IMPERIAL

A META-PROGRAMMING APPROACH TO FUZZING QUANTUM PROGRAMS

Author

Ilan Iwumbwe

CID: 02211662

Supervised by

Dr John Wickerson

Second Marker

Dr Tom Clarke

A Thesis submitted in fulfillment of requirements for the degree of
Master of Engineering in Electronic and Information Engineering

Department of Electrical and Electronic Engineering
Imperial College London

2026

Abstract

Quantum compilers are important pieces of software that perform optimisations on quantum circuits, before running routing and placement algorithms to make them executable on quantum hardware. Therefore, there is a heavy reliance on these compilers which makes it all the more important that they do not alter the semantics of the programs they compiles. Various studies and bug-finding efforts have shown that quantum compilers not only have semantics-changing bugs, but can also crash unexpectedly. Thus, there ought to be efforts made to find and fix these bugs, which moves the field forward because developers would be more likely to adopt quantum software if they can trust its robustness. Even though bug finding tools for quantum compilers have been built before [1] [2], they are not easily extensible for new frameworks, leading to a lot of repeated work when building fuzzers for new frameworks. Additionally, improved fuzzing techniques cannot be easily shared across multiple languages, yet the fundamental problem of finding cleverer ways to generate programs that stress-test quantum compilers isn't so much to do with the exact framework being tested, but more to do with the abstract quantum circuit that gets built.

We present a fuzzing tool, which leverages a novel templating language to generate correct, random and deterministic programs across multiple quantum software stacks which are used to stress-test the compilers. The templating language vastly reduces the time taken to bring-up a fuzzer and find bugs, because in their current state, most quantum programming languages can be described by a set of meta-constructs which know nothing about the specifics of the language.

A differential testing library is also provided, which contains most functions that would be needed to test the generated programs, requiring minimal changes to extend support for new quantum compilers.

The tool has found 10 unique, previously unknown bugs in Pytket, CUDA-Q and Guppy, but also generates circuits for QASM2, QASM3, Pennylane, Cirq and Qiskit.

Plain Language Summary

Quantum compilers are complex systems that bridge the gap between programs written by humans and what the hardware can execute. In the process of doing this, the compilers can make mistakes, which do not always lead to obvious failures, like crashes, but instead change the program in such a way that the output produced is not what the programmer expected. Catching these types of bugs is difficult, because they often crop up when a particular structure of program is fed to the compiler, which means the developers might not think of all possible edge cases while designing their test suites. Moreover, the issue is made worse when you consider the complexity of the compilers. Even though bug finding tools for quantum compilers have been built before [1] [2], they are not easily extensible for new frameworks, leading to a lot of repeated work when building fuzzers for new frameworks. Additionally, improved fuzzing techniques cannot be easily shared across multiple languages, yet the fundamental problem of finding cleverer ways to generate programs that stress-test quantum compilers isn't so much to do with the exact framework being tested, but more to do with the abstract quantum circuit that gets built.

We present a fuzzing tool, which leverages a novel templating language to describe a sort of recipe which the tool uses to generate the programs. This is done because building fuzzing tools requires long engineering hours, yet most components that make up quantum programming languages can be described using the same recipe, with slightly different ingredients. Additionally, if improvements are made to the tool, all programming languages that use the recipes benefit at once.

A differential testing library is also provided, which contains most functions that would be needed to test the generated programs, requiring minimal changes to extend support for new quantum compilers.

The tool has found 10 unique, previously unknown bugs in Pytket, CUDA-Q and Guppy, but also generates circuits for QASM2, QASM3, Pennylane, Cirq and Qiskit.

Declaration of Originality

I hereby declare that the work presented in this thesis is my own unless otherwise stated. To the best of my knowledge the work is original and ideas developed in collaboration with others have been appropriately referenced. I acknowledge the use of Gemini Pro (Google, `https://gemini.google.com/`) to help me understand parts of the Multi-Level Intermediate Representation (MLIR) syntax that I was unfamiliar with such that I could explain the cause of the bug found. For some of the more complex LaTeX figures, I used generative AI to provide the syntax needed to draw those figures, which I then used as a template in my script that runs experiments and saves the LaTeX output with the relevant data injected in. I confirm that no content generated by AI has been presented as my own work.

Copyright Declaration

Acknowledgments

A special thanks to my parents for giving me the opportunity to do what I love at the highest level. I would also like to thank my closest friends for the good times, and for inspiring me to improve and push the boundaries of my capabilities. I would also thank Dr John Wickerson for his advice, suggestions and willingness to discuss ideas that have shaped this project.

Contents

List of Acronyms

QSS Quantum Software Stacks

NISQ Noisy Intermediate-scale Quantum

CUDA-Q CUDA Quantum

MLIR Multi-Level Intermediate Representation

LLVM Low Level Virtual Machine

QIR Quantum Intermediate Representation

BB Bivariate Bicycle

QPU Quantum Processing Unit

DAG Directed Acyclic Graph

AST Abstract Syntax Tree

KS Kolmogorov-Smirnov

eCDF Emperical Cumulative Distribution Function

MAP Elites Multi-dimensional Archive of Phenotypic Elites

UCB Upper Confidence Bound

AFL American Fuzzy Lop

PUT Program Under Test

CPU Central Processing Unit

GPU Graphics Processing Unit

MCVE minimal, complete and verifiable example

REPL read, eval, print Loop

List of Figures

1 Introduction

Contents

1.1 History of quantum computation

At the turn of the 20th century, a series of crises had arisen in physics - the theories at the time (now known as classical physics) were predicting outcomes that did not make sense, such as infinite energies from high frequency radiation [3] and electrons spiralling into the nucleus as they lost energy [4]. Such problems were initially solved by adding ad-hoc hypotheses to classical physics, but as the understanding of atoms and radiation grew, these hypotheses became more complex and ultimately failed to generalise. This resulted in the creation of the modern theory of *quantum mechanics* - a mathematical framework for the construction of physical theories [5].

While physicists were redefining the laws of nature, a parallel revolution was occurring in mathematics and logic. In his seminal 1936 paper, Alan Turing developed an abstract computational model, now known as the *Turing machine* [6] and showed that there is a *Universal Turing Machine* that can be used to simulate any other Turing machine. The *Church-Turing thesis* later posited

that if an algorithm can be performed by any piece of hardware, say a personal computer, then there is an equivalent algorithm for a Universal Turing Machine that performs the same algorithm.

Turing's computational model and its broad acceptance formed the foundation for the development of the first computers from electronic components. In 1947, hardware development saw yet another seismic leap with the development of the transistor, and has continued to grow in a trend predicted by *Moore's Law*, which roughly states that computational power will double every 2 years at constant cost. The reality of computer development has held true to Moore's Law for many decades now, but as transistor sizes become smaller, the functioning of electrical devices starts to be affected by quantum effects. One possible solution to this is to move to a different computing paradigm, one provided by the theory of quantum computation which is based on quantum mechanics. As it turns out, an ordinary computer is capable of simulating a quantum computer, but not efficiently (in less than exponential time).

In 1985, David Deutsch [7] recognized that computation is fundamentally a physical process, and therefore, the theoretical limits of a computer should be dictated by the most complete physical theory available: quantum mechanics. By extending Turing's work, Deutsch defined the *Universal Quantum Turing Machine*; the theoretical framework for quantum computation, and also demonstrated that a quantum computer could, in principle, solve certain problems faster than any classical counterpart, effectively challenging the strong form of the Church-Turing thesis - *any algorithmic process can be simulated efficiently using a Turing machine.* Researchers then begun working on quantum hardware to prove this theoretical speed-up, and in time, the focus shifted towards managing, and abstracting away the complexity of quantum systems. This necessitated a robust layer of software to bridge the gap between high-level logic and hardware execution.

1.2 The Problem

The quantum computing field has seen a shift from lab experiments to mainstream adoption over the past decade due to the development of Quantum Software Stacks (QSS) that facilitate the writing, compilation and running of quantum programs. Some of these include IBM's Qiskit [8], Google's Cirq [9] Quantinuum's Pytket [10], and NVIDIA's CUDA Quantum (CUDA-Q) [11]. These stacks, similar to their classical counterparts, can be broken down into 3 main components: the high level language, the compiler, the hardware, simulator or emulator.

For quite some time, quantum hardware has been in the Noisy Intermediate-scale Quan-

tum (NISQ) era, characterized by a few (hundreds to thousands) noisy qubits [12]. However, this has changed in recent years due to advancements in error-correction and fault tolerance. Historically, the surface code was the best way to achieve fault tolerance for physical qubits, the main disadvantage being the high overhead incurred in using them. In 2023, IBM introduced a family of Bivariate Bicycle (BB) codes which could preserver a 12 logical qubits for a million cycles using roughly 288 physical qubits, whereas the surface code would require nearly 3,000 to achieve the same error threshold (above which the qubit state is corrupted) [13]. Since then, quantum hardware has become more capable, mainly because it is able to have more physical qubits with less overhead for fault tolerance.

In the early NISQ era, compilers were therefore relied upon to reduce circuit depth, which reduces the chance of qubit state corruption due to decoherence, and to reduce multi qubit gate count, as such gates are harder to perform fault tolerance for. Compilers now also have to be able to handle more complex quantum systems, while orchestrating error correction, and possibly even chiplet architectures, where multiple QPUs are used to run an algorithm.

Therefore, quantum compilers ought to be robust and correct, yet this is not the case. A study by [14] gathered and inspected a set of 223 real-world bugs from 18 open-source quantum computing platforms. Their study showed that a significant fraction of these bugs (39.9%) are quantum-specific, and change the semantics of programs written by developers, making them more difficult to catch than bugs that cause the compiler to crash.

1.3 Current solutions

There have been many attempts to solve this problem by adapting automated testing techniques used in classical compilers to the quantum domain. These are some of the examples

- **QDiff**: QDiff introduced one of the first differential testing frameworks for quantum compilers used by Qiskit, Pyquill and Cirq [1]. It starts from a seed corpus of 6 handwritten quantum programs and applies semantics-preserving mutations to generate 730 equivalence classes, then applied semantics-modifying mutations to generate 15000 test cases. Because the result from running a quantum program is probabilistic, QDiff uses the Kolmogorov-Smirnov (KS) test to compare the output probability distributions and if they diverge beyond a certain threshold, a potential bug is flagged. Additionally, QDiff introduced a technique to filter out certain circuits that would lead to unreliable divergence in the output distributions due to

noise, and thus are not worthwhile to run on hardware or noisy simulators. Their paper cited a 66% reduction in testing time as a result. After running for a period of 7 days, 4 simulator crashes and 2 IBM hardware bugs were uncovered.

- **MorphQ**: MorphQ introduced a program generator capable of producing a large and diverse set of valid Qiskit programs and a set of program transformations that exploit quantum-specific metamorphic relationships such that the program output is predictable [15]. This sidesteps the oracle problem, which is that of not knowing what the program output will be before testing, and therefore, having no comparison point to use to flag potential bugs. MorphQ uncovered 13 crash bugs after running for a period of about 30 days.

- **QuteFuzz**: QuteFuzz introduced a program generator capable of producing a large set of Qiskit, Cirq and Pytket programs with subroutines and deeply nested control flow. Since the output of the quantum program is unknown, the quantum program with no optimisation passes applied to it was treated as the oracle, and circuits optimised with increasingly aggressive passes were compared against it using the KS test. If the output distributions diverge, a potential bug is flagged. After running in multiple 48 hour periods, running up to 100,000 circuits, 17 bugs were uncovered [2].

While these tools made significant strides in uncovering compiler crashes, simulator crashes and semantic bugs, they still lack in extensibility and general applicability. Tools like QDiff rely on a set of handwritten seed programs which may not always be available for any QSS being tested, MorphQ is tightly coupled to Qiskit, making it difficult to add support for new frameworks. QuteFuzz suffers from the same problem especially when adding support for new language constructs.

1.4 Proposed solution

The work presented aims to build a fuzzing tool that is capable of generating programs for multiple quantum computing platforms, with a test harness that feeds the programs to the compilers. Support for new platforms should be seamless and require much less engineering effort than it would have taken to build a fuzzing tool for that platform from scratch.

My fuzzing tool, qf++ [1], introduces a templating language in which the structure of the programs can be described. The language is made expressive enough to be able to describe language

[1] `https://github.com/QuteFuzz/QFpp`

constructs across the main QSS frontends. The templating language also allows users to very easily control circuit sizes, control flow depth, which language constructs are generated at varying levels of granularity, and make syntactic changes to the programs. The key philosophy is to give the user as much control as possible in the templating language, and keep them away from the generation source code as much as possible.

The generation engine, written in C++, parses the template and builds an AST on which mutation passes can be run, before emitting the program code. The testing backend is written in Python. It consists of a minimal differential testing library, which requires new programs to define a single function in a class that inherits from the base library class. The differential testing library treats the unoptimised, generated program as the oracle, and its output (probability distribution or state-vector) is compared with that of the same circuit compiled with increasingly complex optimisation passes. *Interesting* circuits are saved in a report with a seed that can be used to reproduce the test case, where we define interesting circuits as those that

- show deviations in outputs that are expected to be the same
- show errors related to compiler crashes
- timeout during compilation

The test cases are manually reduced to get a minimal, complete and verifiable example (MCVE) that is reported to the software vendors.

The tool also aims to be platform-agnostic, increasing is capability to support new architectures and features. If a new quantum stack is developed tomorrow, it would probably share a lot of structural commonalities with existing languages, and therefore, getting a fuzzer up and running for it using this tool should be much easier than it would have been building one from scratch. Moreover, doing so would represent a lot of repeated work, because the fundamental program generation techniques are platform-agnostic. qf++ bridges this gap by acting as one of the first extensible, open-source, differential testing frameworks to target multiple stacks, including those for which external testing efforts have not been explored such as CUDAQ.

Having been an author on QuteFuzz, I was very interested in improving the extensibility of the tool. I felt that creating a templating language was the next natural progression, as it gives the foundation for creating a language-agnostic generation engine based on AST generation. With an AST, performing various mutations on the programs becomes much easier, which opens the

opportunity to explore new circuit generation techniques. qf++ is a complete rewrite of QuteFuzz that aims to achieve these goals.

1.5 Report Overview

Below is an overview of the report chapters and what information they will contain

- **Chapter 2**: Background information about fuzzing and quantum computing
- **Chapter 3**: Project specification and scope
- **Chapter 4**: Technical implementation details and design decisions
- **Chapter 5**: Presenting and interpreting results obtained
- **Chapter 6**: Evaluating the success of the project
- **Chapter 7**: Conclusions and future work
- **Chapter 8**: User guide

2 Background

Contents

This sections provides background knowledge on quantum computing and fuzzing. This is needed to understand the programs that are generated, the root causes behind the bugs that were found, and why certain design decisions were taken.

2.1 Qubits

In classical computers, we need physical systems to store bits, for example: as tiny charges on nanometer scale capacitors. These are abstracted away, such that most people programming computers think in terms of 1s and 0s. Similarly, the qubit is the abstraction used for the basic unit

of computation in quantum computers. The state of a qubit is a *vector* in 2-D space, there are 2 computational basis states $|0\rangle$ and $\langle 1|$, which correspond to 0 and 1 bits in classical computation.

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{2.1}$$

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \tag{2.2}$$

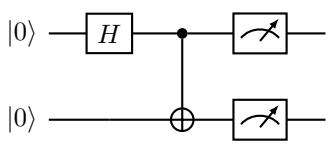
In general, the state of a qubit $|\psi\rangle$ is a complex vector in 2-D space, for example:

$$|\psi\rangle = \frac{i+1}{2}|0\rangle + \frac{i}{\sqrt{2}}|1\rangle = \begin{bmatrix} \frac{i+1}{2} \\ \frac{i}{\sqrt{2}} \end{bmatrix} \tag{2.3}$$

The state 2.3 is said to be in *superposition*: a linear combination of computational basis states. The co-effecients $\frac{i+1}{2}$ and $\frac{i}{\sqrt{2}}$ are called *amplitudes*. The *normalisation constraint* states that the sum of squares of the amplitudes of a qubit's quantum state must be 1. Therefore, we can fully describe the quantum state as a unit vector in 2-D complex vector space. This space is also known as the *state space*.

2.2 Quantum wires

In diagrams showing quantum circuits, a quantum wire shows the progression of time from left to right, where each gate application transforms the qubit's state.



(2.4)

The box with an H is the symbol for a Hadamard gate, the filled dot connected to the empty dot is a CNOT gate, and the last 2 boxes are symbols for measure gates. In the following sections, we will see what each of the components above mean and what they do.

2.3 Quantum gates

In classical computing, we have logic gates, which allow us to perform computation on logical bits, and in the same way, we have quantum gates which allow us to perform computation on qubits. Some have analogues in classical computing while others do not.

2.3.1 The quantum NOT gate - X gate

This takes the state $|0\rangle$ and transforms it into state $|1\rangle$. If we have a superposition of states, it acts linearly, so for a state $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$

$$X|\psi\rangle = \alpha|1\rangle + \beta|0\rangle \tag{2.5}$$

As a 2-D matrix, we can represent the X-gate as

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \tag{2.6}$$

2.3.2 The Hadamard gate - H gate

The H gate acts in the following way on the computational basis states

$$H|0\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \tag{2.7}$$

$$H|1\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}} \tag{2.8}$$

On a superposition of states, it acts linearly

$$H(\alpha|0\rangle + \beta|1\rangle) = \alpha H|0\rangle + \beta H|1\rangle \tag{2.9}$$

The matrix representation is

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{2.10}$$

With the H gate, we come to the first deviation from classical computation, and an insight into where the computational advantage in quantum computation comes from. A classical computer can only model distinct states at any given time i.e 0 or 1. After applying the H gate on a qubit, the resulting state is not in a distinct state, it appears to be in both computational basis states at the same time, it is in a superposition of these states. Whereas classical computers can model the final measured state of the qubit, quantum computers can also model this superposition state. This allows us to build quantum circuits which manipulate all possible inputs simultaneously, by "spreading" the state of a single qubit over all possible computational basis states before manipulating it further.

Additionally, consider that applying the Hadamard twice on a qubit is the same as identity. We already saw in 2.7 what we get after $H|0\rangle$. Now when we apply it again

$$H\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}}\right) = \frac{1}{\sqrt{2}}\left(\frac{|0\rangle + |1\rangle}{\sqrt{2}} + \frac{|0\rangle - |1\rangle}{\sqrt{2}}\right) \tag{2.11}$$

We see that the coefficients of $|1\rangle$ cancel, while those of $|0\rangle$ reinforce, leading to a final state of $|0\rangle$. The same can be done to show that $HH|1\rangle = |1\rangle$. We can see that using the H gate to first give us a set of states we manipulate, then cleverly combining and canceling states to give the state that has the desired answer is important when coming up with quantum algorithms.

In general, single qubit gates can be represented as 2x2 unitary matrices, U. A matrix U is unitary if

$$U^\dagger U = I \tag{2.12}$$

where $U^\dagger$ is the complex transpose

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}^\dagger = \begin{bmatrix} a^* & c^* \\ b^* & d^* \end{bmatrix}$$

Like the H gate, the Y and Z gates also allow us to "spread out" the quantum state, there

matrix presentations are given below

$$Y := \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$$

The Z gate leaves $|0\rangle$ unchanged, but converts $|1\rangle$ into $-|1\rangle$.

$$Z := \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Following the unitarity definition at 2.12, we can also see that the rotation matrix is a valid 2 qubit quantum gate

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Unitary matrices preserve the length of their inputs, i.e $||U|\psi\rangle|| = ||\,|\psi\rangle\,||$. This is true for any vector $|\psi\rangle$ in d dimensional space. This is great for quantum gates - a quantum state has to be normalized, therefore after applying a quantum gate on it, the resulting vector is still normalized, hence still a legitimate quantum state.

2.3.3 CNOT gate

Shown in 2.16 as a filed dot on the control qubit, and an empty dot on the target qubit, this gate allows for qubits to interact with each other, a phenomenon known as *entanglement*.

Two-qubit systems like 2.16 have 4 computational basis states, namely: $|00\rangle, |01\rangle, |10\rangle, |11\rangle$.

$$|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}, |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}, |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}, |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.13}$$

To see where these states come from, consider the tensor products of all computational basis

states for a 1-qubit system

$$|0\rangle \otimes |0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \begin{bmatrix} 1 \\ 0 \end{bmatrix} \\ 0 \begin{bmatrix} 1 \\ 0 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \tag{2.14}$$

And so on for the other multiplications $|0\rangle \otimes |1\rangle$, $|1\rangle \otimes |0\rangle$, $|1\rangle \otimes |1\rangle$.

We can also have a superposition of these states: $\alpha |00\rangle + \beta |01\rangle + \gamma |10\rangle + \delta |11\rangle$. We still have the same normalization condition as before, i.e $|\alpha|^2 + |\beta|^2 + |\gamma|^2 + |\delta|^2 = 1$.

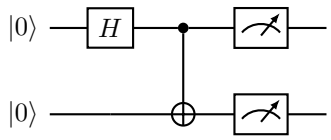
The CNOT gate works in the following way - if the control bit of the qubit is set to 1, then flip the target bit, otherwise do nothing.

$$|00\rangle \to |00\rangle, |01\rangle \to |01\rangle, |10\rangle \to |11\rangle, |11\rangle \to |10\rangle$$

As a unitary matrix, we write it as

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.15}$$

With these definitions, we can follow through the computation of the 2-qubit system above, re-written again below for reader convenience.



(2.16)

- The H gate has the effect $|0\rangle \to \frac{|0\rangle + |1\rangle}{\sqrt{2}}$
- The second qubit isn't affected, so we get $|00\rangle \to \frac{|00\rangle + |10\rangle}{\sqrt{2}}$
- The $|00\rangle$ state isn't affected by the CNOT, the $|10\rangle$ state becomes $|11\rangle$. We end up with

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}}$$

A run-through of this computation using linear algebra can be found in A.1.

What do we use this final state for? First we need to measure the qubit.

2.4 Qubit measurement

It is forbidden by the laws of quantum physics to observe the quantum state of any system. To get information from a quantum computer, we perform *measurement in the computational basis*. Consider a single qubit in state $\alpha|0\rangle + \beta|1\rangle$. Measuring this qubit gives a classical bit of information: it gives 0 with probability $|\alpha|^2$ and gives 1 with probability $|\beta|^2$. Moreover, measuring the quantum state disturbs it, creating a "posterior state". If the measurement result is 0, then the quantum state becomes $|0\rangle$, if the measurement result is 1, then the quantum state becomes $|1\rangle$. Additionally, after measurement, α and β are no longer accessible.

2.5 Computational basis and the Bloch sphere

Them Bloch sphere is a geometrical representation of quantum states, which before now we have only represented using linear algebra [16]. First, consider again the general form of a quantum state

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \tag{2.17}$$

Because each of the coefficients are complex numbers, which we can write out in polar form, we can rewrite 2.17 as

$$|\psi\rangle = r_\alpha e^{i\theta_\alpha}|0\rangle + r_\beta e^{i\theta_\beta}|1\rangle \tag{2.18}$$

We then factor out $e^{i\theta_\alpha}$ to get

$$|\psi\rangle = e^{i\theta_\alpha}(r_\alpha|0\rangle + r_\beta e^{i(\theta_\beta - \theta_\alpha)}|1\rangle) \tag{2.19}$$

We can ignore the $e^{i\theta_\alpha}$ at the front because it has no observable effects on the qubit's state. As we saw in section 2.4, the observed output is dependent on the squared amplitude of the qubit

state. This term has an amplitude of 1, and hence does not affect the output probabilities when measuring the qubit.

$$|\psi\rangle = r_\alpha|0\rangle + r_\beta e^{i(\theta_\beta - \theta_\alpha)}|1\rangle \tag{2.20}$$

Then replacing $\phi = \theta_\beta - \theta_\alpha$

$$|\psi\rangle = r_\alpha|0\rangle + r_\beta e^{i\phi}|1\rangle \tag{2.21}$$

Next, due to the normalisation condition, we know that the sum of the squares of the real coefficients must equal 1, i.e., $r_\alpha^2 + r_\beta^2 = 1$. This constraint allows us to parameterize the coefficients using trigonometric identities. We define $r_\alpha = \cos(\frac{\theta}{2})$ and $r_\beta = \sin(\frac{\theta}{2})$.

Finally, we arrive at the canonical Bloch sphere representation:

$$|\psi\rangle = \cos\frac{\theta}{2}|0\rangle + e^{i\phi}\sin\frac{\theta}{2}|1\rangle = \begin{bmatrix} \cos\frac{\theta}{2} \\ e^{i\phi}\sin\frac{\theta}{2} \end{bmatrix} \tag{2.22}$$

The numbers θ and ϕ (which are both real numbers) represent a point on a three-dimensional sphere as shown in 2.1 [5]

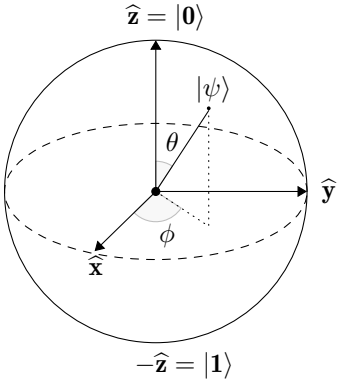


Figure 2.1: Bloch sphere representation of a single qubit's state

Lines of latitude

ϕ is the relative phase. In equation 2.22, it is the argument to a complex exponential, which implies that changing it causes the state to repeat with a period of 2π. On the Bloch sphere, this is a rotation around the Z axis, which would not change the measurement probabilities of the vector

Poles

In the derivation, we used $\frac{\theta}{2}$ as the poles in order to make orthogonal states appear as exact opposite poles on the Bloch sphere (π apart). $|0\rangle$ and $|1\rangle$ are $\frac{\pi}{2}$ apart as they are orthogonal, so moving from $|0\rangle$ to $|1\rangle$ requires the vector to rotate by $\frac{\pi}{2}$. Using the $\frac{1}{2}$ scale factor accounts for this automatically.

Measurements and resets can only happen on a qubit that is on the Z-axis, because the computational basis states $|0\rangle$ and $|1\rangle$ lie on the Z axis [17] [5].

2.6 Clifford gateset and the universal gateset

The Clifford gateset is a group of quantum operations defined by the H, S, and CNOT gates [5]. The Pauli group $\mathbf{P_n}$ is made up of tensor products of the 4 basic Pauli matrices: X, Y, Z, I. In general, if you have n qubits, there are 4^n possible Pauli matrices. We have the identity

$$Y = iXZ \tag{2.23}$$

where the i implies an additional global phase factor, which we ignore as it has no effect on the final result amplitudes. Therefore, we say X and Z are the generator matrices for the Pauli gateset.

By definition, a matrix U is said to be a Clifford matrix if it satisfies the transformation

$$UPU^\dagger \in \mathbf{P_n} \tag{2.24}$$

where P is a Pauli matrix.

We do not have to check all 4 Pauli matrices due to the identity in equation 2.23

$$U(Y)U^\dagger = U(XZ)U^\dagger = (UXU^\dagger)(UZU^\dagger) \tag{2.25}$$

We only need to check what the matrix does to the X and Z gates only. The computational cost goes down from 4^n to $2n$. This concept forms the foundation for why clifford gates are so efficient to simulate even for classical computers. The Gottesman–Knill theorem [18] proves that any Clifford-only circuit can be efficiently simulated on a classical computer in polynomial time.

To form the universal gateset, the Clifford gates are usually combined with the T gate (a $\frac{\pi}{4}$ rotation along the Z-axis). The Clifford gates are relatively cheap to implement and protect from errors, while the T gate gives the quantum advantage needed to run complex algorithms like Shor's factoring algorithm [19].

2.7 The Quantum computing system

The quantum computing system is made up of the following layers, ordered here in bottom-up fashion [20]

- **Quantum hardware**: Implementations of physical bits using trapped ions, superconducting circuits, photonics, and other technologies [21]. This is what is commonly referred to as the QPU.
- **Control and measurement layer**: Hardware responsible for pulse generation and readout
- **Compiler and runtime layer**: Software that takes abstract quantum programs and translates them into machine executable instructions
- **Orchestration layer**: Input preparation, output reading, job scheduling. A purely classical layer
- **Application layer**: Algorithms that solve domain-specific problems.

These layers are tightly intertwined, and choices made at one layer have implications on how the entire system will work.

2.7.1 Quantum hardware

The most dominant hardware technologies are trapped ion and superconducting circuits. The latter give you high fidelity gates, qubits with long coherence times, and excellent qubit connectivity, while sacrificing gate execution speeds and having tighter timing constraints. This implies that a QPU built using trapped ion qubits simplifies qubit routing problems for the compiler, but also makes it more difficult to schedule instructions in such a way that timing is met. On the other hand, superconducting qubits, used by platforms such as IBM Quantum [22], offer strong integration with classical control electronics and faster gate operations, while suffering form high noise and

decoherence, and limited qubit connectivity. Compilers therefore need to optimise qubit mapping and gate scheduling to allow program results to be obtained before decoherence effects corrupt them.

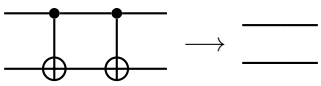
2.7.2 Quantum compilation

A modern quantum software stack generally operates in 3 phases

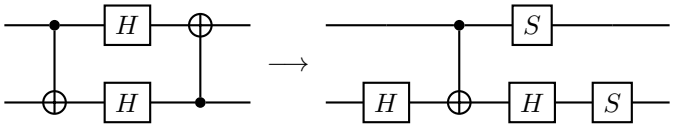
High level synthesis

This is the front-end of the compiler. When a quantum program is written in a high-level language it is first converted into an abstract representation, often a Directed Acyclic Graph (DAG), on which it performs hardware-agnostic optimisations. For instance, it might notice that there is an operation like `CX(q0, q1); CX(q0, q1)` and remove it because it simplifies down to the identity matrix.

This is part of a family of optimisations knows as *peephole optimisations*, which work the same way they do in classical compilers; a sliding window goes over the instruction graph looking for specific patterns of sub-circuits and replacing equivalent circuits (with lower gate count and circuit depth). Some of these known decompositions are shown in 2.26 and 2.27



(2.26)



(2.27)

Another pass that might happen is the decomposition of unitary matrices into sequences of local unitaries with fewer qubit entanglements. The Cartan decomposition [23] gives way to transform any arbitrary unitary matrix on n qubits in terms of one qubit and two qubit operations.

Optimisation and routing

Hardware has strict qubit connectivity constraints which the compiler must work with to make the circuit it is compiling executable. These constraints can be represented as a graph, where nodes

are qubits, and physical connections form edges. A hypothetical 12 qubit QPU is shown in figure 2.2, with logical qubits mapped.

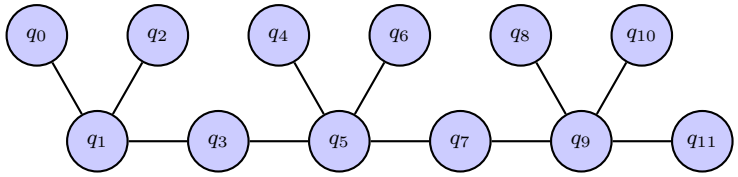
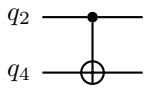


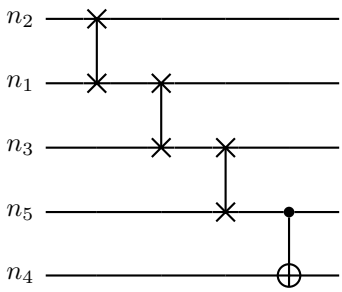
Figure 2.2: Hypothetical 12 qubit QPU

The circuit needs to be dynamically modified such that it satisfies these connectivity constraints, which is done by permuting the mapping of logical to physical qubits using SWAP gates. Because SWAP gates are prone to noise errors, the compiler needs to use the fewest possible number of SWAP gates. The cirucit in 2.28 is one with an entanglement between logical qubits q_2 and q_4, which are not physically connected as seen in the mapping in figure 2.2.



(2.28)

The physical qubits are distance 4 apart, and therefore would require 3 SWAPs to interact, therefore, the physical nodes would have to go through the transformation shown in 2.29.



(2.29)

Lowering and pulse generation

The compiler needs to lower generic gates into the gateset supported by the hardware it is compiling for. The control and measurement layer generates analog pulses that act on the qubits in different ways depending on the gate.

Different providers have different architectural philosophies regarding how their compiler infrastructure is built. Pytket's compiler, `tket` [10] was built to be a counterpart to Low Level Virtual

Machine (LLVM)[24]; you give it a circuit written in a front-end like Cirq [9] or Qiskit [8] which is translated into the `tket` IR. A set of rewrites are performed on this IR, which involves hardware-agnostic optimisations followed by hardware aware mapping and routing to prepare the circuit for execution [10]. The end product of this process is a kernel: a circuit that can be executed on the chosen target device, of which there are many options including IBM Q [22] and Rigetti [25].

The `nvq++` compiler that comes with CUDA-Q is built on top of LLVM and MLIR. It lowers both classical and quantum instructions into the same AST, therefore being better suited to optimise dynamic circuits based on classical data, whereas classical control flow in languages such as Qiskit is static. A separate construct is provided to express dynamic control flow that works in the quantum world. It performs its optimisation passes on the `Quake` MLIR dialect, before outputting Quantum Intermediate Representation (QIR) [26].

2.8 Quantum program simulation

Quantum simulators can be used in place of quantum processors to run quantum circuits, as they are cheaper and more easily accessible. One major downside to using them is the efficiency penalty. Consider that in order for classical hardware to simulate quantum mechanics, the probability amplitudes must be tracked for all possible states every qubit can be in. The number of states grows exponentially.

Every time a quantum gate is applied to a qubit, a large matrix-vector multiplication needs to be carried out - the matrix represents the circuit state, the vector contains 2^n complex numbers. Therefore, with every new qubit added to the circuit doubles the workload. If the circuit consists entirely of gates and measurements at the end, then the simulator can calculate the unitary matrix representing the entire circuit once then perform the matrix-vector product to get the final output distribution. Each shot becomes a random sampling from this distribution. This can be done in a fraction of a millisecond especially with fast caching in today's Central Processing Unit (CPU)s and even with Graphics Processing Unit (GPU) acceleration. 2.1 shows one such program.

Listing 2.1: Example of a simple program to simulate

```
import numpy as np
from pytket import Circuit, Qubit
from pytket.circuit import Unitary1qBox

raw = [
    (-0.45185305463425651, -0.28618722701421401),
    (-0.33173301426847457, 0.77709645177156639),
    (-0.74158821740205827, -0.40493530831732105),
```

```
    (0.17769978713734222, -0.5044770535314137),
]
U = np.array([complex(r, i) for r, i in raw], dtype=complex).reshape(2, 2)
unitary_7 = Unitary1qBox(U)

main_circuit = Circuit()
sing_3793 = Qubit("sing_3793",0)
main_circuit.add_qubit(sing_3793)
sing_3806 = Qubit("sing_3806",0)
main_circuit.add_qubit(sing_3806)
sing_3832 = Qubit("sing_3832",0)
main_circuit.add_qubit(sing_3832)

main_circuit.CX(sing_3806, sing_3832)
main_circuit.add_unitary1qbox(unitary_7,sing_3806)
main_circuit.H(sing_3806)

main_circuit.measure_all()
```

However, if the circuit contains any mid-circuit measurements, the simulator cannot compute the unitary matrix beforehand, because each shot could evaluate the conditional differently. The simulator has to collapse the statevector, pick a branch, and continue performing the matrix multiplications, which has to be repeated from scratch on every shot. The circuit in 2.1 is still so small that performing conditionals on all the statements would still run in a reasonable amount of time. However, more complex programs with deeply nested control flow both in the main cicuit and in subroutines would take longer to run. This is why qf++ generates relatively small programs by default (hundreds of lines long) which allows it to generate the complex constructs that would normally slow down the simulator, but are more likely to induce bugs in the compiler.

2.9 Fuzzing

Fuzzing is an automated software testing technique that involves generating malformed or unexpected inputs into a computer program then monitoring it for exceptions, crashes, memory leaks, failing assertions, or any information leaks which could create security vulnerabilities. The idea originated from a 1988 class by Professor Barton Miller at the University of Wisconsin, where he asked his Advanced Operating Systems class to test the reliability of UNIX command line programs by having them run on a series of random inputs until they crashed [27]. This laid the groundwork for tools such as American Fuzzy Lop (AFL) [28] and AFL++ [29], which instrument any binary at compile time and use genetic algorithms to discover new test cases that trigger previously unseen paths in the instrumented binary. AFL has gone on to find numerous security vulnerabilities such as those in the UNIX Bash shell as reported in Shellshock [30].

Fuzzers have evolved significantly since then, mainly in program generation technique and

program awareness.

2.9.1 Program awareness

This axis classifies fuzzers by how much information they have about the Program Under Test (PUT).

- **Black-box fuzzing**: The fuzzer knows nothing about the internal code structure of the program, and generates random inputs following format specifications. While these types of fuzzers are often fast and easy to scale, they struggle to reach the deep, complex code paths of the PUT. qf++ falls into this category
- **White-box fuzzing**: This is the other end of the spectrum, where the fuzzer uses techniques such as symbolic execution and program analysis to systematically explore all possible execution paths. In the latter case, instead of passing input values to the PUT, the inputs are assigned symbols and equations in terms of those symbols are passed to the PUT. In the case of conditionals, white-box fuzzers can record the set of possible execution paths and under which constraints those paths are found, which can be used to dynamically generate inputs that satisfy said constraints. Consider the example in 2.2, where we try rendering an image by first checking whether it is stored in the correct format before calling the potentially buggy function.

Listing 2.2: Example showing the advantage of white-box fuzzing

```
void parse_image(char* input) {
    if (input[0] == 0xDE && input[1] == 0xAD &&
        input[2] == 0xBE && input[3] == 0xEF) {
        /* some bug exisits in this function */
        render_pixels(input);
    } else {
        return;
    }
}
```

With black-box fuzzing, we would need the 32-bit input to match exactly `0xDEADBEEF` which has a $1/2^{32}$ chance of happening. Thus, the fuzzer will miss this bug most of the time. Even though white-box fuzzing is better at exercising deep code paths, it is time consuming and computationally expensive.

- **Grey-box fuzzing**: This technique tries to find a middle ground between the 2 previous techniques. It does not require full source analysis, but it leverages lightweight binary instrumentation to track coverage and use it as feedback on whether the generated inputs successfully unlocked new code paths in the PUT.

2.9.2 Program generation

This axis classifies fuzzers by the techniques they use to generate inputs

- **Mutation-based fuzzing**: This starts with a corpus of existing valid inputs, and performs random mutations on them to generate malformed inputs that are passed to the compiler
- **Generation-based fuzzing**: This generates inputs from scratch, relying on a pre-programmed knowledge of the expected format of the PUT. qf++ falls under this category.

This section has gone through background knowledge on quantum computing and fuzzing, while contextualising the project within these fields.

3

Project Specification

Contents

This section will go through the core aims of the project, then breaks down its main themes: program generation, differential testing, statevector comparison testing and tool extensibility and support.

The core aim of this project is to write a tool that generates random, but valid quantum programs in the language provided by the major QSS providers, compile, then run them on shots-based or statevector simulators. Circuits that crash the compiler or simulator and those that produce results deviating from the uncompiled baseline are logged in a final report. Interesting test cases are reduced manually before reporting the bug to a vendor.

The tool also provides a generation mode which produces a corpus that consists of diverse, high quality test cases by leveraging an implementation of MAP Elites [31], mutation passes on the AST and quality heuristics calculated statically.

The main contributions of the project are stated below:

- Creation of a BNF-like templating language with expressive features that allow circuit-based program description
- An easily extensible fuzz testing system and differential testing library

- Exploration of a new generation technique that makes use of a genetic algorithm which produces variants in a predefined search space of possible programs

3

3.1 Program Generation

This works in two phases:

- grammar build time (analogous to compile time), when the grammar templates are parsed and the grammar is built into a data structure
- AST build time (analogous to run time), when the data structure is used to build the final AST

A context data structure is used to keep track of constructs which affect how other parts of the AST are generated, for example, if the tool decides to generate 3 qubits to use in the circuit, only those 3 qubits must be available in the resource pool for later gates to use, and they also restrict whether certain gates and subroutines can be generated - a subroutine that expects 4 qubits as arguments can no longer be generated, as the result would be an incorrect circuit. This is achieved with *branch constraints*, which limit choice on which branch in the grammar can be taken if it would lead to a malformed circuit. Branch constraints are created at runtime, because they depend upon the choices that happen to be made by the generator. Constraints can also be specified in the grammar to control how certain parts of the grammar are built. This allows constraints that are known while writing the template to be specified immediately, then dynamically evaluated at runtime depending on context. One example of this is when specifying the structure of gate arguments, you know that the number of qubit nodes in the AST equals the number of qubit arguments required by the gate, therefore, you might write something like

```
1 gate_op_args = qubit_list;
2 qubit_list = qubit (", " qubit)[GET_GATE_QUBITS - 1];
```

where `GET_GATE_QUBITS` returns the number of qubits the current gate requires when resolved at runtime.

The combination of context, branch constraints, and in-grammar constraints guarantees **correctness** in the final program.

Randomness is guaranteed by the fact that with the exception of constraints added to ensure correctness, all the other choices about how to build the AST are left up to the generator. In order to ensure reproducibility, the pseudo-random random number generator is seeded before running the tool allowing for entire batches of test cases to be reproduced by reusing the same seed. This is especially useful when exploring interesting test cases after nightly runs.

Determinism is an interesting consideration when fuzzing quantum programs, because any single run yields non-deterministic results by design. However, running the same program multiple times produces a deterministic probability distribution. Say we generated the program shown in 3.1.

Listing 3.1: Hadamard gate applied on a qubit in Pytket

```
from pytket import Circuit, Qubit

from diff_testing.pytket import pytketTesting

main_circuit = Circuit()
sing_2496 = Qubit("sing_2496", 0)
main_circuit.add_qubit(sing_2496)

main_circuit.H(sing_2496)
main_circuit.measure_all()

pt = pytketTesting()
pt.opt_ks_test(main_circuit, 0)
```

As we saw in section 2.3.2, applying an H gate on qubit state $|0\rangle$ results in a state

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

signifying a 50-50 chance of the resulting state being $|0\rangle$ or $|1\rangle$. Figure 3.1 shows the probability distribution after running for 1000 shots.

A probability distribution like that in figure 3.1 is the expected output after running the program 3.1.

3.2 Differential testing with probability distributions

For differential testing, we compare probability distributions using KS test, which coverts the shots results into an Emperical Cumulative Distribution Function (eCDF), $\hat{F}(x)$. The eCDF of 3.1 is shown by the blue dotted orange graph in 3.2. It is ”emperical” because it is an estimate of the

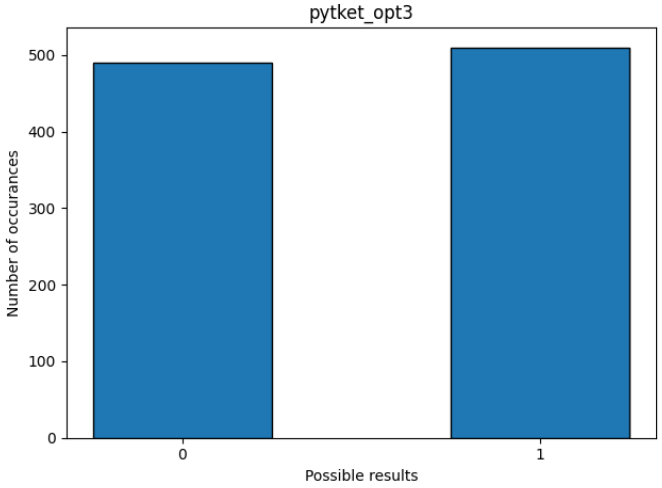


Figure 3.1: Probability distribution of H gate

true CDF $F(x)$, which cannot be obtained as that would require running in infinite number of shots.

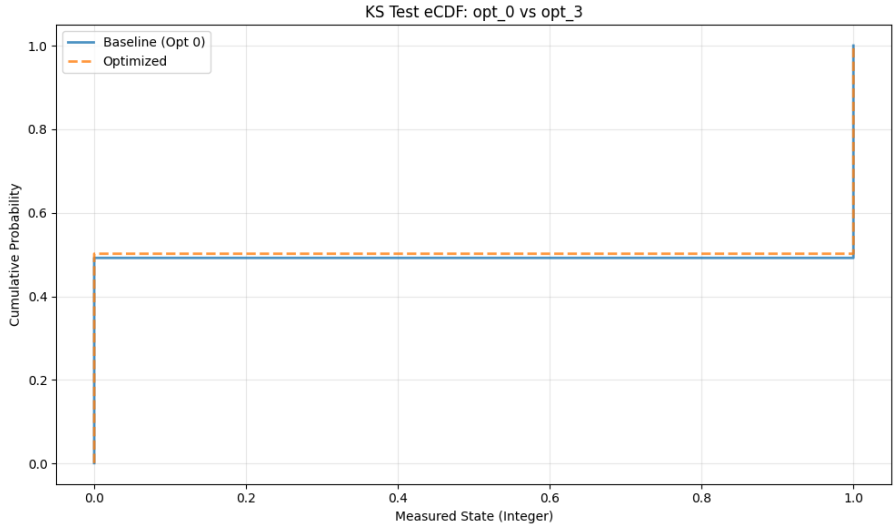


Figure 3.2: eCDF view of probability distributions for optimisation levels 0 and 3 overlayed

The null hypothesis is

$$H_0 : F_1(x) = F_2(x) \quad \text{for all } x$$

The alternative hypothesis is

$$H_a : F_1(x) \neq F_2(x) \quad \text{for at least one } x$$

The KS test assumed H_0 is true, only rejecting it if $D > D_{critical}$

The maximum vertical distance between the 2 graphs is calculated

$$D = \max |\text{eCDF}_A(x) - \text{eCDF}_B(x)|$$

The critical distance is calculated as

$$D_{critical} = c(\alpha)\sqrt{\frac{n_1 + n_2}{n_1 n_2}}$$

where in our case $n_1 = n_2 = 1000$, the number of shots, and

$$c(\alpha) = \sqrt{-\frac{1}{2}\ln\left(\frac{\alpha}{2}\right)}$$

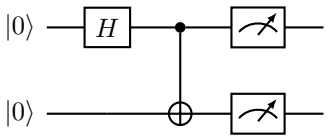
with α being the false-positive rate.

In the differential testing library, an implementation from the `scipy` Python library is used [32]. It returns the p-value, which if less than $\alpha = 0.01$, signifies a deviation between the distributions which with 99% certainty is caused by a bug as opposed to statistical noise.

The test adjusts dynamically depending on the number of shots used, making it mathematically robust even at a relatively low shots value, in comparison with different tests like chi-squared test, which need more data per bin in order to make a meaningful comparison.

3.3 Statevector comparison

Consider the circuit 3.1



(3.1)

This is a 2-qubit system whose initial state is

$$|00\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^T$$

a 4-dimensional unit vector

After applying the H and CNOT gates, the final state is

$$\frac{1}{\sqrt{2}}\begin{bmatrix}0\\1\\0\\1\end{bmatrix} = \frac{1}{\sqrt{2}}\begin{bmatrix}0\\1\\0\\0\end{bmatrix} + \frac{1}{\sqrt{2}}\begin{bmatrix}0\\0\\0\\1\end{bmatrix} = \frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|11\rangle \tag{3.2}$$

also, a 4-dimensional unit vector. Appendix A.1 works out this final state using linear algebra.

We can compare 2 such vectors by taking their dot product, a result of 1 implying they are the same. If the compiler has a semantics-changing bug, it would manifest as a different dot product because the final state would be altered. Circuits without mid-circuit measurements and control flow can be run on statevector simulators to get this final state, instead of running on shot-based simulators, which is a slower process. Performing the dot product is a closest we can get to knowing whether the circuit semantics are still the same with 100% certainty.

3.4 Tool support and extensibility

The generation engine is built in such a way that it knows nothing about the specific language it is generating. Adding support for new targets becomes a matter of writing the template for the program, while using specific keywords for some constructs that need to be tracked by the generation engine, for example, the number of qubit definitions in a circuit.

Additionally, one needs to write a class with a function which when given a circuit, runs it on a backend and returns a dictionary where each key is a possible output of the program, and the value is the number of times that value is observed.

This section has gone through the core aims of the project, explained each of them in turn, and given a general overview of the core parts of the project.

4 Implementation

Contents

This section describes key design decisions that were made, then gives an in-depth account of how the templating language is defined, how the program AST is generated, how the AST passes are defined, how MAP Elites is implemented as an alternative program generation technique and how the differential testing library is implemented. Lastly, an overview of the entire testing loop is provided.

4.1 Analysis and Design

These are some of the main design choices I made and my reasoning behind them.

4.1.1 Terminal-based interaction with generation engine

The generation engine is run once and is interacted with by passing commands to it.

Listing 4.1: Terminal interface for the generation engine

```
$ ./qf
   ____  ______
  / __ \/ ____/___  ____
 / / / / /_  / __ \/ __ \
/ /_/ / __/ / /_/ / /_/ /
\___\_\/_/ /_.___/ .___/
                /_/   /_/

    Quantum Compiler Fuzzing Engine

Type 'h' for help.

Built pytket
Built guppy
Built cirq
Built cudaq
Built qasm2
Built qasm3
Built pennylane
Built qiskit

>
```

To generate a program, one passes the grammar name and an entry point into the grammar, for example, `pytket program` will generate an AST with `program` as its root. This means that any rule defined in the grammar can be used as an entry point, for example, `pytket qubit_op` would generate a qubit operation statement. One issue to consider is that some of these rules only produce meaningful ASTs in certain conditions, for instance, a pytket qubit operation is defined as shown in 4.2 which means at some point, it reaches the term `qubit`, which needs to return a random qubit from those that have been defined in the current circuit. By using `qubit_op` as an entry point, the rule which triggers the creation of qubits is skipped. To reconcile this, any node that relies on certain information to have been created beforehand is made in such a way that if that information does not exist, a dummy node is returned instead.

Listing 4.2: Examples showing pytket qubit op definition

```
1 qubit_op = gate_op | subroutine_op;
2
3 gate_op = ""[GET_CIRCUIT_NAME] "(" gate_op_args ")";
4
5 gate_op_args = (float_list ", ")[GET_GATE_PARAMS > 0]
6     qubit_list ("," bit)[GET_GATE_BITS];
7
8 qubit_list = (qubit ",")[GET_GATE_QUBITS];
```

This allows any entry point to still be used without breaking the generator, which is useful to

have for debugging purposes - often, one wants to see what effect the templates have when the AST is finally parsed into a program.

The full list of supported commands is given in 4.3

Listing 4.3: Supported commands

```
- print-tokens: Print tokens parsed from set grammar
- print-grammar: Print grammar data structure for set grammar
- map-elites: Toggle map elites algorithm
- seed: Set global seed
- info: Print circuit info
- quit \ Ctrl-D: Quit
- step: Step through program generation node by node in the AST
- render: Render AST(s) for program(s) generated
- <ENTER>: Enter on your keyboard to write the program to a file
- grammar-name entry-point: Generate program from template given by
    grammar-name starting from entry-point
```


With these commands, users can do the following without having to rerun the tool, or re-compile any code

- Quickly try generating some other program by simply passing a different set of commands
- See whether the lexer is tokenising the template correctly
- See whether the grammar data structure stores the template as expected
- Switch to map elites mode, an alternative to randomly sampling the grammar
- Set the seed when trying to reproducing previous previous results
- Print out all `Circuit` nodes in the AST to monitor what state they contain. This and other special node types are explained in 4.3.2
- Switch to step mode, which prints out in ASCII the AST as it is being created every time enter is pressed

This allows for quick and easy debugging.

4.1.2 Building grammar and AST separately

One might be able to eagerly build the AST on the fly while parsing the grammar templates. I opted not to do this as I preferred having the grammar in its own data structure I could test and refactor independent of anything else. Additionally, this separation is more efficient in the

sense that you only need to ”compile” the template once, then use it as a recipe to build as many ASTs as you want. The alternative would involve parsing the grammar from scratch each time. Having a separate grammar data structure also means you can use it for different things very easily once it has been built. An example is how passes, which are explained further in section 4.4, are implemented. The pass receives the built AST and grammar separately, and can refer to the grammar using any entry point to create a new node then add it to the AST.

4.2 Templating

The templating language is defined in 4.4

Listing 4.4: Definition of templating language

```
<character> ::= letter | digit | symbol | space

<identifier> ::= [a-z][A-Z][a-zA-Z0-9]*

<quoted_string> ::= '"' <character>* '"' | "'" <character>* "'"

<var_expr> ::=
    GET_CIRCUIT_NAME
    | GET_CIRCUIT_KIND
    | GET_GATE_NAME
    | GET_GATE_SOURCE
    | GET_GATE_QUBITS
    | GET_GATE_BITS
    | GET_GATE_PARAMS
    | GET_TOTAL_QUBITS
    | GET_TOTAL_BITS
    | GET_MAT_POS "(" <additive_expr> "," <additive_expr> ")"
    | UNIFORM "(" <additive_expr> "," <additive_expr> ")"
    | AST_HAS_NODE "(" <factor> ")"
    | MAKE_INTEGER
    | MAKE_FLOAT
    | MAKE_VAR
    | GET_DEF_NAME
    | GET_DECL_NAME
    | GET_SIZE
    | GET_INDEX

<prop_access_expr> ::=
    <identifier> ".from_reg"
    | <identifier> ".from_sing"
    | <identifier> ".is_register"
    | <identifier> ".is_singular"
    | <identifier> ".is_used_at_least_once"
    | <identifier> ".in_ext_scope"
    | <identifier> ".in_int_scope"
    | <identifier> ".name"
    | <identifier> ".index"
    | <identifier> ".size"

<mod_expr> ::=
    "int(" <expr> ")"
    | "str(" <expr> ")"
    | "kind(" <expr> ")"

<factor> ::=
```

```
        "(" <expr> ")"
        | digit+
        | digit+(.digit+)?
        | <var_expr>
        | <prop_access_expr>
        | <mod_expr>
        | <identifer>

    <multiplicative_expr> ::=
        <multiplicative_expr> "*" <factor>
        | <multiplicative_expr> "/" <factor>
        | <factor>

    <additive_expr> ::=
        <additive_expr> "+" <multiplicative_expr>
        | <additive_expr> "-" <multiplicative_expr>
        | <multiplicative_expr>

    <logic_expr> ::=
        <logic_expr> ">" <math_expr>
        | <logic_expr> "<" <math_expr>
        | <logic_expr> ">=" <math_expr>
        | <logic_expr> "<=" <math_expr>
        | <logic_expr> "==" <math_expr>
        | <logic_expr> "&&" <math_expr>
        | <logic_expr> "||" <math_expr>
        | <logic_expr> "!=" <math_expr>

    <else_block> ::=
        "else" ":" <block_expr>
        | "else" <block_expr>
        | "else " <if_expr>

    <if_expr> ::= "if" <logic_expr> <block_expr> <else_block>?

    <iterable> ::=
        ALL_QUBITS
        | ALL_BITS
        | ALL_PARAMS
        | ALL_QUBIT_DEFS
        | ALL_BIT_DEFS
        | ALL_PARAM_DEFS
        | ALL_GATE_QUBIT_DEFS
        | ALL_GATE_BIT_DEFS

    <for_expr> ::= "for" <identifier> "in" <iterable>

    <expr> ::= <for_expr> | <if_expr> | <rule_def> | <logic_expr>

    <wildcards> ::= "+" | "*" | "?"

    <scope-res> ::=
        "EXTERNAL::"
        | "INTERNAL::"
        | "GLOB::"

    <term> ::=
        <quoted_string>("["expr"]")?
        | (<scope-res>)?<identifier>("["expr"]")?
        | "["expr"]"
        | (<scope-res>)?<identifier>(<wildcards>)?
        | RESET
        | INDENT_LEVEL
        | "(" <term> ")"
        | LPAREN
        | RPAREN
        | LBRACK
        | RBRACK
```

```
        | LBRACE
        | RBRACE
        | COMMA
        | SPACE
        | DOT
        | SINGLE_QUOTE
        | DOUBLE_QUOTE
        | EQUAL
        | NL

    <branch> ::=
        <term>*
        | "ci<" <term>+ ">"
        | "si<" <term>+ ">"

    <special_rule_identifier> ::=
        circuit | sub_circuit | unitary_1q_def | unitary_2q_def
        | subroutine_defs | compound_stmts | compound_stmt
        | qubit_op | gate_op | subroutine_op | qubit | register_qubit
        | singular_qubit | bit | register_bit | singular_bit | param
        | register_param | singular_param | qubit_def | register_qubit_def
        | singular_qubit_def | bit_def | register_bit_def | singular_bit_def
        | param_def | register_param_def | singular_param_def | primitive_gate
        | h | x | y | z | t | tdg | s | sdg | v | vdg | project_z
        | cx | cy | cz | ch | xx | yy | zz | swap | cswap | toffoli
        | phased_x | u1 | u2 | u3 | rx | ry | rz | u | measure
        | classical_expr | bool_expr | uint_expr | if_stmt | do_while_stmt
        | while_stmt | for_stmt

    <rule_def> ::=
        <identifier> "=" (<branch>)* ";"
        | <identifier> "+=" (<branch>)* ";"
        | <special_rule_identifier> "=" (<branch>)* ";"
        | <special_rule_identifier> "+=" (<branch>)* ";"

    <qf> ::= (<rule_def>)+
        | "EXTERNAL" "{" (<rule_def>)+ "}"
        | "INTERNAL" "{" (<rule_def>)+ "}"
```

Program templates written in the language are first tokenised in the lexer, which produces a `std::vector<Token>`, where the `Token` struct contains the string literal of the parsed token, as well as its kind. The token kind is used by the grammar builder to know how to create the `Grammar` data structure and by the AST builder to assign specific node kinds to the node created from the corresponding token.

The `Grammar` data structure can be broken down as follows, where only the essential contents are shown here:

- `Grammar` contains `std::vector<std::shared_ptr<Rule>>`
- `Rule` contains
  - `std::vector<Branch>`
  - `Scope`
- `Branch` contains `std::vector<Term>`

- `Term` contains
  - `std::variant<std::weak_ptr<Rule>, std::string>` - the value of the term
  - `std::shared_ptr<Expr>` - the expression associated with the term

Consider the program in 4.2

Listing 4.5: Example program

```
program = qubit_op+;

qubit_op = gate_op | subroutine_op;

gate_op = primitive_gate gate_op_args;

primitive_gate = h;

h = "H";
```

- Pointers to `program`, `qubit_op`, `gate_op`, `primitive_gate` and `h` would be stored in the grammar
- The `qubit_op` rule would have branches `gate_op` and `subroutine_op`
- The term `gate_op` is a weak pointer to the `gate_op` rule

A `std::weak_ptr<Rule>` is used inside the term in order to prevent memory leaks caused by circular dependencies. Consider the program 4.2

Listing 4.6: Example program showing circular dependencies

```
compound_stmts = compound_stmt+;

compound_stmt =
    qubit_op
    | if_stmt;

if_stmt =
    'with ' ""[GET_CIRCUIT_NAME] '.if_test(' classical_expr
    ') as else_' INDENT_LEVEL ':' NL ci<compound_stmts>
    si<('with else_' INDENT_LEVEL ':' NL ci<compound_stmts>)>?;
```

If the term were `std::shared_ptr<Rule>`, whilst trying to destroy the grammar object, the reference to `compound_stmts` in `if_stmt` would prevent destruction of the `compound_stmts` rule in grammar, leading to a memory leak.

Another important data structure is `Current` which contains a stack, the top of which has the pointer to the rule definition that is currently being built (the current rule), and `Branch` which is the branch currently being built (the current branch). Each time the token `|` is met, it signifies

the end of a branch. The current branch is moved inside the current rule to prepare for the next branch.

When the `=` token is met, it signifies the start of a rule definition. The previous token is used as the name of the rule, and a `std::shared_ptr<Rule>` is pushed to the top of the stack in `Current`. On the right-hand side, anything that fits the definition of a term as shown for `<term>` in 4.4 is used to create a `Term` before being added to the current branch.

A stack is used to allow creation of anonymous rules when terms inside parentheses are met. For example, consider the program in 4.2

```
subroutine_defs = (  (sub_circuit | unitary_1q_def | unitary_2q_def) NL);
                  1  2                                               3   4
```

`subroutine_defs` is already on top of the stack when the opening parenthesis `1` is met, which triggers the push of a new rule, `NR_0`, on the stack. When `2` is met, another rule called `NR_1` is pushed to the stack. Now, everything that is parsed as a `Term` is added to the current branch, and on meeting the `|` token, the current branch is moved inside `NR_1`. When `3` is met, `NR_1` is popped from the stack, and a term containing it is added to the current branch, which is now associated to `NR_0`. The `NL` term will also end up in `NR_0`. Finally, when `4` is met, `NR_0` is popped from the stack, and a term containing it is added to the current branch. When `;` is met, the rule on top of the stack is added to the grammar.

This makes writing rule definitions easier for the user, because, for instance, one does not have to always create a rule to define a complex structure inside the branch, the parentheses implicitly creates the rule.

Rules can be used as terms before they have been defined. This is done to ensure no strict order is imposed, allowing for quick prototyping and creation of the templates. When a rule declaration is met whose pointer cannot be found inside the grammar, a new pointer is created and used, whereas if a pointer is found, that pointer is used.

4.2.1 Scope

Each rule has an associated scope, of which there's three; internal, external and global. These are essentially namespaces, allowing you to reuse the same rule name to define something different depending on what you want. Because these are used primarily for differentiating between resource

definitions within a circuit block (internal) and resource definitions that are function arguments, and therefore need to be assigned to when the function is called (external), `INTERNAL` and `EXTERNAL` are the only namespaces one can define. In addition to these two, `GLOB` scope can also be resolved.

Each scope kind is stored in an enum with a unique value that is a power of 2.

Listing 4.7: Enum for scope values

```
enum class Scope {
    NONE = 0,
    GLOB = BIT32(0),
    EXT = BIT32(1),
    INT = BIT32(2),
};

#define scope_matches(a, b) ((a & b) != Scope::NONE)
```

This allows scope comparisons to be made using bit-masking logic. For example, if we want to find the rule `qubit_def` above, but don't care which scope it is in, then setting the searching scope to `Scope::INTERNAL | Scope::EXTERNAL | Scope::GLOB` works.

When the grammar is being built, the rule declaration scope (any scope specified using `::` by rules on the right-hand side of `=`) and rule definition scope (scope specified on the left of `{`) are kept track of separately, both defaulting to global. The scope assigned to rule definitions is global unless it is specified by the namespace. The scope assigned to rule declarations is that of its parent rule unless specified by the namespace. `INTERNAL{...}` would be a scope specification for rule definitions, while `INTERNAL::` would be a scope specification for rule declarations.

Consider the example 4.8

Listing 4.8: Exmple program showing rule scoping

```
pi = "M_PI";
pi_multiples = pi | "("pi"/2.0)" | "("pi"/4.0)";
_float = ""[MAKE_FLOAT] | param | pi_multiples;

INTERNAL {
    float_no_param = ""[MAKE_FLOAT] | GLOB::pi_multiples;

    qubit_def = singular_qubit_def | register_qubit_def;
    param_def = singular_param_def;

    singular_param_def = "double " ""[GET_DEF_NAME] " = " float_no_param ";";

    singular_qubit_def = "cudaq::qubit " ""[GET_DEF_NAME] ";";
    register_qubit_def = "cudaq::qarray<" ""[str(GET_SIZE)] ">"
        ""[GET_DEF_NAME] ";";
}

EXTERNAL {
    qubit_def = singular_qubit_def;

    singular_qubit_def = "cudaq::qubit& " GET_DEF_NAME;
}
```

We specify definitions for `qubit_def` in both scopes, so these will resolve to two different rules in the grammar. When `EXTERNAL::qubit_def` is written as a declaration, the second definition is the one that is found. Note that when declaring `pi_multiples`, `GLOB::` had to be used, because otherwise, it inherits the scope of its parent, so when the declaration is met, it tries to find a rule pointer for a rule `pi_multiples` with internal scope, yet we want it to use the rule defined outside the braces.

4.2.2 Expressions

Defined as `<expr>` in listing 4.4, expressions are anything written in-between square brackets in the language, and are associated with terms. When an expression is evaluated, it returns `Expr_type`, a variant defined as

```
using Rule_list = std::vector<std::shared_ptr<Rule>>;
using Expr_type = std::variant<int, bool, float, std::string,
    std::shared_ptr<Rule>, Rule_list>;
```

Assuming the evaluation result is x, the following happens to a term depending on which type it evaluates to

- `int`: the term is copied $n = x$ times
- `bool`: if $x = True$, the term is left in the branch, otherwise, it is removed
- `std::string`: a `Term` containing a string is created, with the value being x
- `std::shared_ptr<Rule>`: a `Term` containing a rule is created, with the value being x
- `Rule_list`: a `std::vector<Term>` is created, where each term contains a rule x_i, where $0 < i < size(x)$

If the term expression evaluates to `float`, an error is thrown.

These are the supported expression types and the way in which they are evaluated

- `IntExpr`: Stores an integer. On evaluation, returns the integer stored
- `FloatExpr`: Stores a float. On evaluation, returns the float stored
- `VarExpr`: Stores a context resolvable variable and a set of arguments if it takes any. On evaluation, the variable is resolved using context and the stored arguments if needed. For

instance, `GET_MAT_POS(x, y)` gets the value at row x, column y in a random unitary matrix (which is stored as a circuit) and returns it as a string. Therefore, using this is semantically meaningful when the current circuit is a unitary matrix definition, because otherwise, it returns a string ”0”.

- `RuleExpr`: Stores the name of a rule. On evaluation, it returns the `std::shared_ptr<Rule>` that name is bound to within context
- `BlockExpr`: Stores a list of expressions. On evaluation, it evaluates each one in a loop, returning the value of the last expression it evaluates
- `PropertyAccessExpr`: Stores the name of the object, and the name of the access function. On evaluation, it gets the object to which the name is bound to within context, and calls a specific function defined by that object depending on the property access.
- `BinExpr`: Stores the left-hand side expression, right-hand side expression and binary op. On evaluation, the types of both expressions are checked to make sure they are the same, and either `int` or `bool`, then operated on depending on the operator
- `IfExpr`: Stores the condition, true and false expressions. If the condition evaluates to true, then the true block is evaluated, otherwise, the false block is evaluated if it has been provided.
- `ForExpr`: Stores the iterator name, the iterable, and the body expression. On evaluation, a for loop is performed over the iterables, on each iteration, binding the iterable to the iterator within context, then evaluating the body.
- `AssignExpr`: Stores the `std::shared_ptr<Rule>` that is defined. On evaluation, we create a new rule which will store the evaluated branches of the stored rule. This essentially creates a copy of the rule where all evaluations have been made in the given context.
- `ModExpr`: Stores the modifier, and the expression that we want to be modified. It evaluates the expression, then depending on the modifier, makes changes to the result before returning the final result. One use of this is to change the return type, such that the term replacement works differently - `""[def.size]` would copy the empty string term $def.size$ times because it evaluates to an integer. If we instead want to write the size itself, we can convert the size to a string `""[str(def.size)]`, which performs a replacement of the empty string term.

Consider the example

Listing 4.9: Example showing for loop expression

```
qubit_defs_measure =
    ""[
    for qdef in ALL_QUBIT_DEFS {
        if (qdef.in_int_scope) {
            singular_qubit_def_measure =
            "result('res_" ""[qdef.name] "', measure(" ""[qdef.name] "))" NL;
            register_qubit_def_measure =
            "mz_" ""[qdef.name] " = measure_array(" ""[qdef.name] ")" NL;
            if (qdef.is_singular): singular_qubit_def_measure
            else: register_qubit_def_measure
        }
    }
    ]
    ;
```

When the grammar is being built, the opening square bracket signals the start of an expression. Once parsed, the expression AST will look as shown in figure 4.1

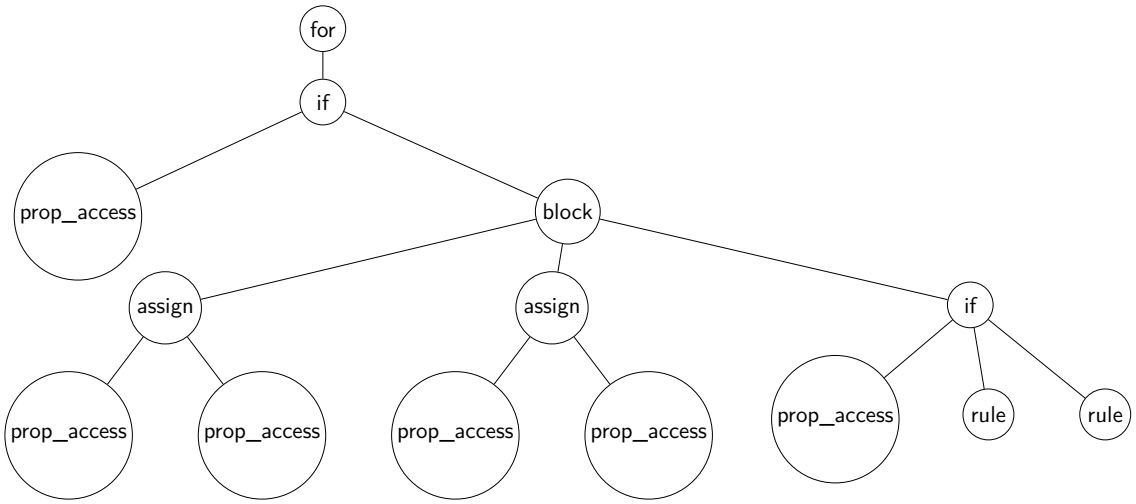


Figure 4.1: AST for the expression in listing 4.9

The expression AST is then evaluated at runtime using context, for instance, `ALL_QUBIT_DEFS` returns all qubit definitions generated by the current circuit node. When evaluating the `for` node above, a for loop is performed over the qubit definitions. On each iteration, the qubit definition is bound to the iterator `def` within context, such that when the `var_access` nodes are evaluated downstream, they can look inside context, grab the correct qubit definition and return the relevant data depending on the access function used. The return type of the `for` node is `Rule_list`; the return type of `block` is that of `if`, because the block always returns the last expression it evaluates. The return type of `if` is `std::shared_ptr<Rule>`, so you end up with as many rule pointers at the there are qubit definitions in the circuit.

4.3 Program AST generation

Program AST generation follows these steps

1. Entry point into grammar is provided. This is a rule name which forms the root of the AST. The rule pointer is embedded into a term, then passed to the node factory
2. The term contains a rule pointer, a random branch is picked, while satisfying any branch constraints that might be imposed. If the term contains a terminal (string, integer, float), a terminal node is returned
3. For each term in the branch, the expression, if it has one, is evaluated, then a node is created from the term depending on its type. The returned node will sometimes have branch constraints added to ensure the branch chosen to create its children has certain properties that maintain correctness
4. The child node and child term are passed as arguments on the next recursive call to the node factory, continuing the process from step 2

Before explaining how the node factory works, we need to first understand how branch constraints work

4.3.1 Branch constraints

A branch constraint contains two elements: a token kind, and the number of occurrences it must have in a branch. When picking a random branch from a rule, the branch must also satisfy this constraint in order to be picked. Say I have a rule definition `primitive_gate = x | y | u1 | u2;`, and for whatever reason, I want to pick a specific gate, say the X gate. I can do that by setting a branch constraint which forces the branch with the `x` term to be picked. The definition of the `Branch_constraint` struct is shown in listing 4.10.

Listing 4.10: Definition of the `Branch_constraint` struct

```
1 struct Branch_constraint {
2     public:
3         Branch_constraint(){}
4 
5         Branch_constraint(Token_kind _singleton_term_token_kind,
6             unsigned int _n_occurances
7         ):
8             singleton_term_token_kind(_singleton_term_token_kind),
```

```
            n_occurances(_n_occurances)
        {}

        inline bool passed(const Branch& branch){
            return branch.count_rule_occurances(singleton_term_token_kind)
            == n_occurances;
        }

    private:
        Token_kind singleton_term_token_kind;
        unsigned int n_occurances;
};
```

4

When a branch is being picked, the branch constraint is used to pick the branch that passes the constraint as shown in listing 4.11.

Listing 4.11: Picking a branch by satisfying constraints

```
Branch Rule::pick_branch(std::shared_ptr<Node> rule_as_node){
    size_t size = branches.size();

    bool valid_branch_exists = false;

    for (const auto& branch : branches) {
        if (rule_as_node->branch_satisfies_constraints(branch)) {
            valid_branch_exists = true;
            break;
        }
    }

    if(valid_branch_exists){
        Branch branch = branches[uniform_uint(size - 1)];

        while(!rule_as_node->branch_satisfies_constraints(branch)){
            branch = branches[uniform_uint(size - 1)];
        }

        return branch;

    } else {

        #ifdef DEBUG
        if(rule_as_node->has_branch_constraint()){
            std::cout <<
            "No branch exisits for this rule that satisfies constraints"
            << std::endl;
            std::cout << *this << std::endl;
            rule_as_node->print_branch_constraint(std::cout);
        }
        #endif

        return Branch();
    }
}
```

Note that we first check whether any valid branch exists to prevent starting an infinite loop.

4.3.2 Node factory

The core of AST node creation is the `make_child` function which maps the term to its corresponding node type. Most terms return the base node type, but some return special node types which carry additional state depending on the rule embedded in the term. These special nodes are outlined below.

- `circuit` and `sub_circuit` produce `Circuit` nodes. The types of resources are qubits, bits, and parameters. These circuit nodes store their resource definitions, the resources that can are derived from those definitions

- `unitary_1q_def` and `unitary_2q_def` also produce `Circuit` nodes but these store a random unitary matrix, which is generated using Gram-Schmidt column orthogonalisation of a complex Gaussian random matrix. When `GET_MAT_POS` is used as shown in 4.12, it gets the complex number at the corresponding (x, y) coordinate in the matrix in the current circuit.

Listing 4.12: Example showing creation of random 1q unitary

```
unitary_1q_def =
"CUDAQ_REGISTER_OPERATION("                                                    NL
""[GET_CIRCUIT_NAME] ", "                                                      NL
"    1,    // Number of qubits"                                               NL
"    0,    // Number of params"                                               NL
"    (std::vector<std::complex<double>>{"                                      NL
"           {"""[GET_MAT_POS(0,0)] "},{" ""[GET_MAT_POS(0,1)] "},"           NL
"           {"""[GET_MAT_POS(1,0)] "},{" ""[GET_MAT_POS(1,1)] "} "           NL
"    })"                                                                       NL
");"                                                                           NL
;
```

- `qubit_op` produces a `Qubit_op` node, which tracks the name of the circuit that called it and stores a pointer to its associated `Gate` node, set later when a gate token is encountered.

- Gate tokens (`h`, `cx`, `rz`, etc.) produce `Gate` nodes. Each `Gate` stores a `Gate_info` struct looked up from the `SUPPORTED_GATES` table, which records the number of qubits, bits and parameters required. This information is later used by expression evaluation and branch constraint enforcement.

- `subroutine_op` either produces a `Gate` referencing a previously built sub-circuit or unitary, or redirects to a `gate_op` term if no applicable subroutine exists in the current context. There is a check which verifies that at least one previously built circuit has compatible resource counts with the current circuit before committing to a subroutine call.

- `qubit`, `bit` and `param` produce clones of `Resource` nodes drawn at random from the current circuit's resource collection, marking them as used to prevent reuse within the same gate operation. The `Resource` node will have a branch constraint that forces the child node to be a register or singular resource depending on what was picked.

- `register/singular qubit`, `bit` and `param` produce clones of their parents, which are also guaranteed to be `Resource` nodes.

- `qubit_def`, `bit_def` and `param_def` produce `Resource_def` nodes, which randomly decides whether to use a register or singular definition depending on which corresponding grammar rules are non-empty. For instance, if the rule `register_qubit_def` has no branches, it is taken that the user wants to disable creation of register qubit definitions, so only singular definitions are created. One needs to define at least one of register or singular resource definition rules. The definition is stored in the current `Circuit`, which also expands it into individual `Resource` entries. The `Resource_def` node will have a branch constraint that forces the child node to be a register or singular resource definition depending on what was created.

- `register/singular qubit_def`, `bit_def` and `param_def` produce clones of their parents, which are also guaranteed to be `Resource_def` nodes.

- `primitive_gate` produces a `Primitive_gate` node that pre-computes branch constraints at construction time, forbidding any gate that requires more qubits or bits than are available in the current circuit.

- `compound_stmt` produces a `Compound_stmt` node. At nesting depth zero, a branch constraint is added that requires exactly one `Qubit_op` child, preventing further control flow nesting.

- `RESET` does not produce a meaningful node but has the side effect of resetting qubit and bit usage within context, allowing the same resources to be selected again in subsequent gate operations.

With this in mind, certain rules have definitions that aren't supposed to be changed, as far as the terms contained in the branches go.

Listing 4.13: Rules that must be followed

```
# other
program = .... circuit .... |  ... body ...;
```

```
body = .... compound_stmts ... | .... some_rule ... ;
some_rule = ... compound_stmts ...;

# resource definitions
qubit_def = singular_qubit_def | register_qubit_def;
bit_def = singular_bit_def | register_bit_def;
param_def = singular_param_def | register_param_def;

# primitive gates
primitive_gate = h| x | y | z | ....;

# resources
qubit = singular_qubit | register_qubit;
bit = singular_bit | register_bit;
param = singular_param | register_param;

# compound stmt
compound_stmt = qubit_op | if_stmt;

if_stmt =
    'with ' ""[GET_CIRCUIT_NAME] '.if_test(' classical_expr ')
    as else_' INDENT_LEVEL ':' NL ci<compound_stmts>
    si<('with else_' INDENT_LEVEL ':' NL ci<compound_stmts
```

Having the setup in 4.13 makes it such that a branch constraint on `compound_stmt` can find a branch just a `qubit_op` term. Note however, that one can put whatever they want around the term, and the branch constraint will still work, for instance `compound_stmt = some_rule " = " some_other_rule qubit_op NL | if_stmt;` would still work.

The lexer also recognizes certain rule names and assigns them specific token kinds such that the corresponding nodes can be found in the AST, which is useful when performing mutations on the AST. Note the `#other` block in 4.13, which states that `circuit` or `body` must be used as children of the `program` rule. This is because they are looked for in the AST as the main compilation unit. `compound_stmts` must also be used to define the block of operations in all circuits. This is the node on which most mutation passes apply.

Because some of the rule definitions are meant to be ”constant” as shown above, while others

will have the same structure in most languages, a separate `common.qf` file is imported automatically by all templates written. Any of the rules defined in the common library can be overwritten (`rule = .....;`) or appended to (`rule += .....;`).

4.3.3 Indentation

When generating control flow blocks in languages like Qiskit, one needs to respect indentation. This can be set in the templates using:

- `ci<some_rule>`: Child indent. It creates a tab whose length is defined by the current control flow depth, and adds the tab in-front of each child node created from `some_rule`

`si<some_rule>`: Self indent. It creates a tab whose length is defined by the current control flow depth, and adds the tab in-front of the node created from `some_rule`

Listing 4.14: Example showing indentation application

```
1    compound_stmts = (compound_stmt NL)[UNIFORM(15,25)];
2
3    if_stmt =
4    'with ' ""[GET_CIRCUIT_NAME] '.if_test(' classical_expr ') as else_'
5    INDENT_LEVEL ':' NL ci<compound_stmts>
6    si<('with else_' INDENT_LEVEL ':' NL ci<compound_stmts>)>?;
```

`ci<compound_stmts>` adds the tab in-front of each node under the `compound_stmts` node (which would be anonymous due to the use of brackets `compound_stmts = (compound_stmt NL);`).

`si<('with else_' INDENT_LEVEL ':' NL ci<compound_stmts>)>` adds the tab in-front of `with else_` only, because an anonymous rule is created that defines all the terms within the brackets `('with else_' INDENT_LEVEL ':' NL ci<compound_stmts>)`. Using `si<...>` without the brackets would add an indentation before each of the terms instead.

4.4 AST passes

Once the AST has been built, a set of passes can be used to mutate the AST in various ways. A `Pass` receives:

- the AST

- the grammar from which the AST was built
- the node kind k on which the pass is performed
- the name of the rule in the grammar from which a node of kind k is built
- the ratio of all nodes of kind k in the AST on which the pass should apply
- a map of node kind k' to branch constraint. The branch constraints are added to nodes of kind k' that are built while creating new nodes from the grammar

It also implements a function `Pass::apply`, which collects all nodes in the AST of the given kind k, then loops over the collected nodes, calling `apply_blockwise`, a pure virtual function that must be implemented by any pass derived from the base `Pass` class. The nodes are collected first then looped over second to ensure that the passes are only activated for the nodes present in the AST after it is built for the first time. If application of the block-wise function is performed within the collection loop, new nodes created by some mutations will also be found, which could lead to a situation where the pass runs indefinitely. The grammar is needed because it is the source of truth about the structure that must be followed when creating the AST.

From this base class, the following set of passes are built

- `Add_child`: Creates a new child node for the given node of kind k
- `Erase_child`: Removes a child node for the given node of kind k
- `Replace_block`: Finds the node of kind k, then replaces it with a new node formed by the given rule r in the grammar
- `Mutate_on_condition`: Receives a `Pass` and a condition which when true, that pass' implementation of `apply_blockwise` is called
- `Add_gate_chain`: Receives a vector of n gate kinds and adds qubit operations that use those gates on the same qubits. This implies that all gates in the chain must expect the same number of input qubits
- `Remove_gate_chain`: Removes a pair of qubit ops if they are *interesting*, which is determined by a checker function that is given to this pass. For example, you might want to remove gate pairs that are inverses of each other (applying them in order on the same qubit is the same as doing nothing to the qubit), which can be done with this pass

- `Combine`: Receives a list of passes and applies them in order
- `Dead_subs`: Performs reachability analysis to find which subroutines are never reached from the main circuit, and therefore can safely be deleted from the AST

The `Dead_subs` pass was initially written to circumvent an error in CUDA-Q, which would not allow compilation of programs that contained subroutine definitions that were never called anywhere in the program. Consider the code in 4.15

Listing 4.15: Example showing need for dead code elimination for subroutines

```
#include <cudaq.h>
#include <iostream>
#include <cmath>
#include <vector>
#include <complex>

struct sub_1 {
void operator()(cudaq::qubit& sing_64,cudaq::qubit& sing_69,
    cudaq::qubit& sing_74,cudaq::qubit& sing_79, double sing_85) __qpu__ {
    unitary_0(sing_64);
    if (!mz(sing_69)){
        unitary_0(sing_69);
        z(sing_64);
        if (!mz(sing_64)){
            swap<cudaq::ctrl>(sing_64, sing_74, sing_79);
}       else {
            rx<cudaq::ctrl>(1.735096, sing_69, sing_74);
}
        s(sing_64);
}   else {
        r1((M_PI/4.0), sing_69);
}
    z<cudaq::ctrl>(sing_69, sing_74);
}};

struct sub_2 {
void operator()(cudaq::qubit& sing_732,cudaq::qubit& sing_737,
    cudaq::qubit& sing_742,cudaq::qubit& sing_747, double sing_753) __qpu__ {
    if (!mz(sing_732)){
        if (!mz(sing_737)){
            u3(M_PI, 2.666755, 2.350925, sing_732);
        } else {
            y<cudaq::ctrl>(sing_737, sing_742);
        }
        t<cudaq::adj>(sing_732);
    } else {
        rx<cudaq::ctrl>(sing_753, sing_732, sing_737);
    }
    y<cudaq::ctrl>(sing_737, sing_732);
}};

struct main_circuit {
uint64_t operator()() __qpu__ {
    cudaq::qubit sing_1454;
    cudaq::qubit sing_1461;
    cudaq::qarray<2>reg_1468;

    double sing_1480 = 5.739464;
    double sing_1490 = 4.899212;
    double sing_1500 = 8.411851;

    if (!mz(sing_1454)){
```

```
53          if (mz(sing_1461)){
54              x<cudaq::ctrl>(sing_1461, reg_1468[0], sing_1454);
55              sub_2{}(sing_1454, reg_1468[0], reg_1468[1], sing_1461, 4.044616);
56          }
57          swap(sing_1454, sing_1461);
58      } else {
59          s(sing_1461);
60      }
61  }};
```

The compile has one phase which goes over the AST, finds all kernels and registers them to the CUDA-Q runtime such that they can be run on the QPU or simulator. A separate pass C++ pass is done in order to compile the classical constructs. The compiler will see that `sub1` is never used, hence performing dead code elimination. At the linker phase, the linker will have the generated registration function for all kernels, but will fail to find the kernel definition because it will have been removed. The solution is to perform dead code elimination on the AST generated by qf++ before sending the program to the compiler.

4.5 MAP Elites

The default AST generation strategy is random sampling of the grammar. The tool also implements a generation mode based on the Multi-dimensional Archive of Phenotypic Elites (MAP Elites) algorithm - a type of search algorithm that aims to produce a large diversity of high performing, yet qualitatively different solutions in a user-defined search space [31]. The algorithm roughly works as shown in 1

Note that it may not be possible to completely fill the archive, because

- Depending on the feature space defined, some cells may not physically impossible to obtain
- The mutation passes defined may not be able to reach certain cells

In my implementation, I define the feature space on these dimensions:

- Inverse pair count - the number of gate pairs on the same qubits that are inverses of each other; 20 bins
- Entanglement density - the ratio of all qubit ops that operate on more than 1 qubit at once; 10 bins
- Non-Clifford density - the ratio of qubit ops that are not part of the Clifford gate set; 10 bins

Algorithm 1 MAP-Elites AST Generation

Require: Number of seed genomes N_{genomes}, Target grammar G

procedure GENERATOR::MAP_ELITES(N_{genomes}, G)
    ▷ Step 1: Define search space qualitatively
    $\mathcal{A} \leftarrow$ Initialize `Archive`

    ▷ Steps 2, 3 & 4: Initialize, explore, and populate
    ARCHIVE::FILL_ARCHIVE($\mathcal{A}, N_{\text{genomes}}, G$)
    return $\mathcal{A}$
end procedure

procedure ARCHIVE::FILL_ARCHIVE($\mathcal{A}, N_{\text{genomes}}, G$)
    Initialize $\mathcal{A}$ with $N_{genomes}$ random seed programs

    $n_{evals} \leftarrow 200,000$
    $patience \leftarrow 2000$ ▷ Evals in which at least one new cell must be discovered
    while $i < n_{evals}$ & $j < patience$ **do**
        $p \leftarrow$ Select a random program from $\mathcal{A}$
        $p' \leftarrow$ ARBITER::APPLY(p, G) ▷ Explore by mutating program

        $c' \leftarrow$ Calculate feature cell for p'
        if $\mathcal{A}[c']$ is empty **then**
            $\mathcal{A}[c'] \leftarrow p'$ ▷ Mutant lives in new cell
            $j = j + 1$
        else if QUALITYSCORE(p') > QUALITYSCORE($\mathcal{A}[c']$) **then**
            $\mathcal{A}[c'] \leftarrow p'$ ▷ Replace with higher quality mutant
        end if
        $i = i + 1$
    end while
end procedure

An additional bin for counts that exceed the maximum is always added, therefore the feature space has a size

$$S = 21 * 11 * 11 = 2541$$

The quality is calculated as a weighted sum of these metrics

- Interaction graph diversity - The ratio of all multi qubit operations that operate on unique qubit entanglements; $w = 0.8$
- Reducibles density - the ratio of adjacent qubit ops that are either commutative or inverses of each other $w = 0.2$

My rationale for the these choices is the following - the goal of MAP Elites in this case is to find programs that increase the likelihood of finding bugs in compilers, therefore, the feature space should be defined in such a way that bug-inducing programs can be feasibly found at either end of the spectrum of each axis in the feature space, and the quality should be higher for programs

that would be harder for a quantum compiler stack to deal with.

Programs with high interaction graph diversity are harder to compile because they introduce non-local qubit interactivity, which increases the likelihood of SWAP insertion. Circuits with high density of known inverse or commutative pairs are more likely to exercise the compilers reduction passes. However, this metric is given a lower weighting to dis-incentivize trivially reducible sequences in the circuit.

The compiler behaves differently when given a circuit filled with many single qubit gates and when given another with many multi qubit gates. Therefore, to generate test cases that could exercise both parts of the compiler, entanglement density is used as a feature. The same goes for circuits with many clifford gates vs those with less, and circuits with many inverse pairs vs those with less. The general idea for the feature space is to define axis that could produce circuit phenotypes that would exercise different code paths in the compiler.

4.5.1 Arbiter

The archive filling stage uses the passes described in 4.4 to define mutations that perform specific tasks, for example, the `Add_gate_chain` is used to define mutations that add known self-inverse pairs to the program

Listing 4.16: Code snippet showing use of `Add_gate_chain` pass

```
const std::vector<Token_kind> SELF_INVERSE_PAIRS = {
    H, X, Y, Z, CX, CY, CZ, SWAP, CCX, CSWAP, TOFFOLI};

for (const auto& token_kind : SELF_INVERSE_PAIRS){
    std::vector<Token_kind> pair = {token_kind, token_kind};

    arbiter.add(std::to_string(token_kind) + "--" + std::to_string(token_kind),
        [pair](Ast_entry& e, std::shared_ptr<Grammar> g, float r) {
            return std::make_unique<Add_gate_chain>(e, g, r, pair);
        }
    );
}
```

The fill phase halts after 200,000 runs or if 2000 runs have passed since a new cell was discovered - whichever happens first. On each iteration, the arbiter picks the mutation with the highest Upper Confidence Bound (UCB) [33] score calculated as

$$a(i,t) = s_{i,t} + \sqrt{\frac{2\log(t)}{p_{i,t}}}$$

where $p_{i,t}$ is the number of trials for mutation i at iteration t, $s_{i,t}$ is the success rate of arm i at

time step t, calculated as the fraction of $p_{i,t}$ trials found new cells. By using this scoring mechanism, the arbiter finds by itself the right balance of exploration of mutations that haven't been used often with exploitation of mutations that have recorded high success rates. The blockwise ratio of each mutation is also dynamically chosen by the arbiter depending on the success rate - if success rate increases, the block-wise ratio is increased by a small delta i.e the mutation affects larger parts of the AST, otherwise, the block-wise ratio is reduced.

4

4.6 Differential testing

A minimal base class which provides most of the functionality needed for differential testing is written, reducing the work needed to add support for testing new stacks. The only requirement for testing a new stack is the implementation of a class that inherits `Base`, and implements the function `_get_counts`, which compiles the given code at some optimisation level and returns a dictionary mapping each possible output to the number of times it is observed. This function is used to perform KS-test [34] between the distribution formed from outputs at that optimisation level and the baseline. Statevector comparison can be used instead if supported by the framework; this requires implementation of the `_get_statevector` function, which runs the given program on a statevector simulator, and returns the output statevector.

Plotting functionality is also provided, which is useful for visualizing output distributions and their eCDF.

The corresponding testing class is then called within the templates

Listing 4.17: Calling testing function

```
imports =
    ...
    "from diff_testing.qiskit import qiskitTesting" NL
    ...
    ;

program = imports NL[2] subroutine_defs NL circuit NL compiler_call;


compiler_call =
    make_param_dict NL
    ""[GET_CIRCUIT_NAME] ".measure_all()" NL
    "qt = qiskitTesting(" ""[GET_CIRCUIT_NAME] ","
    ""[str(GET_CIRCUIT_ID)] ", param_dict)" NL testing_method;

testing_method = opt_ks_test;

opt_ks_test = "qt.opt_ks_test()" NL;
```

4.7 Testing loop

A run script is provided, making it easy to run the generation and testing of programs while marking interesting ones and creating a report for them. The entire algorithm runs as shown in 2

Algorithm 2 QuteFuzz Execution Pipeline

Require: Target framework grammars G, Number of tests N, Significance level α, Evaluation mode *mode*
Ensure: Set of interesting circuits I

$I \leftarrow \emptyset$
CleanAndBuildFuzzer ▷ Compiles the C++ generation core
for each grammar $g \in G$ **do**
▷ Phase 1: Program generation
    $C \leftarrow$ GenerateTests(g, N)
▷ Phase 2: Concurrent validation
    for each circuit $c \in C$ **in parallel do**
        $res \leftarrow$ RunCircuit(c) ▷ Compile and execute program
        if res indicates a crash **or** timeout **then**
            $I \leftarrow I \cup \{c\}$
        else if res successfully evaluated **then**
            $KS, DP \leftarrow$ ParseMetrics(res) ▷ Regex search for ks values or dot product
▷ Evaluate KS p-value or dot product
            if $\min(KS) < \alpha$ **or** $DP \neq 1.0$ **then**
                $I \leftarrow I \cup \{c\}$
            end if
        end if
    end for
▷ Phase 3: Save interesting circuits if running nightlies
    if *mode* = NIGHTLY **and** $|I| > 0$ **then**
        SaveInterestingCircuits(I)
    end if
end for
return I

After running in nightly mode, the report is saved in a time-stamped directory which contains copies of all interesting circuits, a file containing the regression seed, and a file containing a report detailing why each of the stored circuits were marked as interesting.

This section has described key design decisions that were made, has given an in-depth account of how the templating language is defined, how the program AST is generated, how the AST passes are defined, how MAP Elites is implemented as an alternative program generation technique, and how the differential testing library is implemented. Lastly, an overview of the entire testing loop is provided.

4

5 Results

Contents

This section presents coverage results for the CUDA-Q compiler to compare programs generated MAP Elites with those generated using random grammar sampling and compare coverage obtained from different parts of the compiler. A few interesting bugs are chosen and explained, starting from the programs that caught them, their root cause, and how they were fixed. The MAP Elites algorithm is used to provide insights into the kinds of programs generated by the fuzzer, and finally, the test loop efficiency results are presented.

The tool currently supports generation and testing for

For compilers that do not follow the traditional ”optimisation-level” setup, I define a series of pass pipelines of increasing complexity. One needs to be careful with this, because some of the passes applied to quantum programs are mandatory, in the sense that without them, the circuit will not be able to run on a QPU or emulator. Consider the array conversion pass in CUDA-Q

Target Framework	Differential Testing Strategy
Pytket	KS test across optimisation levels 0–3
Qiskit	KS test across optimisation levels 0–3
Cirq	KS test across 3 custom transpilation levels
PennyLane	KS test across 4 transform pipelines of increasing complexity
CUDA-Q	KS test across pass pipelines of increasing complexity
OpenQASM2	KS test p-value agreement at all opt levels for Cirq, Pytket and Qiskit
Guppy	KS test across 3 transform pipelines of increasing complexity

Table 5.1: Supported quantum software stacks and their differential testing approaches.

Listing 5.1: Snippet from CUDA-Q compiler script

```
if ${ENABLE_ARRAY_CONVERSION}; then
    RUN_OPT=true
    OPT_PASSES=$(add_pass_to_pipeline "${OPT_PASSES}"
    "func.func(quake-add-metadata,constant-propagation,lift-array-alloc),
    quake-propagate-metadata,globalize-array-values,canonicalize,
    get-concrete-matrix")
fi
```

When we define a custom unitary matrix

Listing 5.2: CUDA-Q custom unitary

```
using namespace std::complex_literals;
CUDAQ_REGISTER_OPERATION(
unitary_0,
    1,    // Number of qubits
    0,    // Number of params
    (std::vector<std::complex<double>>{
       {-0.16543969906842199, 0.47833421588821279},
       {0.19679964894805152, 0.83969993572506729},
       {0.7586982038862421, 0.41012573596709212},
       {-0.5049791889390931, 0.034204310483580157}
    })
);
```

the vector needs to be converted into a static, global, array allocation because there's no heap memory or an operating system to dynamically allocate that memory on a quantum computer or emulator.

Below are short descriptions of the passes at each optimisation level in CUDA-Q. I only describe the CUDA-Q passes because the results reported in this section will be obtained after running that compiler since it was the only one I could instrument within the time constraints. A "+" indicates passes I added, "-" indicates disabled passes, _ indicates nothing was done.

- **Level 0**: -aggressive inlining -variable coalescing -lambda lifting
- **Level 1**: -aggressive inlining

- **Level 2**: _
- **Level 3**: +loop unrolling +loop peeling

5.1 Code coverage

In order to examine the quality of test cases produced by the tool, I compiled the CUDA-Q compiler from source with coverage instrumentation. Higher coverage implies better test cases, because they would be able to test more independent parts of the compiler, hence increase the probability of finding bugs.

The 4 statistics used are function coverage, line coverage, region coverage and branch coverage [35] from least to most granular. The code in 5.3 will be used as a reference to explain each of these statistics.

Listing 5.3: Example to showcase the different forms of coverage being collected

```
1  // target_function.cpp
2
3  int calculate_value(int y, int z) {
4      int x = 0;
5
6      if (y == 2 || z < 10) {
7          x = 1;
8      } else {
9          x = 2;
10     }
11
12     int a = (x == 1) ? 100 : 200;
13
14     return a;
15 }
```

- **Function coverage**: The percentage of fractions that have been executed, where a function is considered to have been executed if any of its instantiations were executed. If the `calculate_value` function is called at any point, we would get 100% function coverage for the `target_function.cpp` file.
- **Line coverage**: The percentage of code lines that have been executed at least once, where only lines within a function body are considered as code lines. If we have a call `calculate_value(5, 10);` somewhere, line 4 and 6 are executed, the condition evaluates to false, line 7 is not executed, but line 9 is executed. line 12 and 14 are executed. So 83% line coverage for this file.

- **Region coverage**: This is an LLVM-specific statistic which looks at regions of the AST and asks whether they were executed at least once. This is especially useful for statements with multiple things happening on the same line, for instance, line 12 in the example code. If the function calls are such that `x == 1` is always true, the line coverage would be 100% for line 12, but the region coverage is 66% because the false part of the ternary operator is never executed.

- **Branch coverage**: The fraction of all possible branch outcomes that have been executed. Consider the condition in line 6; it contains 2 conditions, which means there are 4 possible outcomes:

  1. `y == 2` is true, we short-circuit to line 7
  2. `y == 2` is false we evaluate `z < 10`
  3. `z < 10` is false, we execute line 9
  4. `z < 10` is true, we execute line 7

  Achieving 100% branch coverage would imply all true/false decisions like that went in all possible outcomes at least once

My coverage results are split into full coverage - reports for the entire repository, cudaq coverage - reports for the frontend, optimiser, and codegen parts of the compiler and eigen coverage - reports for the eigen linear algebra library. This is done to ensure that the results can be interpreted within the correct context while focusing on the important parts. When collecting cudaq coverage, the following parts are disregarded

- Any `.inc` files were ignored because these are generated from high level dialect descriptions written by developers in TableGen [36]. This is specific to LLVM / MLIR compilers. Ideally we only want to view coverage for files that we are interested in testing for bugs, generated files are not one of them because a bug in this file implies either a bug in `mlir-tablegen` or a mistake in how the dialect is described, both of which are not our interest.

- Any internal utility functions are left out

- IR verifier functions are left out

- `/cudaq/tools` is left out as this contains implementations for the command line tools `cuda-translate`, `cuda-quake` and `cuda-opt`

The coverage tables shown below are colour-coded such that for a coverage percentage **x**, the cell will have a colour:

- **Red** if **x** $< 50\%$
- **Yellow**: if $50\% \leq$**x** $\leq 75\%$
- **Green**: if **x** $> 75\%$

5.2 Comparing MAP Elites implementation to random sampling

Figure 5.1 shows overall coverage for the core compiler. The graph compares coverage outputs obtained when the grammar templates are randomly sampled in comparison to when MAP Elites is used.

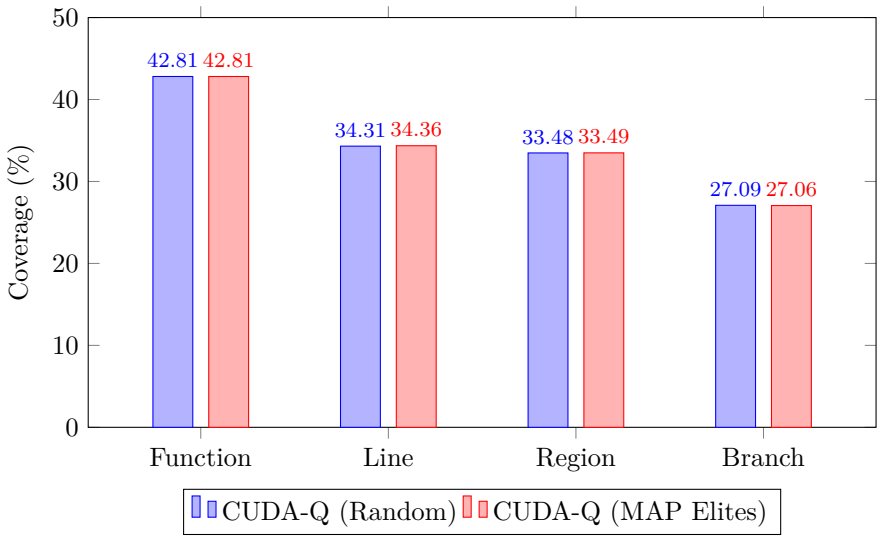


Figure 5.1: Code coverage metrics for CUDA-Q compiler core

The coverage results for random sampling were obtained after running a suite 50 programs, whereas the coverage results for map elites were obtained after seeding the archive with 70 random programs, then running the algorithm, which yielded 43 final programs.

These are essentially the same coverage values across all statistics. This could be explained by considering that the mutations performed on the circuit do not modify the topology of the circuit that much - they only add reducible inverse pairs, commutative pairs, replace cliffords with

non-cliffords and vice versa. In the limit, the resulting circuits will look much like those produced by randomly sampling the grammars. Perhaps a different set of mutation passes (and therefore a different feature space definition) would produce circuits that cover different parts of the compiler. Additionally, it is worth noting that the mutations can only add what is already defined in the grammars i.e a mutation might choose to add a `for_stmt` only if it is defined in the grammar. Therefore, a more straightforward way to improve coverage results could be to consider which constructs would interact with certain parts of the compiler, then defining templates for them. The experiment in 5.2.1 tries to do that.

5

5.2.1 Experiment - Improving coverage via feedback

A CUDA-Q `compound_stmt` is defined as shown below

```
compound_stmt  = qubit_op ";" | if_stmt;

qubit_op       = gate_op | subroutine_op;

if_stmt =
    'if (' '!'? 'mz(' qubit '){' NL ci<compound_stmts> "}" si<(else_stmt)?>;
else_stmt =
    'else {' NL ci<compound_stmts> "}";
```

yet I still see coverage for a loop unrolling pass in table 5.2

File Name	Function Cov.	Line Cov.	Region Cov.	Branch Cov.
lib/Optimizer/Transforms/LoopPeeling.cpp	0.00% (0/4)	0.00% (0/74)	0.00% (0/29)	0.00% (0/20)
lib/Optimizer/Transforms/LoopUnroll.cpp	50.00% (3/6)	39.06% (25/64)	40.91% (9/22)	20.00% (3/15)

Table 5.2: Code coverage metrics for loop-related files

The reason for this is that I generate for-loops to assign values to my parameter arrays and measure off the qubits before returning the final result.

```
# Register parameter definition
INTERNAL {
    # after initing the array, use a for loop to assign floats to the array
    register_param_def =
        "double " ""[GET_DEF_NAME] "[" ""[str(GET_SIZE)] "];" NL[2]
        "for(int i = 0; i < " ""[str(GET_SIZE)] "; i++){" NL
        ""[GET_DEF_NAME]"[i] = " float_no_param";" NL
        "}" NL
    ;
}

# Qubit measurment
measures =
    "uint64_t packed_result = 0; int bit_offset = 0;" NL[2]

    ""[
    for qdef in ALL_QUBIT_DEFS {
        if (qdef.in_int_scope) {
```

```
            if (qdef.is_singular){
                singular_measure =
                    "if (mz(" ""[qdef.name] "))"
                    "{ packed_result |= (1ULL << bit_offset); }" NL
                    "   bit_offset += 1;" NL;

                singular_measure

            } else {
                register_measure =
                    "for (auto b : mz(" ""[qdef.name] ")){" NL
                    "   if (b) packed_result |= (1ULL << bit_offset);" NL
                    "   bit_offset++;" NL
                    "}" NL;
                register_measure
            }
        }
    }
    ]
    "return packed_result;"
    ;
```


After looking through the source code [1], we see that the loop peeling pass in CUDA-Q is only activated when we have do-while loops in the program. We can increase the loop unrolling coverage my adding more for loops in the programs, and increase the loop peeling coverage by adding do-while loops.

To test these hypotheses, I redefine `compound_stmt` as shown below

```
compound_stmt  = qubit_op ";" | if_stmt | for_stmt | do_while_stmt;

qubit_op       = gate_op | subroutine_op;

if_stmt =
    'if (' '!'? 'mz(' qubit '){' NL ci<compound_stmts> "}" si<(else_stmt)?>;
else_stmt =
    'else {' NL ci<compound_stmts> "}";

for_stmt =
    "stride = " ""[str(UNIFORM(2, 4))] ";" NL
    "for (int i = 1; i < " loop_bound "; i += stride) {" NL
    si<compound_stmt> NL
    "}";

# wrapped in {} block scope so nested loops don't have
# naming collisions for 'loop_ctr'
do_while_stmt =
    "{" NL
    ("int loop_ctr = 0;" NL)
    ("do {" NL)
    (ci<compound_stmts>)
    ("loop_ctr++;" NL)
    ("} while (" '!'? "mz(" RESET qubit ") && loop_ctr < "
        ""[str(UNIFORM(2, 3))] ");" NL)
    "}";

loop_bound =
    ""[
    if (GET_CIRCUIT_KIND == kind(sub_circuit)){
        var_lp = "loop_bound";
        var_lp
```

[1] https://github.com/NVIDIA/cuda-quantum/blob/main/cudaq/lib/Optimizer/Transforms/LoopPeeling.cpp

```
33      } else {
34          rand_lp = ""[str(UNIFORM(5, 10))];
35          rand_lp
36      }
37      ];
```

In order to ensure that the while loop terminates, I create a loop counter which counts up on each loop iteration. After either 2 or 3 iterations, the loop is guaranteed to terminate. Table 5.3 shows the new coverage results for loop-related files.

File Name	Function Cov.	Line Cov.	Region Cov.	Branch Cov.
lib/Optimizer/Transforms/LoopPeeling.cpp	100.00% (4/4)	91.89% (68/74)	93.10% (27/29)	90.00% (18/20)
lib/Optimizer/Transforms/LoopUnroll.cpp	66.67% (4/6)	67.19% (43/64)	77.27% (17/22)	66.67% (10/15)

Table 5.3: New code coverage metrics loop-related files

This experiment shows that the MAP Elites algorithm is only as strong as the grammar definition and that better results can be obtained by tweaking the generated programs depending on the coverage feedback. The templating language makes that easy to do manually.

5.3 Comparing coverage of different compiler sections

This section will look at the core compiler coverage results for specific parts of the compiler and compare them. Table 5.4 shows these results, obtained after running a suite of 50 CUDA-Q programs, **without** the improvements added in the experiment from section 5.2.1.

Compiler Section	Function Cov.	Line Cov.	Region Cov.	Branch Cov.
Frontend	65.33% (196/300)	41.83% (2500/5976)	41.89% (1651/3941)	32.94% (922/2799)
Optimizer (Excl. Codegen)	42.14% (667/1583)	34.44% (7833/22747)	32.87% (4110/12505)	26.74% (2281/8529)
Codegen	31.52% (156/495)	28.34% (2066/7289)	24.50% (598/2441)	19.07% (295/1547)
Overall Total	42.85% (1019/2378)	34.43% (12399/36012)	33.67% (6359/18887)	27.17% (3498/12875)

Table 5.4: Aggregated code coverage metrics by compiler section

There are several systemic reasons that could explain why we get seemingly low coverage.

Opt-in passes

Not all aggressive passes are enabled by default in the CUDA-Q compiler. Consider the improvement experiment we performed in section 5.2.1, where we saw that we had low coverage in the loop peeling pass, looked at the source code and saw that it operates on a do-while loop, which we did not have, then added it to the template. One also needs to add the loop peeling pass to the

pass pipeline for it to be triggered, hence why my CUDA-Q optimisation pipeline specifically adds that pass as described earlier.

CodeGen fragmentation

The codegen section having the least coverage is to be expected, because I only target quantum simulators, so only the MLIR to LLVM path is tested, yet there are other supported targets like QASM.

Pass preconditions

Some compiler passes fire when certain preconditions are met. For instance, consider that the experiment in section 5.2.1 only exercises the loop peeling with do-while loops in the program.

5.4 Bugs found

To test the bug finding capabilities of the tool, a set of nightly runs were setup to generate programs in all supported languages. Saved interesting test cases were manually reduced until a MCVE was obtained and used to create a bug report. The bugs shown in 5.5 are all the unique bugs that were found. If an already reported bug was found again through a new test case, it is not counted. Most of the bugs were crashes in different parts of the compiler, while some were semantics modifying bugs.

QSS	Bug in	Status
Pytket	optimiser[2]	fixed
Pytket	optimiser[3]	ack
CUDA-Q	parser[4]	ack
CUDA-Q	parser[5]	fixed
CUDA-Q	parser[6]	_
CUDA-Q	MLIR→ QIR translator[7]	_
Qiskit	basis translator[8]	ack, duplicate
Pytket	parser[9]	ack
Pytket	routing pass[10]	ack
Guppy	compiler[11]	ack

Table 5.5: Status of reported compiler bugs

On average, the bugs were found after running 100,000 programs for each QSS. The next sections will go through the different classes of bugs that were found, using the most interesting examples to explain them, and discuss fixes if any.

5.4.1 Parsing bugs

This section will go through bugs that occur while the program is being parsed.

MLIR type mismatch

The MCVE reported for this bug in CUDA-Q is shown in 5.4.

Listing 5.4: MCVE for a CUDA-Q bug

```
1  struct main_circuit {
2  uint64_t operator()() __qpu__ {
3      cudaq::qarray<2>reg_1190;
4
5      int64_t packed_result = 0; int bit_offset = 0;
6
7      for (bool b : mz(reg_1190)){
8         if (b) packed_result |= (1ULL << bit_offset);
9         bit_offset++;
10     }
11
12     return packed_result;
13 }
14 };
```

Running this through the compiler would cause a crash in the `cuda-quake` verifier caused by the use of the range-based for-loop on line 7.

```
'cc.store' op failed to verify that type of value matches element type
of pointer
```

The relevant MLIR snippet is

```
%18 = "cc.alloca"() <{elementType = i1}> : () -> !cc.ptr<i1>
%19 = "cc.load"(%17) : (!cc.ptr<!cc.measure_handle>) -> !cc.measure_handle
"cc.store"(%19, %18) : (!cc.measure_handle, !cc.ptr<i1>) -> ()
```

[2] https://github.com/Quantinuum/tket/issues/2109
[3] https://github.com/Quantinuum/tket2/issues/1417
[4] https://github.com/NVIDIA/cuda-quantum/issues/4562
[5] https://github.com/NVIDIA/cuda-quantum/issues/4600
[6] https://github.com/NVIDIA/cuda-quantum/issues/4689
[7] https://github.com/NVIDIA/cuda-quantum/issues/4694
[8] https://github.com/Qiskit/qiskit/issues/16251
[9] https://github.com/Quantinuum/tket/issues/2179
[10] https://github.com/Quantinuum/tket/issues/2180
[11] https://github.com/Quantinuum/tket2/issues/1632

The AST visitor lowers `mz(reg_1190)` to `!cc.stdvec<!cc.measure_handle>`, therefore, each element will be of type `!cc.measure_handle`. `cc.store` tries to map the result to `!cc.ptr<i1>` which is a pointer to a boolean. The verifier rejects this and crashes.

Running the code with `auto` instead of `bool` works, the relevant MLIR is

```
cc.store %9, %8 : !cc.ptr<!cc.measure_handle> loc(#loc6)
%10 = cc.load %8 : !cc.ptr<!cc.measure_handle> loc(#loc8)
%11 = quake.discriminate %10 : (!cc.measure_handle) -> i1 loc(#loc8)
```

This works because the type is `!cc.ptr<!cc.measure_handle>` and the discriminate is inserted when `b` is used in the `if` statement.

Stack overflow

The MCVE provided for this bug is provided in 5.5

Listing 5.5: MCVE for a CUDA-Q bug

```
1  #include <cudaq.h>
2  #include <iostream>
3  #include <cmath>
4  #include <vector>
5  #include <complex>
6
7  // using namespace std::complex_literals;
8
9  CUDAQ_REGISTER_OPERATION(
10 unitary_3,
11     1,    // Number of qubits
12     0,    // Number of params
13     (std::vector<std::complex<double>>{
14            0.311740+0.286260i,-0.905661+0.025514i,
15            0.903697+0.064850i,0.323887-0.272442i
16     })
17 );
18
19 struct main_circuit {
20 uint64_t operator()() __qpu__ {
21     cudaq::qubit sing_2256;
22     unitary_3(sing_2256);
23     return mz(sing_2256);
24 }};
25
26 int main(int argc, char* argv[]) {
27     auto kernel = main_circuit{};
28     return 0;
29 }
```

While Clang builds the AST and meets the complex literals, it needs to know what `i` means. If the namespace is added (line 7 un-commented), it knows to call the standard library function that builds a `std::complex` object using the real and imaginary parts of a complex number. In

this case, the real part is 0.0, and imaginary is 0.5. If the namespace is left out, C++ falls back to the C type `_Complex` which is a single raw value. Hence, 1 value is pushed onto the argument stack for `std::complex` as opposed to 2.

Looking at the stack trace, the function that looks at C++ constructs is `VisitCXXConstructExpr`, in which we can see the root cause. Consider the relevant snippet from that function shown in 5.6.

The first fix by the developers [12] simply checked whether there were exactly 2 arguments on the stack and threw an error if not. This fix [13] dynamically creates real part and casts to the correct type.

Listing 5.6: Source of bug in CUDA-Q source code

```
// ...

if (ctorName == "complex") {
    Value imag = popValue();
    Value real = popValue();
    return pushValue(mlir::complex::CreateOp::create(
        builder, loc, ComplexType::get(real.getType()), real, imag));
}

// ...
```

We see that an assumption is made that all complex types must have both imaginary and real parts, hence 2 values are popped off the stack. Since only 1 value was pushed in the first place, we get a stack overflow error.

MLIR lowerer crash

The MCVE posted for this bug in CUDA-Q is show in 5.7

Listing 5.7: MCVE that causes MLIR lower crash in CUDA-Q

```
#include <cudaq.h>
#include <iostream>
#include <cmath>
#include <vector>
#include <complex>

using namespace std::complex_literals;

CUDAQ_REGISTER_OPERATION(
unitary_8,
    2,    // Number of qubits
    1,    // Number of params
    (std::vector<std::complex<double>>{
           {-0.22643183121344332, -0.28170883523153889},
           {-0.40854489588865311, -0.23182703999464191},
           {0.56339148737673705, -0.32040801291409404},
```

[12] https://github.com/NVIDIA/cuda-quantum/pull/4627
[13] https://github.com/NVIDIA/cuda-quantum/pull/4664/changes

```
            { -0.20139985753987433, 0.43368523623851524},
            {-0.22948364985872663, -0.11759429313636136},
            {-0.049390029386749718, -0.5540633368296668},
            { -0.70294020354468045, 0.017020556405626705},
            { -0.22127619883545602, 0.28408712603137459},
            {-0.36077134196874977, 0.0075805935264369184},
            {0.65103075550871925, 0.21282493600854432},
            { 0.12668650115104951, 0.23502338199955278},
            { 0.042338015172833761, 0.57234002849825372},
            {-0.75314714195456511, -0.32470532890086296},
            {0.012258908382733833, -0.025218178145389653},
            { 0.058968964561439352, 0.10368852236494151},
            { 0.14012413361190376, -0.54100478124798823}
    })
);


struct main_circuit {
uint64_t operator()() __qpu__ {
    cudaq::qarray<2>reg_1063;
    unitary_8(0.8, reg_1063[0], reg_1063[1]);
    return 0;
}};

int main(int argc, char* argv[]) {
    auto kernel = main_circuit{};
    return 0;
}
```

This caused the crash in the `cuda-quake` MLIR lowerer. Setting the number of expected parameters to 0 and calling the unitary matrix without parameters compiles as expected. At the time of writing, this bug has not yet been acknowledged, and no fixes have been suggested.

5.4.2 Semantics changing bugs

This section will go through bugs that alter the result unexpectedly.

Bug in Pytket simplification pass

The MCVE I posted for this bug was further reduced by the developers to the one shown in 5.8, which is expected to have a final state of $|00\rangle$.

Listing 5.8: Reduced example showing pytket bug

```
from pytket import Circuit
from pytket.passes import GreedyPauliSimp

c = Circuit(2).H(1).Sdg(1).SWAP(0, 1).Reset(0).measure_all()
GreedyPauliSimp().apply(c)

print(c.get_commands())
```

All resets and measures on qubits act in the Z computational basis. Since the qubit state is not always in the Z-basis, the compiler needs to map the qubit state to the Z basis, perform the RESET, then map the qubit back to the basis it was in before. This can be represented mathematically like

$$P = CZC^\dagger$$

where $C^\dagger$ is the sequence of operations mapping to the Z basis, Z is the operation we want to perform, C is the sequence of operations mapping back to the original basis; the complex transpose of $C^\dagger$. Following the circuit in 5.8, the basis changes until we reach the RESET are as follows

- H: $Z \to X$
- $S^\dagger$: $X \to -Y$

Therefore, the target basis is $-Y$. To execute this safely, the compiler should have generated $C^\dagger$ to map $-Y \to Z$, executed the reset, and then applied C to map $Z \to -Y$.

However, the commands printed out after running the `GreedyPauliSimp` pass reveal a critical ordering error:

```
[Measure q[0] --> c[1];, Sdg q[1];, H q[1];, Z q[1];, Reset q[1];,
Z q[1];, H q[1];, S q[1];, H q[1];, S q[1];, Z q[1];, Measure q[1] --> c[0];]
```

Notice that the implicit SWAP reroutes the logical RESET to physical `q[1]`. The compiler erroneously implements the sequence as C; RESET; $C^\dagger$. It applies C (Sdg; H; Z) first, destroys the state with the Reset, and then applies $C^\dagger$ (Z; H; S). Because the basis changes are applied in reverse, the qubit state vector ends up along the negative Z axis. When `measure_all()` is called, the compiler flushes the tracking frame to the physical circuit with a final H; S; Z sequence to align the qubit for a Z-basis measurement. Due to the prior state corruption, this flush rotates the qubit to the positive Y axis, which corresponds to the state $\frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|01\rangle$, completely diverging from the expected outcome.

The fix for this bug [14] swapped the order back to the correct sequence.

[14] `https://github.com/Quantinuum/tket/pull/2110`

5.5 MAP Elites exploration

The coverage results in figure 5.1 showed that the current MAP Elites implementation converges to random sampling in the limit. Focusing on CUDA-Q, this section will visualize the MAP elites archive to show the different kinds of programs generated by the fuzzer. This section will also briefly consider the average cell occupancy and quality as the number of seed programs increases.

The feature space is defined by:

- Entanglement density - the ratio of all qubit ops that operate on more than 1 qubit at once; 10 bins
- Non-Clifford density - the ratio of qubit ops that are not part of the Clifford gate set; 10 bins

Two of the three dimensions outlined in 4.5 are chosen such that we get a 2-D graph we can intuitively interpret, instead of mapping the 3-D space to 2-D. The number of cells in the archive is $11 * 11 = 121$

The quality is calculated as a weighted sum of these metrics:

- Interaction graph diversity - The ratio of all multi qubit operations that operate on unique qubit entanglements; $w = 0.8$
- Reducibles density - the ratio of adjacent qubit ops that are either commutative or inverses of each other $w = 0.2$

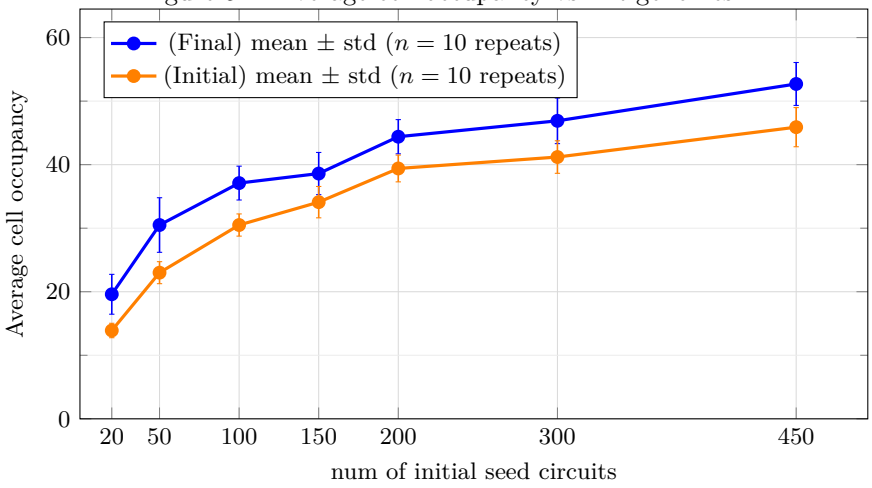
All results in this section are **without** the improvements made in the experiment from section 5.2.1.

5.5.1 MAP elites occupancy and quality

Figure 5.2 shows the average cell occupancy as the number of seed programs increases in the final archive. It compares the initial occupancy after initializing the archive with some seed programs with the final occupancy after running the mutation passes. The errors bars are at $\pm$ 1 standard deviation.

We see that the initial archive sizes are lower than the number of seeds, which implies that most of the programs have the same phenotype, and hence fall in the same cell combining entanglement

Figure 5.2: Average cell occupancy vs init genomes



density and non-clifford density. The yield is becomes worse the higher the number of seeds. This makes sense, because each of the random seeds are generated the same way i.e no changes are made between runs in the grammar templates. Therefore, the higher the number of seeds, the more pronounced the same issue of the generated programs converging to the same phenotype becomes, which is why the yield becomes much worse.

The blue graph shows the final cell occupancies after running the mutation loop. These are all higher as one would expect, but not by much. This implies that the mutation passes defined are rarely discover new program phenotypes in the feature space.

Figure 5.3: Average cell quality vs init genomes

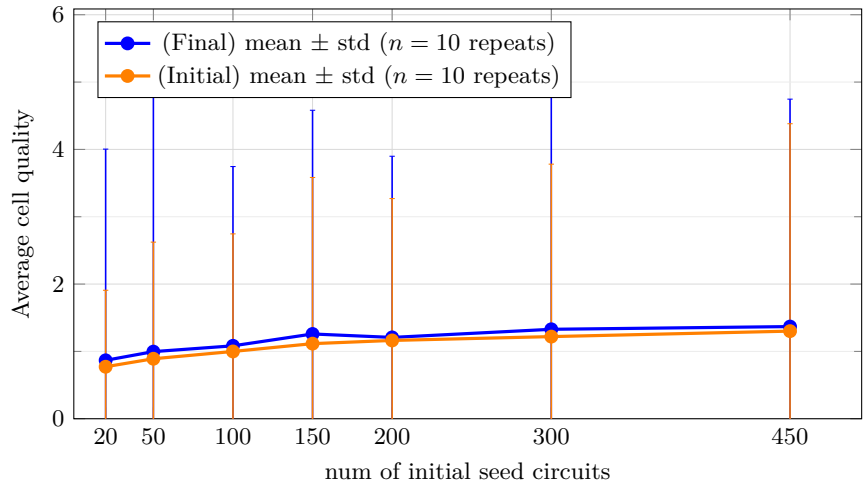


Figure 5.3 shows the average cell quality as the number of seed programs increases. The weak correlation imply that there is essentially no relationship between these two quantities. This makes sense, because the seed population is randomly chosen independent of previous runs, increasing it

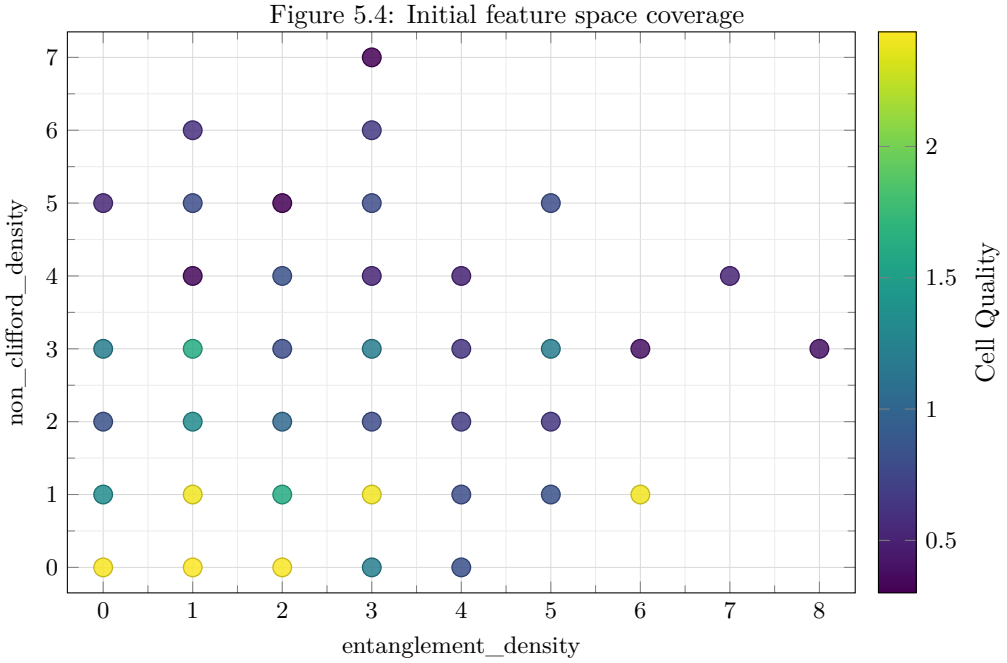
has no bearing on the quality of programs generated. However, I would expect a positive correlation between the **final** archive size and the average cell quality, because as implied by 5.2, most programs map to the same cells in the archive, therefore they can only replace the old programs if they have better quality.

We also see a high variance in the quality score for both initial and final archives. High variance in the initial archive implies that the initial mean quality is highly dependent on how lucky we get with the seed population generated. High variance in the final archive implies that the mutations do not have much of an effect in increasing the quality if it happens to initialize at a poor population.

5.5.2 Visualising MAP Elites archive

The visualisations below were obtained by running the fuzzer with MAP Elites enabled for a seed population of 400. The yield after mapping to the feature space was 39 as shown in 5.4 and the final yield after mutation passes was 47 as shown in 2.22.

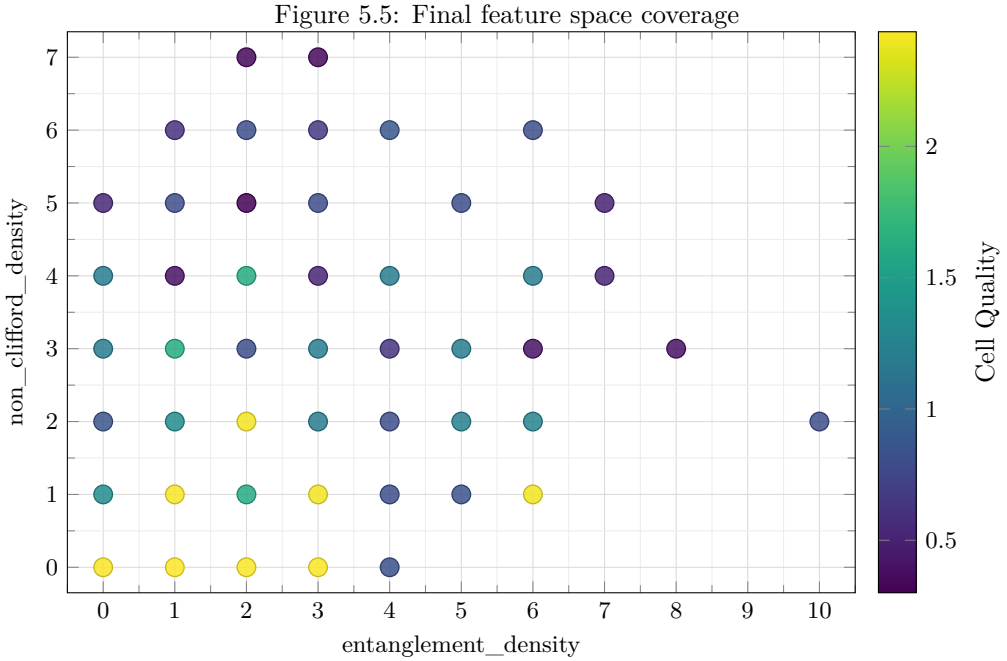
Figure 5.4: Initial feature space coverage



Comparing the initial feature space in Figure 5.4 with the final feature space in Figure 5.5 provides visual confirmation of the fuzzer's behaviour when using the MAP Elites algorithm. Several key observations can be made regarding the distribution and quality of the generated quantum programs:

- **Phenotype clustering:** Both the initial and final archives show a strong concentration of

Figure 5.5: Final feature space coverage



programs in the lower-left to mid-regions of the feature space (low to medium entanglement and non-Clifford densities). The top-right quadrant (high entanglement and high non-Clifford density) remains largely unoccupied. This implies that randomly generated template circuits naturally favour a balanced mix of single-qubit, multi-qubit, Clifford, and non-Clifford gates, making extreme topological edge cases statistically rare.

- **Marginal mutation impact:** While the mutation passes successfully increased the archive yield from 39 to 47, the expansion of the boundary is minimal. For instance, the mutation loop successfully pushed the maximum entanglement density boundary slightly outwards (discovering a new program at $x = 10, y = 2$) and filled in sparse gaps (such as $x = 4, y = 6$ and $x = 6, y = 6$). However, it failed to drive the population deep into uncharted areas. This visually reinforces the earlier conclusion that the current suite of mutation passes are not disruptive enough to radically alter a circuit's core topology. Additionally, we also see that there are programs within $7 \leq x \leq 10$ that are missed (forming islands), and we can be sure that they should be possible to generate, but are perhaps statistically rare due to the way the grammar is setup, and the mutation passes not being able to make smaller steps in the entanglement density axis.

- **Quality score distribution:** There is an inverse relationship between the coordinates and the quality score - the highest quality scores are tightly clustered at the lower bounds of the axes. As programs become highly dense with non-Clifford or entangling gates, their interac-

tion graph diversity and reducible density (the metrics driving the quality score) inherently drop, resulting in the darker regions observed at higher coordinates. An explanation for this could be that even though most of the mutation passes specifically add reducible gate pairs to the program, most of them are single qubit gate pairs, therefore, if entanglement density increases, the number of single qubit gate pairs must be lower, hence the quality goes down. This signals that it is worth creating new quality scores that are independent of the feature dimensions, such that quality is spread across the feature space.

Ultimately, visualizing the feature space demonstrates that if the target compiler is suspected to have bugs specifically related to extremely dense, highly entangled non-Clifford circuits, relying solely on standard MAP Elites mutations will be insufficient. Moreover, because we have seen that most of the circuits in the initial feature space end up in the same cells, the same can be said about the random sampling generation. Instead, the generator templates must be explicitly biased to produce those structures with higher probability.

One way to improve the discovery of unexplored parts of the space could be to introduce crossover, where 2 random genomes g_0 and g_1 are picked from the archive. Their corresponding feature vectors are $\widehat{\mathbf{v}}_0$ and $\widehat{\mathbf{v}}_1$. A vector $\widehat{\mathbf{v}}_k = 0.5(\widehat{\mathbf{v}}_0 + \widehat{\mathbf{v}}_1)$ is calculated, and used to find $\widehat{\mathbf{v}}_{C'} = argmin_{\widehat{\mathbf{v}}_i : i<n} \|\widehat{\mathbf{v}}_k - \widehat{\mathbf{v}}_i\|^2$, which is in the feature space.

5.6 Test loop efficiency

All results below are obtained **without** MAP Elites, just random sampling of the grammar. All results below are obtained **with** the improvements made in the experiment from section 5.2.1. This was done because the improvements contained nested CUDA-Q constructs, therefore the generated programs would contain nested for-loops and while-loops, which gives a better measure of generation efficiency because the program AST is bigger, and gives a higher-bound of evaluation time because nested control flow is hard to simulate efficiently.

5.6.1 Generation efficiency

This section will focus on the efficiency and throughput of the fuzzer. The results shown in 5.6 are obtained by generating CUDA-Q programs in series using an Intel Core Ultra 7 255H CPU.

Looking back at the templating language, the rule that has the biggest effect on the program size is `compound_stmts`, because as shown in 5.6.1, it is nested within definitions of control flow statements.

```
compound_stmts = (compound_stmt NL)[UNIFORM(3, 6)];
compound_stmt  = qubit_op ";" | if_stmt | for_stmt | do_while_stmt;

qubit_op       = gate_op | subroutine_op;

if_stmt =
    'if (' '!'? 'mz(' qubit ')){' NL ci<compound_stmts> "}" si<(else_stmt)?>;
else_stmt =
    'else {' NL ci<compound_stmts> "}";

for_stmt =
    "stride = " ""[str(UNIFORM(2, 4))] ";" NL
    "for (int i = 1; i < " loop_bound "; i += stride) {" NL
    si<compound_stmt> NL
    "}";

# wrapped in {} block scope so nested loops don't have
# naming collisions for 'loop_ctr'
do_while_stmt =
    "{" NL
    ("int loop_ctr = 0;" NL)
    ("do {" NL)
    (ci<compound_stmts>)
    ("loop_ctr++;" NL)
    ("} while (" '!'? "mz(" RESET qubit ") && loop_ctr < "
        ""[str(UNIFORM(2, 3))] ");" NL)
    "}";

loop_bound =
    ""[
    if (GET_CIRCUIT_KIND == kind(sub_circuit)){
        var_lp = "loop_bound";
        var_lp
    } else {
        rand_lp = ""[str(UNIFORM(5, 10))];
        rand_lp
    }
    ];
```

In order to properly measure the program efficiency, I also change the number of `compound_stmt` nodes in the program from 3 to 9. A static value is used as opposed to a random one (`[UNIFORM(3, 6)]` in the template) as this is a more principled way of measuring the performance. The results of the run are shown in figure 5.6.

We see that the fuzzer algorithm is $O(n)$ for any given $n_{compound\_stmts}$ but as expected, runs slower when asked to generate larger programs.

Figure 5.7 measures the program throughput by plotting the number of circuits on the y-axis, while also changing $n_{compound\_stmts}$.

We see an initial upshot in throughput because as the number of circuits increases, the fixed cost of grammar parsing is amortised. The plateau gives the throughput of the fuzzer, which

Figure 5.6: Total generation time (s) vs number of circuits

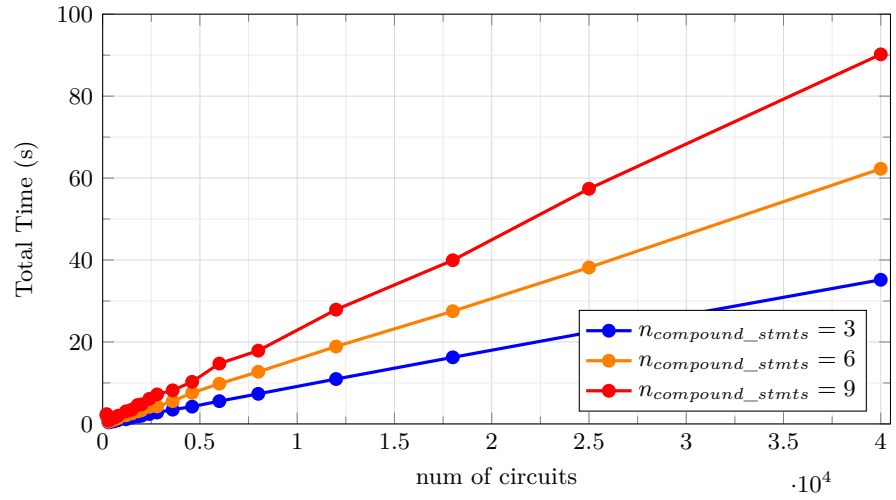
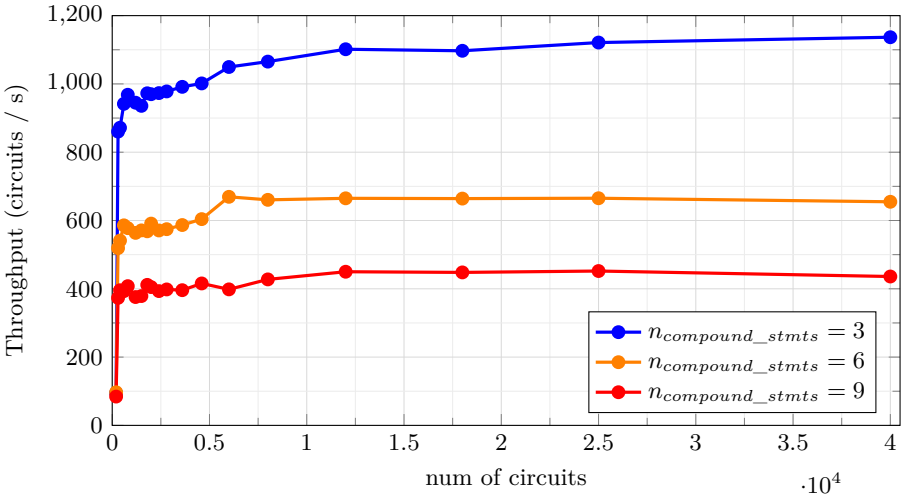


Figure 5.7: Throughput



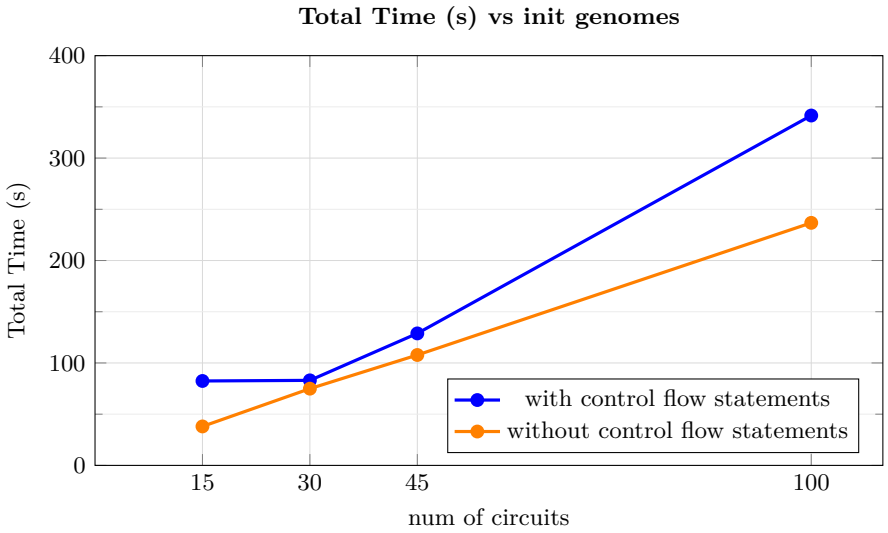
reduces the bigger the circuits get, as expected.

5.6.2 Evaluation efficiency

Evaluation is a combination of the compilation and execution phase of the fuzzing pipeline. Figure 5.8 shows the evaluation time of CUDA-Q programs with $n_{compound\_stmts} = 3$, running on 16 Intel Core Ultra 7 255H CPUs in parallel. The compiler was built **without** coverage instrumentation as that would add overhead that pollutes the results.

The evaluation times are much higher than the generation times, making circuit compilation and evaluation the main bottleneck to fuzzing campaigns. This makes sense, because as stated earlier, classical compilers cannot efficiently simulate quantum circuits, especially those with deeply

Figure 5.8: Total time taken to validate CUDA-Q circuits



nested control flow structures.

This section presented coverage results for the CUDA-Q compiler to compare programs generated MAP Elites with those generated using random grammar sampling and compare coverage obtained from different parts of the compiler. A few interesting bugs were chosen and explained and the MAP Elites algorithm was used to provide insights into the kinds of programs generated by the fuzzer. Finally, the test loop efficiency results were presented.

6 Evaluation

Contents

This section will provide an overview of the extent to which the tool achieves its main goal - providing a meta-programming approach to quantum compiler fuzzing, while also going through the tool's limitations.

6.1 Tool support and extensibility

Table 6.1 shows the lines of code required to add support for each of the target frameworks, combining the template file and the differential testing code. Only 2 files are counted: the template file where the grammar goes and the python file where the differential testing code goes.

The complexity of fuzzing new languages is vastly reduced. If a new language is developed, or massive refactors happen in an existing language, using qf++ to test the compiler is a good option, as the alternative would require massive architectural rewrites or building new tools from scratch. The differential testing strategy — using the unoptimized program as an execution oracle against increasingly aggressive optimization passes—proved effective in sidestepping the oracle problem. Furthermore, the hybrid C++ and Python architecture successfully balanced the high-performance requirements of continuous AST generation with the rapid prototyping and API integration required for executing the compiled quantum circuits.

Target Framework	Lines of code
Pytket	397
Qiskit	320
Cirq	335
PennyLane	130
CUDA-Q	406
OpenQASM2	283
OpenQASM3	246 (diff testing file (116 lines) is shared with QASM2)
Guppy	269

Table 6.1: Lines of code needed to add language support

However, the tool relies heavily on intra-compiler differential testing. While this is highly effective for catching bugs introduced during optimization passes, it assumes that the unoptimized baseline compilation is fundamentally correct, which may not always be true for deeply embedded semantic bugs. Furthermore, the current implementation of the generative grammar is highly uniform; while it produces valid code, it does not inherently prioritize the generation of deeply nested or highly specific circuit topologies that might trigger edge-case compiler crashes.

6.2 Alternative generation techniques

The results presented show that the current MAP Elites implementation does not produce diverse, high-quality programs as intended, but it gives interesting insights on the types of programs the fuzzer is biased towards given the templates it defines. Major improvements to the algorithm could include refinement of the quality metrics, such that they are truly independent of the feature axes, and implementing a crossover feature which could allow discovery into the islands of the feature space as shown in figure 5.5.

6.3 Bugs that the tool misses

Even though the tool finds several bugs, there is potentially a space of bugs that are missed. For example, bugs that would not show up when performing differential testing would be missed. Issue 16271 [1] is a bug report that manifests as a change in the global phase of the circuit after performing the `BasicSwap` pass. Brecause the tool works by comparing output distributions, which would not

[1] `https://github.com/Qiskit/qiskit/issues/16271`

change if the global phase changes, such a bug would be missed. One way to catch such bugs would be to add a new testing avenue to the tool that considers phase differences as bugs.

Another class of missed bugs are those triggered when certain functions are called with unexpected inputs. Consider issue 16268 [2] which initialises two multi-qubit quantum gates and checks whether they are commutative.

```
cc = CommutationChecker()
xx = PauliProductRotationGate(Pauli("XX"), 0.1)
summed = SparsePauliOp(["XXI", "IXI", "IIZ"])
evo = PauliEvolutionGate(summed, time=42.2)
cc.commute(evo, [0, 2, 2], [], xx, [1, 2], [])
```


The arguments `[0, 2, 2]` are qubits, and the duplicate argument represents an edge case that triggers a bug. The tool generates qubit operations such that the arguments never repeat such that the parser accepts the subroutine calls. It could be the case that this pass is triggered deep in the pipeline after the parser ensures that the user has not used duplicate qubits, and therefore assumes it will always be called with that precondition. However, calling the pass on its own uncovers this assumption and triggers the bug. Because qf++ focuses on breadth and language-agnostic design, the testing pipeline does not contain specific passes, but such changes can be made if one desires to test one particular language in more depth.

[2] https://github.com/Qiskit/qiskit/issues/16268

6

7

Conclusions and future directions

Contents

7.1 Conclusions

qf++ successfully demonstrates a novel approach to quantum fuzzing by introducing a custom templating language which allows it to achieve its primary objective: providing a seamless, highly extensible framework that can generate complex program structures for vastly different QSS environments. MAP Elites assisted generation offers interesting insights about the kinds of programs the fuzzer is biased towards, but ultimately fails to do much better than random sampling of the grammar when it comes increasing to diversity and quality of generated programs.

Coming up with a language definition that would effectively describe the language constructs required for a range of different languages was the most technically difficult part of the project. Initially, I decided to get the templates to work for Pytket and Qiskit as those have less expressive

features than some of the other languages that are supported. Once I switched to a more expressive language, Guppy, I realised there were a lot of limitations to the language - it had too many ”special rules” that had to be defined as a must in the template, it could not properly describe qubits defined within subroutines and those defined as arguments to the subroutine, it could not easily reference parts of the AST built earlier later at some other node (which was needed because certain decisions on the new node depended on what the earlier node was), it was not particularly easy to write, among others. These are brief summaries of how these problems were solved

7.1.1 Special rule reduction

This was an important decision to make, because any rule that must be defined reduces the control the user has on their programs, and makes building them a more complicated process. I knew that there was no way to make it such that the generator did not know the exact types of certain rules, because it needs to know when to trigger certain generation loops for example, it needs to know when to generate qubit definitions for the circuit, therefore, there needs to be some kind of qubit definitions node, which implies a special rule for qubit definitions. After singling out the node kinds that needed to be known by the generator, I shifted my focus to rules that the generator did not necessarily need to know about if the language was more expressive. For example, there used to be a special rule `qubit_list` which created a node in the AST whose children were the qubits needed by some gate being used as shown in 7.1.

Listing 7.1: Old qubit_list definition

```
1    qubit_list = qubit | qubit "," qubit | qubit "," qubit "," qubit;
```

A branch constraint (section 4.3.1) would be added such that the branch chosen matches the expected number of qubits. The key insight was realising that such rules can be resolved automatically while the AST is being built, which is how term expressions (section 4.2.2) were created.

7.1.2 Rule scoping

In order to generate qubit definitions as function arguments as well as as local variable definitions, the key insight was to implement a way of having different versions of the same rule. One could create 2 rule names that the lexer recognises, perhaps `internal_qubit_definitions` and `external_qubit_definitions`, then have the AST builder trigger qubit definition creation and

the relevant node on both rules, but that would create the same problem described earlier. This led to the implementation of scopes, described in section 4.2.1, which maintains the use of 1 rule name, and removes the need for repeating code in the AST generator.

Listing 7.2: Example of rule scoping

```
INTERNAL {
    float_no_param = ""[str(MAKE_FLOAT)] | GLOB::pi_multiples;

    qubit_defs = (qubit_def NL)[UNIFORM(3, 4)];
    param_defs = (param_def NL)[UNIFORM(3, 4)];

    singular_qubit_def = "cudaq::qubit " ""[GET_DEF_NAME] ";";
    register_qubit_def = "cudaq::qarray<" ""[str(GET_SIZE)] ">"
        ""[GET_DEF_NAME] ";";
}

EXTERNAL {
    qubit_defs = (qubit_def NL)[UNIFORM(3, 4)];
    param_defs = (param_def NL)[UNIFORM(3, 4)];

    singular_qubit_def = "cudaq::qubit& " ""[GET_DEF_NAME];

    singular_param_def = "double " ""[GET_DEF_NAME];

    # some defs restricted to singular to make passing of arguments easier
    # (do not have to check for types and register length to match)
    qubit_def = singular_qubit_def;
    param_def = singular_param_def;
}
```

Additionally, this can be used for any other rule definition, which makes this a more scalable solution. However, `INTERNAL` and `EXTERNAL` are keywords that attach specific scopes to the resources that are generated. An improvement would be to generalise scoping to any namespace chosen by the user, then internally assign each new namespace to its own flag such that the scope filtration still works.

7.1.3 Referencing earlier nodes via context

The key insight to allow decisions to be made later based on earlier node creation was to make the term expressions as expressive as possible, and resolve them via context. The getters defined under `<var_expr>` in listing 4.4 achieve that, for example: `GET_SIZE` looks at the context and finds the current resource definition then gets its size, `GET_GATE_QUBITS` gets the current gate's number of qubit arguments which is often used to automatically expand the `qubit_list` rule at runtime.

```
qubit_list = qubit (", " qubit)[GET_GATE_QUBITS - 1];
```

7.1.4 Ease of use

In order to make the language easy to write, one major limitation early on was how nested terms were parsed.

```
a = hello " = world + 42" ("where did " ((def1 (", "))) def2 ("time"));
def1 = "everyone go";
def2 = "bingo?";
```

The parser did not handle complex bracket nesting as shown above. The solution to this problem was to create anonymous rules on the fly once "(" is met, and push a new rule building context onto a stack. This is described in section 4.2.

7.2 Future directions

7

While qf++ lays a strong foundation for platform-agnostic quantum fuzzing, there are several key areas where the tool can be expanded to increase its bug-finding capabilities and ease of use.

7.2.1 Language improvements

There are still limitations in the language that would make it difficult to support highly expressive languages without significant code additions, because its syntax and grammar were developed iteratively as the need arose.

One such limitation is that expressions within grammar rules are evaluated shallowly: rule assignments inside expressions, such as the `AssignExpr` construct, cannot themselves contain nested expressions that refer to the same scope. This prevents certain natural patterns, such as that shown in 7.3, where a decision to index into a qubit is made depending on whether it is a register or not qubit.

Listing 7.3: Templating language limitation

```
identities =
[
    for qb in ALL_QUBITS {
    id =
    GET_CIRCUIT_NAME ".append(["                                    NL
    "cirq.I(" ""[qb.name] ("[" ""[qb.index] "]")[qb.from_reg] ")," NL
    "])"                                                            NL
    ;
    id
    }
]
;
```

The above has to be written like

Listing 7.4: Bypassing templating language limitation

```
identities =
[
    for qb in ALL_QUBITS {
        if (qb.from_reg){
            id1 =
                ""[GET_CIRCUIT_NAME] ".append([" NL
                "cirq.I(" ""[qb.name] "[" ""[qb.index] "]" ")," NL
                "])" NL
                ;
            id1
        } else {
            id2 =
                ""[GET_CIRCUIT_NAME] ".append([" NL
                "cirq.I(" ""[qb.name] ")," NL
                "])" NL
                ;
            id2
        }
    }
]
;
```


7.2.2 Cross-QSS differential testing

Currently, the differential testing backend evaluates a single compiler against itself across different optimization levels. A extension of this work would be to implement cross-platform (Cross-QSS) differential testing.

Because the generation engine already uses a unified templating representation to express identical quantum semantics across different platforms, the framework is uniquely positioned to compile and execute the exact same conceptual circuit on two entirely different toolchains—for example, comparing the state-vector output of Cirq against that of Pytket. This would not only uncover bugs within optimization passes but also expose fundamental discrepancies in how different vendors interpret quantum semantics, native gate sets, endianness, and mid-circuit measurements.

7.2.3 Weighted context-free grammar

The current generation engine builds the AST by selecting valid structural permutations. To improve the depth and quality of the generated programs, the underlying grammar could be upgraded to a weighted context-free grammar.

By assigning dynamically adjustable probabilities to the grammar rules, the generation engine could be guided to change the overall structure of the emitted programs. For instance, researchers

could tune the probabilities to favour nested conditional branches, force an unusually high density of multi-qubit entangling gates, or generate exceedingly deep circuits. This transition from uniform generation to probabilistically guided generation could be the key to stress-testing specific compiler modules, such as routing algorithms or instruction schedulers, ultimately leading to higher code coverage and the discovery of deeper logic bugs.

7.2.4 Automated test case reduction

As mentioned in chapter 1, when an interesting circuit triggers a compiler crash or a semantic deviation, the resulting test case is manually reduced to create a MCVE that is reported to the software vendors. For a fuzzer to be adopted into automated CI/CD pipelines, this reduction process must be automated. Future iterations of qf++ could implement quantum-aware Delta Debugging algorithms to automatically remove gates, qubits, and classical control flow from the generated program until the MCVE is found, saving significant developer time.

8 User guide

Contents

8.1 Environment Setup

qf++ requires Python 3.12, CMake, Clang, and a set of system packages. The setup script handles dependency installation, virtual environment creation via uv, and optional compilation of external libraries:

```
# Install system dependencies, create venv, and sync Python packages
python3 -m scripts.setup

# Optionally build external quantum compiler libraries (e.g. CUDA-Q)
python3 -m scripts.setup --libs tket cuda-quantum
```

The framework can also be run inside a pre-built Docker container, which bundles all dependencies:

```
docker pull ghcr.io/qutefuzz/qutefuzz-env:latest
./scripts/docker/run.sh
```

8.2 Running the Fuzzer

The C++ fuzzer binary is built and invoked via the `scripts/qf.py` runner, which handles compilation, test generation, and differential testing in a single command:

```
uv run -m scripts.qf --num-tests 10 --grammars pytket cudaq guppy
```

This compiles the fuzzer, feeds the requested grammars to the interactive read, eval, print Loop (REPL) with the entry point ”program”, generates the specified number of circuits, and runs each output program in parallel. `--nightly` flag observes output to detect interesting circuits and save them for later examination. The fuzzer REPL can also be invoked directly:

```
./build/qf
> pytket program   # select grammar and entry point
> 5                # generate 5 circuits
> quit
```

8.3 Reproducing Experiments

The `scripts/experiments.py` script runs the MAP-Elites scaling experiment, sweeping over a specified list of initial seed genome counts and repeating each configuration a fixed number of times to obtain stable statistics:

```
uv run -m scripts.experiments \
    --grammars pytket \
    --num-tests 1 5 10 20 50 \
    --n_repeats 10
```

Results are written to a JSON file under the grammar's output directory. On subsequent runs, passing `--map-elites-json` with the path to an existing results file skips re-execution and regenerates the LaTeX figures directly from cached data, ensuring that plots can be reproduced without re-running the full experiment.

A

Explanations

A.1 Where does the final state of the circuit come from?

We have a 2 qubit system whose initial state is assumed to be $|00\rangle = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}^T$.

We have a Hadamard gate acting on qubit 0, leaving qubit 1 unchanged. The unitary matrix for this can be calculated by taking the tensor product of the H and I matrices:

$$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & 1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ 1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & -1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{bmatrix} \tag{A.1}$$

Completing the multiplication:

$$\frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \tag{A.2}$$

Next, the first qubit is left unchanged, while the second has an X gate applied to it.

$$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 0\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ 0\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 1\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{bmatrix} \tag{A.3}$$

Completing the multiplication:

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{A.4}$$

Taking the product of matrix A.4 premultiplied by matrix A.2 gives the unitary matrix representing the circuit:

$$U = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 & 1 & 0 & -1 \\ 1 & 0 & 1 & 0 \\ 0 & -1 & 0 & -1 \\ 1 & 0 & -1 & 0 \end{bmatrix} \tag{A.5}$$

The final state of the circuit can be found by multiplying U with the initial qubit state, giving:

$$\frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + \frac{1}{\sqrt{2}} \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \frac{1}{\sqrt{2}}|01\rangle + \frac{1}{\sqrt{2}}|11\rangle \tag{A.6}$$

Bibliography

[1] W. Jiyuan, Z. Qian, H. X. Guoqing, and K. Miryung, "Qdiff: Differential testing of quantum software stacks," *IEEE/ACM Automated Software Engineering (ASE)*, 2021.

[2] B. Z. L. Ilan Iwumbwe and J. Wickerson, "Qutefuzz: Fuzzing quantum compilers using randomly generated circuits with control flow and subcircuits," *PlanQC*, 2025.

[3] N. H. (https://physics.stackexchange.com/users/219989/nadav-harel), *In simple terms: How does the quantisation of energy solve the ultraviolet catastrophe?* Physics Stack Exchange, URL:https://physics.stackexchange.com/q/770425 (version: 2023-07-02). eprint: `https://physics.stackexchange.com/q/770425`. [Online]. Available: `https://physics.stackexchange.com/q/770425`.

[4] fgoudra (https://physics.stackexchange.com/users/131812/fgoudra), *Electron spiraling into nucleus*, Physics Stack Exchange, URL:https://physics.stackexchange.com/q/413047 (version: 2022-10-14). eprint: `https://physics.stackexchange.com/q/413047`. [Online]. Available: `https://physics.stackexchange.com/q/413047`.

[5] M. A. N. and I. L. C., *Quantum Computation and Quantum Information*. Cambridge University Press, 2002.

[6] A. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. 42, no. 1, pp. 230–265, 1936. DOI: `10.2307/2268810`.

[7] D. David, "Quantum theory, the church-turing principle and the universal quantum computer," in *Proceedings of the Royal Society of London*, 1985.

[8] A. Javadi-Abhari et al., *Quantum computing with Qiskit*, 2024. DOI: `10.48550/arXiv.2405.08810`. arXiv: `2405.08810 [quant-ph]`.

[9] C. Developers, *Quantumlib/cirq: Cirq v0.9.1*, version v0.9.1, Oct. 2020. DOI: `10.5281/zenodo.4064322`. [Online]. Available: `https://doi.org/10.5281/zenodo.4064322`.

[10] S. Seyon, D. Silas, C. Alexander, S. Will, E. Alec, and D. Ross, "A retargetable compiler for nisq devices," *IOPScience*, 2020.

[11] T. C.-Q. development team, *Cuda-q*, version 0.14.0, Mar. 2026. DOI: 10.5281/zenodo.19057431. [Online]. Available: https://doi.org/10.5281/zenodo.19057431.

[12] J. Preskill, *Quantum computing in the nisq era and beyond*, 2018. arXiv: 1801.00862. [Online]. Available: https://arxiv.org/abs/1801.00862.

[13] . Sergey, W. C. Andrew, M. G. Jay, M. Dmitri, P. R., and J. Y. Theodore, *High-threshold and low-overhead fault-tolerant quantum memory*, 2023. arXiv: 2308.07915. [Online]. Available: https://arxiv.org/abs/2308.07915.

[14] P. Matteo and P. Michael, *Bugs in quantum computing platforms: An empirical study*, 2022. arXiv: 2110.14560. [Online]. Available: https://arxiv.org/pdf/2110.14560.

[15] P. Matteo and P. Michael, *Morphq: Metamorphic testing of the qiskit quantum computing platform*, 2022. arXiv: 2206.01111. [Online]. Available: https://arxiv.org/abs/2206.01111.

[16] P. Viswanath, *Quantum states and the bloch sphere*. [Online]. Available: https://medium.com/quantum-untangled/quantum-states-and-the-bloch-sphere-9f3c0c445ea3.

[17] A. Matuschak and M. Nielsen, *Quantum computing for the very curious*, San Francisco(2019). [Online]. Available: https://quantum.country/qcvc.

[18] E. C. Michael, *On the significance of the gottesman-knill theorem*, 2013. arXiv: 1310.0938. [Online]. Available: https://arxiv.org/abs/1310.0938.

[19] W. S. Peter, *Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer*, 1995. arXiv: 9508027. [Online]. Available: https://arxiv.org/abs/quant-ph/9508027.

[20] Everydaysworld, 2025. [Online]. Available: https://everydaysworld9980.medium.com/design-and-implementation-of-a-quantum-computing-system-fd49a99c1240.

[21] T. C.-Q. development team. [Online]. Available: https://nvidia.github.io/cuda-quantum/latest/using/backends/hardware.html.

[22] M. AbuGhanem, *Ibm quantum computers: Evolution, performance, and future directions*, 2024. arXiv: 2410.00916v1. [Online]. Available: https://arxiv.org/abs/2410.00916v1.

[23] K. Navin and J. G. Steffen, "Cartan decomposition of su(2n) and control of spin systems," in *Chemical Physics*, 2001. [Online]. Available: https://api.semanticscholar.org/CorpusID:95061665.

[24] L. Chris and A. Vikram, "LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation," San Jose, CA, USA, Mar. 2004, pp. 75–88.

[25] [Online]. Available: https://www.rigetti.com/.

[26] QIR Alliance, *QIR Specification*, Also see https://qir-alliance.org, 2021. [Online]. Available: https://github.com/qir-alliance/qir-spec.

[27] B. P. Miller, L. Fredriksen, and B. So, "An empirical study of the reliability of unix utilities," *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990, ISSN: 0001-0782. DOI: 10.1145/96267.96279. [Online]. Available: https://doi.org/10.1145/96267.96279.

[28] *American fuzzy lop*. [Online]. Available: https://lcamtuf.coredump.cx/afl/.

[29] F. Andrea, M. Dominik, E. Heiko, and H. Marc, "AFL++: Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, USENIX Association, Aug. 2020.

[30] N. Perlroth, *Security experts expect 'shellshock' software bug in bash to be significant*, 2014. [Online]. Available: https://www.nytimes.com/2014/09/26/technology/security-experts-expect-shellshock-software-bug-to-be-significant.html.

[31] M. Jean-Baptiste and C. Jeff, *Illuminating search spaces my mapping elites*, 2015. arXiv: 1504.04909. [Online]. Available: https://arxiv.org/abs/1504.04909.

[32] S. developers, *Ks_2samp*. [Online]. Available: https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mstats.ks_2samp.html.

[33] Y. Michelle, P. Deric, and L. Ayaz, *The multi-armed bandit problem, An exploration of epsilon greedy and ucb1*. [Online]. Available: https://cse442-17f.github.io/LinUCB/.

[34] N. I. of Standards and Technology, *Kolmogorov-smirnov goodness-of-fit test*. [Online]. Available: https://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm.

[35] T. C. Team, *Sourcebasedcodecoverage*. [Online]. Available: https://clang.llvm.org/docs/SourceBasedCodeCoverage.html#interpreting-reports.

[36] T. L. developers, *Tablegen overview*. [Online]. Available: https://llvm.org/docs/TableGen/.