} \rightharpoonup \mathbb{S}$);                    // Initial abstraction
    $\mu_\alpha^{refined} \leftarrow$ UnaryApply($\mu$, Abstract[$\alpha$]);
    repeat
        $\mu_\alpha \leftarrow \mu_\alpha^{refined}$;
        $\mu'_\alpha \leftarrow C_S(\mu_\alpha)$;
        $\tau \leftarrow \emptyset$ (type $\tau\colon \mathbb{S} \rightharpoonup \mathbb{T}_\mathbb{S}$);                    // Update
        $\sigma \leftarrow \emptyset$ (type $\sigma\colon \mathbb{S} \rightharpoonup \mathbb{S}$);                 // Refinement substitution
        $\mu_\alpha^{refined} \leftarrow$ Apply($\mu_\alpha, \mu'_\alpha$, Refine[$\tau, \sigma, \alpha$]);
    until $\mu_\alpha = \mu_\alpha^{refined}$;
    return $(\mu_\alpha, \tau)$;
end

Function Abstract($val\colon \mathbb{C}$) : $\mathbb{S}$ is
    Data: $\alpha\colon \mathbb{C} \rightharpoonup \mathbb{S}$
    if $\alpha(val) = \bot$ then
        let $s_{new} \in \mathbb{S} \setminus rng(\alpha)$ be a fresh symbolic variable;
        $\alpha \leftarrow \alpha \cup \{val \mapsto s_{new}\}$;
    end
    return $\alpha(val)$;
end

Function Refine($lhs\colon \mathbb{S}$, $rhs\colon \mathbb{T}_\mathbb{S}$) : $\mathbb{S}$ is
    Data: $\tau\colon \mathbb{S} \rightharpoonup \mathbb{T}_\mathbb{S}, \sigma\colon \mathbb{S} \rightharpoonup \mathbb{S}, \alpha\colon \mathbb{C} \rightharpoonup \mathbb{S}$
    if $\tau(lhs) = \bot$ then
        $\tau \leftarrow \tau \cup \{lhs \mapsto rhs\}$;
    end
    else if $\not\models \tau(lhs) = rhs$ then
        if $\sigma(lhs) = \bot$ then
            let $s_{new} \in \mathbb{S} \setminus rng(\alpha)$ be a fresh symbolic variable;
            $\sigma \leftarrow \sigma \cup \{lhs \mapsto s_{new}\}$;
        end
        return $\sigma(lhs)$;
    end
    return $lhs$;
end
```



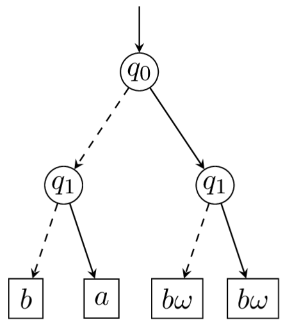
(a) The simulated circuit with two iterations of loop operation C



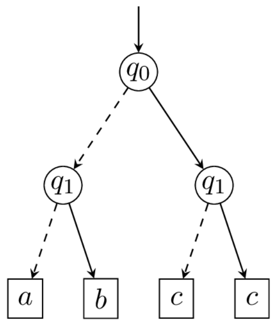
(b) Initial state μ_0



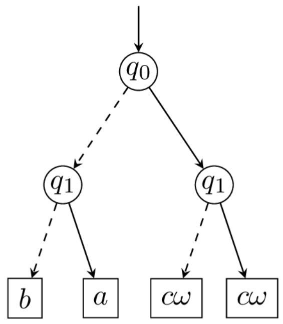
(c) Initial abstraction μ_α^0 of μ_0



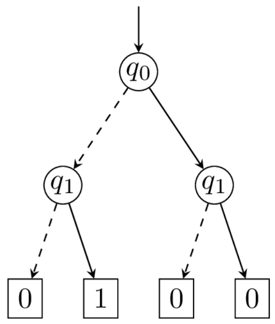
(d) Result $\mu_\alpha'^0$ we get after applying C_S to μ_α^0



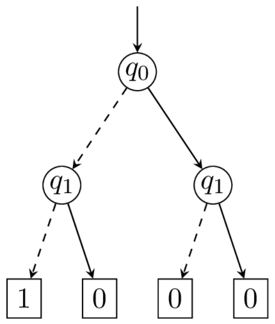
(e) Refined abstraction μ_α^1 of μ_0



(f) Result $\mu_\alpha'^1$ we get after applying C_S to μ_α^1



(g) Evaluated result μ_1 after the first iteration



(h) Evaluated result μ_2 after the second iteration

Figure 5.3: Example of symbolic loop execution (uses full trees instead of MTBDDs for clarity)

Chapter 6

Experimental Evaluation

This chapter presents the experimental results of the performance of the implemented simulator `MEDUSA` compared to other state-of-the-art tools. The experiments consisted of simulating two sets of benchmark circuits: LOOPS and STRAIGHTLINE. The aim of the benchmark set LOOPS was to evaluate the impact of using symbolic execution for loop simulation, and the aim of the benchmark set STRAIGHTLINE was to compare the proposed MTBDD-based approach itself with the state of the art. We show that the implemented MTBDD-based simulator not only keeps up with current state-of-the-art simulators in terms of performance, but also that the symbolic loop simulation leads to a noticeable acceleration of the simulation that far exceeds the capabilities of the state of the art.

6.1 Experimental Environment and Used Simulators

All the experiments were conducted on a machine with the following parameters:

OS	Debian GNU/Linux 12 (bookworm)
Number of CPUs	2
CPU model	Intel Xeon X5650 (2.67 GHz)
RAM	32 GiB

The timeout limit was set to 60 minutes. The performance of `MEDUSA` was compared with these state-of-the-art quantum circuit simulators: `SliQSim` [33], `DDSIM` [47] (v1.21.0), `Quasimodo` [30], `Quokka#` [24]. It is worth mentioning again that of the measured simulators, `MEDUSA` and `SliQSim` are the only *accurate* simulators (other simulators rely on floating point complex number representation). For more details on the importance of accurate simulation and all the above-mentioned tools in general, see Chapter 3. In the following, `Quas[`B`]` denotes `Quasimodo` with the backend option B, since `Quasimodo` supports multiple different decision diagram backend options. Note that we performed the experiments only on the `Quasimodo`'s BDD, WBDD (which uses the `DDSIM`'s decision diagram package), and CFLOBDD backend options, because the WCFLOBDD backend is currently in a rather experimental state (some gates have not been implemented yet for this backend option).

It is also worth mentioning that during the experiments with `Quasimodo`, some bugs in this tool were discovered, reported to the authors[1], and subsequently fixed (all results

[1]See `https://github.com/trishullab/Quasimodo/issues/8` and `https://github.com/trishullab/Quasimodo/issues/9`

shown for Quasimodo were measured after fixing these bugs). We refer to the implemented simulator with the symbolic loop simulation enabled as MEDUSA$_{loop}$ and without the symbolic loop simulation enabled as MEDUSA$_{base}$.

6.2 Benchmark Overview

The experiments consisted of measuring the simulation runtime on the above-mentioned tools for various quantum circuits specified in OpenQASM. Specifically, for all the decision diagram-based simulators (all simulators except Quokka#), the time to reach the final quantum state in the respective representation was measured. Since Quokka# does not explicitly compute the representation of the final quantum state, the time to obtain the probability that the first qubit is $|0\rangle$ was measured instead. As was already mentioned, the experiments were conducted on two sets of benchmark circuits, Loops and StraightLine (both sets do not contain any measurement operations in the circuits).

The benchmark set Loops contains quantum circuits with explicitly specified loops with a fixed number of iterations (if some tool did not support the loop syntax, the loops were unfolded for them). These quantum circuits implement either (i) the Grover's search algorithm (Grover), (ii) quantum counting (QC), or (iii) period finding (PF). The circuits implementing Grover's search have a single solution. Note that the circuits implementing (ii) and (iii) do not include the final part with the inverse Quantum Fourier Transform (QFT). This is because the implemented simulator does not currently support rotations by $\frac{\pi}{2^n}$ for an arbitrary $n \in \mathbb{N}$, which is a necessary operation for the inverse QFT. The reason for this is that the used algebraic representation of complex numbers (see Section 4.1) cannot represent rotations that are not multiples of $\frac{\pi}{4}$. However, this could be solved, for example, by using to a finer base rotation than $\frac{\pi}{4}$ to preserve the accuracy of the algebraic representation, or by switching to a floating point representation of complex numbers.

The circuits implementing quantum counting and period finding use the naming convention of form $\langle FR\rangle\_\langle SR\rangle\_\langle MT\rangle$, where FR and SR denote the number of qubits in the first and the second register, respectively, and MT denotes the number of randomly generated multi-control Toffoli gates in the oracle. For all these benchmark circuits it holds that $SR = \lfloor \frac{FR}{2} \rfloor$ and $MT \in \{5, 10, 15\}$.

The second benchmark set, StraightLine, consists only of circuits without loops, namely: circuits implementing (i) the Bernstein-Vazirani algorithm (BV, 99 circuits), (ii) multi-control Toffoli gates (MCToffoli, 97 circuits), (iii) the Grover's search algorithm (MOG, 9 circuits, specifically multi-oracle version with unfolded loops), and also (iv) randomly generated circuits (Random, 97 circuits), (v) benchmarks from the toolkit Feynman [1] (Feynman, 42 circuits), (vi) RevLib [42] reversible circuits (RevLib, 80 circuits), and (vii) modified versions of some of the RevLib benchmarks taken from [33] (RevLib-H, 16 circuits, the modification consists of adding a Hadamard gate at each unassigned input). Benchmark circuits (i)-(iv) are taken from the repository[2] of the AutoQ [8, 7] tool for quantum circuit verification.

6.3 Evaluation of Symbolic Execution Performance Impact

First, let us evaluate the results of the Loops benchmark set. The focus of this benchmark set was to examine the impact of symbolic loop execution, i.e., whether the symbolic execu-

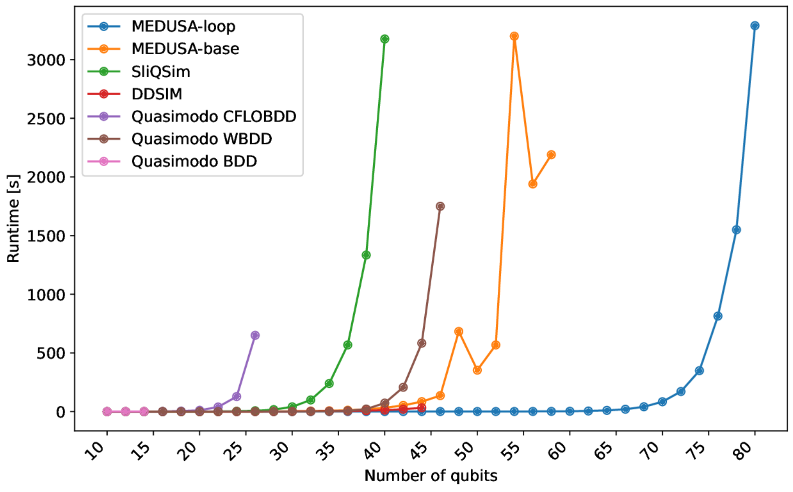
[2]Available at: https://github.com/alan23273850/AutoQ/

Table 6.1: Selected results for the LOOPS benchmark set The columns "#q" and "#G" denote the number of qubits and gates (counted with unfolded loops), respectively. Times are given in seconds ("0" denotes a time <0.5 s), memory in MiB. TO denotes a timeout, num denotes the fastest time, and num denotes the fastest *accurate* simulator (`MEDUSA` or `SliQSim`). `Quokka#` is omitted as it did not finish for any of the circuits in this benchmark set.

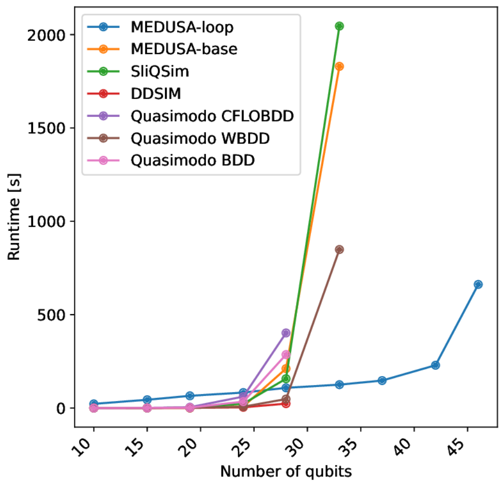
	circuit	#q	#G	$\texttt{MEDUSA}_{loop}$		$\texttt{MEDUSA}_{base}$		`SliQSim`		`DDSIM`		`Quas[CFLOBDD]`		`Quas[WBDD]`		`Quas[BDD]`	
				time	**mem**	**time**	**mem**	**time**	**mem**	**time**	**mem**	**time**	**mem**	**time**	**mem**	**time**	**mem**
GROVER	7	14	480	0	99	0	37	0	12	0	30	0	463	0	444	1	445
	10	20	2,136	0	116	0	40	0	12	0	34	11	570	0	449	TO	TO
	13	26	8,037	0	137	1	47	7	13	0	49	650	3,405	0	459	TO	TO
	16	32	27,956	0	162	4	97	99	14	2	55	TO	TO	1	492	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	84	1,088	TO	TO	32	254	TO	TO	583	1,083	TO	TO
	23	46	461,646	1	200	136	1,735	TO	TO	TO	TO	TO	TO	1,750	1,708	TO	TO
	26	52	1,473,184	1	210	568	4,505	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	29	58	4,676,916	2	215	2,190	10,032	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	33	66	21,328,090	20	220	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	37	74	95,794,310	349	229	TO	TO	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
PF	07_03_10	10	2,294	23	1,891	0	27	0	12	0	30	0	458	0	442	0	440
	13_06_10	19	245,744	66	2,090	2	28	4	32	1	213	10	458	1	446	4	442
	16_08_05	24	1,507,322	83	600	8	24	23	130	4	1,235	61	458	7	449	34	442
	19_09_15	28	39,321,545	109	2,154	247	32	587	3,002	178	31,144	1,580	459	198	452	2,160	455
	22_11_05	33	146,800,628	125	922	1,830	38	2,046	10,293	TO	TO	TO	TO	849	454	TO	TO
	22_11_15	33	448,790,444	128	1,662	3,020	27	TO	TO	TO	TO	TO	TO	2,650	454	TO	TO
	28_14_15	42	37,312,528,274	233	1,935	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	31_15_15	46	277,025,390,495	673	1,973	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
QC	07_03_15	11	6,108	24	2,092	0	42	1	12	0	33	0	459	0	443	1	446
	08_04_10	13	11,999	31	2,115	1	44	4	13	0	38	1	459	0	444	TO	TO
	09_04_10	14	24,032	38	2,127	1	54	15	14	0	46	2	459	0	445	TO	TO
	10_05_05	16	40,937	45	2,115	3	83	60	15	0	42	4	459	0	446	TO	TO
	11_05_05	17	81,898	52	2,116	5	109	TO	TO	0	65	TO	TO	0	447	TO	TO
	12_06_05	19	188,390	219	7,394	TO	TO	TO	TO	941	144	TO	TO	TO	TO	TO	TO
	12_06_15	19	376,760	250	7,691	TO	TO	TO	TO	1,280	294	TO	TO	TO	TO	TO	TO
	13_06_15	20	753,593	919	9,502	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO

tion leads to a noticeable acceleration of the simulation for practically applicable quantum algorithms. Some selected results can be seen in Table 6.1 (for all results, see Table A.1). For each simulator, this table includes the largest circuit in each benchmark subset that it was able to successfully simulate within the timeout limit. The runtimes of the simulators can be also seen in Figure 6.1 (for PF and QC subsets, only the results with a 5 multi-control Toffoli gates in the oracle are shown for clarity). `Quokka#` is not included in the presented results because it did not finish successfully for any of the circuits in the benchmark set (it timed out for the smallest circuit in the GROVER subset and it does not support the multi-control X gate, which is present in all PF and QC circuits).

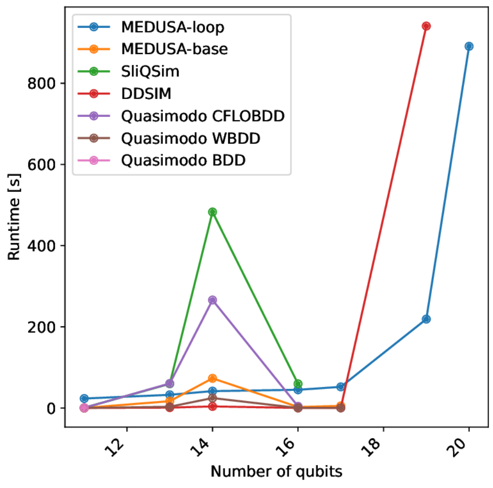
If we take a look at the performance of $\texttt{MEDUSA}_{loop}$ and $\texttt{MEDUSA}_{base}$ (the only difference between them is whether they use symbolic execution or not), we can clearly see that the symbolic execution allows the simulator to scale much better with increasing circuit complexity. Although this speed-up often comes at the cost of higher memory usage, this can probably be further optimized, since it seems that in these simulated circuits it was not



(a) Grover's search



(b) Period finding



(c) Quantum counting

Figure 6.1: Runtimes on the LOOPS benchmark set (the results for quantum counting and period finding show all the circuits with 5 multi-control Toffoli gates in the oracle)

the symbolic execution itself that was the most memory-intensive task, but the subsequent variable evaluation, where there is definitely some room for further optimization. It is important to mention that this leads to the assumption that the symbolic execution would provide a significant performance improvement if extended to other simulators based on decision diagrams.

Not only does the symbolic execution of loops lead to a noticeable acceleration of the simulation for $\texttt{MEDUSA}_{base}$, but $\texttt{MEDUSA}_{loop}$ scaled much better than all other simulators on all benchmark subsets. This is especially true for GROVER ($\texttt{MEDUSA}_{loop}$ was able to simulate circuits with up to 80 qubits, while the best performing simulator other than `MEDUSA` achieved at most a circuit with 46 qubits) and PF ($\texttt{MEDUSA}_{loop}$ was able to simulate a circuit with over 277 billion gates in less than 12 minutes). However, $\texttt{MEDUSA}_{loop}$ did not perform as well on some smaller but still complex circuits (e.g., for QC), because the symbolic representation for these circuits is not trivial, but at the same time the loops in these circuits have relatively few iterations, so symbolic execution tends to cause an extra overhead.

Note that the performance of `MEDUSA`$_{base}$ on the LOOPS benchmark circuits is also quite impressive, although this was not the main goal of these experiments. Especially for GROVER benchmark circuits, `MEDUSA`$_{base}$ scales exceptionally well and surpasses other state-of-the-art tools in the size of the circuits it managed to simulate.

6.4 Evaluation of MTBDD-based Simulator Performance

As mentioned earlier, the second benchmark set STRAIGHTLINE was used to compare `MEDUSA`$_{base}$ with other state-of-the-art simulators. Some selected results are shown in Table 6.2 (for all results, see Table A.2). `MEDUSA`$_{loop}$ is omitted from these results, as it would give exactly the same results as `MEDUSA`$_{base}$ because these circuits do not contain loops. In addition to this, the results for BV and MCTOFFOLI are not presented, since all the circuits in these two categories were trivial for all the simulators—all of them managed to simulate the largest BV circuit (100 qubits, 251 gates) in less than 0.1 seconds and the largest MCTOFFOLI circuit (198 qubits, 197 gates) in less than 0.2 seconds. Note that the errors we encountered using `Quas[CFLOBDD]` and `Quas[BDD]` were caused by not supporting certain gates present in the circuits (specifically, $R_x\left(\frac{\pi}{2}\right)$ and $R_y\left(\frac{\pi}{2}\right)$). `Quas[WBDD]` also does not support the $R_x\left(\frac{\pi}{2}\right)$ and $R_y\left(\frac{\pi}{2}\right)$ gates, but it does support the $\sqrt{X}$ and $\sqrt{Y}$ gates (which are equivalent to $R_x\left(\frac{\pi}{2}\right)$ and $R_y\left(\frac{\pi}{2}\right)$, respectively, up to a global phase factor that can be ignored [25]), so we used these gates instead. Similarly, the simulation errors with `Quokka#` were caused by the fact that `Quokka#` does not support the multi-control X gate, which is present in all REVLIB-H circuits.

If we take a closer look at the performance of `MEDUSA`$_{base}$, it is obvious that this simulator is competitive with the current state of the art, especially if we focus only on accurate simulators. Specifically, `MEDUSA`$_{base}$ is clearly the best available accurate simulator for all the non-trivial FEYNMAN circuits, on the other hand, it struggles with some RANDOM circuits. It is important to mention that REVLIB-H circuits were challenging for all simulators—the selected circuits include only those where none of the tools timed out. `SliQSim` was the best performing simulator on REVLIB-H as it was able to simulate 14 of the circuits, whereas `DDSIM` and `Quas[CFLOBDD]` managed to simulate 6 of the circuits and `MEDUSA`$_{base}$, `Quas[WBDD]`, and `Quas[BDD]` managed to simulate 5 of the circuits. `Quokka#` failed to simulate any of the circuits because, again, these circuits contain the multi-control X gate, which is currently not supported by `Quokka#`.

Combined with the fact that `MEDUSA`$_{base}$ also performed notably well on the LOOPS benchmark set, it is shown that the implemented MTBDD-based simulator is an interesting and useful alternative to the available quantum circuit simulation tools, as it is often complementary to the current state of the art (especially if we consider only accurate simulation tools).

Table 6.2: Selected results for the STRAIGHTLINE benchmark set. 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`). `Quokka#` is not marked 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	40	1.35	13	0.10	71	0.66	460	0.11	502	0.82	450	0.87	45
	gf2^{64}_mult	192	12,731	1.82	66	17.11	20	0.75	126	2.59	464	0.68	601	4.43	462	3.56	149
	gf2^{128}_mult	384	50,043	20.40	231	264.81	38	5.28	235	10.50	478	4.76	1,159	27.60	498	15.39	570
	gf2^{256}_mult	768	198,395	163.00	1,635	TO	TO	41.21	538	43.30	531	38.50	4,989	238.00	633	71.28	2,324
	hwb8	12	6,446	0.16	38	3.70	13	0.03	33	0.95	460	0.04	443	1.03	444	TO	TO
	hwb10	16	31,764	0.80	51	84.20	15	0.21	38	4.72	466	0.22	447	1.56	445	TO	TO
	hwb11	15	87,789	2.64	103	660.93	22	0.49	70	12.80	475	0.52	449	1.51	448	TO	TO
	hwb12	20	171,482	5.80	205	2,568.02	35	1.13	133	27.20	510	1.35	456	6.43	457	3,193.79	1,070
MOG	10	30	2,433	0.20	42	1.26	12	0.08	34	9.08	595	0.05	456	TO	TO	62.68	40
	11	33	3,746	0.36	45	3.12	13	0.13	42	48.80	906	0.08	462	TO	TO	167.01	56
RANDOM	85	85	255	1.00	52	0.47	15	2.12	64	ERR	ERR	0.11	485	ERR	ERR	0.03	12
	86	86	258	15.30	214	0.48	14	2.25	72	ERR	ERR	3.25	553	ERR	ERR	0.07	12
	89	89	267	9.48	105	0.67	14	0.72	66	ERR	ERR	0.59	492	ERR	ERR	0.06	13
	93	93	279	1.68	62	0.32	13	0.18	68	ERR	ERR	0.11	494	ERR	ERR	0.05	12
	94	94	282	79.60	337	0.78	18	4.45	76	ERR	ERR	74.30	521	ERR	ERR	0.08	13
	97	97	291	5.70	118	0.42	13	1.47	78	ERR	ERR	0.42	525	ERR	ERR	0.03	13
	99	99	297	9.58	173	0.38	12	2.61	79	ERR	ERR	0.67	526	ERR	ERR	0.08	13
REVLIB	apex5_290	1,025	2,909	1.75	62	0.37	44	1.03	536	0.26	467	1.33	1,214	3.95	516	2.11	73
	cps_292	923	2,763	1.19	58	0.21	31	1.25	485	0.22	465	1.09	1,035	2.82	528	1.39	60
	frg2_297	1,219	3,724	2.32	93	0.50	49	1.52	633	0.32	469	1.90	1,307	6.25	498	2.15	84
	seq_314	1,617	5,990	4.96	98	1.35	109	4.11	835	0.54	477	3.71	1,776	14.00	537	3.65	124
REVLIB-H	add64_184	193	385	0.20	204	0.03	14	0.10	118	0.10	460	0.08	545	0.06	446	ERR	ERR
	cpu_register_32_405	328	1,978	0.46	214	0.09	15	0.42	195	0.57	469	0.70	668	0.33	457	ERR	ERR
	e64-bdd_295	195	516	1.98	239	2.49	14	2.00	127	0.62	477	0.54	614	1.91	496	ERR	ERR
	ex5p_296	206	736	7.61	283	12.03	21	3.57	132	0.99	490	1.15	691	6.23	549	ERR	ERR
	hwb9_304	170	774	33.00	663	13.51	20	12.17	114	3.61	560	4.90	1,105	21.70	570	ERR	ERR

Chapter 7

Conclusion

In this thesis, we presented a novel approach to quantum circuit simulation on classical computers and introduced a tool implementing this approach. The implemented simulator, called `MEDUSA`, is accurate and based on MTBDDs. An essential feature of `MEDUSA` is symbolic execution of loops, which leads to a significant acceleration of the simulation as shown by the conducted experiments, where no other state-of-the-art simulator managed to scale nearly as well. For circuits that do not contain loops, `MEDUSA` has proven to not only keep up with other simulators in general, but for some circuits it even demonstrates superior performance—especially when considering only accurate simulators. Based on this thesis, a paper [6] (see Appendix B) was written and submitted to ICCAD'24 at the time of writing.

In terms of future work, it would be useful to perform certain quantum operations more efficiently (e.g., qubit measurement, or control gates other than control phase gates) and to better optimize the final evaluation during symbolic execution to be less memory demanding. However, these are only minor improvements. A more substantial improvement would be to extend the current simulation technique to support rotations by $\frac{\pi}{2^n}$ for an arbitrary $n \in \mathbb{N}$ while maintaining accuracy, or to implement some additional optimization preprocessing procedure that would take advantage of the fact that some operations can be done more efficiently in simulation than on a real quantum computer. Another possible direction would be to extend the functionality of the implemented tool for the purpose of quantum circuit verification.

Bibliography

[1] Amy, M. Towards Large-scale Functional Verification of Universal Quantum Circuits. In: Selinger, P. and Chiribella, G., ed. *Proceedings 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Canada, 3-7th June 2018*. 2018, vol. 287, p. 1–21. EPTCS. DOI: 10.4204/EPTCS.287.1. Available at: https://doi.org/10.4204/EPTCS.287.1.

[2] Arute, F., Arya, K., Babbush, R., Bacon, D., Bardin, J. C. et al. Quantum supremacy using a programmable superconducting processor. *Nature*. October 2019, vol. 574, no. 7779, p. 505–510. DOI: 10.1038/s41586-019-1666-5. ISSN 1476-4687. Available at: https://doi.org/10.1038/s41586-019-1666-5.

[3] Bernstein, E. and Vazirani, U. V. Quantum complexity theory. In: Kosaraju, S. R., Johnson, D. S. and Aggarwal, A., ed. *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing, May 16-18, 1993, San Diego, CA, USA*. ACM, 1993, p. 11–20. DOI: 10.1145/167088.167097. Available at: https://doi.org/10.1145/167088.167097.

[4] Brassard, G., Høyer, P. and Tapp, A. Quantum Counting. In: Larsen, K. G., Skyum, S. and Winskel, G., ed. *Automata, Languages and Programming, 25th International Colloquium, ICALP'98, Aalborg, Denmark, July 13-17, 1998, Proceedings*. Springer, 1998, vol. 1443, p. 820–831. Lecture Notes in Computer Science. DOI: 10.1007/BFB0055105. Available at: https://doi.org/10.1007/BFb0055105.

[5] Bryant, R. E. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. Computers*. 1986, vol. 35, no. 8, p. 677–691. DOI: 10.1109/TC.1986.1676819. Available at: https://doi.org/10.1109/TC.1986.1676819.

[6] Chen, T.-F., Chen, Y.-F., Jiang, J.-H. R., Jobranová, S. and Lengál, O. Accelerating Quantum Circuit Simulation with Symbolic Execution and Loop Summarization. *Submitted to ICCAD'24*. 2024.

[7] Chen, Y., Chung, K., Lengál, O., Lin, J. and Tsai, W. AutoQ: An Automata-Based Quantum Circuit Verifier. In: Enea, C. and Lal, A., ed. *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III*. Springer, 2023, vol. 13966, p. 139–153. Lecture Notes in Computer Science. DOI: 10.1007/978-3-031-37709-9_7. ISBN 978-3-031-37709-9. Available at: https://doi.org/10.1007/978-3-031-37709-9_7.

[8] Chen, Y., Chung, K., Lengál, O., Lin, J., Tsai, W. et al. An Automata-Based Framework for Verification and Bug Hunting in Quantum Circuits. *Proc. ACM*

Program. Lang. New York, NY, USA: Association for Computing Machinery. June 2023, vol. 7, PLDI, p. 1218–1243. DOI: 10.1145/3591270. Available at: https://doi.org/10.1145/3591270.

[9] COECKE, B. and DUNCAN, R. Interacting Quantum Observables. In: ACETO, L., DAMGÅRD, I., GOLDBERG, L. A., HALLDÓRSSON, M. M., INGÓLFSDÓTTIR, A. et al., ed. *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.* Berlin, Heidelberg: Springer, 2008, vol. 5126, p. 298–310. Lecture Notes in Computer Science. DOI: 10.1007/978-3-540-70583-3_25. ISBN 978-3-540-70583-3. Available at: https://doi.org/10.1007/978-3-540-70583-3_25.

[10] CROSS, A. W., BISHOP, L. S., SMOLIN, J. A. and GAMBETTA, J. M. *Open Quantum Assembly Language.* 2017. DOI: 10.48550/arXiv.1707.03429. Available at: https://doi.org/10.48550/arXiv.1707.03429.

[11] DIEKS, D. Communication by EPR devices. *Physics Letters A.* November 1982, vol. 92, no. 6, p. 271–272. DOI: https://doi.org/10.1016/0375-9601(82)90084-6. ISSN 0375-9601. Available at: https://www.sciencedirect.com/science/article/pii/0375960182900846.

[12] DIJK, T. van and POL, J. van de. Sylvan: Multi-Core Decision Diagrams. In: BAIER, C. and TINELLI, C., ed. *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings.* Springer, 2015, vol. 9035, p. 677–691. Lecture Notes in Computer Science. DOI: 10.1007/978-3-662-46681-0_60. Available at: https://doi.org/10.1007/978-3-662-46681-0_60.

[13] EINSTEIN, A., PODOLSKY, B. and ROSEN, N. Can Quantum-Mechanical Description of Physical Reality Be Considered Complete? *Phys. Rev.* American Physical Society. May 1935, vol. 47, p. 777–780. DOI: 10.1103/PhysRev.47.777. Available at: https://link.aps.org/doi/10.1103/PhysRev.47.777.

[14] FEYNMAN, R. P. Simulating physics with computers. *International Journal of Theoretical Physics.* June 1982, vol. 21, no. 6, p. 467–488. DOI: 10.1007/BF02650179. ISSN 1572-9575. Available at: https://doi.org/10.1007/BF02650179.

[15] FUJITA, M., MCGEER, P. C. and YANG, J. C. Multi-Terminal Binary Decision Diagrams: An Efficient Data Structure for Matrix Representation. *Formal Methods in System Design.* April 1997, vol. 10, 2/3, p. 149–169. DOI: 10.1023/A:1008647823331. ISSN 1572-8102. Available at: https://doi.org/10.1023/A:1008647823331.

[16] GANSNER, E. R. and NORTH, S. C. An open graph visualization system and its applications to software engineering. *Softw. Pract. Exp.* 2000, vol. 30, no. 11, p. 1203–1233. DOI: 10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N. Available at: https://doi.org/10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N.

[17] Granlund, T. and the GMP development team. *GNU MP: The GNU Multiple Precision Arithmetic Library* [online]. 6.2.1th ed. 2020 [cit. 2024-04-14]. Available at: http://gmplib.org/.

[18] Grover, L. K. A Fast Quantum Mechanical Algorithm for Database Search. In: Miller, G. L., ed. *Proceedings of the Twenty-Eighth Annual ACM Symposium on the Theory of Computing, Philadelphia, Pennsylvania, USA, May 22-24, 1996*. ACM, 1996, p. 212–219. DOI: 10.1145/237814.237866. Available at: https://doi.org/10.1145/237814.237866.

[19] Herman, D., Googin, C., Liu, X., Sun, Y., Galda, A. et al. Quantum computing for finance. *Nature Reviews Physics*. Springer Science and Business Media LLC. July 2023, vol. 5, no. 8, p. 450–465. DOI: 10.1038/s42254-023-00603-1. ISSN 2522-5820. Available at: http://dx.doi.org/10.1038/s42254-023-00603-1.

[20] Hong, X., Zhou, X., Li, S., Feng, Y. and Ying, M. A Tensor Network based Decision Diagram for Representation of Quantum Circuits. *ACM Trans. Design Autom. Electr. Syst.* New York, NY, USA: Association for Computing Machinery. June 2022, vol. 27, no. 6, p. 60:1–60:30. DOI: 10.1145/3514355. ISSN 1084-4309. Available at: https://doi.org/10.1145/3514355.

[21] Kissinger, A. and Wetering, J. van de. Simulating quantum circuits with ZX-calculus reduced stabiliser decompositions. *Quantum Science and Technology*. IOP Publishing. July 2022, vol. 7, no. 4, p. 044001. DOI: 10.1088/2058-9565/ac5d20. ISSN 2058-9565. Available at: http://dx.doi.org/10.1088/2058-9565/ac5d20.

[22] Kitaev, A. Y. Quantum measurements and the Abelian Stabilizer Problem. *Electron. Colloquium Comput. Complex.* 1996, TR96-003. Available at: https://eccc.weizmann.ac.il/eccc-reports/1996/TR96-003/index.html.

[23] McArdle, S., Endo, S., Aspuru Guzik, A., Benjamin, S. C. and Yuan, X. Quantum computational chemistry. *Reviews of Modern Physics*. American Physical Society (APS). March 2020, vol. 92, no. 1. DOI: 10.1103/revmodphys.92.015003. ISSN 1539-0756. Available at: http://dx.doi.org/10.1103/RevModPhys.92.015003.

[24] Mei, J., Bonsangue, M. and Laarman, A. Simulating Quantum Circuits by Model Counting. In: *CAV'24 (to appear)*. 2024. DOI: 10.48550/ARXIV.2403.07197. Available at: https://doi.org/10.48550/arXiv.2403.07197.

[25] Nielsen, M. A. and Chuang, I. L. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010. ISBN 978-1-107-00217-3.

[26] Niemann, P., Wille, R., Miller, D. M., Thornton, M. A. and Drechsler, R. QMDDs: Efficient Quantum Function Representation and Manipulation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 2016, vol. 35, no. 1, p. 86–99. DOI: 10.1109/TCAD.2015.2459034. Available at: https://doi.org/10.1109/TCAD.2015.2459034.

[27] Niemann, P., Zulehner, A., Drechsler, R. and Wille, R. Overcoming the Tradeoff Between Accuracy and Compactness in Decision Diagrams for Quantum Computation. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 2020, vol. 39,

no. 12, p. 4657–4668. DOI: 10.1109/TCAD.2020.2977603. Available at: https://doi.org/10.1109/TCAD.2020.2977603.

[28] Raedt, H. D., Jin, F., Willsch, D., Nocon, M., Yoshioka, N. et al. Massively parallel quantum computer simulator, eleven years later. *Computer Physics Communications*. 2019, vol. 237, p. 47–61. DOI: 10.1016/J.CPC.2018.11.005. ISSN 0010-4655. Available at: https://doi.org/10.1016/j.cpc.2018.11.005.

[29] Sistla, M., Chaudhuri, S. and Reps, T. W. CFLOBDDs: Context-Free-Language Ordered Binary Decision Diagrams. *CoRR*. 2022, abs/2211.06818. DOI: 10.48550/ARXIV.2211.06818. Available at: https://doi.org/10.48550/arXiv.2211.06818.

[30] Sistla, M., Chaudhuri, S. and Reps, T. W. Symbolic Quantum Simulation with Quasimodo. In: Enea, C. and Lal, A., ed. *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part III*. Springer, 2023, vol. 13966, p. 213–225. Lecture Notes in Computer Science. DOI: 10.1007/978-3-031-37709-9_11. Available at: https://doi.org/10.1007/978-3-031-37709-9_11.

[31] Sistla, M., Chaudhuri, S. and Reps, T. W. Weighted Context-Free-Language Ordered Binary Decision Diagrams. *CoRR*. 2023, abs/2305.13610. DOI: 10.48550/ARXIV.2305.13610. Available at: https://doi.org/10.48550/arXiv.2305.13610.

[32] Thanos, D., Villoria, A., Brand, S., Mei, A.-J. Q. J., Coopmans, T. et al. A Knowledge Compilation Map for Quantum Information. In: *SPIN'24 (to appear)*. 2024. Available at: https://spin-web.github.io/SPIN2024/assets/preproceedings/SPIN2024-paper6.pdf.

[33] Tsai, Y., Jiang, J. R. and Jhang, C. 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, 2021, p. 439–444. DOI: 10.1109/DAC18074.2021.9586191. Available at: https://doi.org/10.1109/DAC18074.2021.9586191.

[34] Vidal, G. Efficient Classical Simulation of Slightly Entangled Quantum Computations. *Physical Review Letters*. American Physical Society (APS). October 2003, vol. 91, no. 14. DOI: 10.1103/physrevlett.91.147902. ISSN 1079-7114. Available at: http://dx.doi.org/10.1103/PhysRevLett.91.147902.

[35] Vinkhuijzen, L., Coopmans, T., Elkouss, D., Dunjko, V. and Laarman, A. LIMDD: A Decision Diagram for Simulation of Quantum Computing Including Stabilizer States. *Quantum*. September 2023, vol. 7, p. 1108. DOI: 10.22331/Q-2023-09-11-1108. ISSN 2521-327X. Available at: https://doi.org/10.22331/q-2023-09-11-1108.

[36] Vinkhuijzen, L., Coopmans, T. and Laarman, A. A Knowledge Compilation Map for Quantum Information. *CoRR*. 2024, abs/2401.01322. DOI: 10.48550/ARXIV.2401.01322. Available at: https://doi.org/10.48550/arXiv.2401.01322.

[37] Vinkhuijzen, L., Grurl, T., Hillmich, S., Brand, S., Wille, R. et al. Efficient Implementation of LIMDDs for Quantum Circuit Simulation. In: Caltais, G. and Schilling, C., ed. *Model Checking Software - 29th International Symposium, SPIN 2023, Paris, France, April 26-27, 2023, Proceedings*. Springer, 2023, vol. 13872, p. 3–21. Lecture Notes in Computer Science. DOI: 10.1007/978-3-031-32157-3_1. Available at: https://doi.org/10.1007/978-3-031-32157-3_1.

[38] Vojnar, T. *Static Analysis and Verification (SAV), lecture Binary Decision Diagrams: Slides for the course at Faculty of Information Technology, Brno University of Technology* [online]. 2023 [cit. 2024-01-20]. Available at: https://www.fit.vutbr.cz/study/courses/SAV/public/.

[39] Wei, C., Tsai, Y., Jhang, C. and Jiang, J. R. Accurate BDD-based unitary operator manipulation for scalable and robust quantum circuit verification. In: Oshana, R., ed. *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*. New York, NY, USA: ACM, 2022, p. 523–528. DOI: 10.1145/3489517.3530481. Available at: https://doi.org/10.1145/3489517.3530481.

[40] Wetering, J. van de. *ZX-calculus for the working quantum computer scientist*. December 2020. ArXiv preprint arXiv:2012.13966. Available at: http://dx.doi.org/10.1088/2058-9565/ac5d20.

[41] Wille, R., Burgholzer, L., Hillmich, S., Grurl, T., Ploier, A. et al. The basis of design tools for quantum computing: arrays, decision diagrams, tensor networks, and ZX-calculus. In: Oshana, R., ed. *DAC '22: 59th ACM/IEEE Design Automation Conference, San Francisco, California, USA, July 10 - 14, 2022*. ACM, July 2022, p. 1367–1370. DOI: 10.1145/3489517.3530627. Available at: https://doi.org/10.1145/3489517.3530627.

[42] Wille, R., Grosse, D., Teuber, L., Dueck, G. W. and Drechsler, R. 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, 2008, p. 220–225. DOI: 10.1109/ISMVL.2008.43. Available at: https://doi.org/10.1109/ISMVL.2008.43.

[43] Wille, R., Meter, R. V. and Naveh, Y. IBM's Qiskit Tool Chain: Working with and Developing for Real Quantum Computers. In: Teich, J. and Fummi, F., ed. *Design, Automation & Test in Europe Conference & Exhibition, DATE 2019, Florence, Italy, March 25-29, 2019*. IEEE, 2019, p. 1234–1240. DOI: 10.23919/DATE.2019.8715261. Available at: https://doi.org/10.23919/DATE.2019.8715261.

[44] Wootters, W. K. and Zurek, W. H. A single quantum cannot be cloned. *Nature*. October 1982, vol. 299, no. 5886, p. 802–803. DOI: 10.1038/299802a0. ISSN 1476-4687. Available at: https://doi.org/10.1038/299802a0.

[45] Zhong, H.-S., Wang, H., Deng, Y.-H., Chen, M.-C., Peng, L.-C. et al. Quantum computational advantage using photons. *Science*. 2020, vol. 370, no. 6523, p. 1460–1463. DOI: 10.1126/science.abe8770. Available at: https://www.science.org/doi/abs/10.1126/science.abe8770.

[46] Zulehner, A., Hillmich, S. and Wille, R. How to Efficiently Handle Complex Values? Implementing Decision Diagrams for Quantum Computing. In: Pan, D. Z., ed. *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2019, Westminster, CO, USA, November 4-7, 2019.* ACM, 2019, p. 1–7. DOI: 10.1109/ICCAD45719.2019.8942057. Available at: https://doi.org/10.1109/ICCAD45719.2019.8942057.

[47] Zulehner, A. and Wille, R. Advanced Simulation of Quantum Computations. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 2019, vol. 38, no. 5, p. 848–859. DOI: 10.1109/TCAD.2018.2834427. Available at: https://doi.org/10.1109/TCAD.2018.2834427.

Appendix A

All Experimental Results

Table A.1: Results for the LOOPS benchmark set. The columns "#q" and "#G" denote the number of qubits and gates (counted with unfolded loops), respectively. Times are given in seconds ("0" denotes a time <0.5 s), memory in MiB. TO denotes a timeout, num denotes the fastest time, and num denotes the fastest *accurate* simulator (`MEDUSA` or `SliQSim`). `Quokka#` is omitted as it did not finish for any of the circuits in this benchmark set.

				`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	5	10	170	0	89	0	30	0	12	0	29	0	458	0	442	0	441
	6	12	301	0	94	0	35	0	12	0	30	0	459	0	443	0	442
	7	14	480	0	99	0	37	0	12	0	30	0	463	0	444	1	445
	8	16	813	0	104	0	38	0	12	0	30	1	478	0	445	TO	TO
	9	18	1,319	0	109	0	38	0	12	0	31	4	511	0	447	TO	TO
	10	20	2,136	0	116	0	40	0	12	0	34	11	570	0	449	TO	TO
	11	22	3,337	0	122	0	42	1	12	0	34	39	774	0	450	TO	TO
	12	24	5,163	0	134	0	44	3	12	0	39	128	1,228	0	453	TO	TO
	13	26	8,037	0	137	1	47	7	13	0	49	650	3,405	0	459	TO	TO
	14	28	12,115	0	145	1	56	17	13	1	50	TO	TO	0	470	TO	TO
	15	30	18,618	0	153	2	69	40	13	1	52	TO	TO	0	472	TO	TO
	16	32	27,956	0	162	4	97	99	14	2	55	TO	TO	1	492	TO	TO
	17	34	42,334	0	167	7	116	239	15	3	64	TO	TO	2	532	TO	TO
	18	36	63,133	0	174	12	185	568	17	5	75	TO	TO	7	539	TO	TO
	19	38	94,876	0	180	20	256	1,334	19	8	89	TO	TO	22	612	TO	TO
	20	40	140,721	0	187	32	387	3,176	25	12	118	TO	TO	73	769	TO	TO
	21	42	210,367	0	191	53	779	TO	TO	20	173	TO	TO	207	794	TO	TO
	22	44	310,367	0	196	84	1,088	TO	TO	32	254	TO	TO	583	1,083	TO	TO
	23	46	461,646	1	200	136	1,735	TO	TO	TO	TO	TO	TO	1,750	1,708	TO	TO
	24	48	678,601	1	205	683	3,669	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	25	50	1,005,355	1	206	353	3,553	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	26	52	1,473,184	1	210	568	4,505	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	27	54	2,174,689	1	212	3,200	11,103	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	28	56	3,178,178	1	213	1,940	10,042	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	29	58	4,676,916	2	215	2,190	10,032	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	30	60	6,819,806	3	216	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	31	62	10,008,932	5	219	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO

Continued on next page

Table A.1 (continued from previous page)

	circuit	#q	#G	`MEDUSA`$_{loop}$		`MEDUSA`$_{base}$		`SliQSim`		`DDSIM`		`Quas[CFLOBDD]`		`Quas[WBDD]`		`Quas[BDD]`	
				time	**mem**	**time**	**mem**	**time**	**mem**	**time**	**mem**	**time**	**mem**	**time**	**mem**	**time**	**mem**
Grover	32	64	14,566,326	10	220	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	33	66	21,328,090	20	220	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	34	68	30,985,878	41	222	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	35	70	45,276,660	84	224	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	36	72	65,677,990	171	226	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	37	74	95,794,310	349	229	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	38	76	138,767,877	814	234	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	39	78	202,070,979	1,550	242	TO	TO	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
	41	82	425,106,417	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
PF	07_03_05	10	897	22	204	0	22	0	12	0	29	0	457	0	442	0	440
	07_03_10	10	2,294	23	1,891	0	27	0	12	0	30	0	458	0	442	0	440
	07_03_15	10	3,437	23	2,017	0	35	0	12	0	32	0	458	0	442	0	440
	10_05_05	15	17,402	44	1,922	0	31	0	13	0	34	1	458	0	445	0	441
	10_05_10	15	34,793	44	2,084	0	39	1	15	0	40	1	464	0	449	1	446
	10_05_15	15	52,184	45	2,104	0	38	1	16	0	47	2	462	0	447	1	443
	13_06_05	19	106,497	65	1,992	1	26	2	21	0	97	4	458	0	446	2	441
	13_06_10	19	245,744	66	2,090	2	28	4	32	1	213	10	458	1	446	4	442
	13_06_15	19	384,991	66	2,112	21	102	82	51	11	329	21	588	33	590	137	481
	16_08_05	24	1,507,322	83	600	8	24	23	130	4	1,235	61	458	7	449	34	442
	16_08_10	24	3,407,837	88	2,116	16	40	67	260	14	2,712	137	461	15	458	186	451
	16_08_15	24	5,439,422	87	2,111	34	64	470	412	48	4,336	222	472	281	593	459	479
	19_09_05	28	9,961,473	108	2,092	211	28	157	799	23	8,289	402	458	48	451	286	443
	19_09_10	28	26,214,370	109	2,133	172	32	392	1,976	61	21,286	1,050	459	130	452	1,630	450
	19_09_15	28	39,321,545	109	2,154	247	32	587	3,002	178	31,144	1,580	459	198	452	2,160	455
	22_11_05	33	146,800,628	125	922	1,830	38	2,046	10,293	TO	TO	TO	TO	849	454	TO	TO
	22_11_10	33	310,378,445	127	1,387	1,480	39	TO	TO	TO	TO	TO	TO	1,810	454	TO	TO
	22_11_15	33	448,790,444	128	1,662	3,020	27	TO	TO	TO	TO	TO	TO	2,650	454	TO	TO
	25_12_05	37	1,107,296,249	147	2,081	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	25_12_10	37	2,348,810,196	147	2,140	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	25_12_15	37	3,388,997,557	166	2,211	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	28_14_05	42	9,395,240,954	229	1,152	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	28_14_10	42	22,548,578,249	238	1,680	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	28_14_15	42	37,312,528,274	233	1,935	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	31_15_05	46	79,456,894,971	662	1,351	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	31_15_10	46	171,798,691,792	693	1,810	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	31_15_15	46	277,025,390,495	673	1,973	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	34_17_05	51	876,173,328,368	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
QC	07_03_05	11	3,822	23	2,033	0	40	1	12	0	31	0	459	0	443	1	446
	07_03_10	11	4,965	24	2,058	0	40	1	12	0	32	0	458	0	443	0	446
	07_03_15	11	6,108	24	2,092	0	42	1	12	0	33	0	459	0	443	1	446
	08_04_05	13	8,684	32	2,992	17	506	60	21	1	41	60	952	3	587	TO	TO
	08_04_10	13	11,999	31	2,115	1	44	4	13	0	38	1	459	0	444	TO	TO
	08_04_15	13	15,314	31	2,128	1	50	5	13	0	33	122	1,542	0	444	TO	TO
	09_04_05	14	17,389	42	3,581	73	2,325	483	80	4	52	266	2,085	24	1,020	TO	TO
	09_04_10	14	24,032	38	2,127	1	54	15	14	0	46	2	459	0	445	TO	TO
	09_04_15	14	30,675	38	2,126	2	62	20	14	0	37	970	5,546	0	446	TO	TO
	10_05_05	16	40,937	45	2,115	3	83	60	15	0	42	4	459	0	446	TO	TO
	10_05_10	16	56,282	61	6,006	872	3,487	TO	TO	24	61	TO	TO	694	5,054	TO	TO
	10_05_15	16	77,765	45	2,164	6	110	TO	TO	1	65	TO	TO	0	447	TO	TO
	11_05_05	17	81,898	52	2,116	5	109	TO	TO	0	65	TO	TO	0	447	TO	TO
	11_05_10	17	112,603	96	6,661	TO	TO	TO	TO	98	90	TO	TO	TO	TO	TO	TO
	11_05_15	17	155,590	52	2,163	TO	TO	TO	TO	1	125	TO	TO	TO	TO	TO	TO

Continued on next page

Table A.1 (continued from previous page)

	circuit	#q	#G	MEDUSA$_{loop}$ time	MEDUSA$_{loop}$ mem	MEDUSA$_{base}$ time	MEDUSA$_{base}$ mem	SliQSim time	SliQSim mem	DDSIM time	DDSIM mem	Quas [CFLOBDD] time	Quas [CFLOBDD] mem	Quas [WBDD] time	Quas [WBDD] mem	Quas [BDD] time	Quas [BDD] mem
QC	12_06_05	19	188,390	219	7,394	TO	TO	TO	TO	941	144	TO	TO	TO	TO	TO	TO
	12_06_10	19	258,005	238	7,536	TO	TO	TO	TO	1,227	200	TO	TO	TO	TO	TO	TO
	12_06_15	19	376,760	250	7,691	TO	TO	TO	TO	1,280	294	TO	TO	TO	TO	TO	TO
	13_06_05	20	376,807	891	8,891	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	13_06_10	20	516,054	934	9,166	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	13_06_15	20	753,593	919	9,502	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
	14_07_05	22	884,705	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO

Table A.2: Results for the StraightLine benchmark set. 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). Quokka# is not marked as the fastest because it does not compute the quantum state representation.

	circuit	#q	#G	MEDUSA$_{base}$ time	MEDUSA$_{base}$ mem	SliQSim time	SliQSim mem	DDSIM time	DDSIM mem	Quas [CFLOBDD] time	Quas [CFLOBDD] mem	Quas [WBDD] time	Quas [WBDD] mem	Quas [BDD] time	Quas [BDD] mem	Quokka# time	Quokka# mem
BV	1	2	6	0.02	21	0.01	12	0.00	29	0.02	456	0.00	438	0.00	439	0.07	11
	2	3	8	0.00	21	0.01	12	0.00	29	0.00	457	0.00	438	0.00	439	0.00	11
	3	4	11	0.00	21	0.01	12	0.00	29	0.00	456	0.00	439	0.00	439	0.00	11
	4	5	13	0.00	21	0.01	12	0.00	29	0.00	458	0.00	439	0.00	439	0.00	11
	5	6	16	0.00	21	0.01	12	0.00	29	0.00	456	0.00	440	0.00	439	0.00	11
	6	7	18	0.00	21	0.01	12	0.00	29	0.01	456	0.01	440	0.00	439	0.00	11
	7	8	21	0.00	22	0.01	12	0.00	29	0.01	456	0.01	441	0.00	439	0.00	11
	8	9	23	0.00	22	0.01	12	0.00	29	0.01	456	0.01	441	0.00	439	0.00	11
	9	10	26	0.00	23	0.01	12	0.00	29	0.01	456	0.01	442	0.00	439	0.00	11
	10	11	28	0.01	23	0.01	12	0.00	29	0.01	456	0.01	442	0.00	439	0.00	11
	11	12	31	0.01	23	0.01	12	0.00	29	0.01	458	0.01	443	0.00	439	0.00	11
	12	13	33	0.01	23	0.01	12	0.00	29	0.01	458	0.01	443	0.00	440	0.00	11
	13	14	36	0.01	23	0.01	12	0.00	29	0.01	458	0.01	444	0.00	440	0.00	11
	14	15	38	0.01	24	0.01	12	0.00	29	0.01	457	0.01	444	0.00	440	0.00	11
	15	16	41	0.01	25	0.01	12	0.00	29	0.01	458	0.01	445	0.00	440	0.00	11
	16	17	43	0.01	25	0.01	12	0.00	29	0.01	458	0.01	445	0.00	440	0.00	11
	17	18	46	0.01	25	0.01	12	0.00	29	0.01	458	0.01	446	0.00	440	0.01	11
	18	19	48	0.01	25	0.01	12	0.00	29	0.01	458	0.01	446	0.00	440	0.01	11
	19	20	51	0.01	26	0.01	12	0.00	29	0.01	457	0.01	447	0.00	440	0.01	11
	20	21	53	0.01	26	0.01	12	0.00	29	0.02	458	0.01	447	0.00	440	0.01	11
	21	22	56	0.01	26	0.01	12	0.00	29	0.02	458	0.01	448	0.00	440	0.01	11
	22	23	58	0.01	27	0.01	12	0.00	29	0.02	458	0.01	448	0.00	440	0.01	11
	23	24	61	0.01	27	0.01	12	0.00	29	0.02	458	0.01	449	0.00	440	0.01	11
	24	25	63	0.01	27	0.01	12	0.00	29	0.02	457	0.01	449	0.00	440	0.01	11
	25	26	66	0.01	28	0.01	12	0.00	29	0.02	458	0.01	450	0.01	440	0.01	11
	26	27	68	0.01	28	0.01	12	0.00	29	0.02	458	0.01	450	0.01	440	0.01	11
	27	28	71	0.01	29	0.01	12	0.00	29	0.02	458	0.01	451	0.01	440	0.01	11
	28	29	73	0.07	29	0.01	12	0.00	29	0.02	457	0.01	451	0.01	440	0.01	11

Continued on next page

Table A.2 (continued from previous page)

				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
	29	30	76	0.02	29	0.01	12	0.00	29	0.02	458	0.01	452	0.01	440	0.01	11
	30	31	78	0.01	29	0.01	12	0.00	29	0.02	458	0.01	453	0.01	440	0.01	11
	31	32	81	0.01	30	0.01	12	0.00	29	0.02	458	0.01	453	0.01	440	0.01	11
	32	33	83	0.01	30	0.01	12	0.00	41	0.02	458	0.01	453	0.01	440	0.01	11
	33	34	86	0.01	31	0.01	12	0.00	39	0.03	458	0.01	454	0.01	440	0.01	11
	34	35	88	0.01	31	0.01	12	0.00	42	0.03	458	0.01	454	0.01	440	0.01	11
	35	36	91	0.01	31	0.01	12	0.00	41	0.03	458	0.01	455	0.01	440	0.01	11
	36	37	93	0.01	32	0.01	12	0.00	41	0.03	458	0.01	455	0.01	440	0.01	11
	37	38	96	0.01	32	0.01	12	0.00	42	0.03	458	0.01	456	0.01	441	0.01	11
	38	39	98	0.01	32	0.01	12	0.01	41	0.03	458	0.01	457	0.01	441	0.01	11
	39	40	101	0.02	33	0.01	12	0.01	44	0.03	458	0.01	457	0.01	440	0.01	11
	40	41	103	0.02	33	0.01	12	0.01	45	0.03	458	0.01	458	0.01	440	0.01	11
	41	42	106	0.02	35	0.01	12	0.01	42	0.03	458	0.01	458	0.01	441	0.01	11
	42	43	108	0.02	34	0.01	12	0.01	46	0.03	458	0.01	459	0.01	441	0.01	11
	43	44	111	0.02	34	0.01	12	0.01	45	0.03	458	0.01	459	0.01	441	0.01	11
	44	45	113	0.02	34	0.01	12	0.01	45	0.03	458	0.01	460	0.01	441	0.01	11
	45	46	116	0.02	34	0.01	12	0.01	48	0.03	458	0.01	460	0.01	441	0.01	11
	46	47	118	0.02	34	0.01	12	0.01	47	0.03	458	0.01	461	0.01	441	0.01	11
	47	48	121	0.02	35	0.02	12	0.01	45	0.03	458	0.02	461	0.01	441	0.01	11
	48	49	123	0.02	35	0.02	12	0.01	48	0.03	458	0.01	462	0.01	441	0.01	11
	49	50	126	0.02	35	0.02	12	0.01	49	0.04	458	0.02	463	0.02	441	0.01	11
	50	51	128	0.02	35	0.02	12	0.01	48	0.04	458	0.02	463	0.02	441	0.01	11
	51	52	131	0.02	37	0.02	12	0.01	49	0.04	458	0.02	464	0.02	441	0.01	11
BV	52	53	133	0.02	36	0.01	12	0.01	48	0.04	458	0.02	464	0.02	441	0.01	11
	53	54	136	0.02	36	0.02	12	0.01	51	0.04	458	0.02	465	0.02	441	0.01	11
	54	55	138	0.02	36	0.02	12	0.01	50	0.04	458	0.02	465	0.02	441	0.01	11
	55	56	141	0.02	36	0.02	12	0.01	53	0.04	458	0.02	466	0.02	441	0.01	11
	56	57	143	0.02	36	0.02	12	0.01	51	0.04	458	0.02	466	0.02	441	0.01	11
	57	58	146	0.02	36	0.02	12	0.01	51	0.04	458	0.02	467	0.02	441	0.01	11
	58	59	148	0.02	38	0.02	12	0.01	52	0.04	458	0.02	467	0.02	441	0.01	11
	59	60	151	0.02	37	0.02	12	0.01	51	0.04	458	0.02	468	0.02	441	0.01	11
	60	61	153	0.02	37	0.02	12	0.01	51	0.04	458	0.02	468	0.02	441	0.01	11
	61	62	156	0.02	37	0.02	12	0.01	53	0.04	458	0.02	469	0.02	441	0.01	11
	62	63	158	0.02	37	0.02	12	0.01	52	0.04	458	0.02	469	0.02	441	0.01	11
	63	64	161	0.02	37	0.02	12	0.01	53	0.04	458	0.02	470	0.03	442	0.01	11
	64	65	163	0.02	37	0.02	12	0.01	53	0.05	458	0.02	470	0.03	442	0.01	11
	65	66	166	0.02	37	0.02	12	0.01	54	0.05	458	0.02	471	0.03	442	0.01	11
	66	67	168	0.02	37	0.02	12	0.01	54	0.05	458	0.02	471	0.03	442	0.01	11
	67	68	171	0.02	41	0.02	12	0.01	55	0.05	458	0.02	472	0.03	442	0.01	11
	68	69	173	0.02	37	0.02	12	0.02	55	0.05	458	0.02	472	0.03	442	0.01	12
	69	70	176	0.03	37	0.02	12	0.02	56	0.05	458	0.02	473	0.03	442	0.01	12
	70	71	178	0.03	37	0.02	12	0.02	56	0.05	458	0.02	474	0.03	442	0.01	12
	71	72	181	0.03	37	0.02	12	0.02	57	0.05	458	0.02	474	0.03	442	0.01	12
	72	73	183	0.03	37	0.02	12	0.02	57	0.05	458	0.02	475	0.03	442	0.01	12
	73	74	186	0.03	37	0.02	12	0.02	58	0.06	458	0.03	475	0.03	442	0.01	12
	74	75	188	0.03	37	0.03	12	0.02	59	0.06	458	0.02	476	0.03	442	0.01	12

Continued on next page

Table A.2 (continued from previous page)

	circuit	#q	#G	MEDUSA$_{base}$		SliQSim		DDSIM		Quas[CFLOBDD]		Quas[WBDD]		Quas[BDD]		Quokka#	
				time	mem	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem
	75	76	191	0.03	37	0.03	12	0.02	59	0.06	458	0.03	476	0.04	442	0.01	12
	76	77	193	0.03	37	0.03	12	0.02	60	0.06	458	0.03	477	0.04	442	0.01	12
	77	78	196	0.04	37	0.03	12	0.02	60	0.06	458	0.03	477	0.04	442	0.01	12
	78	79	198	0.03	37	0.03	12	0.02	60	0.06	458	0.03	478	0.04	442	0.01	12
	79	80	201	0.03	40	0.03	12	0.02	61	0.06	458	0.03	478	0.04	442	0.01	12
	80	81	203	0.03	38	0.03	12	0.02	61	0.06	458	0.03	479	0.04	442	0.01	12
	81	82	206	0.03	37	0.03	12	0.02	62	0.06	458	0.03	479	0.04	442	0.01	12
	82	83	208	0.03	37	0.03	12	0.02	62	0.06	458	0.03	480	0.04	442	0.01	12
	83	84	211	0.03	39	0.03	12	0.02	63	0.06	458	0.03	480	0.04	443	0.01	12
	84	85	213	0.03	37	0.03	12	0.00	64	0.06	458	0.03	481	0.04	443	0.01	12
	85	86	216	0.03	38	0.03	12	0.00	64	0.06	458	0.03	481	0.05	443	0.01	12
	86	87	218	0.03	38	0.03	12	0.00	64	0.06	458	0.03	482	0.05	443	0.01	12
BV	87	88	221	0.03	39	0.03	12	0.00	65	0.06	458	0.03	482	0.05	443	0.01	12
	88	89	223	0.03	38	0.03	12	0.00	65	0.07	458	0.03	484	0.05	443	0.01	12
	89	90	226	0.03	37	0.03	12	0.00	66	0.07	458	0.03	484	0.05	443	0.01	12
	90	91	228	0.03	38	0.03	12	0.00	66	0.07	458	0.03	485	0.05	443	0.01	12
	91	92	231	0.03	37	0.03	12	0.00	67	0.07	458	0.03	485	0.05	443	0.01	12
	92	93	233	0.03	38	0.03	12	0.00	67	0.07	458	0.03	486	0.05	443	0.01	12
	93	94	236	0.03	40	0.03	12	0.00	68	0.07	458	0.03	486	0.06	443	0.01	12
	94	95	238	0.03	38	0.03	12	0.00	68	0.07	458	0.03	487	0.06	443	0.01	12
	95	96	241	0.03	38	0.04	12	0.00	69	0.07	458	0.03	487	0.06	443	0.02	12
	96	97	243	0.03	37	0.04	13	0.00	69	0.07	458	0.03	488	0.06	443	0.01	12
	97	98	246	0.03	38	0.04	13	0.01	70	0.07	458	0.03	488	0.06	444	0.01	12
	98	99	248	0.03	37	0.04	13	0.01	71	0.07	458	0.03	489	0.06	443	0.01	12
	99	100	251	0.03	40	0.04	13	0.01	71	0.07	458	0.04	489	0.06	444	0.02	12
	adder_8	24	330	0.02	34	0.03	12	0.00	29	0.06	458	0.01	449	0.03	441	0.09	13
	barenco_tof_3	5	20	0.00	22	0.01	12	0.00	29	0.00	456	0.00	439	0.00	440	0.01	11
	barenco_tof_4	7	34	0.00	22	0.01	12	0.00	29	0.01	456	0.01	440	0.00	439	0.01	12
	barenco_tof_5	9	50	0.01	23	0.01	12	0.00	29	0.01	456	0.01	441	0.00	440	0.01	12
	barenco_tof_10	19	130	0.01	28	0.01	12	0.00	29	0.03	458	0.01	446	0.01	440	0.03	12
	csla_mux_3	15	70	0.01	25	0.01	12	0.00	29	0.01	458	0.01	444	0.00	440	0.01	12
	csum_mux_9	30	140	0.01	30	0.01	12	0.00	29	0.03	457	0.01	452	0.01	440	0.06	12
	gf2^{4}_mult	12	65	0.02	24	0.01	12	0.00	29	0.02	458	0.01	443	0.00	440	0.03	12
	gf2^{5}_mult	15	97	0.01	26	0.01	12	0.00	29	0.02	458	0.01	444	0.01	440	0.02	12
	gf2^{6}_mult	18	135	0.01	28	0.01	12	0.00	29	0.04	458	0.01	446	0.01	440	0.05	12
	gf2^{7}_mult	21	179	0.02	32	0.01	12	0.00	29	0.04	457	0.01	447	0.01	441	0.04	13
	gf2^{8}_mult	24	243	0.02	32	0.02	12	0.00	29	0.05	458	0.01	449	0.02	441	0.05	13
	gf2^{9}_mult	27	285	0.02	36	0.02	12	0.00	29	0.06	458	0.01	450	0.03	441	0.06	14
	gf2^{10}_mult	30	347	0.03	35	0.03	12	0.00	29	0.07	458	0.01	452	0.04	442	0.08	14
FEYNMAN	gf2^{16}_mult	48	875	0.07	38	0.11	12	0.01	48	0.18	458	0.03	463	0.18	445	0.21	19
	gf2^{32}_mult	96	3,322	0.26	40	1.35	13	0.10	71	0.66	460	0.11	502	0.82	450	0.87	45
	gf2^{64}_mult	192	12,731	1.82	66	17.11	20	0.75	126	2.59	464	0.68	601	4.43	462	3.56	149
	gf2^{128}_mult	384	50,043	20.40	231	264.81	38	5.28	235	10.50	478	4.76	1,159	27.60	498	15.39	570
	gf2^{256}_mult	768	198,395	163.00	1,635	TO	TO	41.21	538	43.30	531	38.50	4,989	238.00	633	71.28	2,324
	grover_5	9	351	0.02	34	0.03	12	0.00	29	0.12	459	0.01	441	0.26	442	TO	TO
	ham15-low	17	213	0.02	30	0.01	12	0.00	29	0.04	458	0.01	445	0.01	440	0.54	12
	ham15-med	17	452	0.02	34	0.04	12	0.00	29	0.10	458	0.01	445	0.07	441	TO	TO
	ham15-high	20	1,798	0.06	38	0.49	12	0.01	30	0.37	458	0.02	447	0.41	443	TO	TO
	hwb6	7	109	0.01	24	0.01	12	0.00	29	0.03	458	0.01	440	0.00	440	13.79	12
	hwb8	12	6,446	0.16	38	3.70	13	0.03	33	0.95	460	0.04	443	1.03	444	TO	TO
	hwb10	16	31,764	0.80	51	84.20	15	0.21	38	4.72	466	0.22	447	1.56	445	TO	TO
	hwb11	15	87,789	2.64	103	660.93	22	0.49	70	12.80	475	0.52	449	1.51	448	TO	TO
	hwb12	20	171,482	5.80	205	2,568.02	35	1.13	133	27.20	510	1.35	456	6.43	457	3,193.79	1,070

Continued on next page

Table A.2 (continued from previous page)

	circuit	#q	#G	MEDUSA$_{base}$ time	MEDUSA$_{base}$ mem	SliQSim time	SliQSim mem	DDSIM time	DDSIM mem	Quas[CFLOBDD] time	Quas[CFLOBDD] mem	Quas[WBDD] time	Quas[WBDD] mem	Quas[BDD] time	Quas[BDD] mem	Quokka# time	Quokka# mem
FEYNMAN	mod5_4	5	23	0.00	22	0.01	12	0.00	29	0.00	456	0.00	439	0.00	439	0.04	11
	mod_adder_1024	28	1,435	0.06	38	0.31	12	0.01	29	0.30	458	0.02	451	0.34	443	0.23	21
	mod_mult_55	9	49	0.01	23	0.01	12	0.00	29	0.01	457	0.01	441	0.00	440	0.01	12
	mod_red_21	11	108	0.01	25	0.01	12	0.00	29	0.02	458	0.01	442	0.00	440	0.04	12
	qcla_adder_10	36	181	0.02	33	0.01	12	0.00	42	0.04	458	0.01	455	0.01	441	0.03	12
	qcla_com_7	24	153	0.01	30	0.01	12	0.00	29	0.03	458	0.01	449	0.01	440	0.10	12
	qcla_mod_7	26	294	0.02	33	0.02	12	0.00	29	0.07	458	0.01	450	0.03	441	TO	TO
	qft_4	5	159	ERR	ERR	0.01	12	0.00	29	ERR	ERR	ERR	ERR	ERR	ERR	0.11	12
	rc_adder_6	14	90	0.01	26	0.01	12	0.00	29	0.02	458	0.01	444	0.00	440	0.04	12
	tof_3	5	15	0.00	21	0.00	12	0.00	29	0.00	458	0.00	439	0.00	439	0.01	11
	tof_4	7	25	0.00	22	0.01	12	0.00	29	0.01	456	0.01	440	0.00	439	0.01	11
	tof_5	9	35	0.01	23	0.01	12	0.00	29	0.01	458	0.01	441	0.00	439	0.01	11
	tof_10	19	85	0.01	27	0.01	12	0.00	29	0.02	458	0.01	446	0.00	440	0.02	12
	vbe_adder_3	10	50	0.01	23	0.01	12	0.00	29	0.01	456	0.01	442	0.00	440	0.03	12
MCTOFFOLI	3	6	5	0.02	21	0.00	12	0.00	29	0.02	457	0.00	440	0.00	439	0.01	11
	4	8	7	0.00	21	0.01	12	0.00	29	0.00	457	0.01	441	0.00	439	0.01	11
	5	10	9	0.00	22	0.01	12	0.00	29	0.00	457	0.01	442	0.00	439	0.01	12
	6	12	11	0.00	22	0.01	12	0.00	29	0.00	456	0.01	443	0.00	440	0.02	12
	7	14	13	0.01	23	0.01	12	0.00	29	0.00	456	0.01	444	0.00	439	0.01	12
	8	16	15	0.01	23	0.01	12	0.00	29	0.00	456	0.01	445	0.00	439	0.01	12
	9	18	17	0.01	23	0.01	12	0.00	29	0.00	456	0.01	446	0.00	440	0.02	12
	10	20	19	0.01	24	0.01	12	0.00	29	0.00	456	0.01	447	0.00	440	0.02	12
	11	22	21	0.01	24	0.01	12	0.00	29	0.00	456	0.01	448	0.00	440	0.02	12
	12	24	23	0.01	24	0.01	12	0.00	29	0.00	456	0.01	449	0.00	440	0.02	12
	13	26	25	0.01	25	0.01	12	0.00	29	0.00	456	0.01	450	0.00	440	0.02	12
	14	28	27	0.01	25	0.01	12	0.00	29	0.00	456	0.01	451	0.00	440	0.02	12
	15	30	29	0.01	25	0.01	12	0.00	29	0.00	456	0.01	452	0.00	440	0.02	12
	16	32	31	0.01	26	0.01	12	0.00	29	0.00	456	0.01	453	0.00	440	0.03	12
	17	34	33	0.01	26	0.01	12	0.00	40	0.00	456	0.01	454	0.00	440	0.03	12
	18	36	35	0.01	26	0.01	12	0.00	41	0.00	456	0.01	455	0.00	440	0.03	12
	19	38	37	0.01	27	0.01	12	0.00	42	0.00	457	0.01	456	0.00	440	0.03	12
	20	40	39	0.01	27	0.01	12	0.00	42	0.00	456	0.01	457	0.00	440	0.03	12
	21	42	41	0.01	28	0.01	12	0.00	46	0.00	456	0.01	458	0.00	440	0.03	12
	22	44	43	0.01	27	0.01	12	0.00	47	0.00	456	0.01	459	0.00	440	0.03	13
	23	46	45	0.01	29	0.01	12	0.00	47	0.00	456	0.01	460	0.00	440	0.03	12
	24	48	47	0.01	27	0.01	12	0.00	49	0.00	456	0.01	461	0.00	440	0.04	12
	25	50	49	0.01	30	0.01	12	0.00	50	0.00	456	0.01	462	0.00	440	0.04	12
	26	52	51	0.01	30	0.01	12	0.00	48	0.00	456	0.01	463	0.00	440	0.04	13
	27	54	53	0.01	30	0.01	12	0.00	49	0.00	457	0.01	464	0.00	440	0.04	13
	28	56	55	0.01	31	0.01	12	0.00	51	0.01	456	0.01	465	0.00	440	0.04	13
	29	58	57	0.03	31	0.01	12	0.00	50	0.02	456	0.01	466	0.00	440	0.06	13
	30	60	59	0.03	31	0.01	12	0.00	51	0.01	456	0.01	467	0.00	440	0.05	13
	31	62	61	0.02	32	0.01	12	0.00	53	0.00	456	0.01	468	0.00	440	0.05	13
	32	64	63	0.02	32	0.01	12	0.00	53	0.00	456	0.01	469	0.00	440	0.05	13
	33	66	65	0.02	32	0.01	12	0.00	54	0.00	457	0.02	470	0.00	440	0.05	13
	34	68	67	0.02	33	0.01	12	0.00	55	0.00	456	0.02	471	0.00	440	0.05	13
	35	70	69	0.02	33	0.01	12	0.00	56	0.00	457	0.02	472	0.00	440	0.05	13
	36	72	71	0.02	33	0.01	12	0.00	57	0.00	457	0.02	473	0.00	441	0.05	13
	37	74	73	0.02	34	0.01	12	0.00	58	0.00	456	0.02	474	0.00	441	0.05	13
	38	76	75	0.02	34	0.01	12	0.00	59	0.00	456	0.02	475	0.00	441	0.05	13

Continued on next page

Table A.2 (continued from previous page)

	circuit	#q	#G	MEDUSA$_{base}$ time	MEDUSA$_{base}$ mem	SliQSim time	SliQSim mem	DDSIM time	DDSIM mem	Quas[CFLOBDD] time	Quas[CFLOBDD] mem	Quas[WBDD] time	Quas[WBDD] mem	Quas[BDD] time	Quas[BDD] mem	Quokka# time	Quokka# mem
	39	78	77	0.02	34	0.01	12	0.00	60	0.00	456	0.02	476	0.00	440	0.06	13
	40	80	79	0.02	34	0.01	12	0.00	61	0.00	457	0.02	477	0.01	441	0.06	13
	41	82	81	0.02	35	0.01	12	0.00	62	0.00	456	0.02	478	0.01	441	0.06	13
	42	84	83	0.02	35	0.01	12	0.00	63	0.00	457	0.02	479	0.01	441	0.06	14
	43	86	85	0.03	35	0.01	12	0.00	64	0.00	456	0.02	480	0.01	441	0.06	14
	44	88	87	0.02	36	0.01	12	0.00	65	0.00	457	0.02	481	0.01	441	0.06	14
	45	90	89	0.02	36	0.01	12	0.00	66	0.00	457	0.02	483	0.01	441	0.07	14
	46	92	91	0.03	36	0.01	12	0.00	67	0.00	456	0.02	483	0.01	441	0.07	14
	47	94	93	0.02	36	0.01	12	0.00	68	0.00	457	0.02	485	0.01	441	0.07	14
	48	96	95	0.02	36	0.01	12	0.00	69	0.00	457	0.02	485	0.01	441	0.07	14
	49	98	97	0.02	36	0.01	12	0.00	70	0.00	457	0.02	486	0.01	441	0.07	14
	50	100	99	0.02	36	0.01	12	0.00	71	0.00	457	0.02	488	0.01	441	0.07	14
	51	102	101	0.03	36	0.01	12	0.00	72	0.00	456	0.02	488	0.01	441	0.07	14
	52	104	103	0.03	37	0.01	12	0.00	73	0.00	456	0.02	489	0.01	441	0.08	14
	53	106	105	0.03	37	0.01	12	0.00	74	0.00	457	0.02	490	0.01	441	0.07	14
	54	108	107	0.03	37	0.01	12	0.00	75	0.00	457	0.02	491	0.01	441	0.08	14
	55	110	109	0.03	37	0.01	12	0.00	76	0.00	456	0.03	493	0.01	441	0.08	14
	56	112	111	0.03	37	0.01	12	0.00	77	0.00	457	0.03	494	0.01	441	0.08	14
	57	114	113	0.03	37	0.01	12	0.00	78	0.00	457	0.03	494	0.01	441	0.08	14
	58	116	115	0.03	37	0.01	12	0.00	79	0.00	457	0.03	495	0.01	441	0.08	15
	59	118	117	0.03	37	0.01	12	0.00	80	0.00	457	0.03	496	0.01	442	0.09	15
MCToffoli	60	120	119	0.03	37	0.01	12	0.00	81	0.00	456	0.03	497	0.01	441	0.09	15
	61	122	121	0.03	37	0.01	12	0.00	82	0.00	457	0.03	498	0.01	442	0.09	15
	62	124	123	0.03	37	0.01	12	0.00	83	0.00	457	0.03	499	0.01	442	0.09	15
	63	126	125	0.03	37	0.01	12	0.00	84	0.00	456	0.03	500	0.01	442	0.09	15
	64	128	127	0.03	37	0.01	12	0.00	85	0.00	457	0.03	501	0.01	442	0.09	15
	65	130	129	0.03	37	0.01	12	0.00	86	0.00	457	0.03	502	0.01	442	0.10	15
	66	132	131	0.03	37	0.01	12	0.00	87	0.00	457	0.03	504	0.01	442	0.09	15
	67	134	133	0.03	37	0.01	12	0.00	88	0.00	456	0.03	505	0.01	442	0.10	15
	68	136	135	0.03	37	0.01	12	0.00	89	0.00	456	0.03	505	0.01	442	0.10	15
	69	138	137	0.03	37	0.01	12	0.00	90	0.00	457	0.03	506	0.01	442	0.10	15
	70	140	139	0.03	38	0.01	12	0.00	91	0.00	456	0.03	508	0.01	442	0.10	15
	71	142	141	0.04	39	0.01	13	0.00	92	0.00	456	0.03	508	0.01	442	0.10	15
	72	144	143	0.04	37	0.01	13	0.00	93	0.00	456	0.03	509	0.01	442	0.10	15
	73	146	145	0.04	38	0.01	13	0.00	94	0.00	456	0.03	511	0.02	442	0.10	16
	74	148	147	0.04	39	0.01	13	0.00	95	0.00	457	0.03	511	0.02	442	0.11	16
	75	150	149	0.04	37	0.01	13	0.00	96	0.00	456	0.04	513	0.01	442	0.11	16
	76	152	151	0.04	37	0.01	13	0.00	97	0.00	457	0.03	513	0.01	442	0.11	16
	77	154	153	0.04	38	0.02	13	0.00	98	0.00	457	0.04	516	0.02	443	0.11	16
	78	156	155	0.04	38	0.01	13	0.00	99	0.00	456	0.04	517	0.02	442	0.11	16
	79	158	157	0.04	38	0.01	13	0.00	100	0.00	457	0.04	518	0.02	443	0.12	16
	80	160	159	0.04	37	0.01	13	0.01	101	0.00	457	0.04	519	0.02	443	0.13	16
	81	162	161	0.04	37	0.01	13	0.01	102	0.00	457	0.04	520	0.02	443	0.11	16
	82	164	163	0.04	40	0.01	13	0.01	103	0.00	456	0.04	521	0.02	443	0.12	16
	83	166	165	0.05	38	0.01	13	0.01	104	0.00	456	0.04	522	0.02	443	0.12	16
	84	168	167	0.04	40	0.01	13	0.01	105	0.00	457	0.04	523	0.02	443	0.12	16

Continued on next page

Table A.2 (continued from previous page)

	circuit	#q	#G	MEDUSA$_{base}$ time	MEDUSA$_{base}$ mem	SliQSim time	SliQSim mem	DDSIM time	DDSIM mem	Quas[CFLOBDD] time	Quas[CFLOBDD] mem	Quas[WBDD] time	Quas[WBDD] mem	Quas[BDD] time	Quas[BDD] mem	Quokka# time	Quokka# mem
MCToffoli	85	170	169	0.05	38	0.01	13	0.01	106	0.00	457	0.04	523	0.02	443	0.12	16
	86	172	171	0.04	38	0.01	13	0.01	107	0.00	457	0.04	525	0.02	443	0.12	16
	87	174	173	0.04	40	0.01	13	0.01	108	0.00	456	0.04	526	0.02	443	0.13	17
	88	176	175	0.04	37	0.01	13	0.01	109	0.00	457	0.04	527	0.02	443	0.13	17
	89	178	177	0.05	37	0.01	13	0.01	110	0.00	457	0.04	528	0.02	443	0.13	16
	90	180	179	0.04	38	0.01	13	0.01	111	0.00	458	0.05	529	0.02	443	0.13	17
	91	182	181	0.05	38	0.01	13	0.01	112	0.00	457	0.05	530	0.02	443	0.13	17
	92	184	183	0.04	38	0.01	13	0.01	113	0.00	457	0.05	530	0.02	443	0.13	17
	93	186	185	0.04	38	0.02	13	0.01	114	0.00	458	0.05	531	0.02	443	0.13	17
	94	188	187	0.05	39	0.01	13	0.01	115	0.00	458	0.05	533	0.02	443	0.14	17
	95	190	189	0.04	38	0.02	13	0.01	116	0.00	458	0.05	533	0.02	444	0.14	17
	96	192	191	0.05	39	0.02	13	0.01	117	0.00	458	0.05	535	0.02	443	0.14	17
	97	194	193	0.05	40	0.02	13	0.01	118	0.00	457	0.05	535	0.02	444	0.14	17
	98	196	195	0.05	38	0.01	13	0.01	119	0.00	458	0.05	537	0.02	444	0.15	17
	99	198	197	0.06	38	0.02	15	0.01	120	0.00	457	0.05	538	0.02	444	0.15	17
MOG	3	9	64	0.01	26	0.01	12	0.00	29	0.01	458	0.01	441	0.00	440	0.03	12
	4	12	123	0.01	31	0.01	12	0.00	29	0.02	458	0.01	443	0.02	440	0.10	12
	5	15	202	0.02	35	0.02	12	0.00	29	0.03	458	0.01	444	0.12	441	0.30	13
	6	18	357	0.03	37	0.04	12	0.01	30	0.11	460	0.01	446	3.67	446	1.04	15
	7	21	552	0.05	38	0.07	12	0.01	30	0.20	462	0.02	448	75.40	473	2.67	17
	8	24	939	0.08	38	0.17	12	0.02	30	1.06	485	0.02	451	45.60	467	8.00	21
	9	27	1,492	0.12	40	0.45	12	0.04	32	3.15	515	0.03	452	TO	TO	20.73	28
	10	30	2,433	0.20	42	1.26	12	0.08	34	9.08	595	0.05	456	TO	TO	62.68	40
	11	33	3,746	0.36	45	3.12	13	0.13	42	48.80	906	0.08	462	TO	TO	167.01	56
Random	3	3	9	0.00	21	0.01	12	0.00	29	ERR	ERR	0.01	438	ERR	ERR	0.03	11
	4	4	12	0.00	21	0.01	12	0.00	29	ERR	ERR	0.00	439	ERR	ERR	0.01	12
	5	5	15	0.00	21	0.01	12	0.00	29	ERR	ERR	0.01	439	ERR	ERR	0.02	12
	6	6	18	0.00	22	0.01	12	0.00	29	ERR	ERR	0.00	440	ERR	ERR	0.01	12
	7	7	21	0.00	22	0.01	12	0.00	29	ERR	ERR	0.01	440	ERR	ERR	0.01	11
	8	8	24	0.00	22	0.01	12	0.00	29	ERR	ERR	0.01	441	ERR	ERR	0.02	12
	9	9	27	0.01	23	0.01	12	0.00	29	ERR	ERR	0.01	441	ERR	ERR	0.00	12
	10	10	30	0.00	22	0.01	12	0.00	29	ERR	ERR	0.01	442	ERR	ERR	0.01	12
	11	11	33	0.01	24	0.01	12	0.00	29	ERR	ERR	0.01	442	ERR	ERR	0.01	12
	12	12	36	0.01	25	0.01	12	0.00	29	ERR	ERR	0.01	443	ERR	ERR	0.01	11
	13	13	39	0.01	24	0.01	12	0.00	29	ERR	ERR	0.01	443	ERR	ERR	0.01	11
	14	14	42	0.01	25	0.01	12	0.00	29	ERR	ERR	0.01	444	ERR	ERR	0.01	12
	15	15	45	0.01	28	0.01	12	0.00	29	ERR	ERR	0.01	444	ERR	ERR	0.02	12
	16	16	48	0.01	29	0.01	12	0.00	29	ERR	ERR	0.01	445	ERR	ERR	0.01	12
	17	17	51	0.01	25	0.01	12	0.00	29	ERR	ERR	0.01	445	ERR	ERR	0.01	12
	18	18	54	0.01	27	0.01	12	0.00	29	ERR	ERR	0.01	446	ERR	ERR	0.02	12
	19	19	57	0.01	31	0.02	12	0.00	30	ERR	ERR	0.01	446	ERR	ERR	0.01	12
	20	20	60	0.02	32	0.02	12	0.00	29	ERR	ERR	0.01	447	ERR	ERR	0.01	12
	21	21	63	0.02	33	0.02	12	0.00	29	ERR	ERR	0.01	447	ERR	ERR	0.01	12
	22	22	66	0.01	27	0.01	12	0.00	29	ERR	ERR	0.01	448	ERR	ERR	0.01	12
	23	23	69	0.02	32	0.02	12	0.00	29	ERR	ERR	0.01	448	ERR	ERR	0.02	12
	24	24	72	0.01	29	0.01	12	0.00	29	ERR	ERR	0.01	449	ERR	ERR	0.01	12

Continued on next page

Table A.2 (continued from previous page)

				`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**
	25	25	75	0.03	36	0.02	12	0.00	29	ERR	ERR	0.01	449	ERR	ERR	0.01	12
	26	26	78	0.02	35	0.04	12	0.00	29	ERR	ERR	0.01	450	ERR	ERR	0.02	12
	27	27	81	0.02	35	0.04	12	0.00	29	ERR	ERR	0.01	450	ERR	ERR	0.02	12
	28	28	84	0.03	37	0.05	12	0.01	29	ERR	ERR	0.01	451	ERR	ERR	0.02	12
	29	29	87	0.02	33	0.02	12	0.00	29	ERR	ERR	0.01	451	ERR	ERR	0.01	12
	30	30	90	0.02	35	0.05	12	0.00	29	ERR	ERR	0.01	452	ERR	ERR	0.02	12
	31	31	93	0.03	37	0.05	12	0.00	29	ERR	ERR	0.01	453	ERR	ERR	0.04	12
	32	32	96	0.02	36	0.04	12	0.01	29	ERR	ERR	0.01	453	ERR	ERR	0.02	12
	33	33	99	0.05	37	0.10	12	0.01	41	ERR	ERR	0.02	454	ERR	ERR	0.03	12
	34	34	102	0.03	37	0.05	12	0.01	41	ERR	ERR	0.01	454	ERR	ERR	0.01	12
	35	35	105	0.04	37	0.09	13	0.00	40	ERR	ERR	0.01	455	ERR	ERR	0.02	12
	36	36	108	0.03	38	0.05	12	0.01	43	ERR	ERR	0.01	456	ERR	ERR	0.03	12
	37	37	111	0.05	38	0.10	12	0.01	40	ERR	ERR	0.01	456	ERR	ERR	0.01	12
	38	38	114	0.02	36	0.05	12	0.00	42	ERR	ERR	0.01	456	ERR	ERR	0.02	12
	39	39	117	0.03	39	0.06	12	0.00	41	ERR	ERR	0.01	457	ERR	ERR	0.01	12
	40	40	120	0.04	37	0.07	12	0.00	43	ERR	ERR	0.01	458	ERR	ERR	0.03	12
	41	41	123	0.04	38	0.12	13	0.01	45	ERR	ERR	0.02	459	ERR	ERR	0.02	12
	42	42	126	0.05	40	0.11	12	0.01	45	ERR	ERR	0.02	459	ERR	ERR	0.02	12
	43	43	129	0.02	35	0.03	12	0.00	45	ERR	ERR	0.01	459	ERR	ERR	0.02	12
	44	44	132	0.03	39	0.07	12	0.00	46	ERR	ERR	0.02	459	ERR	ERR	0.03	12
	45	45	135	0.08	38	0.12	12	0.01	46	ERR	ERR	0.02	461	ERR	ERR	0.01	12
	46	46	138	0.20	40	0.19	12	0.05	46	ERR	ERR	0.03	463	ERR	ERR	0.02	12
RANDOM	47	47	141	0.06	37	0.05	12	0.00	45	ERR	ERR	0.02	461	ERR	ERR	0.02	12
	48	48	144	0.03	37	0.05	12	0.01	47	ERR	ERR	0.02	462	ERR	ERR	0.01	12
	49	49	147	0.04	38	0.13	14	0.01	48	ERR	ERR	0.02	462	ERR	ERR	0.02	12
	50	50	150	0.06	40	0.12	12	0.01	50	ERR	ERR	0.03	463	ERR	ERR	0.01	12
	51	51	153	0.03	40	0.11	14	0.01	48	ERR	ERR	0.02	463	ERR	ERR	0.02	12
	52	52	156	0.04	37	0.07	12	0.01	49	ERR	ERR	0.02	464	ERR	ERR	0.04	12
	53	53	159	0.14	40	0.08	12	0.03	49	ERR	ERR	0.03	465	ERR	ERR	0.02	12
	54	54	162	0.08	37	0.12	12	0.01	49	ERR	ERR	0.02	465	ERR	ERR	0.02	12
	55	55	165	0.18	40	0.21	12	0.04	49	ERR	ERR	0.03	467	ERR	ERR	0.03	12
	56	56	168	0.05	37	0.08	13	0.01	51	ERR	ERR	0.02	466	ERR	ERR	0.04	12
	57	57	171	0.17	42	0.14	12	0.02	52	ERR	ERR	0.03	467	ERR	ERR	0.04	12
	58	58	174	0.06	38	0.09	12	0.02	51	ERR	ERR	0.02	467	ERR	ERR	0.03	12
	59	59	177	0.08	38	0.19	15	0.02	51	ERR	ERR	0.02	468	ERR	ERR	0.05	12
	60	60	180	0.21	42	0.20	12	0.09	51	ERR	ERR	0.05	469	ERR	ERR	0.02	12
	61	61	183	0.08	40	0.09	13	0.02	51	ERR	ERR	0.02	469	ERR	ERR	0.02	12
	62	62	186	0.05	40	0.13	14	0.01	53	ERR	ERR	0.02	469	ERR	ERR	0.02	12
	63	63	189	0.10	40	0.18	14	0.04	53	ERR	ERR	0.03	470	ERR	ERR	0.02	12
	64	64	192	0.05	38	0.12	14	0.01	53	ERR	ERR	0.02	470	ERR	ERR	0.02	12
	65	65	195	0.09	39	0.18	13	0.01	53	ERR	ERR	0.03	472	ERR	ERR	0.02	12
	66	66	198	0.05	38	0.12	15	0.01	54	ERR	ERR	0.02	472	ERR	ERR	0.04	12
	67	67	201	0.63	48	0.24	14	0.26	54	ERR	ERR	0.13	481	ERR	ERR	0.06	12
	68	68	204	0.43	44	0.19	13	0.15	55	ERR	ERR	0.07	477	ERR	ERR	0.04	12
	69	69	207	0.18	42	0.23	14	0.02	55	ERR	ERR	0.03	474	ERR	ERR	0.05	12
	70	70	210	0.16	40	0.14	13	0.04	56	ERR	ERR	0.04	475	ERR	ERR	0.02	12

Continued on next page

Table A.2 (continued from previous page)

				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
Random	71	71	213	0.07	38	0.18	14	0.01	56	ERR	ERR	0.03	475	ERR	ERR	0.02	12
	72	72	216	0.21	42	0.23	13	0.04	57	ERR	ERR	0.04	476	ERR	ERR	0.04	12
	73	73	219	0.38	46	0.24	12	0.22	57	ERR	ERR	0.06	479	ERR	ERR	0.06	12
	74	74	222	0.39	43	0.33	13	0.49	58	ERR	ERR	0.08	480	ERR	ERR	0.04	12
	75	75	225	0.30	42	0.24	12	0.07	59	ERR	ERR	0.05	478	ERR	ERR	0.06	12
	76	76	228	0.41	44	0.18	12	0.09	59	ERR	ERR	0.05	481	ERR	ERR	0.02	12
	77	77	231	0.82	50	0.37	13	0.45	60	ERR	ERR	0.07	481	ERR	ERR	0.03	12
	78	78	234	0.70	46	0.30	14	0.40	60	ERR	ERR	0.14	486	ERR	ERR	0.03	12
	79	79	237	0.31	42	0.29	14	0.12	60	ERR	ERR	0.05	482	ERR	ERR	0.03	12
	80	80	240	0.23	42	0.26	14	0.05	61	ERR	ERR	0.04	481	ERR	ERR	0.03	12
	81	81	243	0.34	44	0.29	12	0.22	62	ERR	ERR	0.06	483	ERR	ERR	0.02	12
	82	82	246	0.22	42	0.21	14	0.05	62	ERR	ERR	0.04	481	ERR	ERR	0.03	12
	83	83	249	0.44	44	0.25	12	0.14	63	ERR	ERR	0.07	484	ERR	ERR	0.03	12
	84	84	252	0.16	39	0.28	13	0.05	63	ERR	ERR	0.04	483	ERR	ERR	0.08	12
	85	85	255	1.00	52	0.47	15	2.12	64	ERR	ERR	0.11	485	ERR	ERR	0.03	12
	86	86	258	15.30	214	0.48	14	2.25	72	ERR	ERR	3.25	553	ERR	ERR	0.07	12
	87	87	261	0.58	46	0.28	12	0.15	64	ERR	ERR	0.06	486	ERR	ERR	0.15	13
	88	88	264	0.15	39	0.30	16	0.04	65	ERR	ERR	0.04	483	ERR	ERR	0.03	12
	89	89	267	9.48	105	0.67	14	0.72	66	ERR	ERR	0.59	492	ERR	ERR	0.06	13
	90	90	270	0.53	44	0.24	13	0.07	66	ERR	ERR	0.06	488	ERR	ERR	0.03	12
	91	91	273	0.33	44	0.25	12	0.03	66	ERR	ERR	0.04	486	ERR	ERR	0.03	12
	92	92	276	0.33	42	0.36	13	0.13	67	ERR	ERR	0.06	489	ERR	ERR	0.07	12
	93	93	279	1.68	62	0.32	13	0.18	68	ERR	ERR	0.11	494	ERR	ERR	0.05	12
	94	94	282	79.60	337	0.78	18	4.45	76	ERR	ERR	74.30	521	ERR	ERR	0.08	13
	95	95	285	0.25	40	0.34	15	0.07	68	ERR	ERR	0.06	488	ERR	ERR	0.03	12
	96	96	288	0.40	44	0.34	14	0.10	69	ERR	ERR	0.06	493	ERR	ERR	0.12	13
	97	97	291	5.70	118	0.42	13	1.47	78	ERR	ERR	0.42	525	ERR	ERR	0.03	13
	98	98	294	0.57	48	0.32	14	0.11	70	ERR	ERR	0.06	494	ERR	ERR	0.03	12
	99	99	297	9.58	173	0.38	12	2.61	79	ERR	ERR	0.67	526	ERR	ERR	0.08	13
RevLib	0410184_169	14	46	0.01	23	0.01	12	0.00	29	0.00	458	0.01	444	0.00	440	0.01	12
	4gt11_82	5	12	0.00	21	0.01	12	0.00	29	0.00	458	0.00	439	0.00	440	0.01	11
	4gt11_83	5	8	0.00	21	0.01	12	0.00	29	0.00	456	0.00	439	0.00	439	0.01	11
	4gt11_84	5	3	0.00	21	0.01	12	0.00	29	0.00	458	0.00	439	0.00	439	0.01	11
	4gt11-v1_85	5	4	0.00	21	0.01	12	0.00	29	0.00	456	0.00	439	0.00	439	0.01	11
	4mod5-bdd_287	7	8	0.00	21	0.01	12	0.00	29	0.00	456	0.01	440	0.00	439	0.01	11
	add8_172	25	32	0.01	24	0.01	12	0.00	29	0.01	458	0.01	449	0.00	440	0.06	12
	add16_174	49	64	0.01	30	0.01	12	0.00	49	0.01	458	0.01	462	0.00	440	0.13	12
	add32_183	97	128	0.03	37	0.01	12	0.00	70	0.02	458	0.03	487	0.01	441	0.39	13
	add64_184	193	256	0.05	37	0.02	13	0.02	118	0.04	459	0.06	537	0.04	445	1.28	15
	alu-v0_27	5	6	0.00	21	0.00	12	0.00	29	0.00	458	0.00	439	0.00	439	0.01	11
	alu-v1_28	5	7	0.00	21	0.01	12	0.00	29	0.00	458	0.00	439	0.00	439	0.01	11
	alu-v1_29	5	7	0.00	21	0.01	12	0.00	29	0.00	456	0.00	439	0.00	439	0.01	11
	alu-v2_33	5	7	0.00	21	0.01	12	0.00	29	0.00	458	0.01	439	0.00	439	0.01	11
	alu-v3_34	5	7	0.00	21	0.01	12	0.00	29	0.00	457	0.00	439	0.00	439	0.01	11
	alu-v3_35	5	7	0.00	21	0.01	12	0.00	29	0.00	458	0.00	439	0.00	439	0.01	11
	alu-v4_37	5	7	0.00	21	0.01	12	0.00	29	0.00	457	0.00	439	0.00	440	0.01	11
	alu-bdd_288	7	9	0.00	19	0.01	12	0.00	29	0.00	456	0.01	440	0.00	439	0.01	11
	apex2_289	498	1,746	0.40	44	0.10	16	0.29	271	0.13	462	0.41	750	0.80	485	0.91	46
	apex5_290	1,025	2,909	1.75	62	0.37	44	1.03	536	0.26	467	1.33	1,214	3.95	516	2.11	73
	avg8_325	320	1,757	0.21	40	0.07	14	0.12	182	0.12	461	0.31	633	0.30	457	TO	TO

Continued on next page

Table A.2 (continued from previous page)

	circuit	#q	#G	MEDUSA$_{base}$		SliQSim		DDSIM		Quas[CFLOBDD]		Quas[WBDD]		Quas[BDD]		Quokka#	
				time	mem	time	mem	time	mem	time	mem	time	mem	time	mem	time	mem
RevLib	avg16_324	576	3,484	0.52	46	0.17	18	0.52	311	0.22	465	1.03	806	1.25	491	TO	TO
	bw_291	87	307	0.03	37	0.02	12	0.01	64	0.02	458	0.03	482	0.02	443	0.14	16
	cnt3-5_179	16	25	0.01	23	0.01	12	0.00	29	0.00	458	0.01	445	0.00	440	0.05	12
	cps_292	923	2,763	1.19	58	0.21	31	1.25	485	0.22	465	1.09	1,035	2.82	528	1.39	60
	cycle10_293	39	78	0.01	29	0.01	12	0.00	41	0.02	458	0.01	456	0.00	440	0.03	12
	decod24-v0_38	4	6	0.00	21	0.01	12	0.00	29	0.00	456	0.00	439	0.00	439	0.03	11
	decod24-v2_43	4	6	0.00	21	0.01	12	0.00	29	0.00	456	0.00	439	0.00	439	0.01	11
	decod24-enable_125	6	9	0.00	21	0.01	12	0.00	29	0.00	456	0.01	440	0.00	440	0.01	11
	decod24-bdd_294	6	11	0.00	21	0.01	12	0.00	29	0.00	456	0.00	440	0.00	439	0.01	11
	e64-bdd_295	195	387	0.06	38	0.02	13	0.03	119	0.03	458	0.07	543	0.06	448	0.11	16
	ex-1_166	3	4	0.00	21	0.00	12	0.00	29	0.00	456	0.00	438	0.00	440	0.01	11
	ex1_226	6	7	0.00	21	0.01	12	0.00	29	0.00	457	0.01	440	0.00	439	0.02	11
	ex5p_296	206	647	0.08	40	0.03	13	0.06	124	0.05	459	0.09	548	0.10	449	0.28	21
	fredkin_6	3	3	0.00	21	0.01	12	0.00	29	0.00	456	0.00	438	0.00	440	0.03	11
	frg2_297	1,219	3,724	2.32	93	0.50	49	1.52	633	0.32	469	1.90	1,307	6.25	498	2.15	84
	ham3_102	3	5	0.00	21	0.01	12	0.00	29	0.00	457	0.00	438	0.00	439	0.01	11
	ham7_106	7	25	0.00	21	0.01	12	0.00	29	0.00	458	0.01	440	0.00	439	0.04	11
	ham7_299	21	61	0.01	24	0.01	12	0.00	29	0.00	457	0.01	447	0.00	440	0.02	12
	ham15_298	45	153	0.01	30	0.01	12	0.00	45	0.02	458	0.01	460	0.01	440	0.04	13
	hwb5_300	28	88	0.01	31	0.01	12	0.00	29	0.01	458	0.01	451	0.00	440	0.04	13
	hwb6_301	46	159	0.02	31	0.01	12	0.00	45	0.01	458	0.01	460	0.01	441	0.07	14
	hwb7_302	73	281	0.03	37	0.02	12	0.01	58	0.03	458	0.02	475	0.02	442	0.13	16
	hwb8_303	112	449	0.05	38	0.02	12	0.02	77	0.04	459	0.04	497	0.04	444	0.21	19
	hwb9_304	170	699	0.09	38	0.03	13	0.05	106	0.06	460	0.08	530	0.08	449	0.36	24
	mini_alu_305	10	20	0.00	22	0.01	12	0.00	29	0.00	456	0.01	442	0.00	440	0.01	11
	mod5adder_306	32	96	0.01	28	0.01	12	0.00	29	0.01	458	0.01	453	0.00	440	0.04	13
	mod5d2_70	5	8	0.00	21	0.01	12	0.00	29	0.00	456	0.00	439	0.00	439	0.01	11
	mod5mils_71	5	5	0.00	21	0.01	12	0.00	29	0.00	456	0.00	439	0.00	439	0.01	11
	one-two-three-v2_100	5	8	0.00	21	0.01	12	0.00	29	0.00	458	0.00	439	0.00	439	0.01	11
	one-two-three-v3_101	5	8	0.00	21	0.01	12	0.00	29	0.00	456	0.00	439	0.00	439	0.01	11
	parity_247	17	32	0.01	23	0.01	12	0.00	29	0.01	458	0.01	445	0.00	439	0.01	11
	pdc_307	619	2,080	0.74	48	0.15	21	0.63	332	0.19	464	0.58	812	1.20	505	1.03	49
	peres_9	3	2	0.00	21	0.00	12	0.00	29	0.00	456	0.00	438	0.00	439	0.01	11
	plus63mod4096_309	23	49	0.01	26	0.01	12	0.00	29	0.00	458	0.01	448	0.00	440	0.01	12
	plus63mod8192_310	25	53	0.01	26	0.01	12	0.00	29	0.00	458	0.01	449	0.00	440	0.01	12
	plus127mod8192_308	25	54	0.01	26	0.01	12	0.00	29	0.00	458	0.01	449	0.00	440	0.01	12
	rd32_270	5	9	0.00	21	0.01	12	0.00	29	0.00	456	0.00	439	0.00	439	0.01	11
	rd32_271	5	9	0.00	21	0.01	12	0.00	29	0.00	456	0.00	439	0.00	439	0.01	11
	rd32_272	5	6	0.00	21	0.00	12	0.00	29	0.00	457	0.00	439	0.00	440	0.01	11
	rd53_138	8	12	0.00	21	0.01	12	0.00	29	0.00	456	0.01	441	0.00	439	0.01	12
	rd53_311	13	34	0.01	23	0.01	12	0.00	29	0.00	458	0.01	443	0.00	440	0.02	12
	rd73_140	10	20	0.00	22	0.01	12	0.00	29	0.00	457	0.01	442	0.00	440	0.01	12
	rd73_312	25	73	0.01	25	0.01	12	0.00	29	0.01	458	0.01	449	0.00	440	0.03	12
	rd84_142	15	28	0.01	23	0.01	12	0.00	29	0.00	456	0.01	444	0.00	440	0.02	12
	rd84_313	34	104	0.01	28	0.01	12	0.00	40	0.01	458	0.01	454	0.00	440	0.04	13
	seq_314	1,617	5,990	4.96	98	1.35	109	4.11	835	0.54	477	3.71	1,776	14.00	537	3.65	124
	spla_315	489	1,709	0.40	44	0.10	17	0.37	266	0.13	462	0.39	747	0.72	485	0.89	46
	sym6_316	14	29	0.01	23	0.01	12	0.00	29	0.00	457	0.01	444	0.00	440	0.02	12
	sym9_146	12	28	0.00	22	0.01	12	0.00	29	0.00	456	0.01	443	0.00	439	0.02	12
	sym9_192	12	28	0.00	22	0.01	12	0.00	29	0.00	457	0.01	443	0.00	440	0.02	12
	sym9_317	27	62	0.01	26	0.01	12	0.00	29	0.01	458	0.01	450	0.00	440	0.03	12
	sys6-v0_111	10	20	0.00	22	0.01	12	0.00	29	0.00	457	0.01	442	0.00	440	0.01	12

Continued on next page

Table A.2 (continued from previous page)

	circuit	#q	#G	MEDUSA$_{base}$ time	MEDUSA$_{base}$ mem	SliQSim time	SliQSim mem	DDSIM time	DDSIM mem	Quas[CFLOBDD] time	Quas[CFLOBDD] mem	Quas[WBDD] time	Quas[WBDD] mem	Quas[BDD] time	Quas[BDD] mem	Quokka# time	Quokka# mem
RevLib	urf1_149	9	11,554	0.46	22	0.33	14	0.04	33	0.12	458	0.07	441	0.16	440	TO	TO
	urf2_152	8	5,030	0.07	22	0.15	13	0.02	34	0.06	458	0.03	441	0.07	440	TO	TO
	urf3_155	10	26,468	0.34	22	0.75	17	0.08	40	0.25	458	0.13	442	0.38	441	2,902.70	881
	urf4_187	11	32,004	0.38	21	0.90	18	0.10	45	0.31	458	0.19	442	0.44	441	TO	TO
	urf5_158	9	10,276	0.43	22	0.29	14	0.03	32	0.12	458	0.06	441	0.15	440	TO	TO
	urf6_160	15	10,740	0.13	23	0.31	14	0.03	32	0.14	458	0.08	445	0.17	441	TO	TO
	xor5_254	6	7	0.02	21	0.00	12	0.00	29	0.01	457	0.02	440	0.00	439	0.12	11
RevLib-H	_443	261	1,701	TO	TO	71.67	154	TO	TO	TO	TO	TO	TO	TO	TO	ERR	ERR
	add64_184	193	385	0.20	204	0.03	14	0.10	118	0.10	460	0.08	545	0.06	446	ERR	ERR
	apex2_289	498	1,803	TO	TO	1,430.24	77	TO	TO	TO	TO	TO	TO	TO	TO	ERR	ERR
	callif_32_439	130	754	TO	TO	1.80	35	TO	TO	TO	TO	TO	TO	TO	TO	ERR	ERR
	cps_292	923	3,165	TO	TO	2,650.41	144	TO	TO	1,390.00	5,788	TO	TO	TO	TO	ERR	ERR
	cpu_alu_16bit_400	405	6,552	ERR	ERR	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	ERR	ERR
	cpu_control_unit_402	392	1,537	TO	TO	474.11	138	TO	TO	TO	TO	TO	TO	TO	TO	ERR	ERR
	cpu_register_32_405	328	1,978	0.46	214	0.09	15	0.42	195	0.62	469	0.71	668	0.34	457	ERR	ERR
	e64-bdd_295	195	516	1.98	239	2.50	14	2.03	127	0.66	477	0.54	614	1.99	496	ERR	ERR
	ex5p_296	206	736	7.61	283	12.15	21	3.57	133	1.03	489	1.15	691	6.42	548	ERR	ERR
	hwb9_304	170	774	33.00	663	13.58	20	12.09	115	3.75	559	4.98	1,105	22.10	570	ERR	ERR
	lu_326	299	831	TO	TO	7.72	33	TO	TO	TO	TO	TO	TO	TO	TO	ERR	ERR
	nestedif2_32_445	263	1,304	TO	TO	316.18	126	TO	TO	TO	TO	TO	TO	TO	TO	ERR	ERR
	pdc_307	619	2,319	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO	ERR	ERR
	spla_315	489	1,747	TO	TO	918.54	167	2,737.81	772	1,810.00	5,642	TO	TO	TO	TO	ERR	ERR
	varops_32_447	224	1,402	TO	TO	94.46	109	TO	TO	TO	TO	TO	TO	TO	TO	ERR	ERR

Appendix B

Submitted Paper

The following pages include a paper that was written on the basis of this thesis and, at the time of writing, submitted to ICCAD'24.

Accelerating Quantum Circuit Simulation with Symbolic Execution and Loop Summarization

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.

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 [20]. 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 [15], 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* [20], *period finding* [23], and *quantum counting* [10]. Current standards for describing quantum circuits, such as the OpenQASM 3.0 format [16], 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 allows symbolic simulation thanks to the use of *multi-terminal binary decision diagrams* (MTBDDs) [5, 17]. 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

We use $\mathbb{B} = \{0, 1\}$ to denote the Booleans and fix a set $\mathbb{X} = \{x_1, \ldots, x_n\}$ of Boolean variables with an implicit 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 $f : \mathbb{B}^n \to S$. If $S = \mathbb{B}$, then f is a *Boolean* function. We use ω to denote 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

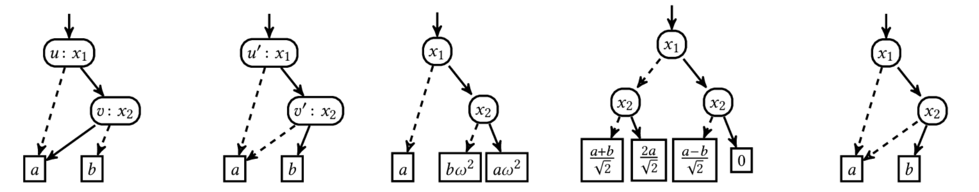
Given an arbitrary nonempty set S with finitely representable elements (in other words, a countable set), we define a *multi-terminal binary decision diagram* (MTBDD) [17][1] as a graph $G = (N, T, low, high, root, var)$ where N is the set of *internal nodes*, $T \subseteq S$ is the set of *leaf nodes* ($T \cap N = \emptyset$, $T \neq \emptyset$), $low, high : N \to (N \cup T)$ are the *low*- and *high*-successor edges, $root \in N \cup T$ is the *root* node, and $var : N \to \mathbb{X}$ 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 $u \in N$ such that $low(u) = high(u)$.

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

[1] sometimes also known as an *algebraic decision diagram* (ADD) [5]; both are generalizations of *reduced ordered binary decision diagrams* (ROBDDs or just BDDs) [11]



(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 CX_2^1 to q**

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

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

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 pseudo-Boolean function with the MTBDD that represents it and also we use a node r to denote the MTBDD rooted from r and vice versa.

The following standard MTBDD operations will be used in the paper. The $\mathtt{apply}(f_1, f_2, op_2)$ 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 $\mathtt{monadic\_apply}(f, op)$ 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 $\mathtt{spawn}(l, h, x)$ function that works as follows: (i) if $l = h$, then the result is l (this enforces the (reducedness) invariant), otherwise (ii) the result is the unique node n such that $low(n) = l$, $high(n) = h$, and $var(n) = x$ (in practice, a cache is used; this enforces the (canonicity) invariant).

2.2 Quantum Computing Fundamentals

Quantum computers are programmed through *quantum gates*, and every time a gate is applied, the global *quantum state* is updated. A *quantum circuit* is a series of gate applications, combined with programming constructs like *loop* or *hierarchical gate definitions* that allow for a more concise presentation [16].

Quantum states: In a traditional computer system with n bits, a state is represented by n Boolean values. In the quantum world, such states are referred to as *computational basis states*. For example, 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 computational 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 that maps each computational basis state in $\mathbb{B}^n$ to a complex amplitude in $\mathbb{C}$ 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: There are two main types of quantum gates that are used in state-of-the-art quantum computers: *single-qubit gates* and *controlled gates*. Our work supports all commonly used gates except the arbitrary rotation single-qubit gate due to the use a precise complex number representation proposed in [37] (cf. Sec. 5). We note that the set of supported gates is much larger than what is required to achieve (approximate) universal quantum computation. One can achieve this with, e.g., either (i) Clifford gates (H, S, and CNOT) and T [8] or (ii) Toffoli and H [3].

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

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

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

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

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 $\binom{l}{h}$ produces the state $\mathrm{X} \cdot \binom{l}{h} = \binom{h}{l}$. In the case of an MTBDD-based representation of $\binom{l}{h}$, 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).

Behaviours of Z, S, and T gates are similar to each other. In particular, applying the gates to $\binom{l}{h}$ produces the states $\mathrm{Z} \cdot \binom{l}{h} = \binom{l}{-h}$, $\mathrm{S} \cdot \binom{l}{h} = \binom{l}{i \cdot h}$, and $\mathrm{T} \cdot \binom{l}{h} = \binom{l}{\omega \cdot h}$, 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 challenging 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 $\binom{l}{h}$ to the state $H \cdot \binom{l}{h} = \frac{1}{\sqrt{2}} \cdot \binom{l+h}{l-h}$. 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 (in this particular case, the result is the same as in Figure 1b).

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 can be seen as a modification of a standard recursive `monadic_apply` function. In particular, it performs a depth-first search (Line 5) until it reaches a node with x_t, in which case it will perform the semantic of the gate on the successors. The semantic differs based on 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 virtually materializing the missing node (with *low* and *high* successors being the same), cf. Line 9. For example, 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 the `monadic_apply` function 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 to the state q, one step would be computing `monadic_apply`$(v, \lambda x(\omega^2 \cdot x))$ and connecting the result to the *high*-successor 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*-successor with $-\omega^2$ and sets it as the new *low*-successor, and (2) it multiplies the *low*-successor with ω^2 and sets it as the new *high*-successor.

For each node at level i, applying the H, $R_X(\frac{\pi}{2})$, or $R_Y(\frac{\pi}{2})$ 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.

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      $l \leftarrow low(node); h \leftarrow high(node); x_i \leftarrow var(node)$;
4      if $i < t$ then
5          $l_{new} \leftarrow$ recurse($l$); $h_{new} \leftarrow$ recurse($h$);
6          return spawn($l_{new}, h_{new}, x_i$);
7      else // $i \geq t$ or a leaf
8          if $i = t$ then $l' \leftarrow l; h' \leftarrow h$;
9          else $l' \leftarrow h' \leftarrow$ node;
10         if $U = X$ then return spawn($h', l', x_t$) ;
11         if $U \in \{T, S, Z\}$ then
12             if $U = T$ then $c \leftarrow \omega$;
13             if $U = S$ then $c \leftarrow \omega^2$;
14             if $U = Z$ then $c \leftarrow -1$;
15             $h_{new} \leftarrow$monadic_apply($h', \lambda x(c \cdot x)$);
16             return spawn($l', h_{new}, x_t$);
17         if $U = Y$ then
18             $l_{new} \leftarrow$monadic_apply($h', \lambda x(-\omega^2 \cdot x)$);
19             $h_{new} \leftarrow$monadic_apply($l', \lambda x(\omega^2 \cdot x)$);
20             return spawn($l_{new}, h_{new}, x_t$);
21         if $U = H$ then
22             $l_{new} \leftarrow$apply($l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x+y))$);
23             $h_{new} \leftarrow$apply($l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x-y))$);
24             return spawn($l_{new}, h_{new}, x_t$);
25         if $U = R_X(\frac{\pi}{2})$ then
26             $l_{new} \leftarrow$apply($l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x - \omega^2 \cdot y))$);
27             $h_{new} \leftarrow$apply($l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (y - \omega^2 \cdot x))$);
28             return spawn($l_{new}, h_{new}, x_t$);
29         if $U = R_Y(\frac{\pi}{2})$ then
30             $l_{new} \leftarrow$apply($l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x-y))$);
31             $h_{new} \leftarrow$apply($l', h', \lambda x, y(\frac{1}{\sqrt{2}} \cdot (x+y))$);
32             return spawn($l_{new}, h_{new}, x_t$);
```

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

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 $\mathrm{CU}_t^c(M_q)$
1  $M_l \leftarrow$ recurse($root$, L);
2  $M_h \leftarrow$ recurse($\mathrm{U}_t(M_q)$, H);
3  return apply($M_l, M_h, \lambda x, y(x + y)$);

4  Function recurse(node, dir)
5      $l \leftarrow low$(node); $h \leftarrow high$(node); $x_i \leftarrow var$(node);
6      if $i < c$ then
7          $l_{new} \leftarrow$ recurse($l$, $dir$); $h_{new} \leftarrow$ recurse($h$, $dir$);
8          return spawn($l_{new}$, $h_{new}$, $x_i$);
9      else // $i \geq c$ or a leaf
10         if $i = c$ then $l' \leftarrow l$; $h' \leftarrow h$ ;
11         else $l' \leftarrow h' \leftarrow$ node;
12         if dir = L then return spawn($l'$, 0, $x_c$) ;
13         else return spawn(0, $h'$, $x_c$) ;
```

using the apply function (Line 3), which will, effectively, combine the two MTBDDs together (one operand of the plus will always be 0). We note that the *Toffoli* gate can be obtained by using the CNOT gate for U. We also note that a specialized more efficient version of the algorithm for phase gates (such as Z, S, and T) can be used, which we omit here because of space constraints.

Memoization. In order to avoid redundant computation, calls to the recurse functions in Algorithms 1 and 2 should be memoized similarly as in standard implementations of apply for (MT)BDDs [11].

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[2], such as various *amplitude amplification* algorithms [9], with the most famous one being Grover's unstructured search [20]. 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).

Intuitively, the optimization works as follows. Consider a circuit with the following loop (in the OpenQASM 3.0 format [16]):

```
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 an MTBDD M_q,

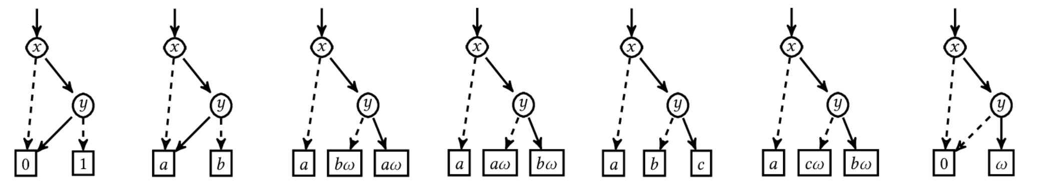
Algorithm 3: Loop summarization

```
   Input: An MTBDD $M_q$, a loop body $C$
   Output: An MTBDD $M_\alpha$ 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 ($M_\alpha, \tau$);

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

16 Function refine(lhs, rhs)
       Data: $\tau\colon \mathbb{S} \rightharpoonup \mathbb{T}_{\mathbb{S}}, \sigma\colon \mathbb{S} \rightharpoonup \mathbb{S}, \alpha\colon \mathbb{C} \rightharpoonup \mathbb{S}$
17     if $\tau$(lhs) $= \bot$ then
18         $\tau \leftarrow \tau \cup \{$lhs $\mapsto$ rhs$\}$;
19     else if $\nvDash \tau$(lhs) = rhs then
20         if $\sigma$(lhs) $= \bot$ then
21             let $s_{new} \in \mathbb{S} \setminus \mathrm{rng}(\alpha)$ be a fresh symbolic var.;
22             $\sigma \leftarrow \sigma \cup \{$lhs $\mapsto s_{new}\}$;
23         return $\sigma$(lhs);
24     return lhs;
```

it will first create an MTBDD M_α with leaves containing symbolic variables (from a set $\mathbb{S}$, which is some infinite set of symbolic names). Then, it will run the circuit C of the loop body with M_α as its input, with operations being done symbolically, i.e., instead of a single number, 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 present in M_q. The assumption is that computational bases with the same value will behave similarly in the algorithm. This does not need to hold, so after M'_α is 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

[2]In particular, 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; the technique can be extended to non-unitary loops and nested loops as well.



(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.

variables in the **Data** declaration of f (passed by reference). Given a (partial) function f of the type $f : X \rightharpoonup Y$, we use $\mathrm{rng}(f)$ to denote the *range* of f, i.e., the set $\{y \in Y \mid \exists x \in X : f(x) = y\}$. 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 toy circuit in Figure 3. The circuit starts in the state q with the qubit x set to 1 and qubit y set to 0. Then, it performs K executions of the loop body C. In each execution of the loop body, first, the T gate gets 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 demonstrated 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. □

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

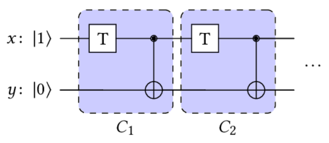


Figure 3: An example circuit for loop summarization

perform the initial abstraction of M_q, obtaining an MTBDD $M_\alpha^{refined}$ 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 Tool publicly available on GitHub. Tool is written in C and uses the Sylvan library [31] for handling MTBDDs (we, however, do not use its multi-threading features) and the GNU GMP library [1] for handling integers of arbitrary length. We use two configurations of Tool: with (Tool$_{loop}$) and without (Tool$_{base}$) loop summarization.

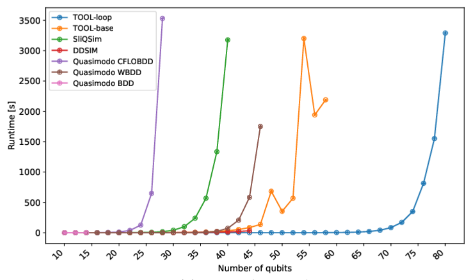
To achieve accuracy, we represent complex numbers algebraically as proposed in [37] and first realized in [30] (used also later in [12, 13]). The algebraic representation is given by the form

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

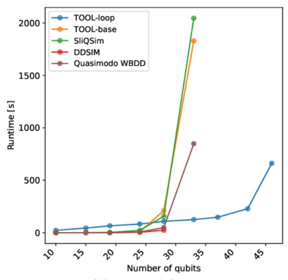
where a, b, c, d, and k are integers. A complex number is then represented by a five-tuple (a, b, c, d, k). Although it only represents a countable subset of $\mathbb{C}$, it can approximate any complex number up to a specified precision and suffices to support a set of quantum gates for universal quantum 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 $(\frac{1}{\sqrt{2}})^k(-d + a\omega + b\omega^2 + c\,\omega^3)$.

6 EXPERIMENTAL RESULTS

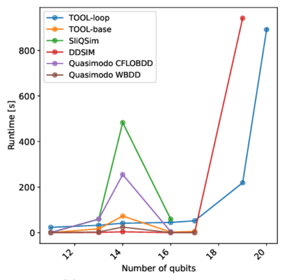
Simulators. We compared the performance of Tool against the following state-of-the-art quantum circuit simulators: SliQSim [30], Quasimodo [28], DDSIM [38] (v1.21.0), and Quokka# [24]. For



(a) Grover's search



(b) Period finding



(c) Quantum counting

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

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 Tool 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 [20] (with a single solution), quantum counting [10], and period finding [23], the last two without the final inverse quantum Fourier transform (QFT)[3]. 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. [10]), and MT denotes the number of randomly generated multi-control Toffoli gates in the oracle. We always set $SR = \lfloor \frac{FR}{2} \rfloor$ 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 [7] (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 [4] (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).

[3]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.

The experiments measured the time it took for the final quantum state to be obtained in the respective representation with the exception of Quokka#, where we measured the time to obtain the probability of the first qubit being zero (Quokka# does not compute the representation of the whole quantum state). The benchmarks did not contain measurements.

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 Tool$_{loop}$ and Tool$_{base}$, which differ only in loop summarization. The results show that in all three algorithms, Tool$_{loop}$ scales much better than Tool$_{base}$—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 that we can expect a similar behaviour if loop 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.*

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 (Tool or SliQSim).

				Tool$_{loop}$		Tool$_{base}$		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

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), Tool$_{loop}$ 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 Tool$_{base}$, 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 Tool$_{loop}$ can scale up to 46 qubits, while the second best ones, Quas[WBDD] and Tool$_{base}$, can scale only to 33 qubits. Let us note the size of the largest period finding circuit that Tool$_{loop}$ 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, Tool$_{loop}$ 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 Tool$_{loop}$, since it would be the same as Tool$_{base}$). Due to space limitations, we present only selected results. We chose circuits that took over one second to finish for three better-performing tools Tool$_{base}$, SliQSim, and DDSIM. However, RevLib-H circuits were challenging for most tools, except for SliQSim which solved 13 cases. Both Tool$_{base}$ 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. In particular, Quas[CFLOBDD] and Quas[BDD] failed on some circuits from Random (because they did not support the S gate) and also (together with Quokka#) on some circuits in RevLib-H (because they do not support the multi-control Toffoli).

The results show that Tool$_{base}$ is competitive to other simulators and in many cases, especially for the particularly challenging benchmarks from Feynman, is the best available accurate simulator. For the Loops benchmark, as mentioned previously, Tool$_{base}$ 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 the other accurate simulator 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 [22], a QMDD variant, called *tensor decision diagrams* (TDDs), is proposed to support contraction operation and allows tensor-network-like quantum circuit simulation. The TDD performance is comparable to DDSIM [22].

SliQSim [30] exploits the standard *reduced ordered binary decision diagrams* (ROBDDs, or just BDDs) [11] to represent quantum states exactly with an algebraic number system and achieves precise quantum operations through Boolean formula manipulation. Note that similarly to Tool, the supported quantum gate set of SliQSim, though universal, is restricted to those algebraically representable.

The paper [13] 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

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 (Tool or SliQSim). We do not mark Quokka# as the fastest because it does not compute the quantum state representation.

				Tool$_{base}$		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

in the same structure. We took inspiration from their extension to symbolic amplitudes in [12] to develop our symbolic execution.

SymQV [6] encodes quantum circuit verification problems into SMT with the theory of real numbers, using variables in trigonometric functions, e.g., sin x, 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 [14], where an extension of array theory, named *the theory of cartesian arrays (CaAL)*, was proposed and used to encode quantum gates. However, empirical results suggest that both methods are effective for small quantum circuits only.

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.* [21] proposed *symbolic tensor decision diagrams* (symTDDs) for symbolically executing and representing quantum circuits and quantum states. However, in quantum circuit simulation, parameters are typically predetermined, rendering the utility of this approach mainly for parameterized quantum circuit equivalence checking rather than simulation.

Quokka# [24] extended the standard stabilizer formalism [18] 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) and variational quantum eigensolver (VQE) circuits.

Although Clifford circuits should be efficiently simulatable according to the Gottesman–Knill theorem [19], 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 QMDD 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 technique by integrating it with other data structures for quantum state simulation. Moreover, we wish to look at the problem of automatically generalizing a computed summary into a *closed-form* formula (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, potentially, use the technique also in the verification framework of [13].

REFERENCES

[1] 2022. GMP: The GNU Multiple Precision Arithmetic Library. https://gmplib.org/
[2] 2024. The AutoQ repository. https://github.com/alan23273850/AutoQ/
[3] Dorit Aharonov. 2003. A Simple Proof that Toffoli and Hadamard are Quantum Universal. https://doi.org/10.48550/arxiv.quant-ph/0301040
[4] 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
[5] 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
[6] 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
[7] 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
[8] P. Oscar Boykin, Tal Mor, Matthew Pulver, Vwani P. Roychowdhury, and Farrokh Vatan. 2000. A new universal and fault-tolerant quantum basis. *Inf. Process. Lett.* 75, 3 (2000), 101–107. https://doi.org/10.1016/S0020-0190(00)00084-3
[9] 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
[10] 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
[11] 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
[12] Yu-Fang Chen, Kai-Min Chung, Ondřej 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
[13] 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
[14] 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.
[15] 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
[16] 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
[17] 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
[18] Daniel Gottesman. 1997. *Stabilizer codes and quantum error correction.* Ph.D. Dissertation. California Institute of Technology.
[19] Daniel Gottesman. 1998. The Heisenberg representation of quantum computers. *arXiv preprint quant-ph/9807006* (1998).
[20] 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
[21] 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
[22] 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
[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 *CAV'24 (to appear)*. https://arxiv.org/abs/2403.07197
[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

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009

Appendix C

Contents of the Included Storage Media

`/`	
`medusa/`	`MEDUSA` implementation
`benchmarks/`	All benchmark circuits
`loops/`	Loops circuits
`straightline/`	StraightLine circuits
`doc/`	Doxygen HTML documentation
`src/`	Source files
`Makefile`	For build purposes
`README.md`	Build and usage instructions
`MEDUSA`	Executable
`thesis-src/`	LaTeX source files of the thesis
`thesis.pdf`	Digital version of the thesis