Institut für Architektur von Anwendungssystemen

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Integration von Quantensimulatoren in QHAna

Sevim Davul

Studiengang:	Software Engineering
Prüfer/in:	Prof. Dr. Dr. h. c. Frank Leymann
Betreuer/in:	Dip.-Ing. Alexander Mandl, Fabian Bühler, M.Sc.
Beginn am:	10. Mai 2023
Beendet am:	10. November 2023

Kurzfassung

Quantencomputer nutzen Prinzipien der Quantenphysik wie Superposition und Verschränkung, um Quantenalgorithmen zu implementieren. Diese Algorithmen können systematisch auf klassischen Computern mittels der QHAna-Plattform simuliert werden, um sie zu testen und ihre Eigenschaften zu untersuchen. Dieser Simulationsprozess basiert auf den Simulatoren Qiskit-Bibliothek. Jedoch gibt es verschiedene andere Implementierungen von Quantencomputer-Simulatoren, die sich in ihrem Funktionsumfang und in ihrer Performance unterscheiden. Um die Simulatoren zu vergleichen werden in dieser Arbeit verschiedene Quantensimulatoren mit Python-Unterstützung systematisch anhand ihres Funktionsumfangs untersucht. Nach einer detaillierten Analyse werden zunächst Simulatoren ausgewählt, die mit OpenQASM kompatibel sind. Diese werden dann als Plugins schrittweise in das QHAna-System integriert. Um die Auswahl für den Nutzer zu erleichtern, erfolgt ein Vergleich der Simulatoren, deren Ergebnisse anschließend in einer Tabelle präsentiert werden. Mithilfe der integrierten Plugins wird schließlich die Leistung der Simulatoren bei der Ausführung von OpenQASM Programmen experimentell untersucht.

Danksagung

Herzlichen Dank an alle, die mich während des Studiums und bei der Erstellung dieser Bachelorarbeit motiviert haben.

Mein Dank gilt insbesondere meinen Betreuern Alexander Mandl und Fabian Bühler, die mich mit hilfreichen Rückmeldungen unterstützt haben.

Ein spezieller Dank geht an meinen Professor Frank Leymann, der mich während des Studiums inspiriert hat.

Mein herzlicher Dank gilt meiner Familie, meinen Freunden und Yildiz Senturk für ihre Unterstützung und Motivation.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Verzeichnis der Codebeispiele

Abkürzungsverzeichnis

CNOT Controlled-NOT. 9, 23

MQT Munich Quantum Toolkit. 34, 35, 38, 40

MUSE MUster Suchen und Erkennen. 30

NISQ Noisy Intermediate-Scale Quantum. 45

OpenQASM Open Quantum Assembly Language. 11, 13, 19

OQC Oxford Quantum Circuits. 48

QASM Quantum Assembly Language. 24

QDK Quantum Development Kit. 47

QHAna Quantum Humanities Analysis Tool. 1, 9, 19

TUM Technical University of Munich. 38

1 Einleitung

Quantencomputing bietet das Potenzial, viele Probleme effizienter als klassische Computer zu lösen [VBLW21]. Dabei verwenden Quantencomputer bestimmte Prinzipien der Quantenphysik wie Superposition und Verschränkung, wodurch sie in der Lage sein könnten, Aufgaben schneller, präziser und mit geringerem Energieaufwand als herkömmliche Computer auszuführen [WBLS21] [WBBL]. Dies wird insbesondere bei Aufgaben wie der Faktorisierung ganzer Zahlen mittels Shors Algorithmus [Sho94] oder der Datenbanksuche durch Grovers Suche [Gro96] deutlich, bei denen Quantencomputer eine erhebliche Beschleunigung im Vergleich zu klassischen Computern zeigen können [Des23].

Quantencomputing ist ein Forschungsfeld, das sich kontinuierlich weiterentwickelt und geeignete Werkzeuge erfordert. Allerdings sind echte Quantencomputer aufgrund ihrer hohen Kosten und Seltenheit noch nicht weit verbreitet [Des23]. Aus diesem Grund stützt sich ein Großteil der Forschung im Bereich des Quantencomputings auf die Simulation von Quantencomputing auf klassischen Computern [Des23]. Simulatoren erlauben es, die Algorithmen wie Shor und Grover zu untersuchen, die auf Quantencomputern ausgeführt werden [Kom20].

QHAna ist eines der Werkzeuge, die es ermöglichen, Quantenalgorithmen zu simulieren. Die Simulationen auf dieser Plattform werden über eigene Plugins realisiert. Zurzeit ist allerdings nur ein Plugin für Qiskit verfügbar. Es wäre wünschenswert, dass auch andere Simulatoren über QHAna genutzt werden können. Daher ist es ein Ziel dieser Arbeit, weitere Simulatoren als Plugins in QHAna zu integrieren, um die Auswahl und Flexibilität für die Benutzer zu erhöhen. Dies ermöglicht es Personen ohne vertiefte Kenntnisse im Bereich des Quantencomputings, die Vorteile von Quantenalgorithmen zu erkennen und grundlegendes Wissen in den Quantum Humanities zu sammeln [Bar22].

Quantenalgorithmen können in spezifischen Sprachen wie Open Quantum Assembly Language (OpenQASM) beschrieben werden, die universelle physikalische Quantenschaltkreise repräsentieren können [CBSG17]. Obwohl es mehrere solcher Sprachen gibt, fokussiert sich diese Arbeit auf OpenQASM. Dies liegt daran, dass OpenQASM von den meisten Quantencomputersimulatoren unterstützt wird und somit eine breite Interoperabilität zwischen verschiedenen Plattformen ermöglicht.

Die Simulatoren bieten nicht alle dieselben Funktionen und es ist nicht garantiert, dass ein bestimmter Simulator für jeden Anwendungsfall geeignet ist oder in QHAna integriert werden kann. Daher ist eine vergleichende Bewertung der Simulatoren notwendig. Diese Arbeit zielt darauf ab, eine systematische Untersuchung von Quantensimulatoren durchzuführen, die das OpenQASM 2.0-Format unterstützen. Nach einer detaillierten Analyse der Simulatoren werden geeignete ausgewählt und als Plugins in das QHAna-System integriert. Zudem wird ein Vergleich der ausgewählten Simulatoren anhand unterschiedlicher Kriterien in Form einer Tabelle vorgestellt. Mit der Integration dieser Simulatoren in QHAna werden verschiedene experimentelle Untersuchungen durchgeführt.

Die Leistungsvergleiche der Simulatoren, die auf die Ergebnisse dieser Studien basieren, werden in Diagrammen dargestellt. Diese vergleichenden Analysen erleichtern es den Nutzern, zu entscheiden, welcher Simulator je nach Anwendungsfall im QHAna-System ausgewählt werden sollte.

Die Struktur dieser Arbeit ist wie folgt: In Kapitel 2 werden die Grundlagen des Quantencomputings dargelegt, wobei speziell auf Konzepte wie Qubits, Quantenschaltkreise und die OpenQASM-Sprache eingegangen wird. Außerdem wird auch QHAna ausführlicher vorgestellt. Im dritten Kapitel wird das Vorgehen der Arbeit und die Auswahl der Quantensimulatoren vorgestellt. Eine detaillierte Untersuchung verschiedener Quantensimulatoren wie Qiskit Aer, CirqSimulator, MQT-DDSIM, Pytket_QulacsBackend, Pennylane default.qubit, Amazon Braket Local Simulator findet im vierten Kapitel statt und die Ergebnisse werden in einem tabellarischen Vergleich dargestellt. Kapitel 5 beschreibt zunächst den Integrationsprozess der Simulatoren als Plugins in QHAna. Anschließend befasst es sich mit der Analyse und dem Vergleich der Leistung dieser Simulatoren. Nach der Diskussion der Ergebnisse werden im folgenden Kapitel relevante Forschungsarbeiten vorgestellt, gefolgt von einer Zusammenfassung.

2 Grundlagen

In diesem Kapitel werden essenzielle Konzepte des Quantencomputings vorgestellt, unter anderem werden Themen wie Qubits, Quantenschaltkreise, Quantenoperationen und Open Quantum Assembly Language (OpenQASM) behandelt. Zudem wird auf die Anwendung spezifischer Tools wie das Quantum Humanities Analysis Tool (QHAna) und die Relevanz von Quantensimulatoren eingegangen.

2.1 Quantencomputing

Quantencomputing entwickelt sich täglich weiter und bietet die Möglichkeit, im Vergleich zu klassischen Computern viele Probleme aus verschiedene Bereichen effizienter oder mit höherer Präzision zu lösen, zum Beispiel Optimierung oder maschinelles Lernen [VBL+21] [SHM+23]. Allerdings kann der Fernzugriff auf diese Computer aufgrund der hohen Nachfrage herausfordernd sein [JP22]. In den letzten Jahren ist die Anzahl der Quantenhardware-Anbieter, darunter Unternehmen wie IBM, Rigetti, D-Wave, Google und Microsoft, gestiegen [VBLW21]. Zudem wächst das Interesse in diesem Bereich auch in der Forschung. Einige Institutionen haben die Technologie durch Cloud-Zugang zu Quantencomputern zugänglicher gemacht, wobei insbesondere Cloud-Service-Anbieter wie Amazon Web Services (AWS) durch die Integration von Kapazitäten zum Ausführen von Quantenalgorithmen einen bedeutenden Schritt in diesem Bereich unternommen haben [AJLT22].

Quantencomputing stellt ein neues Paradigma dar, welches auf den fundamentalen Prinzipien der Quantenmechanik aufbaut und dabei zentrale Konzepte wie Superposition und Verschränkung nutzt [SHM+23]. In der Quantenmechanik ermöglicht das Prinzip der Superposition die gleichzeitige Repräsentation mehrerer klassischer Zustände durch einen Quantenzustand; parallel dazu ist die Quantenverschränkung ein Effekt, bei dem Veränderungen im Zustand eines Qubits sofort Veränderungen im Zustand eines anderen auslösen, auch wenn sie räumlich weit voneinander entfernt sind [JP22].

2.1.1 Quantenbits

Im Bereich des klassischen Computings wird der Zustand eines Systems durch Bits dargestellt, die entweder den Wert 0 oder 1 haben können. Im Gegensatz dazu wird in einem Quantencomputer der Zustand durch sogenannte Quantenbits oder kurz Qubits, beschrieben; diese können die Basiszustände $|0\rangle$ oder $|1\rangle$ in der Dirac-Notation [DiV00] annehmen [GFH+23]. Diese Qubits sind das elementarste Konzept im Quantencomputing und ermöglichen eine Superposition von

Zuständen, wobei der Wert eines Qubits $|x\rangle$ eine beliebige Kombination der Werte 0 und 1 sein kann [BLF+21].

Der Zustand eines Qubits ist durch einen zweidimensionalen Vektor $|x\rangle$ charakterisiert [WBLS21]:

$$|x\rangle = \alpha|0\rangle + \beta|1\rangle.$$

Hierbei sind α und β komplexen Zahlen ($\alpha, \beta \in \mathbb{C}$) und die Summe ihrer quadrierten Beträge ist gleich eins [WBLS21]:

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

Diese repräsentieren lineare Kombinationen oder Superpositionen der Zustände $|0\rangle$ und $|1\rangle$. Diese Beschreibung verwendet die Dirac-Notation, wobei die Basiszustände wie folgt dargestellt werden [WBLS21]:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \text{ und } |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Die Menge $\{|0\rangle, |1\rangle\}$ bildet eine Basis des zweidimensionalen Vektorraums des Qubits. Die komplexen Zahlen α und β sind Amplituden des Quantensystems und bestimmen die Wahrscheinlichkeit der Messergebnisse [WBLS21]: Mit einer Wahrscheinlichkeit von $|\alpha|^2$ ergibt eine Messung $|0\rangle$ und mit einer Wahrscheinlichkeit von $|\beta|^2$ ergibt sie $|1\rangle$ [WBLS21]. Diese beiden Möglichkeiten müssen sich zu 1 addieren, da es keine anderen Messergebnisse gibt. Wenn $\alpha, \beta \neq 0$, befindet sich das Qubit in einer Superposition von $|0\rangle$ und $|1\rangle$ [WBLS21].

Zum Beispiel ist der sogenannte $|+\rangle$ Zustand, welcher eine gleiche Superposition von $|0\rangle$ und $|1\rangle$ darstellt, durch folgende Gleichung definiert [WBLS21] [Qis23h]:

$$|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle.$$

Die Wahrscheinlichkeit für beide Messergebnisse ($|0\rangle$ und $|1\rangle$) ist gleich und kann folgendermaßen berechnet werden:

$$\left|\frac{1}{\sqrt{2}}\right|^2 = 0.5.$$

Beim Messen des $|-\rangle$ Zustandes:

$$|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$$

ergeben sich genau die gleichen Wahrscheinlichkeiten für beide Resultate

$$\left|\frac{1}{\sqrt{2}}\right|^2 = 0.5 \quad \text{und} \quad \left|-\frac{1}{\sqrt{2}}\right|^2 = 0.5.$$

Dies impliziert, dass die $|+\rangle$ und $|-\rangle$ Zustände in Bezug auf Messungen in der Standardbasis identisch sind.

Ein Quantenregister, das aus n Qubits besteht, kann theoretisch gleichzeitig 2^n verschiedene klassische Zustände repräsentieren. In der Quantenmechanik sind diese Qubits jedoch oft verschränkt, sodass der Zustand eines Qubits von den Zuständen der anderen Qubits im Register abhängen kann [PMSF23].

Ein Beispiel für einen verschränkten Zustand ist [WBLS21]:

$$|x\rangle = \frac{1}{\sqrt{2}}|00\rangle + 0|01\rangle + 0|10\rangle + \frac{1}{\sqrt{2}}|11\rangle.$$

In Vektorform wird dieser Zustand als

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\1\end{bmatrix}$$

dargestellt. Bei einer Messung ergibt sich entweder das Ergebnis $|00\rangle$ oder $|11\rangle$, wobei beide eine Wahrscheinlichkeit von

$$\left|\frac{1}{\sqrt{2}}\right|^2 = \frac{1}{2}$$

aufweisen. Bei einer Messung des linken Qubits im obigen Zustand kann aufgrund der Verschränkung direkt auf den Zustand des rechten Qubits geschlossen werden.

2.1.2 Quantenoperationen

Quantencomputer verwenden Quantengatter um Qubits zu manipulieren. Dies ist analog zu den klassischen Logikgattern wie UND, ODER und NICHT, die auf Bits wirken [WBLS21]. Berechnungen werden häufig als Quantenschaltkreise dargestellt, die die Sequenz der Quantengatter visualisieren, die auf jedes Qubit angewendet werden; diese Gatter können auf ein oder mehrere Qubits gleichzeitig wirken [WBLS21]. Die Quantengatter werden durch unitäre Matrizen der Größe $2^n \times 2^n$ beschrieben, wobei n die Anzahl der vom Gatter verwendeten Qubits ist. Diese Matrizen werden schrittweise auf einen Anfangszustand angewendet, um den Algorithmus zu berechnen [Com22].

Eine Matrix, die komplexe Zahlen enthält und für die $UU^\dagger = U^\dagger U = I$ gilt, wird als unitär bezeichnet [Qis]. In diesem Zusammenhang symbolisiert I die Identitätsmatrix und $U^\dagger = \overline{U^T}$ ist als die konjugierte Transponierte von U. Dies impliziert, dass das Inverse einer unitären Matrix gleich ihrer konjugierten Transponierten ist (3) [Com22].

Tabelle 2.1 zeigt eine Auswahl wichtiger Ein-Qubit-Quantengatter und ihre entsprechenden Matrizen, während Tabelle 2.2 jene mit mehreren Qubits darstellt.

Die X oder Pauli-X Operation wird auch als Bit Flip oder NOT-Operation bezeichnet, weil sie diese Aktion auf Bits induziert [Qis]:

$$X|0\rangle = |1\rangle \text{ und } X|1\rangle = |0\rangle$$

Tabelle 2.1: Einige wichtige Ein-Qubit-Quantengatter [Com22]

Gattername	Matrix
I oder Identität	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
H oder Hadamard	$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
X oder NOT	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Y	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Z	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
S	$\begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$
T	$\begin{bmatrix} 1 & 0 \\ 0 & e^{\frac{\pi i}{4}} \end{bmatrix}$

Die Z-Operation, häufig als Pauli-Z-Gatter bezeichnet, induziert einen Phasenflip auf den Quantenzustand. Ihre Wirkung auf die Basiszustände kann wie folgt beschrieben werden [Qis]:

$$Z|0\rangle = |0\rangle \quad \text{und} \quad Z|1\rangle = -|1\rangle.$$

Die Hadamard-Operation ist eine weitere in der Quantencomputing häufig verwendete Operation und ist in Tabelle 2.1 dargestellt. Wie bereits oben erwähnt, ist die Hadamard-Operation essentiell für das Erzeugen von Superpositionen.

Phasenoperationen in der Quantencomputing werden durch die Matrix $P(\theta)$ repräsentiert, abhängig von der Wahl eines bestimmten reellen Wertes, θ. Zur Veranschaulichung dienen verschiedene Beispiele [Qis]. Die Identitätsmatrix $P(0) = I$ und die Matrix $S = P\left(\frac{\pi}{2}\right) = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$, die eine Phasenverschiebung von $\frac{\pi}{2}$ verursacht, sind zentrale Beispiele solcher Operationen. Ebenso relevant ist die Matrix $Z = P(\pi) = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$, die eine Phasenverschiebung von π vollzieht und somit das Vorzeichen des amplitudenkomplexen Teils des Qubits ändert [Qis].

Um die Unitarität einer Matrix zu überprüfen, kann als Beispiel die Matrix $S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}$ verwendet werden:

$$SS^\dagger = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & -i \end{bmatrix}$$

mit entsprechende Matrixmultiplikationen ergibt sich:

$$\begin{bmatrix} 1 \cdot 1 + 0 \cdot 0 & 1 \cdot 0 + 0 \cdot (-i) \\ 0 \cdot 1 + i \cdot 0 & 0 \cdot 0 + i \cdot (-i) \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I$$

In diesem Beispiel ist $S^\dagger$ das konjugierte Transponierte von S und das Ergebnis ist die Identitätsmatrix I, was die Unitarität von S bestätigt [Qis].

Tabelle 2.2: Einige wichtige Mehrqubit Quantengatter [Com22]

Gatternname	Matrix
CNOT oder kontrolliert-X	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$
SWAP	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$
Kontrollierte-z	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$
Toffoli	$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$

Das Toffoli-Gatter, auch als Controlled-Controlled-X Gatter bekannt, ist ein Drei-Qubit-Quantengatter. Seine Wirkung auf Basiszustände wird durch die Gleichung

$$|a\rangle|b\rangle|c\rangle \rightarrow |a\rangle|b\rangle|c \oplus (a \wedge b)\rangle$$

ausgedrückt, wobei $a, b, c \in \{0, 1\}$ und $\wedge$ den logischen UND-Operator repräsentiert [Com22]. Die Matrix, die das Toffoli-Gatter repräsentiert, wird in Tabelle 2.2 präsentiert.

Der Controlled-NOT (CNOT)-Operator führt eine wichtige Operation zwischen zwei Qubits durch, indem er eine kontrollierte Inversion ausführt [ZW19]. Seine Darstellung wird in Tabelle 2.2 dargestellt. Bei dieser Operation fungiert ein Qubit als Kontrollqubit. Das andere Qubit, auch als Zielqubit bezeichnet, wird komplementiert, wenn sich das Kontrollqubit im Basiszustand $|1\rangle$ befindet

[ZW19]. Ist das Kontrollqubit im Basiszustand $|0\rangle$, besteht die resultierende Matrix aus der 2×2 Identitätsmatrix [ZW19]. Befindet es sich im Zustand $|1\rangle$, wird die Matrix zu der einzelnen Qubit-Matrix X [ZW19]. Um die Wirkung einer Quantenoperation auf einen spezifischen Quantenzustand zu bestimmen, muss die zugehörige Matrix U mit dem entsprechenden Zustandsvektor multipliziert werden [ZW19].

2.2 Quantenschaltkreise: OpenQASM

Ein Quantenschaltkreis für n Qubits besteht aus einer Reihe von Quantengattern, die auf die Qubits angewendet werden [FM20] [NC10].

In einem Quantenschaltkreis, in dem mehrere Qubits vertikal in einem Quantenschaltkreisdiagramm dargestellt sind, fungiert für jedes Qubit eine horizontale Linie als Zeitachse, die von links nach rechts interpretiert wird [ZW19]. Auf dieser Linie weisen Boxen die spezifische Quantenoperation aus, die durchgeführt werden muss [ZW19].

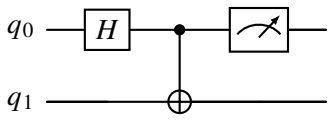


Abbildung 2.1: Ein Beispiel für einen Quantenschaltkreis.

Dieser Quantenschaltkreis enthält zwei Qubits, q_0 und q_1. Beide Qubits beginnen im Grundzustand $|0\rangle$, weshalb der Anfangszustand des Quantenschaltkreises $|\psi\rangle = |00\rangle$ ist [ZW19]. Zunächst wird q_0 einer Hadamard-Transformation unterzogen, die durch ein Kästchen mit dem Buchstaben H symbolisiert wird. Dieser Schritt bringt das Qubit in einen Superpositionszustand [ZW19]. Anschließend wird eine CNOT-Operation durchgeführt. Hierbei dient q_0 als Kontroll-Qubit, repräsentiert durch das Symbol $\bullet$ und q_1 als Ziel-Qubit, dargestellt durch das Symbol $\oplus$. Am Ende der Operationen wird das Qubit q_0 gemessen, wodurch seine Superposition in einen der beiden Basiszustände kollabiert [ZW19]. Das bedeutet, das Messergebnis versetzt das Qubit entweder in den Zustand 0 oder 1.

Um Quantenschaltkreise zu beschreiben, werden high-level Quantensprachen wie Scaffold, Quipper, Quil, ProjectQ, Quantenassemblysprachen wie OpenQASM 2.0 oder Quantenschaltkreisdiagramm verwendet [WMN].

2.2.1 OpenQASM

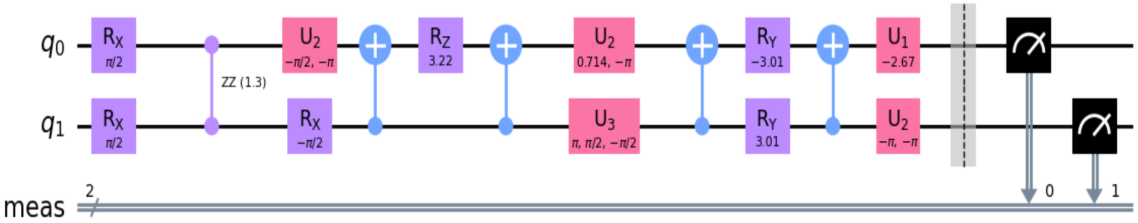
Wie bei klassischen Computern befindet sich der vom Compiler erstellte Code auf der Assembly-Ebene [KAF+]. Ein Assembler, Quantum Assembler (QASM) genannt, wird für diesen Zweck erweitert [KAF+]. Quantum Assembly Language (QASM) ist eine Quanten-Assembly-Sprache, die ursprünglich zur Darstellung von Quantenschaltkreisen entwickelt wurde [KAF+].

QASM unterstützt die meisten üblichen Ein-Qubit-, Zwei-Qubit- und Drei-Qubit-Gatter. Ein zentraler Bestandteil ist die Messung des Zustands des Qubits, welche ein klassisches Bit erzeugt [KAF+]. Das so erhaltene klassische Bit kann das Ergebnis von Quantenberechnungen darstellen oder beispielsweise in Fehlerkorrektur-Quantenschaltkreisen, dazu verwendet werden, um über binär gesteuerte Gatter weitere Quantenoperationen bedingt auszuführen [KAF+].

OpenQASM besitzt die Fähigkeit universelle physikalische Quantenschaltkreise zu repräsentieren. Die definierten Gattersets werden durch einen hierarchisch Mechanismus bestimmt, der neue unitäre Gatter unter Verwendung von eingebauten Gattern und zuvor definierten Subroutinen ermöglicht [CBSG17]. Mit dieser Methodik verwendet man das eingebaute Gatterset, um hardwareunterstützte Operationen über Standardheaderdateien zu definieren [CBSG17]. Der Subroutinenmechanismus ermöglicht eine begrenzte Wiederverwendung des Codes, indem komplexere Operationen hierarchisch definiert werden [CBSG17]. Zudem werden dieser Struktur Anweisungen hinzugefügt, die eine Quanten-Klassik-Schnittstelle modellieren, insbesondere Messung, Zustandsrücksetzung und grundlegendes klassisches Feedback [CBSG17].

```
// Benchmark was created by MQT Bench on 2023-06-29
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg meas[2];
rx(pi/2) q[0];
rx(pi/2) q[1];
rzz(1.304903297657757) q[0],q[1];
u2(-pi/2,-pi) q[0];
rx(-pi/2) q[1];
cx q[1],q[0];
rz(3.2220464314480877) q[0];
cx q[1],q[0];
u2(0.7144206217376547,-pi) q[0];
u3(pi,pi/2,-pi/2) q[1];
cx q[1],q[0];
ry(-3.0062717024781365) q[0];
ry(3.0062717024781365) q[1];
cx q[1],q[0];
u1(-2.6682620430456936) q[0];
u2(-pi,-pi) q[1];
barrier q[0],q[1];
measure q[0] -> meas[0];
measure q[1] -> meas[1];
```

(a) Ein zufälliger Quantenschaltkreis in OpenQASM (2.0) [QBW23].



(b) Ein mittels *Qiskit* generiertes Quantenschaltkreisbeispiel.

Abbildung 2.2: Die Quantenschaltkreis-Darstellung des oben genannten OpenQASM-Codes

Die Abbildung 2.2a, entnommen von [QBW23], stellt ein Beispiel für einen OpenQASM-Quantenschaltkreis dar und Abbildung 2.2b zeigt einen Quantenschaltkreis, der aus dem oben gegebenen OpenQASM-Code mittels der *Qiskit*-Bibliothek erstellt wurde.

In Tabelle 2.3 sind die Hauptanweisungen der OpenQASM Sprache (Version 2.0) zusammengefasst und beschrieben.

Tabelle 2.3: OpenQASM Sprachanweisungen (Version 2.0) [CBSG17]

Anweisung	Beschreibung
OPENQASM 2.0;	Bezeichnet eine Datei im Open QASM Format an
qreg name[size];	Deklariert ein benanntes Register von Qubits
creg name[size];	Deklariert ein benanntes Register von Bits
include "filename";	Öffnet und interpretiert eine andere Quelldatei
gate name(params) qargs { body }	Deklariert ein unitäres Gatter
opaque name(params) qargs;	Deklariert ein opakes Gatter
U(theta,phi,lambda) qubit|qreg;	Wendet eingebautes einzelnes Qubit Gatter an
CX qubit|qreg,qubit|qreg;	Wendet eingebautes CNOT Gatter an
measure qubit|qreg -> bit|creg;	Macht Messungen in Z-Basis
reset qubit|qreg;	Bereitet Qubit(s) in $\|0\rangle$ vor
gatename(params) qargs;	Wendet ein benutzerdefiniertes unitäres Gatter an
if(creg==int) qop;	Wendet bedingt eine Quantenoperation an
barrier qargs;	Verhindert Transformationen über diese Quellzeile

2.3 Quantensimulatoren

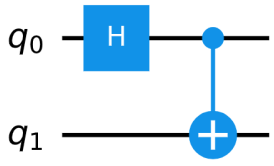
Quantenalgorithmen wie Shor's Factoring, Grover's Search und Quantum Approximate Optimization werden für die Entwicklung realer Anwendungen verwendet und Quantencomputer werden simuliert, um reale Probleme mit den Anwendungen zu lösen [Kom20]. Dabei spielt der Quantensimulator eine entscheidende Rolle: Er modelliert den Quantenschaltkreis und zeigt, wie Quantenanweisungen auf Quantengatter einwirken [Kom20].

Karafyllidis hat das grundlegende Vorgehen des Quantencomputersimulators in [Kar05] im Algorithmus 2.1 dargestellt. Der Simulator nimmt als Eingabe die Anzahl der Qubits im Quantenregister und deren Anfangszustand und führt die festgelegten Rechenschritte durch. Auf dem Quantenschaltkreismodell basierend, visualisiert er nicht nur die Quantenberechnungen und liefert eine grafische Darstellung derselben, sondern stellt auch die Wahrscheinlichkeiten von Messungen grafisch dar.

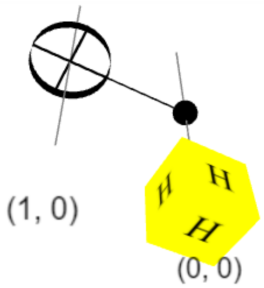
Im Bereich des Quantencomputings besteht eine zentrale Aufgabe darin, Informationen zu codieren und Quantenzustände zu messen [Kom20]. Bei dieser Codierung werden spezifische Quantenbits verwendet, die sich oft in einer Superposition befinden [Kom20]. Bei der Messung eines Quantenbits wird ein spezifischer Zustand ermittelt. Um die Verteilung der Messergebnisse zu bestimmen, wird wiederholt gemessen. Diese Messungen erzeugen Histogramme, die auf den beobachteten Häufigkeiten der Messergebnisse basieren [Kom20].

Algorithmus 2.1 Grundstruktur des Quantencomputersimulators [Kar05]

1: Start
2: Lese Qubit- und Rechenschritt-Anzahl.
3: Lese Anfangszustand des Registers.
4: Lese Quantengatter-Konfiguration für jeden Schritt.
5: Berechne Tensorprodukt des Anfangszustands.
6: **for** $i = 1$ bis festgelegte Anzahl von Schritten **do**
7: Berechne Tensorprodukt der Quantentormatrizen im i-ten Schritt.
8: Aktualisiere Zustand des Registers.
9: Berechne Phase des Zustands.
10: **end for**
11: Gebe Wahrscheinlichkeitsamplituden aus.
12: Gebe Endzustand des Registers aus.
13: Erzeuge Grafik.



(a) Quantenschaltkreis, der mit dem Qiskit-Framework erstellt wurde.

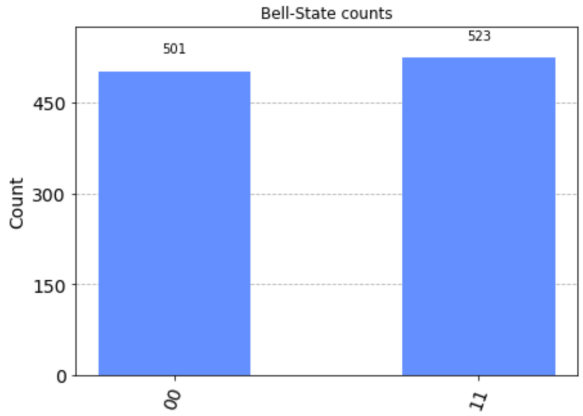


(b) 3D Quantenschaltkreis, der mit dem Cirq-Framework erstellt wurde.

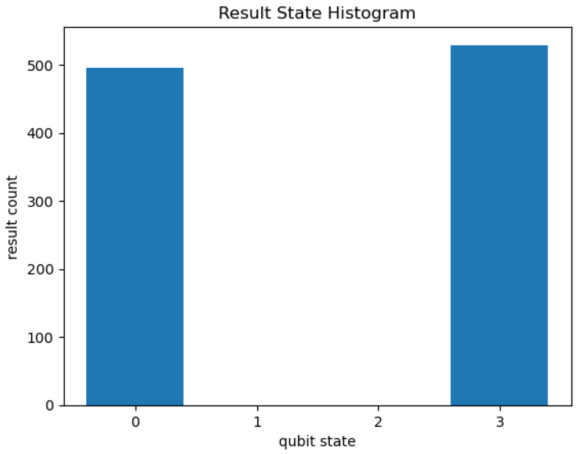
Abbildung 2.3: Quantenschaltkreise, die mit verschiedenen Bibliotheken erstellt wurden und ein Hadamard-Gatter und ein CNOT-Gatter enthalten.

Im Beispiel, das in Abbildung 2.4 dargestellt ist, werden zwei Histogramme präsentiert, die mithilfe verschiedener Quantensimulatoren erstellt wurden. Diese Diagramme zeigen die Ergebnisverteilung eines gegebenen Quantenschaltkreises. In diesem Quantenschaltkreis wurde auf dem ersten Qubit ein Hadamard-Gatter angewendet, wodurch dieses in eine Superposition gebracht wurde. Anschließend führte die Anwendung eines CNOT-Gatter zwischen den Qubits zu einem verschränkten Zustand. Das Experiment wurde mit 1024 Messungen durchgeführt und die Ergebnisse zeigten eine Verteilung auf die Zustände $|00\rangle$ und $|11\rangle$.

Beim Auslesen von Daten aus einem Quantencomputer erhalten wir eine Reihe von Bits, die als natürliche Zahlen dargestellt werden [LB20]. Wiederholte Durchführungen der gleichen Berechnung erzeugen eine Verteilung, in der jeder Wert mehrmals gemessen wird [LB20]. Die Ergebnisse können als Vektor dargestellt werden, in dem jede Position die Häufigkeit (counts) des entsprechenden Messwerts angibt [LB20].



(a) Ergebnis mit dem Aer-Simulator aus der Qiskit-Bibliothek.



(b) Ergebnis mit dem Cirq-Simulator. Hier entspricht die 0 dem Zustand $|00\rangle$ und die 3 dem Zustand $|11\rangle$.

Abbildung 2.4: Darstellung der Ergebnisverteilung als Histogramm.

Simulatoren ermöglichen es auch den Zustandsvektor am Ende einer Simulation zu extrahieren:

```
Statevector: Statevector([0.70710678+0.j, 0.+0.j, 0.+0.j, 0.70710678+0.j], dims=(2, 2)) // qiskit
state vector: [0.70710677+0.j 0. +0.j 0. +0.j 0.70710677+0.j] // cirq
```

Der oben genannte OpenQASM-Code enthält keine Messung. Wenn eine Messung im Code enthalten wäre, würde der Zustand kollabieren, was zu einem definierten Zustand führen würde, wie im folgenden Beispiel mit Qiskit gezeigt:

```
Statevector: Statevector([1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j], dims=(2, 2))
```

2.4 QHAna

Quantum Humanities, die Vision der Kombination von Quantencomputing und digitalen Geisteswissenschaften, stellt ein innovatives Forschungsfeld dar [BL20]. Im Kontext dieser Vision ist das Quantum Humanities Analyse Tool ein Werkzeugsatz für Maschinenlernmethoden, konzipiert sowohl für klassische als auch für Quantenhardware [Qua]. Es wurde für die Nutzung mit dem MUster Suchen und Erkennen (MUSE) Repository [BFL18] entworfen [Qua]. Dieses Projekt verfolgt das Ziel, eine Mustersprache für Kostüme in Filmen zu identifizieren [BFL18]. Zudem beinhaltet es ein Kleidungs- und Kostümrepository, welches die Erfassung, Speicherung und Analyse von Kleidung und Kostümen unterstützt [BFL18]. Hierbei liegt ein besonderer Fokus auf Kostümen in Filmen [BFL18].

QHAna hat das Ziel, die Identifikation von Mustern mittels Datenanalyse zu fördern [Bar22]. Durch den Einsatz sowohl klassischer als auch hybrider Quantenalgorithmen lässt sich ein Vergleich ziehen, um die Vorteile von Quantenalgorithmen zu beurteilen [Bar22]. Dieser Vergleich betont die potenziellen Vorteile des Quantencomputings für die Forschung der digitalen Geisteswissenschaften [Bar22]. Zudem ermöglicht QHAna die Integration von heterogenen Tools verschiedener Quantencomputing-Anbieter in einer einzigen Analysepipeline [Bar22]. Darüber hinaus bietet QHAna für alle Algorithmen eine grafische Oberfläche, wodurch keine Programmierkenntnisse vorausgesetzt werden und somit Personen ohne spezielle Vorkenntnisse ein erleichterter Einstieg in die Anwendungen der Quantum Humanities ermöglicht wird [Bar22].

QHAna wird in vier verschiedenen Repositories entwickelt [Qua23a]:

qhana-ui: Hierbei handelt es sich um die Benutzerschnittstelle von QHAna, die es dem Benutzer ermöglicht, *Experimente* zu kreieren, um Daten mittels vorhandener Algorithmen zu analysieren oder verfügbare Daten mit den Algorithmen zu prüfen.
qhana-backend: Das Backend ist verantwortlich für die Speicherung der Experimentdaten, die für die Benutzeroberflächen benötigt werden.
qhana-plugin-runner: Ein kompakter Wrapper für QHAna-Plugins, der nützliche gemeinsame Funktionen wie zuverlässige backround tasks bereitstellt. Obwohl die Entwicklung von Plugins auch ohne den Plugin-Runner möglich ist, kann dessen Nutzung die Entstehung neuer Plugins beschleunigen.
qhana-plugin-registry: Die Aufgabe der Registry besteht darin, alle aktuell verfügbaren Plugins aufzulisten und sie beispielsweise für die Benutzeroberfläche auffindbar zu machen.

In Abbildung 2.5 dargestellt, zeigt der Experiment Workspace der QHAna UI eine Liste der verfügbaren Plugins auf der linken Seite. Nutzer können das gewünschte Plugin aus dieser Liste auswählen und die Eingabeparameter entsprechend dem ausgewählten Plugin festlegen. Falls eine Datei als Eingabe benötigt wird, kann sie zunächst mit dem 'file-upload' Plugin hochgeladen werden. Nachdem die Eingabeparameter korrekt festgelegt wurden, kann das Experiment ausgeführt werden. Beispielsweise kann, wie in Abbildung 2.5 dargestellt, nach dem Hochladen einer QASM-Datei über das 'file-upload' Plugin diese im Qiskit Simulator Plugin ausgewählt werden. Anschließend kann das Experiment ausgeführt werden und auf der geöffneten Seite wie in Abbildung 2.6 kann auf Informationen wie Eingaben und Ergebnisse zugegriffen werden.

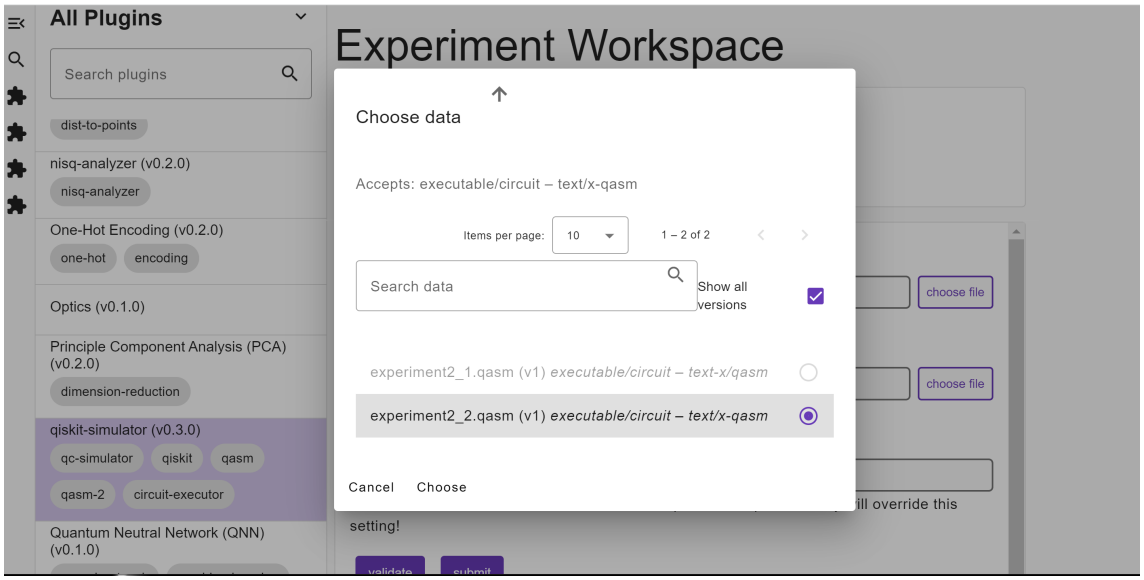


Abbildung 2.5: QHAna UI-Experiment-Workspace Ansicht.

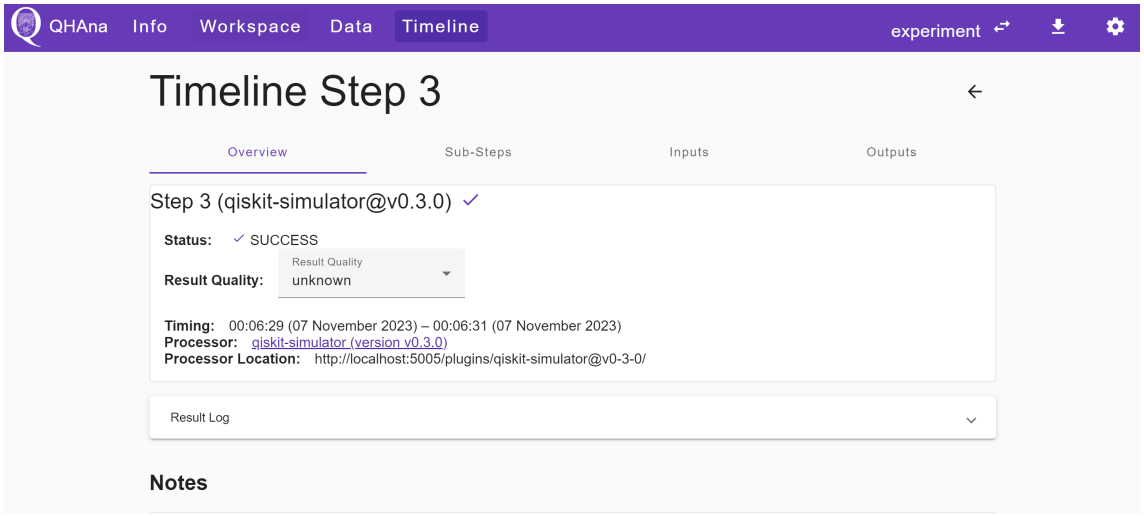


Abbildung 2.6: QHAna UI-Experiment-Timeline Ansicht.

3 Methodik

In diesem Kapitel wird das in dieser Arbeit verfolgte Vorgehen, die eingesetzten Techniken und die Auswahlphase der Simulatoren detailliert beschrieben. Die verwendeten Simulatoren werden anhand definierter Kriterien verglichen. Die für eine detaillierte Bewertung der verwendeten Simulatoren erforderlichen Eigenschaften werden in diesem Abschnitt definiert.

Das Hauptziel dieser Arbeit besteht darin, unterschiedliche Simulatoren zu untersuchen und sie in das QHAna-System zu integrieren. In diesem Kontext wurden insbesondere in Python entwickelte Quantenbibliotheken wie Cirq, QuTIP, ProjectQ, Strawberry Fields, Munich Quantum Toolkit, PennyLane, Pytket, Amazon Braket, Qulacs und Pyquil berücksichtigt. Während der Forschungsphase wurden die in diesen Bibliotheken enthaltenen Simulatoren, die entweder eigenständig sind oder mit anderen Systemen kombiniert werden können, systematisch analysiert. Unter Berücksichtigung der offiziellen Dokumentationen zu diesen Simulatoren wurden sie einem lokalen Testprozess unterzogen. Anschließend wurde entschieden, ob sie in das QHAna-System integriert werden sollten oder nicht. Die ausgewählten Simulatoren wurden dann schrittweise in das QHAna eingebunden.

3.1 Auswahl der Quantensimulatoren

Ein zentrales Kriterium bei der Auswahl der Quantensimulatoren ist, dass das zu verwendende Framework OpenQASM unterstützt wird und kompatibel mit dem Simulator ist. Wie bereits in den Grundlagen erläutert, können Quantenschaltkreise einfach im OpenQASM-Format dargestellt werden und für ihre Simulation sind geeignete Tools erforderlich. Dies bietet dem Benutzer den Vorteil, denselben Input ohne Modifikationen in verschiedenen Simulatoren zu verwenden und die resultierenden Ausgaben systematisch zu vergleichen.

Der Qiskit-Simulator ist bereits in der QHAna-Plattform integriert und verwendet eine bestimmte Schnittstelle. Ein Ziel dieser Arbeit ist es, zu überprüfen, ob andere Simulatoren ebenfalls mit dieser Schnittstelle kompatibel sind, damit alle auf die gleiche Art und Weise in die Plattform integriert werden können. Die Untersuchungen haben mit der Dokumentation von Qiskit begonnen. In der Qiskit-Lernplattform wurden detaillierte Informationen – wie die Eigenschaften des Simulators, seine Funktionsweise, wie er einen Quantenschaltkreis im OpenQASM-Format als Eingabe nimmt und die Simulation durchführt, sowie die für gewünschte Ausgaben notwendigen Methoden – im Jupyter Notebook getestet, um sowohl theoretische als auch praktische Erfahrungen zu sammeln. In diesem Kontext diente der Qiskit-Simulator als Referenzpunkt für andere potenziell auswählbare Simulatoren.

Bei der Untersuchung des Qiskit-Simulators im QHAna System wurde bemerkt, dass dieser verschiedene Ergebnistypen unterstützt. Diese Fähigkeit wurde zu einem wichtigen Kriterium bei der Untersuchung anderer Simulatoren. Manche Ergebnisformate sind für die Simulatorenwahl

grundlegend, während andere besonders beim Vergleich der Ergebnisse hilfreich sind. Obwohl die Histogrammdarstellung in einigen Bibliotheken als zusätzliches Merkmal gesehen wird, wurden lokal mithilfe von Python für alle Simulatoren Histogrammausgaben generiert und evaluiert.

Zusätzlich ist für die verwendeten Quantenbibliotheken eine klare und umfassende Dokumentation für die Benutzer von großer Bedeutung. Ein weiteres Kriterium war die unkomplizierte Installation der Bibliothek, insbesondere durch Befehle, welche ein sofortiges Arbeiten ermöglichen. Besonders beliebte oder häufig in der Literatur erwähnte Simulatoren wurden zuerst betrachtet.

Im Entscheidungsprozess wurden die folgenden Simulatoren ausgewählt, da sie alle vorgegebenen Kriterien erfüllen und erfolgreich in das QHAna integriert wurden: Qiskit Aer (qiskit~=0.43), Cirq-Simulator (cirq~=1.3.0), Pytket-QulacsBackend (pytket-qulacs~=0.29.0), PennyLane-default.qubit, Amazon Braket Local Simulator (qbraid~=0.4) und MQT-DDSIM (mqt.ddsim~=1.18).

3.2 Vergleich von Quantensimulatoren

Im Rahmen dieser Arbeit werden sechs verschiedene Quantensimulatoren untersucht. Dabei zeigte sich, dass sie bestimmte Gemeinsamkeiten und Unterschiede aufweisen. Um einen umfassenden Vergleich zu ermöglichen, werden spezifische Kriterien für jeden Simulator identifiziert, die in Kapitel 4 für den Vergleich der Quantensimulatoren verwendet werden. Diese Kriterien basieren sowohl auf den praktischen Erfahrungen während der Implementierungsphase als auch auf Literaturrecherchen und Analysen der zugehörigen Dokumentationen.

Die Auswahl dieser Kriterien ermöglicht es, die vorhandenen und fehlenden Aspekte jedes Simulators zu bewerten und ihre Anwendbarkeit in verschiedenen Szenarien zu bestimmen. Bei der Auswahl der Kriterien wurde insbesondere darauf geachtet, die für spezifische Anforderungen am besten geeigneten Simulatoren auszuwählen, um einen Vergleich der verschiedenen Simulatoren zu ermöglichen.

Bei manchen Bibliotheken können die Simulatoren durch das Spezifizieren von Backends eingeschränkt werden. Zu diesen Simulatoren gehören Qiskit, MQT-DDSIM und Amazon Braket Local Simulator. Wenn diese nicht speziell angegeben sind, wird für Qiskit Aer und MQT-DDSIM der qasm_simulator und für Amazon der local state vector simulator (braket_sv) verwendet. Für die anderen Simulatoren werden die Kriterien stets mit denselben Simulatoren berücksichtigt. Dazu gehören der Cirq-CirqSimulator, Pytket_QulacsBackend und der PennyLane-default.qubit Simulator.

Die Eigenschaft *Entwickler* ist die Institution, Organisation, Firma oder Universität, die den jeweiligen Simulator entwickelt hat. Unter *Programmiersprache* ist die Sprache angegeben, in der der Simulator entwickelt wurde. Wenn jedoch eine zusätzliche Sprache in Klammern vermerkt ist, wie bei MQT's C++ (Python), dann ist dies eine zusätzliche Sprache mit der der Simulator kompatibel ist.

Der Eintrag *OpenQASM Unterstützung* zeigt, ob ein bestimmter Simulator OpenQASM-Code akzeptieren kann oder nicht. Hierbei ist zu beachten, dass OpenQASM allgemein betrachtet wurde; die meisten Simulatoren unterstützen OpenQASM 2.0, während einige die Version 3.0 unterstützen.

Bei einigen ist eine direkte Umwandlung nicht möglich, weshalb zusätzliche Konverter benötigt werden. Welcher Simulator welche Version unterstützt und wo Probleme auftreten, wird im Kapitel 4 detaillierter beschrieben.

Unter dem Kriterium der *Exakten Simulation* zeigt ein Quantensimulator, ob er in der Lage ist, Quantenschaltkreisen unter idealen Bedingungen und ohne äußere Störungen korrekt und geräuschlos zu simulieren, während das *Noisy Simulation*-Kriterium zeigt, ob der Simulator die Fähigkeit hat, Quantenschaltkreisen unter *noisy* Bedingungen zu simulieren. Unterstützte *noise* Typen, verfügbare Fehlermodelle und Parameteranpassungen werden dabei berücksichtigt.

Die Eigenschaft *counts* zeigt, ob der Simulator die Ausgabe der Verteilung der Messergebnisse unterstützt. *Statevector* gibt an, ob der Simulator in der Lage ist, Ergebnisse als Zustandsvektor zurückzugeben. Bei der Bewertung des *Statevector* Kriteriums wurde für Qiskit Aer und MQT das *StatevectorSimulator-Backend* verwendet.

Beim Vergleichen mit anderen Simulatoren werden Qiskit Aer *UnitarySimulator-Backend* und MQT *UnitarySimulator-Backend* als Simulator verwendet. Die Simulatoren werden danach bewertet, ob sie eine *Unitary Matrix* für einen bestimmten Quantenschaltkreise direkt liefern können.

Für die Bewertung der *Density Matrix* wird untersucht, ob Simulatoren die Fähigkeit besitzen, die Dichtematrix eines gegebenen Quantenschaltkreises direkt bereitzustellen. Speziell für diesen Kriteriumsvergleich werden die `AerSimulator('aer_simulator_density_matrix')` von Qiskit und `LocalSimulator(backend="braket_sv")` von Amazon Braket verwendet.

Die Eigenschaft *Probability* zeigt, ob die Quantensimulatoren in der Lage sind, die Wahrscheinlichkeitsverteilung der Ergebnisse für einen gegebenen Quantenschaltkreis direkt bereitzustellen. Es wird untersucht, ob die Simulatoren diese Wahrscheinlichkeitsverteilung liefern oder ob sie eine klar in den Dokumentationen beschriebene Methode zum Zugriff auf diese Verteilung bieten.

Einige Quantensimulatoren bieten direkt Funktionen zur Darstellung der Simulationsergebnisse in Histogrammen an. In dieser Eigenschaft wird untersucht, ob der Simulator über eine Histogramm-Visualisierungsfunktion innerhalb seiner eigenen Bibliothek verfügt. Dabei sollte jedoch berücksichtigt werden, dass das Fehlen dieser Funktion in einem Simulator nicht bedeutet, dass er diese Fähigkeit nicht durch die Integration von externen Bibliotheken oder Tools erwerben kann.

Während praktischer Arbeiten wurden Unterschiede in den Statevector-Simulationsergebnissen verschiedener Simulatoren (bei Qiskit Aer und MQT *StatevectorSimulator* Backends) beobachtet. Um diese Unterschiede in jedem Simulator zu untersuchen und die Ergebnisbeobachtung zu erleichtern, wurde der Bell-Zustand in OpenQASM 2.0 sowohl mit `measure` als auch ohne Messung simuliert. Einige Simulatoren zeigten das Statevector-Ergebnis, als ob keine Messung durchgeführt worden wäre, obwohl eine Messung im OpenQASM-Code vorhanden war. Andere Simulatoren hingegen aktualisierten ihre Ergebnisse basierend darauf, ob im OpenQASM-Code eine Messung vorhanden war oder nicht. Aufgrund dieser Unterschiede wurde eine Merkmale *Statevector Aktualisierung* hinzugefügt, da solche Unterschiede beim Vergleich der Statevector-Ergebnisse zu Missverständnissen führen könnten. Dieses Kriterium prüft, ob ein Simulator den Statevector nach einer Messung aktualisiert. Es ist möglich, dass das Ergebnis je nach Vorhandensein einer Messung variiert. Diese Ausprägung wird in Kapitel 4 als “beide” markiert. Falls nur der Zustand vor der Messung gezeigt wird, wird der Simulator mit “vor” markiert.

Im nächsten Kapitel werden die Simulatoren anhand der genannten Eigenschaften detaillierter untersucht und die Ergebnisse in der Tabelle 4.1 dargestellt.

In dieser Arbeit erforderte die Auswahl des Simulators einen umfassenden Prozess. Die Entscheidung basierte nicht nur auf Literaturrecherchen, sondern auch auf lokal durchgeführten Implementierungen und den daraus resultierenden konkreten Ergebnissen. Zu Testzwecken wurden die Prototyp Implementierungen der Simulatoren in Jupyter mit OpenQASM-Eingaben ausprobiert und geeignete Prototypen wurden zur Integration in die QHAna-Plattform ausgewählt. Zu Beginn der Arbeit wurden zufällig erstellte Quantenschaltkreisen von einem bis zwanzig Qubits in der Jupyter-Umgebung als Testeingaben verwendet. Bei aufgetretenen Fehlern wurden zusätzliche Untersuchungen durchgeführt und die notwendigen Anpassungen im Code vorgenommen. Die offizielle Dokumentation des Simulators, Anleitungen zur Verwendung und Beispiele waren ebenfalls Faktoren, die die Dauer der Arbeit beeinflusst haben.

4 Quantensimulatoren

Die in Kapitel 3.2 vorgestellten Kriterien und Eigenschaften zur Untersuchung der Quantensimulatoren sind in Tabelle 4.1 dargestellt. Dieses Kapitel beschreibt die Untersuchungsergebnisse genauer.

In dieser Tabelle 4.1 werden die von den Quantensimulatoren unterstützten Eigenschaften mit ✓ , die nicht unterstützten oder bei denen nicht genügend Informationen vorliegen mit ✕ gekennzeichnet. Um ein konkretes Beispiel zu geben, wenn ein Simulator *Unitary Matrix* direkt zurückgeben kann oder eine dokumentierte Methode für den Zugriff auf diese Matrix anbietet, wird er mit ✓ sonst ✕ markiert. Simulatoren, die *Density Matrix* direkt liefern können oder eine klare Methode dazu in ihrer Dokumentation haben, werden mit ✓ markiert.

Zusätzlich wird in der Tabelle bei der Statevector-Aktualisierung *beide* vermerkt, wenn das Ergebnis je nach Vorhandensein einer Messung variiert und *vor*, wenn nur der Zustand vor der Messung angezeigt wird.

4.1 Qiskit

Qiskit [Abr+19] ist eine weitverbreitete Bibliothek für Quantencomputing, entwickelt von IBM und veröffentlicht unter der Apache 2.0-Lizenz [ACCB23]. Geschrieben in Python, bietet es eine nahtlose Integration in Jupyter-Notebooks [AJLT22]. Qiskits Abstraktionsebene über den Typ der ausgewählten Maschine vereinfacht das Design und die Anwendung von Quantenalgorithmen und erhöht die Benutzerfreundlichkeit [ACCB23]. IBM bietet zudem verschiedene Simulatoren und Quantencomputer kostenlos innerhalb eines festgelegten Zeitlimits an [AJLT22].

Qiskit bietet ein umfassendes Werkzeugset, das für die Erstellung grundlegender Quantensysteme und die Nutzung von Quantensimulatoren erforderlich ist [SB23]. Dies trägt zur beschleunigten Entwicklung von Quantenanwendungen bei und bietet dem Nutzer eine anpassungsfähige Plattform, auf welcher er direkt auf Quantenschaltkreise und spezifische Algorithmen von Quantencomputern zugreifen kann [SB23].

In Qiskit wird ein Quantenschaltkreis entworfen [Qis23e]. Anschließend wird diese für einen spezifischen Quantenservice, wie z.B. einen klassischen Simulator, kompiliert [Qis23e]. Nach der Ausführung, die entweder lokal oder in der Cloud stattfinden kann, werden die Ergebnisse analysiert und visualisiert [Qis23e].

Mit der Methode `from_qasm_str()` [Qis23g] kann ein Quantenschaltkreis im 2.0-Format in ein `QuantumCircuit`-Objekt umgewandelt werden. Qiskit unterstützt nicht nur die 2.0-Version von OpenQASM, sondern auch die neuere Version 3.0 [Qis23d].

	Qiskit Aer	**Cirq CirqSimulator**	**Pytket QulacsBackend**	**PennyLane (default.qubit)**	**Amazon Braket Local Simulator**	**MQT DDSIM**
Entwickler	IBM	Google	Quantinuum, Qulacs Team	Xanadu	AWS	TUM
Programmiersprache	Python	Python	Python	Python	Python	C++ (Python)
unterstützt OpenQASM	✓	✓	✓	✓	✓	✓
Exakte Simulation	✓	✓	✓	✓	✓	✓
Noisy simulation	✓	✓	×	teilweise	✓	nicht für Python
Unitary matrix	✓	×	×	×	×	✓
Statevector	✓	✓	✓	✓	✓	✓
Statevektor-Aktualisierung	beide	beide	vor	vor	vor	vor
Probability	×	×	✓	✓	✓	×
Density matrix	✓	×	×	✓	✓	×
Counts	✓	✓	✓	✓	✓	✓
Histogramm	✓	✓	×	×	×	×

Tabelle 4.1: Vergleich verschiedener Quantensimulatoren.

In dieser Arbeit wird der Aer Simulator aus der Qiskit-Bibliothek verwendet. Das Paket Qiskit Aer dient der spezialisierten Simulation von Quantenschaltkreisen und stellt verschiedene Backends für diesen Zweck zur Verfügung, wie z.B. qasm_simulator, unitary_simulator und statevector_simulator[Qis23a].

In der Auflistung 4.1 wird ein Beispiel dargestellt, das die Simulation eines in OpenQASM 2.0 formatierten Quantenschaltkreises mit zwei verschiedenen AerBackens durchführt. Die verschiedenen Ausgabeformate dieser Simulationen sind in Abbildung 4.1 dargestellt. Die QASM-Datei wurde unter Verwendung des [QBW23] für einen zufällig generierten 4-Qubit-Quantenschaltkreis erstellt.

In Qiskit Aer wird der `statevector_simulator` häufig als Backend verwendet [Qis23b]. Er gibt den Quantenzustand als komplexen Vektor der Dimension 2^n zurück, wobei n die Anzahl der Qubits ist [Qis23b]. Nach Auswahl dieses Simulators aus dem Aer-Modul kann man den Quantenschaltkreis mittels der `run`-Methode ausführen [Qis23b]. Das resultierende Ergebnis repräsentiert mit der Methode `result.get_statevector()` den Endzustand des Quantenschaltkreis im `Statevector`-Format von Qiskit [Qis23b]. Mit dem `qasm_simulator` wurde der Quantenschaltkreis mehrfach wiederholt, um Informationen über die Verteilung der Bitfolge zu erhalten [Qis23b]. Die Anzahl der Wiederholungen wird durch das Schlüsselwort Shots bestimmt [Qis23b]. Anschließend wurden die Simulationsergebnisse im Binärformat mit der Methode `get_counts()` abgerufen [Qis23b]. Ähnlich wie beim `statevector_simulator` und `qasm_simulator` bietet Qiskit Aer auch einen `unitary_simulator`. Dieser funktioniert, solange alle Elemente im Quantenschaltkreis unitäre Operationen sind. Dieser Backend berechnet die Matrix, die die Gatter im Quantenschaltkreis repräsentiert [Qis23b]. In AerSimulator gibt es ebenfalls einen Backend, der als `aer_simulator_density_matrix` bezeichnet wird [Qis20a]. Es sei jedoch zu beachten, dass gemäß den Versionshinweisen von Qiskit in zukünftigen Qiskit-Versionen möglicherweise einige Änderungen an Backend-Systemen vorgenommen werden könnten [Qis23f].

In Qiskit gibt es verschiedene Visualisierungsoptionen, einschließlich des Histogramms, der Bloch-Sphäre und Qsphere [Qis23c]. In Abbildung 4.1 zeigt das Histogramm die Verteilung der Messergebnisse, die durch die Simulation eines Vier-Qubit-Quantenschaltkreises mit dem `qasm-Simulator` erzielt wurden. Zudem demonstriert Listing 4.1, wie mit der Methode `plot_histogram()` dieses Histogramm erstellt wird.

In Qiskit gibt es keine direkte Methode im `result` Objekt, um Wahrscheinlichkeiten aus den Ergebnissen zu extrahieren. Trotzdem können diese Wahrscheinlichkeiten entweder indirekt über das `result` Objekt oder direkt mithilfe der `plot_histogram` (Wahrscheinlichkeitshistogrammen) Funktion berechnet werden [Qis20a]. Beispielsweise, wenn eine Simulation mit dem `qasm_Simulator` durchgeführt wird, können aus den resultierenden Messwerten durch Anwendung entsprechender mathematischer Funktionen die Wahrscheinlichkeiten ermittelt werden.

Zusätzlich kann mit `qiskit.providers.aer.noise.NoiseModel` Fehlermodelle für die geräuschbehafteten Simulationen von Quantenschaltkreisen definiert werden [Qis20b].

Listing 4.1 Beispiel für Aer QasmSimulator und StatevectorSimulator

```
from qiskit import QuantumCircuit, execute, Aer
from qiskit.circuit import measure
from qiskit.visualization import plot_histogram

def run_counts_simulation(circuit, shots=1000):
    backend_counts = Aer.get_backend('qasm_simulator')
    result = execute(circuit, backend_counts, shots=shots).result()
    counts = result.get_counts()

    return counts

def run_statevector_simulation(circuit):
    backend_statevector = Aer.get_backend('statevector_simulator')
    statevector = execute(circuit, backend_statevector).result().get_statevector(circuit)

    return statevector

def main(qasm_code):
    circuit = QuantumCircuit.from_qasm_str(qasm_code)
    counts = run_counts_simulation(circuit)
    statevector = run_statevector_simulation(circuit)

    print("State Vector:", statevector)
    if counts:
        print("Counts:")
        display(counts)
        display(plot_histogram(counts))

if __name__ == "__main__":


    with open("random_indep_qiskit_4.qasm", "r") as f:
        qasm_code = f.read()

    main(qasm_code)
```

4.2 The Munich Quantum Toolkit

Das Munich Quantum Toolkit (MQT), ein Projekt, das vom Lehrstuhl für Design Automation der Technischen Universität München entwickelt wurde, bietet verschiedene Designautomatisierungstools und Software für Quantencomputer an [Des].

Das Ziel dieses Projekts ist es, umfassende Lösungen für Designprozesse im gesamten Bereich der Quantensoftware zu bieten [Des]. Dies beinhaltet Unterstützung für Nutzer bei der Realisierung ihrer Quantenanwendungen und die Entwicklung neuer Methoden für die Simulation, Kompilierung und Verifizierung von Quantenschaltkreisen [Des].

MQT beinhaltet einen Quantenschaltkreissimulator namens MQT DDSIM, der auf *decision diagrams* basiert [Des]. Dieser Simulator ist nicht nur eine C++ Bibliothek, sondern ist auch für eine ideale Zugänglichkeit mit einer leicht bedienbaren Python-Schnittstelle ausgestattet [MQT].

```
State Vector: Statevector([-0.        +0.j        , -0.        +0.j        ,
               0.        +0.j        ,  0.        +0.j        ,
               0.        +0.j        ,  0.        +0.j        ,
               0.        +0.j        , -0.31956188-0.94756541j,
               0.        +0.j        ,  0.        +0.j        ,
               0.        +0.j        , -0.        +0.j        ,
              -0.        +0.j        , -0.        +0.j        ,
              -0.        +0.j        ,  0.        +0.j        ],
             dims=(2, 2, 2, 2))
Counts:

{'0100': 39,
 '0001': 134,
 '1001': 233,
 '0101': 64,
 '1000': 69,
 '0000': 92,
 '0010': 27,
 '0111': 67,
 '1100': 101,
 '1101': 70,
 '0110': 88,
 '1111': 11,
 '1110': 4,
 '1011': 1}
```

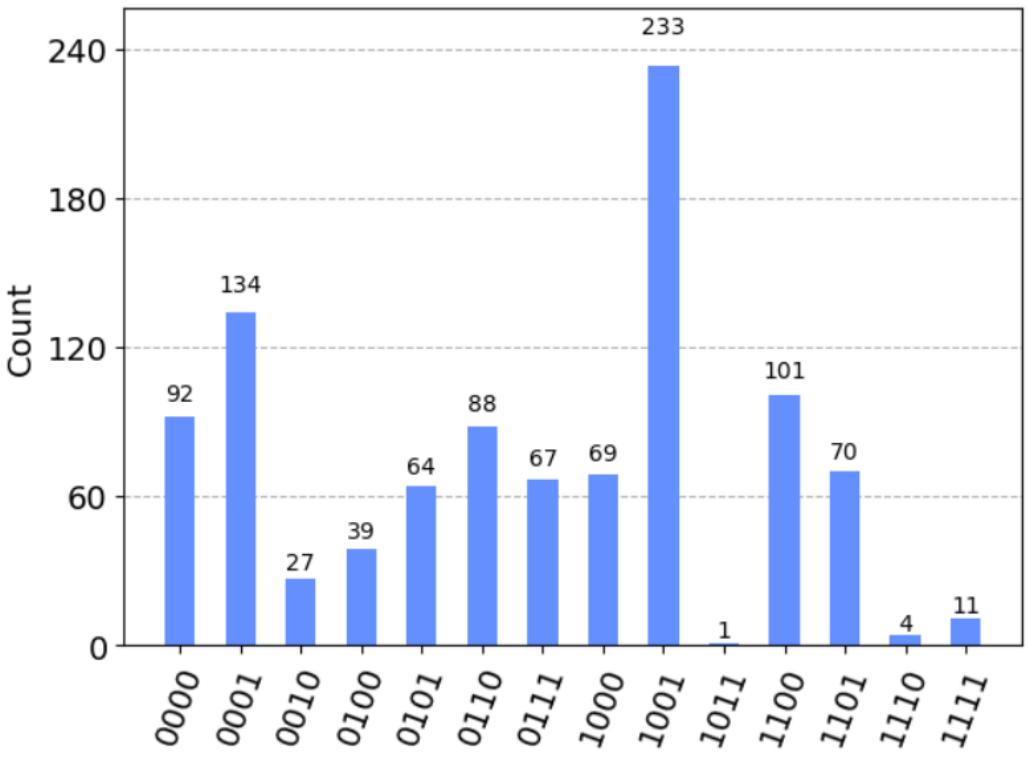


Abbildung 4.1: Qiskit Output

Der MQT DDSIM Simulator unterstützt nicht nur *error-free* und *exact simulations*, sondern auch *weak simulation* und *approximating simulation* [Des23]. Mit der Methode der *weak simulation* liefert der Simulator Messergebnisse statt expliziter Quantenzustände, um eine genauere Simulation von physischen Quantencomputern zu ermöglichen [Des23].

Zusätzlich ist zu beachten, dass der DDSIM Simulator für verrauschte Simulationen aktuell nicht über Python-Bindungen zugänglich ist [MQT23b]. Die `approximating simulation` wurde aufgrund der probabilistischen Natur von physischen Quantencomputern und ihrer inhärenten Widerstandsfähigkeit gegenüber einem gewissen Grad von Fehlern entwickelt [Des23]. Um eine höhere Leistung zu erreichen, werden Knoten aus den *decision diagrams* entfernt, wenn die Einflüsse der Knoten zum Quantenzustand als vernachlässigbar betrachtet werden [Des23]. Dies ermöglicht eine kompaktere Darstellung und somit eine schnellere Simulation, während der Verlust an Genauigkeit fein abgestimmt werden kann [Des23].

DDSIM ermöglicht die Simulation von Quantenschaltkreisen, die im OpenQASM-Format definiert sind [WHB20]. Durch die Integration von Qiskit kann ein Quantenschaltkreis, der im OpenQASM 2.0-Format vorliegt, mittels `QuantumCircuit.from_qasm_str()` eingelesen werden [MQT23a]. Darüber hinaus ist es möglich, den Circuit Simulator unabhängig von Qiskit zu nutzen, indem ein Dateiname, wie im nachstehenden Beispiel gezeigt, übergeben wird [MQT23a]:

```
sim = ddsim.CircuitSimulator("filename.qasm")
```

Mithilfe der Integration in Qiskit können auch Histogramme der gemessenen Verteilung erzeugt werden, wie es in Abbildung 4.1 dargestellt ist.

Im Gegensatz zu Qiskit ignoriert der DDSIM Statevector-Simulator Messungen, die am Ende des Quantenschaltkreises durchgeführt werden. Dadurch wird immer der Zustandsvektor vor abschließenden Messungen zurückgegeben. Ein Beispiel hierfür ist in den Abbildungen 4.2 und 4.3 dargestellt. Zum Vergleich, wenn ein Quantenschaltkreis im OpenQASM 2.0-Format mit dem Qiskit Aer `statevector` Simulator simuliert wird und eine `measure` im Quantenschaltkreis vorkommt, wird der Zustand nach dieser `measure` als einen spezifisch kollabierten Zustand angezeigt. Der bereitgestellte Quantenschaltkreis ist in beiden Fällen gleich.

```
from qiskit import QuantumCircuit, execute, Aer

qasm_code = """
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg meas[2];
h q[0];
cx q[0],q[1];
barrier q[0],q[1];
measure q[0] -> meas[0];
measure q[1] -> meas[1];
"""

circ = QuantumCircuit.from_qasm_str(qasm_code)

print(circ.draw())


backend = Aer.get_backend('statevector_simulator')
job = execute(circ, backend)
state=job.result().get_statevector(circ)

print("Qiskit statevector Result:" ,state)
```

```
q_0: ┤ H ├──■──┤ M ├
q_1: ──────┤ X ├──┤ M ├
meas: 2/═══════════╩═══╩
                   0   1
Qiskit statevector Result: Statevector([1.+0.j, 0.+0.j, 0.+0.j, 0.+0.j],
            dims=(2, 2))
```

Abbildung 4.2: Simulationsergebnis eines Quantenschaltkreises mit Qiskit Aer

```
from qiskit import QuantumCircuit, execute
from mqt import ddsim

qasm_code = """
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg meas[2];
h q[0];
cx q[0],q[1];
barrier q[0],q[1];
measure q[0] -> meas[0];
measure q[1] -> meas[1];
"""

circ = QuantumCircuit.from_qasm_str(qasm_code)

print(circ.draw())

backend = ddsim.DDSIMProvider().get_backend('statevector_simulator')
job = execute(circ, backend)
state=job.result().get_statevector(circ)

print("DDSIM statevector Result:" ,state)
```

```
q_0: ┤ H ├──■──┤ M ├
q_1: ──────┤ X ├──┤ M ├
meas: 2/═══════════╩═══╩
                   0   1
DDSIM statevector Result: [0.70710678+0.j 0.        +0.j 0.        +0.j 0.70710678+0.j]
```

Abbildung 4.3: Simulationsergebnis des gleichen Quantenschaltkreises mit MQT-DDSIM

Der DDSIMProvider bietet aktuell sieben Simulations-Backends an [MQT23c]. Der *QasmSimulator* simuliert Quantenschaltkreisen und generiert die vorgegebene Anzahl von Shots. Das Ergebnis kann aus dem `result`-Objekt mit der Methode `get_counts` abgerufen werden [MQT23c]. Der *StatevectorSimulator* simuliert Quantenschaltkreisen und gibt den entsprechenden Statevector zurück. Das Ergebnis ist über die Methode `get_state` im `result`-Objekt verfügbar [MQT23c]. Der *UnitarySimulator* gibt die dazugehörige unitäre Matrix des Quantenschaltkreises zurück. Das Ergebnis ist über die Methode `get_unitary` im `result`-Objekt verfügbar [MQT23c]. Des Weiteren gibt es auch den *HybridQasmSimulator*, *HybridStatevectorSimulator*, *PathQasmSimulator* und *PathStatevectorSimulator* im Auswahl des DDSIMProviders [MQT23c].

4.3 Cirq

Cirq ist ein Open-Source-Framework zur Programmierung von Quantencomputern [Goo]. Diese auf Python basierende Bibliothek dient dazu, Quantenschaltkreise zu entwerfen und zu optimieren, um sie auf Quantencomputern und Simulatoren auszuführen [Goo].

Cirq verfügt derzeit nicht über eine eigene Quantensprache zur Kommunikation mit Quantenprozessoren, kann aber OpenQASM-Code generieren, um auf IBM's Quantencomputern zu laufen [LaR19a]. Mittels Cirq, insbesondere durch das Modul `from cirq.contrib.qasm_import`, können Quantenschaltkreise im OpenQASM (2.0)-Format importiert werden [Goo23b]. Dennoch befindet sich das QASM Import-Tool von Cirq in einer experimentellen Phase und unterstützt nur einen Teil der OpenQASM-Spezifikationen [Goo23b]. Bestimmte Gatter, beispielsweise `barrier qargs`, `reset qubit/qreg`, `u0(`γ`)`, `cu1`, `crz`, `cu3` und `rzz`, werden aktuell nicht unterstützt [Goo23b].

Cirq unterstützt zwei Typen der Simulation: `Pure State` und `Mixed State`. Während der `cirq.Simulator` (in dieser Arbeit verwendeter Simulator) die `Pure State`-Simulationen unterstützt, ist der `cirq.DensityMatrixSimulator` für die `Mixed State`-Simulationen zuständig [Goo23d]. Der `pure state simulator` und `mixed state simulators` werden für Quantenschaltkreisen verwendet, einschließlich `unitary`, `measurements` und `noise` [Goo23b]. Sie simulieren entweder den reinen Zustand des Quantenschaltkreise oder den gemischten Zustand mit klassischen Wahrscheinlichkeiten [Goo23d]. Der `pure state simulator` unterstützt das Simulieren von Rauschen, solange die Reinheit des Zustands erhalten bleibt [Goo23d].

In Cirq entwickelte Quantenprogramme sind als Quantenschaltkreise konzipiert, die ausschließlich Qubits beinhalten, ohne explizite klassische Register und Cirq ermöglicht es, Qubits als *line* oder als *grid* zu definieren [SY22]. Im in Listing 4.2 gezeigten Codebeispiel wurde zum Beispiel `GridQubit` verwendet. Bei Cirq sind der Quantenschaltkreis und die Quantenregister separate Objekte [SY22]. Der Quantenschaltkreis wird durch Hinzufügen der Operatoren und Qubits erstellt [SY22].

Cirq bietet einen Simulator, der die Ergebnisse von Quantenschaltkreisen bis zu einem Limit von etwa 20 Qubits berechnen kann und mit `cirq.Simulator()` initialisiert wird [Goo23a]. Bei der Verwendung des Simulators gibt es zwei Ansätze: Durch `simulate()` kann direkt auf die resultierende Statevector zugegriffen werden, während `run()` nur Bit-Strings als Resultat liefert und das Verhalten echter Quantengeräte simuliert [Goo23a].

Die Ergebnisse der Simulation werden als `result`, in Form von `counts` (mittels der Methode `result.histogram(key='meas')`) und als `statevector` (durch die Methode `final_state_vector`) erhalten (4.2). Für die Visualisierung wird `cirq.plot_state_histogram(result, plt.subplot())` verwendet, wodurch ein der Cirq-Bibliothek spezifisches Histogramm generiert wird.

In der Programmierung von Quantenschaltkreisen mit Cirq wird ein Schlüsselwert ('key'-Wert) definiert, um den Zugriff auf die Messergebnisse der spezifizierten Qubits zu ermöglichen [SY22]. Die variable `repetitions` gibt die Anzahl der gewünschten Wiederholungen an.

Das Ergebnis der Simulation präsentiert die Messwerte jedes einzelnen Qubits als separate binäre Zeichenkette [SY22]. Um die Messwerte aller Qubits als eine einzelne Zahl zu interpretieren, wird der definierte 'key'-Wert verwendet [SY22]. Dadurch wird das Resultat zu einer Sammlung von Dezimalzahlen, zusammen mit ihren jeweiligen Häufigkeiten, transformiert [SY22]. Dies ermöglicht eine Analyse der Häufigkeit des Auftretens jedes möglichen Messergebnisses [SY22]. Beispielweise, betrachtet man nur einen Teil der Ausgabe des in Listing 4.2 gezeigten Codes, so ergibt sich folgendes:

```
Ergebnisse:
meas=0010000001100100110, 0010000001100100110, 0010000001100100110
Counter({0: 13, 7: 6})
```

Beim Arbeiten mit dem CirqSimulator wurde aufgrund der nicht-binären Darstellung des `counter` wie in dieser Ausgabe und für die Histogramm-Grafik eine zusätzliche Methode hinzugefügt, um die Ergebnisse in binärer Form zu erhalten.

Der Statevector wird durch die methode final_state_vector abgefragt. Dieses Ergebnis kann jedoch je nach Vorhandensein von Messungen im `OpenQASM` Code variieren.

Im Cirq-Framework kann die Dichtematrix für einen gegebenen Quantenschaltkreis mithilfe des `cirq.DensityMatrixSimulator` und der Methode `result.final_density_matrix` abgerufen werden [Goo23c]. Zudem kann die unitäre Matrix einer bestimmten Operation in der Cirq-Bibliothek unabhängig von der Simulation durch die Funktion `cirq.unitary(operation)` bezogen werden [Goo23a].

Cirq ermöglicht es exterene Simulatoren über eine Schnittstelle zu verwenden [Goo23d]. Ein Beispiel dafür ist `qsim` [Qua20], ein Simulator, der in `Cirq` integriert und für tiefe Quantenschaltkreise mit bis zu 30 Qubits optimiert ist. Er implementiert `cirq.SimulatesAmplitudes`, `cirq.SimulatesFinalState` und `cirq.SimulatesExpectationValues`. Während `qsim` 8 GB RAM verbraucht, verdoppelt sich der RAM-Verbrauch mit jedem zusätzlich hinzugefügten Qubit, auch wenn die Simulation größerer Quantenschaltkreise möglich ist [Goo23d].

4.4 Pytket mit QulacsBackend

Pytket, entwickelt von Quantinuum, ist ein Python-Modul, das zur Interaktion mit $t|ket\rangle$, einem Optimierungscompiler für Quantenschaltkreisen, dient [TKE23]. Neben Pytket gibt es mehrere ergänzende Module, die den Zugriff auf verschiedene Quanten-Hardwarekomponenten und Simulatoren vereinfachen [TKE23]. Diese Module unterstützen auch die Integration in einer Vielzahl weitverbreiteter Quantensoftware-Anwendungen [TKE23].

Listing 4.2 Quantensimulation mit Cirq Simulator

```
import cirq
import matplotlib.pyplot as plt

# define qubits
qubits = [cirq.GridQubit(0, i) for i in range(3)]

# Create a circuit
circuit = cirq.Circuit(
    cirq.H(qubits[0]),  # define Hadamard Gatter
    cirq.CNOT(qubits[0], qubits[1]),  #  define CNOT Gatter
    cirq.CNOT(qubits[1], qubits[2]),  # define nother CNOT Gatter
    cirq.measure(*qubits, key='meas')  # define measurements
)

print("Circuit:")
print(circuit)

# Simulate the circuit several times.
simulator = cirq.Simulator()

state = simulator.simulate(circuit).final_state_vector
result = simulator.run(circuit, repetitions=19)
print("Results:")
print(result)
print(result.histogram(key='meas'))
print(state)

vis = cirq.plot_state_histogram(result, plt.subplot())
plt.show()
```

Der $t|ket\rangle$ -Compiler optimiert plattformunabhängige Quantenschaltkreisen und eignet sich besonders für Noisy Intermediate-Scale Quantum (NISQ)-Geräte [Qua23c]. Das Pytket-Paket stellt eine Schnittstelle zu $t|ket\rangle$ bereit und unterstützt verschiedene Quantenschaltkreisstandards [Qua23c]. Pytket ist mit bestimmten Python-Versionen auf Linux, MacOS und Windows kompatibel [Qua23c].

Mithilfe von Pytket ist es möglich, Quantenschaltkreise aus verschiedenen Quellen zu importieren, einschließlich Sprachen wie OpenQASM [Qua23c]. Zum Beispiel kann man mittels des folgenden Codes eine in OpenQASM 2.0 formatierte Quantenschaltkreis in Pytket importieren und sie zur weiteren Simulation vorbereiten.

```
from pytket.qasm import circuit_from_qasm
circ = circuit_from_qasm("filename.qasm")
compiled_circuit = backend.get_compiled_circuit(circ)
```

Nicht alle importierten Quantenschaltkreisen sind ohne Anpassung mit jedem Backend kompatibel. Um solche Inkompatibilitäten zu beheben, ist es notwendig, den Quantenschaltkreis mit Methoden wie `get_compiled_circuits` speziell für einen Backend zu kompilieren. In Pytket ermöglichen

Erweiterungsmodule eine Verbindung zu Quantum-Geräten und Simulatoren unterschiedlicher Anbieter [Qua23e]. Ein Backend in Pytket stellt dabei die Schnittstelle zu einem Quantum-Verarbeitungsgerät oder Simulator dar [Qua23d].

Im Rahmen dieser Arbeit wird das Pytket_QulacsBackend verwendet, ein leistungsfähiger Open-Source-Quantenschaltkreissimulator, der von der Gruppe von Prof. Fujii entwickelt wurde und nun von QunaSys gewartet und weiterentwickelt wird [Qul]. Qulacs bietet eine effiziente Simulation von Quantencomputern durch eine parallele C/C++ Backend-Struktur [Qul]. Es kann Fehler in echten Quantengeräten simulieren und benutzt dabei spezielle Quantengatter [Qul]. Außerdem ermöglicht es die Verwendung von GPUs zur Simulation [Qul]. Allerdings ist es, obwohl Qulacs selbst *noisy simulations* unterstützt, mit der Pytket-Integration nicht möglich.

QulacsBackend unterstützt sowohl Counts (mithilfe der Methode `get_counts`) als auch Statevector (mithilfe der Methode `get_state`) Outputs [Qua23b].

Um die Verteilung der Messergebnise mittels QulacsBackend zu berechnen wird folgender Prozess durchgeführt: Über `process_circuit` wird zuerst der Quantenschaltkreis und die Anzahl der Durchläufe festgelegt und an das ausgewählte Backend (hier Qulacs) gesendet. Anschließend wird mit `get_result` das Ergebnis des Quantenschaltkreises abgerufen und schließlich werden die Messergebnisse mit `get_counts` extrahiert.

```
handle = backend.process_circuit(compiled_circuit, n_shots=shots)
result = backend.get_result(handle)
counts = result.get_counts()
```

Die Methode `get_counts` gibt die Frequenzen der Messergebnisse der Qubits als Tupel zurück. Ein mögliches Ergebnis könnte folgendermaßen aussehen: `Counter({(0, 1): 507, (1, 1): 493})`. Hier zeigt `{(0, 1): 507}`, dass der erste Qubit den Wert 0 und der zweite Qubit den Wert 1 insgesamt 507 Mal angenommen hat. Die Kombination 1 und 1 wurde hingegen insgesamt 493 Mal registriert. Außerdem dient die Methode `probs_from_counts(counts)` dazu, Wahrscheinlichkeiten aus den gegebenen Zählungen zu extrahieren.

Unabhängig vom Vorhandensein einer Messanweisung (`measure`) im OpenQASM (2.0) Code wird das Simulationsergebnis ohne eine durchgeführte Messung dargestellt, wie im DDSIM Simulator.

4.5 PennyLane

PennyLane, entwickelt von Xanadu, ist eine Bibliothek, die für hybride Quantum-Klassik-Neuralnetzwerke konzipiert ist und die automatische Differenzierung von Quantenschaltkreisen ermöglicht [KLB+23] [JR20]. Sie vereint Vorteile sowohl aus Quanten- als auch aus klassischen Modellen und bietet eine Plattform für die Integration von Quantenhardware mit gängigen Frameworks wie PyTorch, TensorFlow und NumPy [Teca]. Darüber hinaus stellt PennyLane Tools für Optimierung und maschinelles Lernen bereit und zeichnet sich durch Geräteunabhängigkeit aus [Ber22]: Ein und dasselbe Quantenschaltkreismodell kann auf verschiedenen Backends ausgeführt werden. Mit Zusatzmodulen wird der Zugang zu einer breiten Palette von Plattformen ermöglicht, einschließlich Strawberry Fields, Amazon Braket, IBM Q, Google Cirq, Rigetti Forest, Microsoft

Quantum Development Kit (QDK) und ProjectQ [Ber22] [Teca]. PennyLane bietet die Möglichkeit, Quantenschaltkreise im OpenQASM-Format zu importieren. Dies geschieht mit Hilfe des PennyLaneQiskit Plugins und den Methoden `from_qasm` und `from_qasm_file` [Tec23]. Es sollte jedoch beachtet werden, dass bestimmte Anweisungen, die spezifisch für externe Frameworks sind, beim Laden eines externen Quantenschaltkreises ignoriert werden könnten [Tec23]. In solchen Fällen werden Warnmeldungen ausgegeben [Tec23].

In dieser Arbeit wurde das Simulationsgerät `default.qubit` aus der PennyLane-Bibliothek verwendet [Ber22]. Der `default.qubit` von PennyLane ist ein einfacher State-Vector-Qubit-Simulator, entwickelt in Python unter Verwendung von JAX, Autograd, Torch und Tensorflow [Tecb]. PennyLane empfiehlt diesen Simulator insbesondere für Optimierungen mit einer reduzierten Anzahl von Qubits oder wenn `stochastic expectation values` verwendet werden sollen [MP22] [Tecb].

Ein Quantenknoten (QNode in PennyLane) kapselt eine Funktion, die mittels Quantenverarbeitung auf einem Gerät läuft, das entweder Quantenhardware oder ein Simulator sein kann [Ber22].

Bei der Implementierung und Simulation wird das Gerät `dev` definiert, auf dem der Quantenknoten ausgeführt wird [Ber22]. PennyLane verwendet den Begriff *wires* für den Verweis auf Quanten-Subsysteme (qubits oder qumodes), da diese in einem Quantenschaltkreis durch horizontale Leitungen repräsentiert werden [Ber22]. Der Dekorierer `@qml.qnode(dev)` wandelt die Funktion `circuit` in einen Quantenknoten gleichen Namens um [Ber22].

Die Funktion, die einen Quantenschaltkreis deklariert, muss folgenden Regeln entsprechen [Ber22]:

- Quantum-Operationen beinhalten, die auf dem Gerät ausgeführt werden.
- Messstatistiken (einschließlich Erwartungswerte, Varianzen und Wahrscheinlichkeiten) von einem oder mehreren Observablen auf separaten *wires* zurückgeben.

Die Messstatistiken sollten als Tuple zurückgegeben werden. Solange mindestens ein Messwert zurückgegeben wird, muss nicht jeder wire gemessen werden. Neben Erwartungswerten unterstützt PennyLane auch [Ber22]:

- Das Zurückgeben von Varianzen (`qml.var()`),
- Wahrscheinlichkeiten (`qml.probs()`),
- Samples (`qml.sample()`)

Simulatorgeräte können auch Zustände (`qml.state()`, `qml.density_matrix()`) zurückgeben [Ber22].

PennyLane's `qml.drawer` Modul bietet Werkzeuge zur Visualisierung von Quantenschaltkreisen an [Xan23]. Dieses Modul ermöglicht insbesondere mit der Funktion `draw_mpl` das Zeichnen von Quantenschaltkreisen mittels *matplotlib* [Xan23]. Jedoch fehlt in diesem Modul direkt eine Funktionalität zum Erstellen von Histogrammen .

Um Rauschen zu simulieren kann mit default.qubit eine X-Rotation mittels klassischer Zufallsparameter verwendet werden [Wak21]. Jedoch wird eine umfassendere Noisy-simulation durch das default.mixed Gerät ermöglicht. Aus diesem Grund wurde in Tabelle für default.qubit device 4.1 'teilweise' angegeben.

4.6 Amazon-Braket

Das Python SDK von Amazon Braket dient zur Erforschung des Quantencomputings [Ama23]. Das als Open-Source-Bibliothek verfügbare SDK ermöglicht die Erstellung von Quanten-Gattern und -Schaltkreisen mit `braket.circuits` und bietet ein Framework zur Interaktion mit Quanten-Hardwaregeräten [Ama23] [Amaa]. Durch die Funktion `Circuit()` können neue Quantenschaltkreise definiert werden. Beispielsweise fügt `circ.h(0)` ein Hadamard-Gatter zum ersten Qubit hinzu [Ama23]. Das SDK ermöglicht die Erweiterung von Quantenschaltkreisen, Definition eigener unitärer Gatter und gibt Information über die Geräteunterstützung [Amaa]. Braket adressiert und überwindet die Barrieren, die mit dem Zugang zu Quantenhardware verbunden sind, indem es einen zentralen Zugriff auf verschiedene Quantencomputing-Technologien bietet [Ama23].

Der Entwicklungsprozess in Braket ist in drei Schritte unterteil [Ama23]: Erstellung, Überprüfung und Implementierung. Dabei werden vollverwaltete Jupyter-Notebook-Umgebungen bereitgestellt und der Zugriff auf effiziente Quantenschaltkreis-Simulatoren sowie eine Vielzahl von Quantencomputern, einschließlich QPUs von IonQ, Oxford Quantum Circuits (OQC), QuEra und Rigetti, ermöglicht. Zusätzlich zu den QPUs bietet Braket auch verschiedene On-Demand- und lokale Simulatoren sowie einen integrierten Simulator an.

Amazon Brakets lokaler Simulator unterstützt OpenQASM 3.0, wobei diese Unterstützung durch das Modul `braket.ir.openqasm` bereitgestellt wird [Ama23]. Während in Amazon Braket mit Quantenschaltkreisen im OpenQASM 3.0 Format gearbeitet werden kann, treten bei der Arbeit mit Quantenschaltkreisen im OpenQASM 2.0 Format auf dem Local Simulator einige Fehler auf. Da die Unterstützung von OpenQASM 2.0 nicht vollständig gegeben ist wurde für Braket zunächst als Cirq-Quantenschaltkreis importiert um ihn dann mittels der Methode to_braket in einen Braket-Quantenschaltkreis umzuwandeln [Dev23b] [Dev23c].

Bei der Simulation von OpenQASM-Code auf der Amazon Braket Plattform traten Umwandlungsfehler oder 'Instruction'-Probleme auf. Allerdings wurden im Rahmen dieser Arbeit bei der Verwendung einfacher Quantenschaltkreise meist Ergebnisse erzielt. Diese Konvertierung mithilfe qBraid [Dev23a] wird in Listing 4.3 im Simulationscode mit dem Braket Local Simulator präsentiert.

In Amazon Braket führt beispielsweise der Befehl `device = LocalSimulator()` zur gleichen Initialisierung des `Local state vector simulator` wie andere äquivalente Befehle [KV22], [Ama23].

Der `Local state vector simulator`, auch bekannt als `braket_sv`, ist ein integraler Bestandteil des Amazon Braket SDK [Ama23]. Dieser lokale Simulator ermöglicht die Implementierung aller im Amazon Braket SDK verfügbaren Gatter, während QPU-Geräte nur eine begrenzte Untermenge davon unterstützen [Ama23]. Informationen zu den von einem Gerät unterstützten Gatter können in den Geräteeigenschaften gefunden werden [Ama23].

Der `Local density matrix simulator` ermöglicht das Simulieren von Quantenschaltkreisen mit Rauschen, indem er verschiedene Gatter-Noise-Operationen einsetzt [Ama23]. Zudem können Noise-Operationen auch auf spezifische Qubits und Gatter in bestehenden Quantenschaltkreisen angewendet werden, die sowohl mit als auch ohne Noise betrieben werden sollen. Abhängig von der Anzahl der spezifizierten Shots kann der braket_dm-Simulator reduzierte Dichtematrizen ausgeben (wenn Shots = 0) [Amab] [Ama23].

Listing 4.3 Simulation von OpenQASM 2.0 Quantenschaltkreisen mit dem Amazon Local Simulator

```
from braket.circuits import Circuit
from braket.devices import LocalSimulator
from qbraid.transpiler.cirq_qasm import from_qasm
from qbraid.transpiler.cirq_braket import to_braket
import matplotlib.pyplot as plt


# QASM code
openqasm2_code = """

OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg meas[2];
h q[0];
cx q[0],q[1];
barrier q[0],q[1];
measure q[0] -> meas[0];
measure q[1] -> meas[1];
"""

# QASM to Cirq
cirq_circuit = from_qasm(openqasm2_code)

# Cirq to Braket
braket_circuit = to_braket(cirq_circuit)

device = LocalSimulator()
result = device.run(braket_circuit, shots=1000)
counts = result.result().measurement_counts
print(counts)
plt.show()
```

Amazon Braket kann verschiedene Arten von Ergebnissen zurückgeben, wenn ein Quantenschaltkreis gemessen wird [Ama23]. Bei Verwendung des lokalen Simulators kann der *Amplitude*-Ergebnistyp die Amplituden spezifizierter Quantenzustände in der Ausgangswellenfunktion zurückgeben, während der *StateVector*-Ergebnistyp den vollständigen Zustandsvektor zurückgibt [Ama23]. Bei der Verwendung der Statevector-Methode zeigt das Simulationsergebnis den Quantenzustand des Quantenschaltkreises bis zu den Messbefehlen an, selbst bei Vorhandensein eines `measure`-Befehls im OpenQASM-Format. Der Statevector-Simulator von Amazon Braket gibt konkret den Zustand direkt vor den Messbefehlen zurück.

Der Quantenschaltkreis wird wie im Simulationscode in Listing 4.3 dargestellt auf dem lokalen Simulator mit 1000 Shots ausgeführt. Die Counts-Ergebnisse können anschließend mit der `measurement_counts` Methode abgerufen werden. Die Ausgabe zeigt sich wie folgt: `Counter('11': 526, '00': 474)`.

5 Implementierung und Leistungsvergleich

In diesem Kapitel wird die Integration der Simulatoren als Plugins in QHAna vorgenommen und es werden Leistungsvergleiche zwischen Simulatoren durchgeführt. Darüber hinaus werden Analysen in verschiedenen Szenarien präsentiert.

5.1 Integration in QHAna

In Kapitel 4 wird jeder der ausgewählten Simulatoren detailliert untersucht. Nach der Auswahl wurden Simulatoren mit OpenQASM-Unterstützung schrittweise als Plugins in die QHAna-Plattform integriert. Zunächst wurden die Simulatoren individuell in einer Jupyter Notebook-Umgebung getestet. Das Hauptziel dieses Tests bestand darin zu überprüfen, ob jeder Simulator eine durch OpenQASM vorgegebener Quantenschaltkreis korrekt simulieren und das counts-Ergebnis zurückgeben kann. Nach erfolgreicher Validierung in der Jupyter Notebook-Umgebung begann die Integration der Simulatoren als Plugins im Plugin Runner Repository.

Während des Integrationsprozesses traten einige Fehler auf. Die meisten dieser Fehler standen in Zusammenhang mit dem Lesen der QASM-Datei und der Rückgabe des Ergebnisses im JSON-Format. Diese Herausforderungen wurden systematisch identifiziert und behoben.

Nach erfolgreicher Integration des ersten Simulators in QHAna wurde dessen Fähigkeit zur statevector-Simulation separat getestet. Nach erfolgreichen Tests wurde diese Simulationsoption zu QHAna hinzugefügt. Dieser Prozess wurde wiederholt, bis alle anderen Simulatoren integriert waren.

Im Laufe der Zeit wurden aufgrund verschiedener Anforderungen zusätzliche Funktionen zu den Simulatoren hinzugefügt.

Die Schnittstelle, die in Abbildung 5.1 dargestellt ist, bietet Zugriff auf die Plugins und Funktionen der QHAna-Plattform. Nach der Integration ausgewählter Simulatoren in das QHAna-System werden diese in der Plugin-Liste angezeigt. Benutzer können durch Auswahl des File Upload-Plugins eine qasm-Datei hochladen. Anschließend kann der gewünschte Simulator aus der Plugin-Liste ausgewählt und die Simulation gestartet werden. Die qasm-Datei wird dem Simulator als URL geben. Über den QHAna Plugin Runner ist der Zugriff auf Plugins mittels GET und POST Methoden möglich.

Die Simulationsergebnisse werden im JSON-Format zurückgegeben. Diese Dateien enthalten Simulations-Metadaten, Counts wie in der Listing 5.1 und Statevector Ergebnisse.

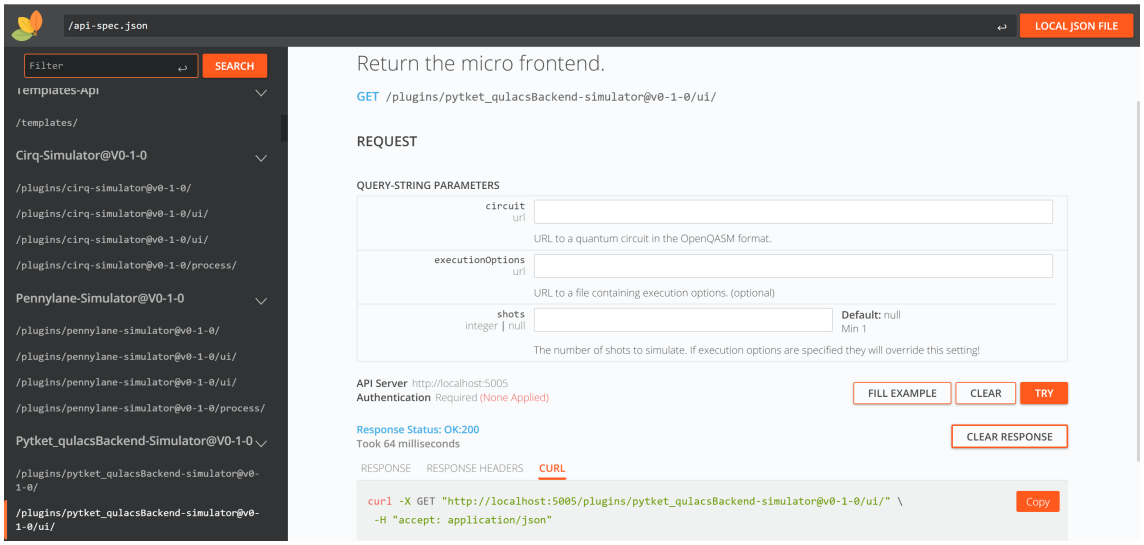


Abbildung 5.1: Ein Ansicht von einem Teil der Plugin-Liste durch das RapiDoc-Interface

Listing 5.1 Simulationsergebnisse im JSON-Format


```
{
  "log": "Finished simulating circuit.",
  "outputs": [
    {
      "contentType": "application/json",
      "dataType": "provenance/trace",
      "href": "http://localhost:5005/files/2/?file-id=HgPK1bYXNIlyx2EvsvlTBYH6fXjUZf1HYYX9FeQSnjY",
      "name": "result-trace.json"
    },
    {
      "contentType": "application/json",
      "dataType": "entity/vector",
      "href": "http://localhost:5005/files/3/?file-id=uQK2PoZ6UceK3kyhfLOsaF_QJj0R0ijbQUA5ffYj8bM",
      "name": "result-counts.json"
    },
    {
      "contentType": "application/json",
      "dataType": "entity/vector",
      "href": "http://localhost:5005/files/4/?file-id=qZDqRrcwM1FRhjwH2c3Om_1UfqbhgMHtvciQvYVQhpE",
      "name": "result-statevector.json"
    },
    {
      "contentType": "application/json",
      "dataType": "provenance/execution-options",
      "href": "http://localhost:5005/files/5/?file-id=NHSYtLKfC4m3p29856v0bYv56ZQhcKR0RG0-P2OvwmI",
      "name": "execution-options.json"
    }
  ],
  "status": "SUCCESS"
}
```

Während der Durchführung der zuvor erwähnten Statevector-Simulation wurden Maßnahmen ergriffen, um Unterschiede zwischen den Simulatoren und potenzielle Komplikationen bei einigen Simulationen zu vermeiden. Wenn der OpenQASM-Code keine Messanweisungen enthält und keine klassischen Bits oder Register definiert sind, resultiert die Ausgabe der Counts in einem JSON-Objekt wie in der Listing 5.2. Dieses enthält einen leeren String als Eintrag und die Anzahl der Shots als Wert.

Listing 5.2 Beispielausgabe der Counts eine Simulation.

```
{"": 1024, "ID": "4a0fd72c-3c75-46b8-9c02-9c14641a76b7"}
```

5.2 Experiment

Nach der Integration der ausgewählten Simulatoren in das QHAna-System wurden alle Experimente auf diesem System durchgeführt. Die integrierten Simulatoren können typischerweise zwei Arten von Ergebnissen zurückgeben: `statevector` und `counts`. In den durchgeführten Experimenten wurde jedoch die Ausführungszeit der verschiedenen Simulatoren bei der Erzeugung von `counts` Ergebnissen mit einer festgelegten Anzahl von Shots evaluiert.

In diesem Zusammenhang ist es wichtig zu beachten, welche Backends bei Simulatoren wie MQT-DDSIM, Qiskit Aer und Amazon Braket Local verwendet wurden. Die in den Experimenten verwendeten Simulatoren sind: Qiskit Aer_QasmSimulator, CirqSimulator, Pyket_QulacsBackend, PennyLane default.qubit, Amazon Braket Local Statevector Simulator und MQT-DDSIM_QasmSimulator. In den Experimenten wurden Quantenschaltkreise im OpenQASM 2.0 Format verwendet. Diese Quantenschaltkreisen wurden mithilfe einer Benchmark-Suite [QBW23] erzeugt. Sie wurden leicht modifiziert, um sicherzustellen, dass sie von jedem Simulator korrekt ausgeführt werden, da manche der ursprünglich erzeugten Instruktionen nicht von jedem Simulator unterstützt werden. Die Experimente wurden mit zwei verschiedenen OpenQASM-Codes mit 10 und 15 Qubits durchgeführt. Die während des Experiments verwendete Zeiteinheit ist als Nanosekunde angegeben und aus Gründen der visuellen Verständlichkeit ist sie auf den Grafiken in der oberen linken Ecke als Exponent dargestellt.

Das erste Experiment, das in Abbildung 5.2 dargestellt ist, wurde mit sechs verschiedenen Simulatoren durchgeführt. Die verwendeten Quantenschaltkreisen basieren auf einem zufällig generierten 10-Qubit OpenQASM 2.0 Code. Die in diesem Quantenschaltkreis verwendeten Gatter sind: `rz`, `sx`, `x`, `cx` und `measure`. Anschließend wurde dieser modifizierte Quantenschaltkreis in allen Simulatoren getestet. Die im Experiment erfassten Zeiten stellen die Zeitspanne dar, die die Simulatoren benötigen, um die `counts` Ergebnisse in Nanosekunden für 1000 shots zu generieren. Für jeden Simulator wurde das Experiment zehn Mal wiederholt. Während dieser Experimente wurden nur die für die `counts`-Ergebnisse benötigten Zeiten aufgezeichnet. Die schwarzen Punkte im Diagramm repräsentieren die einzelnen Zeitaufzeichnungen aus den zehn Wiederholungen für jeden Simulator. Die horizontalen Linien innerhalb des Boxplots zeigen den Median der Simulationszeiten an. Wenn der Abstand zwischen den oberen und unteren Grenzen der Box gering ist, zeigt dies eine konsistente Simulatorleistung an. Ein breiterer Abstand deutet hingegen auf Schwankungen in den Simulatorergebnissen hin.

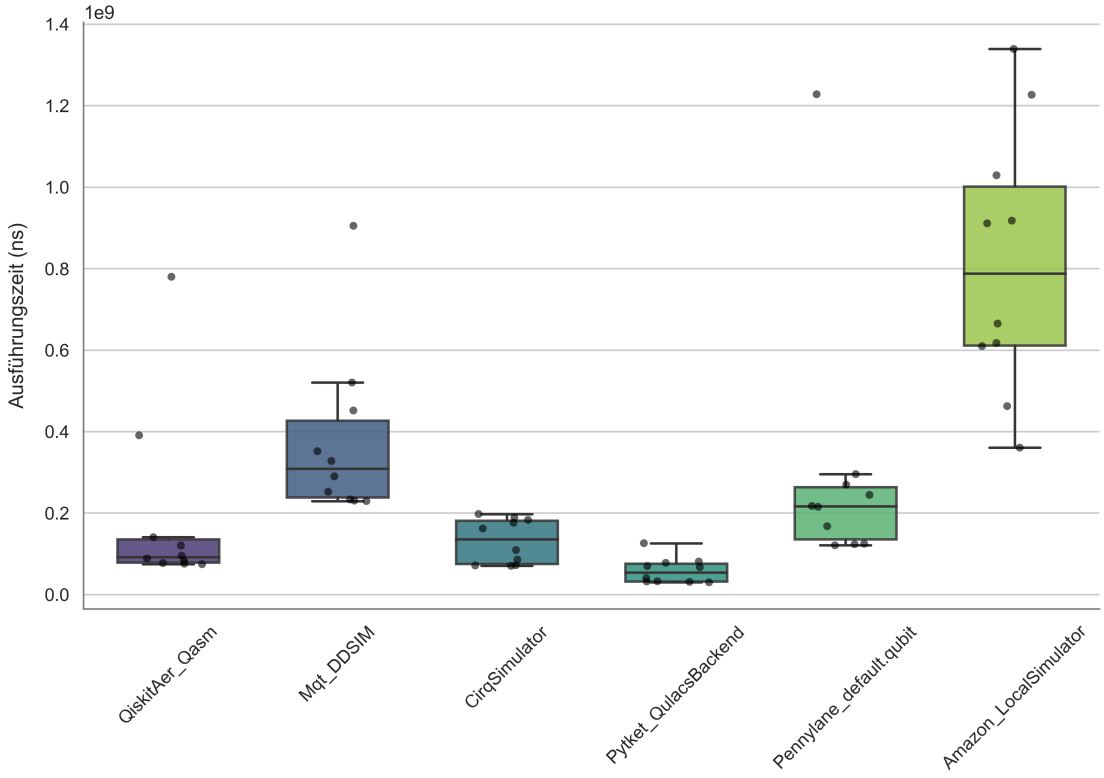


Abbildung 5.2: Vergleich der Ausführungszeiten der Quantensimulatoren.

Um den Einfluss der Anzahl von `shots` auf die Ergebnisse zu bestimmen, wurden die Messungen mit einer erhöhten Anzahl an `shots` wiederholt (Abbildung 5.3). In diesem Experiment wurden 1000000 Shots in 10 Wiederholungen berechnet, um die Verteilung der Ergebnisse zu bestimmen.

In einem weiteren Experiment, um den Einfluss der Anzahl an Qubits auf die Ergebnisse zu untersuchen, wurden die Messungen mit einem 15-Qubit OpenQASM-Code aus [QBW23] durchgeführt. Der Code enthält die folgenden Operationen: `rz`, `sx`, `x`, `cx`, `measure`. Amazon Braket war aufgrund von Inkompatibilität mit diesem komplexen OpenQASM-Code nicht für das Experiment verwendbar. Dennoch wurden in diesem Experiment vier Simulatoren eingesetzt. Die Ergebnisse des Experiments werden in 5.4 dargestellt. Da der MQT-DDSIM Simulator in diesem Experiment eine erheblich längere Ausführungszeit aufweist, werden in Abbildung 5.5 die Ergebnisse für die drei anderen Simulatoren dargestellt. Dadurch können die Unterschiede in den Ausführungszeiten genauer verglichen werden.

Aufgrund der hohen Ausführungszeit wurde dieses Experiment für den MQT-DDSIM_QasmSimulator einmal und für die anderen drei Simulatoren zweimal wiederholt.

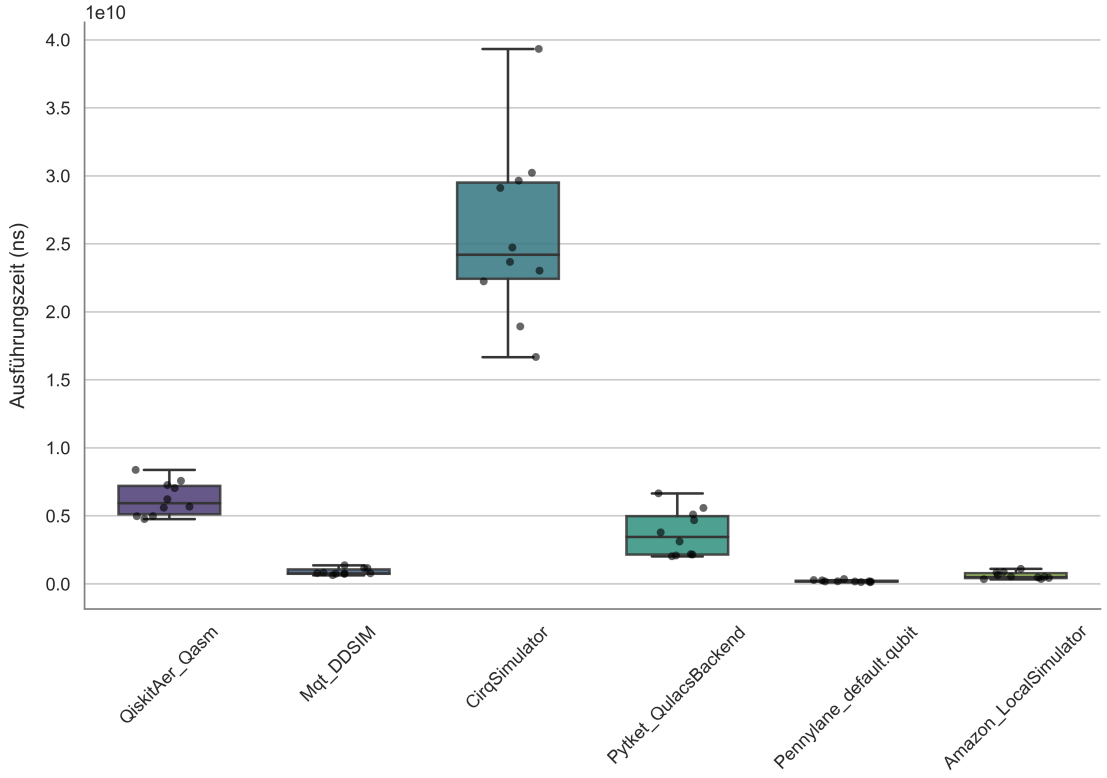


Abbildung 5.3: Vergleich der Ausführungszeiten der Quantensimulatoren mit 10 Qubits 1000000 Shotsanzahl.

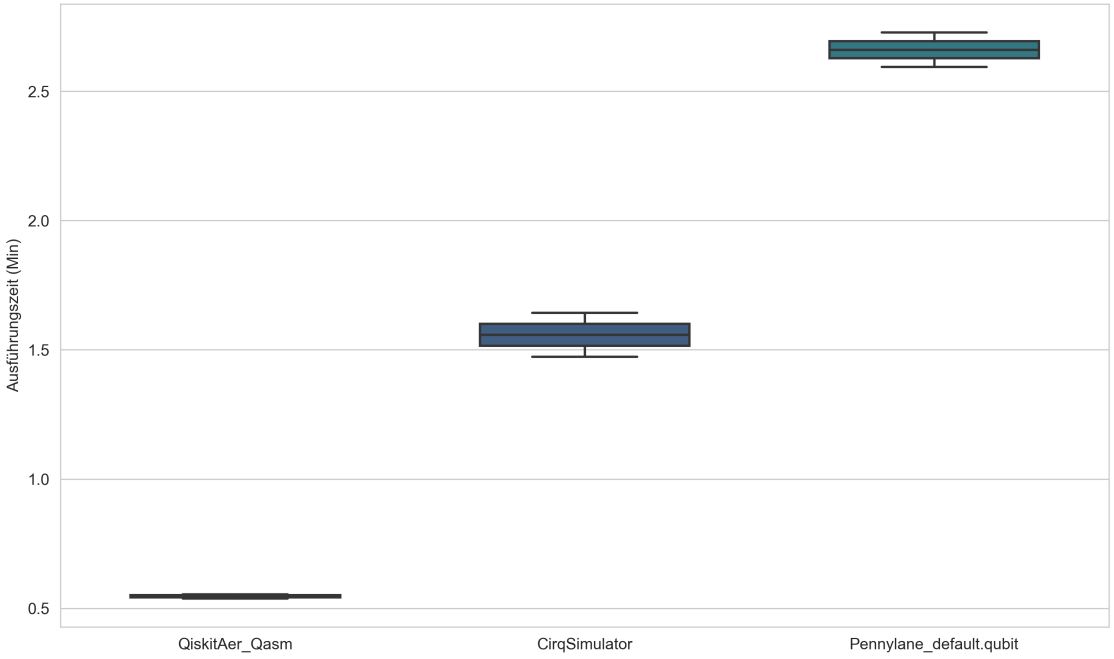


Abbildung 5.5: Vergleich der Ausführungszeiten der Quantensimulatoren in Minuten.

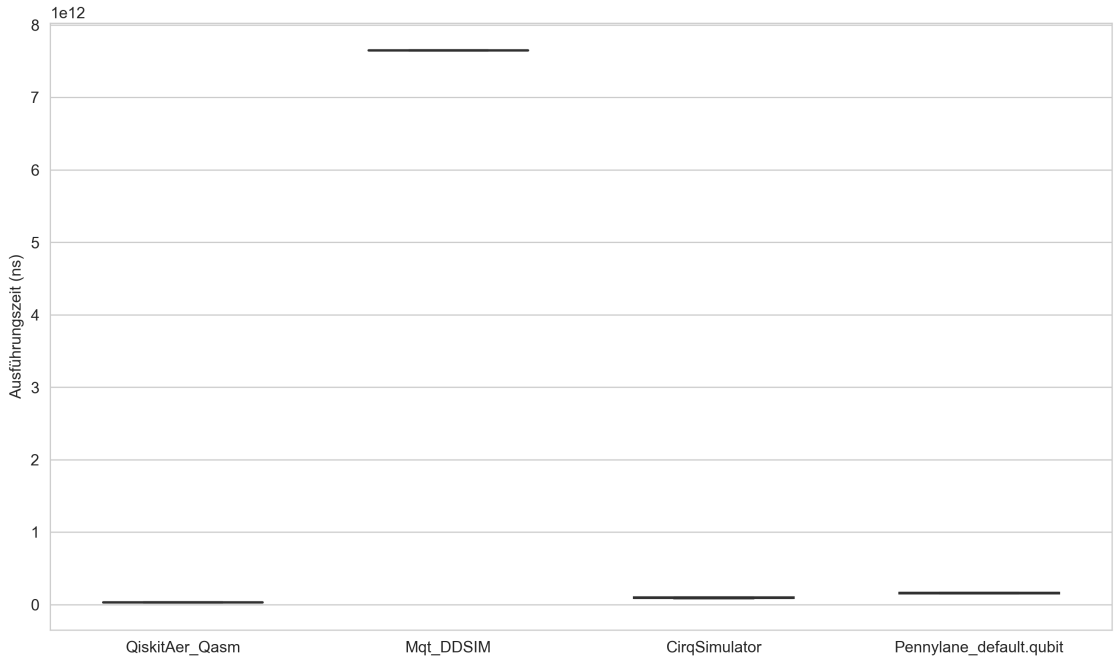


Abbildung 5.4: Vergleich der Ausführungszeiten der Quantensimulatoren mit 15 Qubits und 1000 Shotsanzahl.

5.3 Ergebnisse und Diskussion

5.3.1 Ergebnisse

Im Rahmen dieser Untersuchung wurden gezielte Experimente mit definierten OpenQASM 2.0-Codes durchgeführt, um die Performanz verschiedener Quantensimulatoren zu analysieren. Die dargestellten Ergebnisse, insbesondere die präsentierten Boxplot-Grafiken und die darin horizontal angezeigten Mediane, bieten detaillierte Einblicke in die Ausführungszeiten der Quantensimulatoren. In der Bewertung zeigt ein niedriger Medianwert, dass die Simulationslaufzeit des Simulators schneller ist, während ein größerer Abstand zwischen den Quartilgrenzen auf weniger konsistente Ergebnisse hindeutet. Unter Berücksichtigung aller Ausführungszeiten und der Medianwerte der verschiedenen Simulatoren kann man erkennen, dass Simulator Pytket_QulacsBackend mit einem Median von nur 54122600 Nanosekunden (0,054123 Sekunden) nicht nur am schnellsten arbeitet, sondern auch eine hohe Konsistenz in seinen Ausführungszeiten aufzeigt. Die Qiskit Aer_Qasm und CirqSimulator zeigen mit Medianwerten von 92213000 bzw. 135912900 Nanosekunden (0,092213 bzw. 0,135913 Sekunden) ebenfalls eine schnelle und stabile Leistung, was auf eine geringe Variabilität in ihren Ausführungszeiten hindeutet. Der PennyLane default.qubit arbeitet mit einem Medianwert von 216588150 Nanosekunden (0,216588 Sekunden) relativ schnell, weist jedoch im Vergleich zu den Qiskit Aer_Qasm und CirqSimulator einige Schwankungen in seinen Laufzeiten auf, was auf eine leicht erhöhte Variabilität in seiner Leistung anzeigt. Im Experiment hat der Amazon Braket Local-Simulator mit Medianwerten von 788274150 Nanosekunden (0,788274 Sekunden) die längste Laufzeit. Ihm folgt der MQT-DDSIM Simulator mit 309182500 Nanosekunden (0,309183 Sekunden) und es wurde beobachtet, dass beide im Vergleich zu anderen Simulatoren langsamer

arbeiten. Tatsächlich hat der Amazon Braket Local Simulator unter diesen Experimentsbedingungen die inkonsistentesten Ergebnisse geliefert, was darauf hinweist, dass er in einigen Fällen schnell und in anderen langsam arbeitet.

Bei der Analyse des zweiten Boxplot-Diagramms (5.3) zeigt sich, dass mit der Erhöhung der Anzahl der Shots auf 1.000.000 die Simulationszeiten insgesamt zugenommen haben. Dabei wurde beobachtet, dass die Simulatoren MQT-DDSIM und Amazon Braket Localsimulator, die in den vorherigen Ergebnissen weniger effizient erschienen, bei erhöhter Shot-Anzahl nun eine deutlich bessere Performance im Vergleich zu den anderen Simulatoren aufwiesen. Diese Simulatoren lieferten zudem deutlich konstantere Ergebnisse als die anderen Simulatoren. Dagegen hatte der Simulator CirqSimulator einen Rückgang in der Konsistenz und Performance im Vergleich zu den anderen Simulatoren.

In Abbildung 5.4 sind die Ergebnisse eines weiterführenden Experiments dargestellt. Hierbei wurde ein OpenQASM 2.0-Quantenschaltkreis mit 15 Qubits und erhöhter Komplexität genommen. Im Vergleich zu vorherigen Untersuchungen lässt sich ein genereller Anstieg in den Simulationszeiten über alle Simulatoren beobachten. Besonders auffällig ist die erheblich verlangsamte Geschwindigkeit des Simulators MQT-DDSIM im Vergleich zu den anderen. Zur besseren Visualisierung der Unterschiede wurde in Abbildung 5.5 der Simulator MQT-DDSIM ausgelassen, um die herausragende Effizienz des Simulators Qiskit Aer_QasmSimulator deutlicher zu zeigen.

5.3.2 Diskussion

Bei der Analyse der Ergebnisse aus den durchgeführten Experimenten werden Unterschiede in den Leistungen der einzelnen Simulatoren offensichtlich.

Der Qiskit Aer_QasmSimulator hat in allen Szenarien im Vergleich zum Durchschnitt eine gute Leistung gezeigt. Insbesondere seine Fähigkeit, den OpenQASM 2.0 Code fehlerfrei zu lesen und bei zunehmender Komplexität von Quantenschaltkreisen, schneller als andere Simulatoren Ergebnisse zu liefern, ist bemerkenswert. Es wurde beobachtet, dass der MQT-DDSIM im Allgemeinen langsamer als andere Simulatoren arbeitet aber interessanterweise zeigt er mit einer erhöhten Anzahl von Shots eine bessere Leistung im Vergleich zu anderen. Für Experimente mit einer hohen Anzahl von Shots könnte er daher besonders geeignet sein. Auf der anderen Seite arbeitet CirqSimulator bei 1000 Shots relativ schnell, aber bei einer Erhöhung der Shots wird ein deutlicher Anstieg der Ausführungszeiten im Vergleich zu anderen Simulatoren beobachtet. Zusätzlich zeigte CirqSimulator Schwierigkeiten bei der Verarbeitung bestimmter OpenQASM-Codes. Daher könnte die Auswahl dieses Simulators in Abhängigkeit von den spezifischen Gattern des Quantenschaltkreises vorteilhaft sein. Während des ersten Tests demonstrierte der Simulator pytket_QulacsBackend eine beeindruckende Geschwindigkeit und Konsistenz in der Simulation. Trotz der schnellen Generierung der `counts`-Ergebnisse verlängerte sich die Dauer zur Ergebnisermittlung bei komplexeren Quantenschaltkreisen. Zum Beispiel dauerte bei einem 10-Qubit-Experiment die `counts`-Simulation lediglich ungefähr 0,07 Sekunden, wohingegen die Gesamtausführung des Codes nahezu 6,21 Sekunden in Anspruch nahm. Eine plausible Erklärung hierfür könnte die Anwendung der `get_compile`-Methode im Simulationscode sein, die zusätzliche Kompilierungszeiten erfordert. Für eine tiefergehende Bewertung sind detailliertere Untersuchungen erforderlich. Des Weiteren besteht die Möglichkeit, dass der Qulacs-Simulator mit OpenQASM 2.0 Code auch ohne den Einsatz von Pytket effektiver agiert. Die beachtenswerte Leistung des Simulators bei der Verarbeitung des 10-Qubit-Codes mittels Pytket

darf allerdings nicht vernachlässigt werden. Der PennyLane default.qubit Simulator hat ebenfalls eine starke Performance in den Tests gezeigt, allerdings unterstützt der Simulator `Measure`-Gatter im OpenQasm Code nicht. Obwohl der Amazon Braket Local Simulator im Vergleich zu anderen Simulatoren langsamer und inkonsistenter ist, hat er bei einer steigenden Anzahl von Shots eine bessere Leistung als die anderen gezeigt. Am Ende kann man sehen, dass jeder Simulator seine eigenen Stärken und Beschränkungen hat, die je nach Anwendungskontext und Anforderungen bewertet werden sollten.

6 Verwandte Arbeiten

In diesem Kapitel werden einige der relevanten Literatur und Technologien für diese Arbeit vorgestellt.

Quetschlich et al. [QBW23] haben MQT Bench vorgestellt, einen Bestandteil des Munich Quantum Toolkits (MQT). Diese Benchmark-Suite integriert verschiedene Algorithmen, Compiler, native Gatter-Sets und Zielgeräte und umfasst insgesamt mehr als 70.000 Benchmark-Quantenschaltkreise, die von 2 bis 130 Qubits auf vier Abstraktionsebenen reichen. Um die umfangreiche Menge von Benchmarks zu verwalten, bieten die Autoren sowohl eine Web-Oberfläche als auch ein Python-Paket an. MQT Bench zielt darauf ab, empirische Bewertungen für den gesamten Quantensoftware-Stack zu erleichtern.

Bergholm et al. [Ber22] haben PennyLane als ein leistungsstarkes Werkzeug für Forschungen im Bereich der Quantencomputing und Quantenmaschinenlernen vorgestellt. Dieses Tool bietet Möglichkeiten zur Erforschung von variationalen Quantenschaltkreisen und Quantenalgorithmen.

Barzen und Leymann [BL22] betonten die Komplexität im Entwicklungsprozess von Quantenanwendungen. Sie erklärten, dass es einen Bedarf an Unterstützungswerkzeugen beim Vergleich verschiedener Implementierungen von Quanten- und klassischen Algorithmen gibt. In diesem Kontext wurde QHAna als ein nützliches Werkzeug im Bereich des quantenmaschinellen Lernens vorgestellt.

Pandey et al. [PMSF23] haben die Prinzipien des Quantencomputings untersucht, welches auf Konzepten der Quantenphysik basiert und Informationen durch Quantenzustände repräsentiert. In ihrer Arbeit implementieren und simulieren sie diese Prinzipien mithilfe von Open-Source-Softwareentwicklungskits wie Cirq und Qiskit. Dabei gehen sie auf die Implementierungsdetails von Quantenprinzipien ein und bieten Darstellungen wie Visual Circuit, State vector, Q-sphere und Visual Probabilistic.

Sekić und Yakaryılmaz [SY22] evaluierten SDKs für Quantenalgorithmen wie Qiskit, ProjectQ, Cirq und Forest. Ihre Untersuchung konzentrierte sich auf die Generierung, Durchführung, Messung und Visualisierung von Quantenschaltkreisen, klassische und quantenmechanische Operationen.

Suzuki et al. [Suz21] stellten Qulacs vor, einen schnellen und vielseitigen Simulator. Sie präsentierten das Grundkonzept und die beabsichtigten Anwendungen von Qulacs. Die Effizienz des Qulacs-Simulators wurde unter verschiedenen Bedingungen wie Einzel-Thread- und Multi-Thread-Verarbeitung sowie GPU-Beschleunigung im Vergleich zu anderen vorhandenen Bibliotheken wie Qiskit untersucht. Beobachtungen zeigen, dass Qulacs in vergleichenden Untersuchungen eine hohe Ausführungsgeschwindigkeit bei einer kleinen Qubit-Anzahl zeigt, auch ohne zusätzliche Optimierung.

In einer Untersuchung [KLB+23] von Kim et al. wurden verschiedene Quantencomputing-Umgebungen und Frameworks für Quanten-Neuralen Netzwerke analysiert. Sie verglichen die Performance von Simulatoren wie Qiskit (mit QASM und Statevector), Amazon Braket und Pennylane. Dabei wurden Unterschiede in der Verarbeitungszeit je nach Anzahl der Qubits festgestellt.

LaRose [LaR19b] hat verschiedene Quantensoftware-Plattformen analysiert und miteinander verglichen. Er untersuchte Anforderungen, Installation, Sprachsyntax, Bibliotheksunterstützung sowie die Performance von Quantensimulatoren. Zudem betrachtete er für unterstützte Plattformen die zugehörige Hardware, Quanten-Assembler-Sprachen und Compiler. In einer weiteren Untersuchung [LaR19a] analysierte LaRose die Leistungsfähigkeit von Cirqs Simulatoren mithilfe von zufälligen Quantenschaltkreisen mit unterschiedlichen Qubit Zahlen. Er betrachtete die Dokumentation, Sprachsyntax und wie Cirq Quantencomputer und Simulatoren unterstützt. Des Weiteren untersuchte er fortgeschrittene Funktionen von Cirq und stellte beispielhafte Anwendungen vor.

7 Zusammenfassung und Ausblick

Diese Arbeit zeigt, wie sich Quantencomputer von klassischen Computern unterscheiden, indem sie auf grundlegenden Prinzipien der Quantenphysik basieren, wie zum Beispiel der Superposition und Verschränkung. Es wird gezeigt, wie diese Prinzipien in Quantensimulatoren berücksichtigt werden und sich in den Ergebnissen von Statevector und Counts zeigen, unterstützt durch visuelle Darstellungen.

QHAna bietet eine Plattform, die es ermöglicht, verschiedene Quantenalgorithmen durch die Verwendung von Plugins zu untersuchen und zu simulieren. Neben einem bestehenden Plugin für Qiskit wurden in dieser Arbeit weitere, Python-unterstützte Quantensimulatoren systematisch evaluiert und als neue Plugins integriert. Für die Integration in QHAna wurden verschiedene Quantensimulatoren mit Python-Unterstützung systematisch untersucht. Zu diesen Simulatoren gehören Qiskit Aer, CirqSimulator, MQT-DDSIM, Pytket_QulacsBackend, PennyLane default.qubit und Amazon Braket Local Simulator. Nach einer Literaturrecherche wurden bestimmte Simulatoren einzeln lokal getestet. Diese Simulatoren wurden basierend auf verschiedenen Kriterien für die Integration in das QHAna-System ausgewählt. Ein entscheidendes Auswahlkriterium war die Unterstützung von OpenQASM 2.0. Dadurch konnten die Ergebnisse aus verschiedenen Simulatoren mit derselben Eingabe leicht verglichen werden. Basierend auf den erhaltenen Ergebnissen und den Eigenschaften der Simulatoren wurde in Kapitel 4 eine Vergleichstabelle der Simulatoren präsentiert. Die Kriterien dieser Tabelle basieren hauptsächlich auf den von den Simulatoren unterstützten Funktionen. Sie berücksichtigt ebenfalls die während der Implementierungsphase gewonnenen Erkenntnisse, wie die diversen Ausgabeformen des Statevectors durch unterschiedliche Simulatoren. Diese Tabelle zielt darauf ab, dem Nutzer die Auswahl eines geeigneten Simulators zu erleichtern. Nach einer detaillierten Untersuchung wurden diese Simulatoren in das QHAna-System integriert und liefern zwei verschiedene Ergebnistypen: Counts und Statevector. Mit diesen Simulatoren wurden verschiedene Experimente durchgeführt und ihre Performance mit gegebenen OpenQASM-Codes ausgewertet.

Diese Arbeit ist für Anwender von Quantensimulatoren von bedeutung, da sie es in zukunft ermöglicht einen für spezifische Bedürfnisse geeigneten Simulator auszuwählen. Die Integration als Plugin innerhalb von QHAna macht außerdem den auswahl von Quantensimulatoren für die Benutzer einfacher. Die Unterstützung von OpenQASM 2.0 in diesen Simulatoren ermöglicht es, Quantenschaltkreise mittels QHAna und verschiedener Simulator-Plugins direkt zu simulieren, ohne eine Konvertierung vornehmen zu müssen. Zukünftige Entwicklungen in QHAna könnten sowohl Erweiterungen für OpenQASM 3.0 Unterstützung als auch neue Plugins für Simulatoren ohne OpenQASM-Unterstützung umfassen. Die am Ende dieser Arbeit durchgeführten Experimente haben zudem die Leistung der Simulatoren verglichen, wobei das Verhalten des jeweiligen Simulators in verschiedenen Szenarien entscheidend war. Im ersten Experiment lieferte der Pytket_QulacsBackend Simulator schnell und konsistente Ergebnisse, während die Simulatoren MQT-DDSIM und Amazon Braket im Vergleich langsamer waren. Andererseits verbesserten die Simulatoren, die zunächst länger für die Ausführungszeit brauchten, ihre Leistung erheblich, nachdem die Anzahl der Shots

erhöht wurde. Der Qiskit Aer_QasmSimulator hat im anderen Experiment mit einem komplexeren Quantenschaltkreis von 15 Qubits im Vergleich zu den CirqSimulator und PennyLane default.qubit eine bessere Leistung gezeigt und funktionierte problemlos mit dem OpenQASM-Programm. Die Simulatoren können je nach Geschwindigkeit, Konsistenz und Kompatibilität mit OpenQASM, der Anzahl der Qubits oder einer hohen Anzahl an Shots unterschiedliche Ergebnisse zeigen. Dies zeigt die Wichtigkeit, die Wahl des Simulators entsprechend der Forschungsziele, der spezifischen Eingabedaten oder der Komplexität der Quantenschaltkreise zu treffen.

Literaturverzeichnis

[Abr+19] H. Abraham et al. *Qiskit: An Open-source Framework for Quantum Computing*. 2019. URL: https://doi.org/10.5281/zenodo.2562110 (zitiert auf S. 37).

[ACCB23] A. Andreev, G. H. Cattan, S. Chevallier, Q. Barthélemy. „pyRiemann-qiskit: A Sandbox for Quantum Classification Experiments with Riemannian Geometry“. In: *Research Ideas and Outcomes* 9.e101006 (2023) (zitiert auf S. 37).

[AJLT22] S. Agarwal, G. A. Jaoude, A. Leider, C. C. Tappert. „Comparing Quantum Computing Platforms“. In: *Advances in Information and Communication*. Hrsg. von K. Arai. Cham: Springer International Publishing, 2022, S. 423–441 (zitiert auf S. 19, 37).

[Amaa] Amazon Braket. *Amazon Braket SDK for Python*. URL: https://github.com/amazon-braket/amazon-braket-sdk-python (zitiert auf S. 48).

[Amab] Amazon Braket. *Simulating Noise on Amazon Braket*. URL: https://github.com/amazon-braket/amazon-braket-examples/blob/main/examples/braket_features/Simulating_Noise_On_Amazon_Braket.ipynb (zitiert auf S. 48).

[Ama23] Amazon Web Services. *Amazon Braket Entwicklerhandbuch*. German. 2023. URL: https://docs.aws.amazon.com/pdfs/braket/latest/developerguide/braket-developer-guide.pdf#braket-send-to-local-simulator (zitiert auf S. 48, 49).

[Bar22] J. Barzen. „From Digital Humanities to Quantum Humanities: Potentials and Applications“. In: *Quantum Computing in the Arts and Humanities: An Introduction to Core Concepts, Theory and Applications*. Hrsg. von E. R. Miranda. Cham: Springer International Publishing, 2022, S. 1–52. ISBN: 978-3-030-95538-0. DOI: 10.1007/978-3-030-95538-0_1. URL: https://doi.org/10.1007/978-3-030-95538-0_1 (zitiert auf S. 17, 30).

[Ber22] Bergholm et al. *PennyLane: Automatic differentiation of hybrid quantum-classical computations*. 2022. arXiv: 1811.04968 [quant-ph] (zitiert auf S. 46, 47, 59).

[BFL18] J. Barzen, M. Falkenthal, F. Leymann. „Wenn Kostüme sprechen könnten: MUSE - Ein musterbasierter Ansatz an die vestimentäre Kommunikation im Film“. In: *Digital Humanities. Perspektiven der Praxis*. Berlin: Frank und Timme, Mai 2018, S. 223–241. ISBN: 978-3-7329-0284-2. URL: http://www2.informatik.uni-stuttgart.de/cgibin/NCSTRL/NCSTRL_view.pl?id=INBOOK-2018-05&engl=0 (zitiert auf S. 30).

[BL20] J. Barzen, F. Leymann. „Quantum Humanities: A First Use Case for Quantum-ML in Media Science“. In: *Digitale Welt* 4 (2020), S. 102–103. URL: https://doi.org/10.1007/s42354-019-0243-2 (zitiert auf S. 30).

[BL22] J. Barzen, F. Leymann. „Quantencomputing als Integrationsproblem: Quantenanwendungen sind in der Praxis immer hybride“. In: *Chancen und Risiken von Quantentechnologien: Praxis der zweiten Quantenrevolution für Entscheider in Wirtschaft und Politik*. Hrsg. von A. Wilms, F. Neukart. Wiesbaden: Springer Fachmedien Wiesbaden, 2022, S. 115–123. ISBN: 978-3-658-37534-8. DOI: 10.1007/978-3-658-37534-8_12. URL: https://doi.org/10.1007/978-3-658-37534-8_12 (zitiert auf S. 59).

[BLF+21] J. Barzen, F. Leymann, M. Falkenthal, D. Vietz, B. Weder, K. Wild. „Relevance of Near-Term Quantum Computing in the Cloud: A Humanities Perspective“. In: *Cloud Computing and Services Science*. Hrsg. von D. Ferguson, C. Pahl, M. Helfert. Cham: Springer International Publishing, 2021, S. 25–58. ISBN: 978-3-030-72369-9 (zitiert auf S. 20).

[CBSG17] A. W. Cross, L. S. Bishop, J. A. Smolin, J. M. Gambetta. *Open Quantum Assembly Language*. 2017. arXiv: 1707.03429 [quant-ph] (zitiert auf S. 17, 25, 27).

[Com22] E. F. Combarro. „Quantum Computing Foundations“. In: *Quantum Software Engineering*. Cham: Springer International Publishing, 2022, S. 1–24. ISBN: 978-3-031-05324-5. URL: https://doi.org/10.1007/978-3-031-05324-5_1 (zitiert auf S. 21–23).

[Des] C. for Design Automation TU Munich. *MQT: Munich Quantum Toolkit*. https://www.cda.cit.tum.de/research/quantum/mqt/ (zitiert auf S. 40).

[Des23] C. for Design Automation TU Munich. *Quantum Simulation*. Chair of Databases, Information Systems and Data Analysis. 2023. URL: https://www.cda.cit.tum.de/research/quantum_simulation/ (zitiert auf S. 17, 41, 42).

[Dev23a] qBraid Development Team. *qBraid SDK Overview*. 2023. URL: https://docs.qbraid.com/en/latest/sdk/overview.html (zitiert auf S. 48).

[Dev23b] qBraid Development Team. *qBraid Transpiler API Documentation*. qBraid. 2023. URL: https://docs.qbraid.com/en/latest/api/qbraid.transpiler.html#braket-conversions-qbraid-transpiler-cirq-braket (zitiert auf S. 48).

[Dev23c] qBraid Development Team. *qBraid Transpiler API Documentation*. qBraid. 2023. URL: https://docs.qbraid.com/en/latest/api/qbraid.transpiler.html#qasm-conversions-qbraid-transpiler-cirq-qasm (zitiert auf S. 48).

[DiV00] D. P. DiVincenzo. „The Physical Implementation of Quantum Computation“. In: *Fortschritte der Physik* 48.9-11 (2000), S. 771–783. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/1521-3978%28200009%2948%3A9/11%3C771%3A%3AAID-PROP771%3E3.0.CO%3B2-E (zitiert auf S. 19).

[FM20] A. Fatima, I. L. Markov. *Faster Schrödinger-style simulation of quantum circuits*. 2020. arXiv: 2008.00216 [quant-ph] (zitiert auf S. 24).

[GFH+23] T. Grurl, J. Fuß, S. Hillmich, L. Burgholzer, R. Wille. „Arrays vs. Decision Diagrams: A Case Study on Quantum Circuit Simulators“. In: *Secure Information Systems, University of Applied Sciences Upper Austria, Austria* (2023). http://iic.jku.at/eda/research/quantum/ (zitiert auf S. 19).

[Goo] Google Quantum AI. *Cirq: A Python library for designing, simulating, and running quantum circuits*. https://quantumai.google/cirq (zitiert auf S. 43).

[Goo23a] Google Quantum AI. *Basics of Cirq*. 2023. URL: https://quantumai.google/cirq/start/basics (zitiert auf S. 43, 44).

[Goo23b] Google Quantum AI. *Cirq Interoperability*. 2023. URL: https://quantumai.google/cirq/build/interop (zitiert auf S. 43).

[Goo23c] Google Quantum AI. *Cirq: final_density_matrix*. 2023. URL: https://quantumai.google/reference/python/cirq/final_density_matrix (zitiert auf S. 44).

[Goo23d] Google Quantum AI. *Simulation in Cirq*. 2023. URL: https://quantumai.google/cirq/simulate/simulation (zitiert auf S. 43, 44).

[Gro96] L. K. Grover. *A fast quantum mechanical algorithm for database search*. 1996. arXiv: quant-ph/9605043 [quant-ph] (zitiert auf S. 17).

[JP22] Y. Jungjarassub, K. Piromsopa. „A Performance Optimization of Quantum Computing Simulation using FPGA“. In: *2022 19th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*. 2022, S. 1–4. DOI: 10.1109/ECTI-CON54298.2022.9795571 (zitiert auf S. 19).

[JR20] F. Jendrzejewski, M. Rudolph. „Can we run quantum circuits on ultra-cold atom devices?“ In: *Authorea* (2020). URL: https://doi.org/10.22541%2Fau.159069218.83978492 (zitiert auf S. 46).

[KAF+] N. Khammassi, I. Ashraf, X. Fu, C. G. Almudever, K. Bertels. „QX: A High-Performance Quantum Computer Simulation Platform“. In: () (zitiert auf S. 24, 25).

[Kar05] I. Karafyllidis. „Quantum computer simulator based on the circuit model of quantum computation“. In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 52.8 (2005), S. 1590–1596. DOI: 10.1109/TCSI.2005.851999 (zitiert auf S. 27, 28).

[KLB+23] H. Kim, S. Lim, A. Baksi, D. Kim, S. Yoon, K. Jang, H. Seo. *Quantum Artificial Intelligence on Cryptanalysis*. Cryptology ePrint Archive, Paper 2023/004. 2023. URL: https://eprint.iacr.org/2023/004 (zitiert auf S. 46, 60).

[Kom20] B. Kommadi. *Quantum Computing Solutions: Solving Real-World Problems Using Quantum Computing and Algorithms*. 2020. ISBN: 978-1-4842-6515-4. URL: https://doi.org/10.1007/978-1-4842-6516-1 (zitiert auf S. 17, 27).

[KV22] A. Khan, M. Versaggi. *Quantum Computing Experimentation with Amazon Braket: Explore Amazon Braket quantum computing to solve combinatorial optimization problems*. Packt Publishing, 2022. ISBN: 9781800564091. URL: https://books.google.de/books?id=LqF-EAAAQBAJ (zitiert auf S. 48).

[LaR19a] R. LaRose. *Overview and Comparison of Gate Level Quantum Software Platforms*. Techn. Ber. März 2019. URL: https://quantumcomputingreport.com/review-of-the-cirq-quantum-software-framework/ (zitiert auf S. 43, 60).

[LaR19b] R. LaRose. „Overview and Comparison of Gate Level Quantum Software Platforms“. In: *Quantum* 3 (März 2019), S. 130. ISSN: 2521-327X. DOI: 10.22331/q-2019-03-25-130. URL: https://doi.org/10.22331/q-2019-03-25-130 (zitiert auf S. 60).

[LB20] F. Leymann, J. Barzen. „The bitter truth about gate-based quantum algorithms in the NISQ era“. In: *Quantum Science and Technology* 5.4 (Sep. 2020), S. 044007. DOI: 10.1088/2058-9565/abae7d. URL: https://dx.doi.org/10.1088/2058-9565/abae7d (zitiert auf S. 28).

[MP22] J. Mancilla, C. Pere. „A Preprocessing Perspective for Quantum Machine Learning Classification Advantage in Finance Using NISQ Algorithms“. In: *Entropy* 24.11 (Nov. 2022), S. 1656. URL: https://doi.org/10.3390/e24111656 (zitiert auf S. 47).

[MQT] MQT-DDSIM. *DDSim Installation Documentation*. Chair for Design Automation, Technical University of Munich. URL: https://mqt.readthedocs.io/projects/ddsim/en/latest/Installation.html (zitiert auf S. 40).

[MQT23a] MQT-DDSIM. *Circuit Simulator*. Chair for Design Automation. 2023. URL: https://mqt.readthedocs.io/projects/ddsim/en/latest/simulators/CircuitSimulator.html#circuit-simulator (zitiert auf S. 42).

[MQT23b] MQT-DDSIM. *NoiseAwareSimulator Documentation*. Chair for Design Automation. 2023. URL: https://mqt.readthedocs.io/projects/ddsim/en/latest/simulators/NoiseAwareSimulator.html (zitiert auf S. 42).

[MQT23c] MQT-DDSIM. *Usage of MQT DDSIM*. Chair for Design Automation. 2023. URL: https://mqt.readthedocs.io/projects/ddsim/en/latest/Usage.html (zitiert auf S. 43).

[NC10] M. Nielsen, I. Chuang. *Quantum Computation and Quantum Information*. Cambridge, UK, 2010. URL: https://www.cambridge.org/9781107002173 (zitiert auf S. 24).

[PMSF23] R. Pandey, P. Maurya, G. D. Singh, M. S. Faiyaz. „Simulating Quantum Principles: Qiskit Versus Cirq“. In: *Quantum Computing: A Shift from Bits to Qubits*. Springer Nature Singapore, 2023, S. 333–348. URL: https://doi.org/10.1007/978-981-19-9530-9_18 (zitiert auf S. 21, 59).

[QBW23] N. Quetschlich, L. Burgholzer, R. Wille. „MQT Bench: Benchmarking Software and Design Automation Tools for Quantum Computing“. In: *Quantum* (2023). MQT Bench is available at https://www.cda.cit.tum.de/mqtbench/ (zitiert auf S. 26, 39, 53, 54, 59).

[Qis] Qiskit. *Single Systems: Unitary*. URL: https://learn.qiskit.org/course/basics/single-systems#single-systems-unitary (zitiert auf S. 21–23).

[Qis20a] Qiskit. *Aer Provider*. Qiskit Development Team. 2020. URL: https://qiskit.org/documentation/stable/0.24/locale/de_DE/tutorials/simulators/1_aer_provider.html?highlight=aer_simulator_density_matrix (zitiert auf S. 39).

[Qis20b] Qiskit. *Qiskit NoiseModel Documentation*. Qiskit Development Team. 2020. URL: https://qiskit.org/documentation/stable/0.24/locale/de_DE/stubs/qiskit.providers.aer.noise.NoiseModel.html?highlight=noisemodel#qiskit.providers.aer.noise.NoiseModel (zitiert auf S. 39).

[Qis23a] Qiskit. *Getting Started with Qiskit*. Qiskit. 2023. URL: https://qiskit.org/documentation/tutorials/circuits/1_getting_started_with_qiskit.html (zitiert auf S. 39).

[Qis23b] Qiskit. *Getting Started with Qiskit*. 2023. URL: https://qiskit.org/documentation/getting_started.html (zitiert auf S. 39).

[Qis23c] Qiskit. *Plotting Data in Qiskit*. Qiskit. Okt. 2023. URL: https://qiskit.org/documentation/tutorials/circuits/2_plotting_data_in_qiskit.html (zitiert auf S. 39).

[Qis23d] Qiskit. *Qiskit API Documentation: OpenQASM 3*. 2023. URL: https://qiskit.org/documentation/apidoc/qasm3.html (zitiert auf S. 37).

[Qis23e] Qiskit. *Qiskit Introduction Tutorial*. 2023. URL: https://qiskit.org/documentation/locale/de_DE/intro_tutorial1.html (zitiert auf S. 37).

[Qis23f] Qiskit. *Qiskit Release Notes*. 2023. URL: https://qiskit.org/documentation/release_notes.html (zitiert auf S. 39).

[Qis23g] Qiskit. *QuantumCircuit Method Documentation*. Qiskit. Okt. 2023. URL: https://qiskit.org/documentation/stubs/qiskit.circuit.QuantumCircuit.html#qiskit.circuit.QuantumCircuit.from_qasm_str (zitiert auf S. 37).

[Qis23h] Qiskit. *Single Systems - Quantum Measure*. 2023. URL: https://learn.qiskit.org/course/basics/single-systems#single-systems-quantum-measure (zitiert auf S. 20).

[Qua] Quantil. *QHAna: User Guide*. URL: https://quantil.readthedocs.io/en/latest/user-guide/qhana/ (zitiert auf S. 30).

[Qua20] Quantum AI team and collaborators. *qsim*. Sep. 2020. DOI: 10.5281/zenodo.4023103. URL: https://doi.org/10.5281/zenodo.4023103 (zitiert auf S. 44).

[Qua23a] Quantil. *QHAna New: User Guide*. 2023. URL: https://quantil.readthedocs.io/en/latest/user-guide/qhana-new/ (zitiert auf S. 30).

[Qua23b] Quantinuum. *Comparing Simulators Example in pytket*. 2023. URL: https://github.com/CQCL/pytket/blob/main/examples/comparing_simulators.ipynb (zitiert auf S. 46).

[Qua23c] Quantinuum. *Pytket API: Getting Started*. 2023. URL: https://cqcl.github.io/tket/pytket/api/getting_started.html (zitiert auf S. 45).

[Qua23d] Quantinuum. *pytket Extensions Documentation*. Available at: https://cqcl.github.io/pytket-extensions/api/index.html. 2023 (zitiert auf S. 46).

[Qua23e] Quantinuum. *pytket-extensions API Documentation*. 2023. URL: https://cqcl.github.io/pytket-extensions/api/index.html (zitiert auf S. 46).

[Qul] Qulacs. *Qulacs Documentation*. URL: https://docs.qulacs.org/en/latest/ (zitiert auf S. 46).

[SB23] A. R. Shinde, S. P. Bendale. „Evolution of Quantum Machine Learning and an Attempt of Its Application for SDN Intrusion Detection“. In: *Quantum Computing: A Shift from Bits to Qubits*. 2023, S. 437–456. URL: https://doi.org/10.1007/978-981-19-9530-9_22 (zitiert auf S. 37).

[SHM+23] R. D. M. Simoes, P. Huber, N. Meier, N. Smailov, R. M. Füchslin, K. Stockinger. „Experimental evaluation of quantum machine learning algorithms“. In: *IEEE Access* 11 (Jan. 2023), S. 6197–6208. DOI: 10.1109/ACCESS.2023.3236409 (zitiert auf S. 19).

[Sho94] P. Shor. „Algorithms for quantum computation: discrete logarithms and factoring“. In: *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 1994, S. 124–134. DOI: 10.1109/SFCS.1994.365700 (zitiert auf S. 17).

[Suz21] Suzuki et al. „Qulacs: a fast and versatile quantum circuit simulator for research purpose“. In: *Quantum* 5 (Okt. 2021), S. 559. DOI: 10.22331/q-2021-10-06-559 (zitiert auf S. 59).

[SY22] M. Scekic, A. Yakaryilmaz. „Comparing Quantum Software Development Kits for Introductory Level Education“. In: *Baltic Journal of Modern Computing* 10.1 (2022), S. 87–104. URL: https://doi.org/10.22364/bjmc.2022.10.1.06 (zitiert auf S. 43, 44, 59).

[Teca] X. Q. Technologies. *PennyLane Documentation - Features*. URL: https://docs.pennylane.ai/en/stable/#features (zitiert auf S. 46, 47).

[Tecb] X. Q. Technologies. *PennyLane Plugins Documentation*. URL: https://pennylane.ai/plugins/ (zitiert auf S. 47).

[Tec23] X. Q. Technologies. *Introduction to quantum circuits*. 2023. URL: https://docs.pennylane.ai/en/stable/introduction/circuits.html (zitiert auf S. 47).

[TKE23] TKET Development Team. *pytket*. 2023. URL: https://pypi.org/project/pytket/ (zitiert auf S. 44).

[VBL+21] D. Vietz, J. Barzen, F. Leymann, B. Weder, V. Yussupov. „An Exploratory Study on the Challenges of Engineering Quantum Applications in the Cloud“. In: *University of Stuttgart, Institute of Architecture of Applications Systems* (2021) (zitiert auf S. 19).

[VBLW21] D. Vietz, J. Barzen, F. Leymann, K. Wild. „On Decision Support for Quantum Application Developers: Categorization, Comparison, and Analysis of Existing Technologies“. In: *Computational Science – ICCS 2021*. Hrsg. von M. Paszynski, D. Kranzlmüller, V. V. Krzhizhanovskaya, J. J. Dongarra, P. M. A. Sloot. Cham: Springer International Publishing, 2021, S. 127–141. ISBN: 978-3-030-77980-1 (zitiert auf S. 17, 19).

[Wak21] D. Wakeham. *How to Simulate Noise with PennyLane*. PennyLane. 2021. URL: https://pennylane.ai/blog/2021/05/how-to-simulate-noise-with-pennylane/ (zitiert auf S. 47).

[WBBL] B. Weder, J. Barzen, M. Beisel, F. Leymann. „Provenance-Preserving Analysis and Rewrite of Quantum Workflows for Hybrid Quantum Algorithms“. In: *SN Computer Science* (). URL: https://doi.org/10.1007/s42979-022-01625-9 (zitiert auf S. 17).

[WBLS21] M. Weigold, J. Barzen, F. Leymann, M. Salm. „Encoding patterns for quantum algorithms“. In: *IET Quantum Communication* 2.4 (2021), S. 141–152. URL: https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/qtc2.12032 (zitiert auf S. 17, 20, 21).

[WHB20] R. Wille, S. Hillmich, L. Burgholzer. „JKQ: JKU Tools for Quantum Computing“. In: *International Conference On Computer Aided Design*. 2020 (zitiert auf S. 42).

[WMN] R. Wille, R. V. Meter, Y. Naveh. „IBM's Qiskit Tool Chain: Working with and Developing for Real Quantum Computers (Special Session Summary)“. In: (Zitiert auf S. 24).

[Xan23] Xanadu Quantum Technologies. *PennyLane Documentation: qml.drawer Module*. Xanadu Quantum Technologies. 2023. URL: https://docs.pennylane.ai/en/stable/code/qml_drawer.html (zitiert auf S. 47).

[ZW19] A. Zulehner, R. Wille. „Advanced Simulation of Quantum Computations“. In: *Trans. on CAD of Integrated Circuits and Systems* 38.5 (2019), S. 848–859. DOI: 10.1109/TCAD.2018.2834427 (zitiert auf S. 23, 24).

Alle URLs wurden zuletzt am 10. 11. 2023 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift