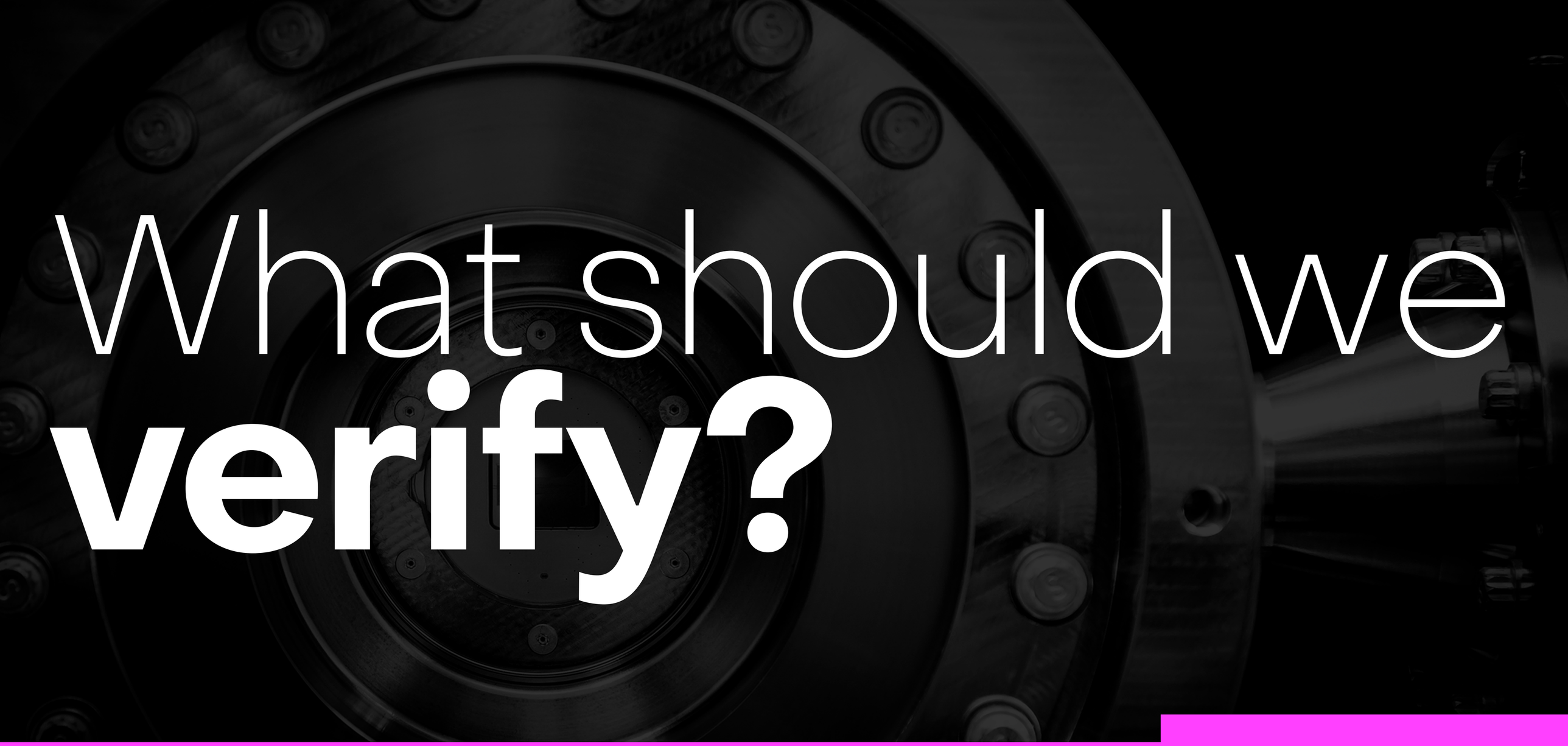
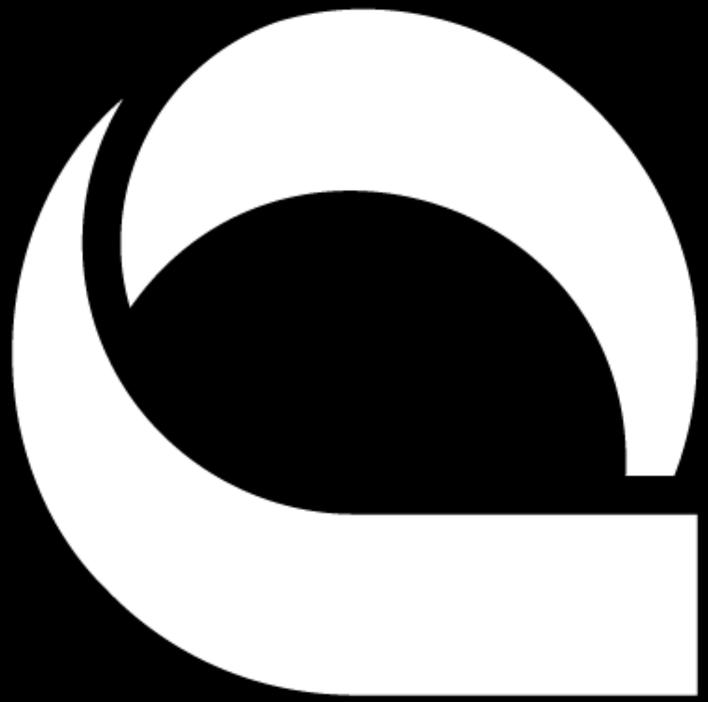


What should we **verify**?

Ross Duncan / 21 July 2025 / VQC

QUANTINUUM



QUANTINUUM

>500

people worldwide


Oxford
London
Cambridge
Munich
Minnesota
Colorado
Tokyo

Applications

Condensed matter simulations
Quantum chemistry
Topological data analysis
Monte Carlo integration
Optimisation
... more!

Tooling

Compilers
Programming langauges
Simulators / Emulators
Error mitigation
Web services
.... more!

Hardware

QCCD ion traps

Quantinuum Ltd

Quantum Software and Technologies

184 followers · Cambridge, UK · http://www.quantinuum.com

Overview | Repositories 131 | Projects | Packages | People 2

Pinned

lambeq Public
A high-level Python library for Quantum Natural Language Processing
Python ☆ 439 ⑂ 106

Qermit Public
Python module for running error-mitigation protocols with the pytket quantum SDK
Python ☆ 31 ⑂ 9

tket Public
Source code for the TKET quantum compiler, Python bindings and utilities
C++ ☆ 242 ⑂ 48

tket2 Public
Version 2 of the TKET quantum compiler
Rust ☆ 20 ⑂ 3

pytket-quantinuum Public
pytket-quantinuum, extensions for pytket quantum SDK
Python ☆ 21 ⑂ 13

hugr Public
Hierarchical Unified Graph Representation for quantum and classical programs
Rust ☆ 15 ⑂ 4

H2

Trapped Ion Quantum Processor

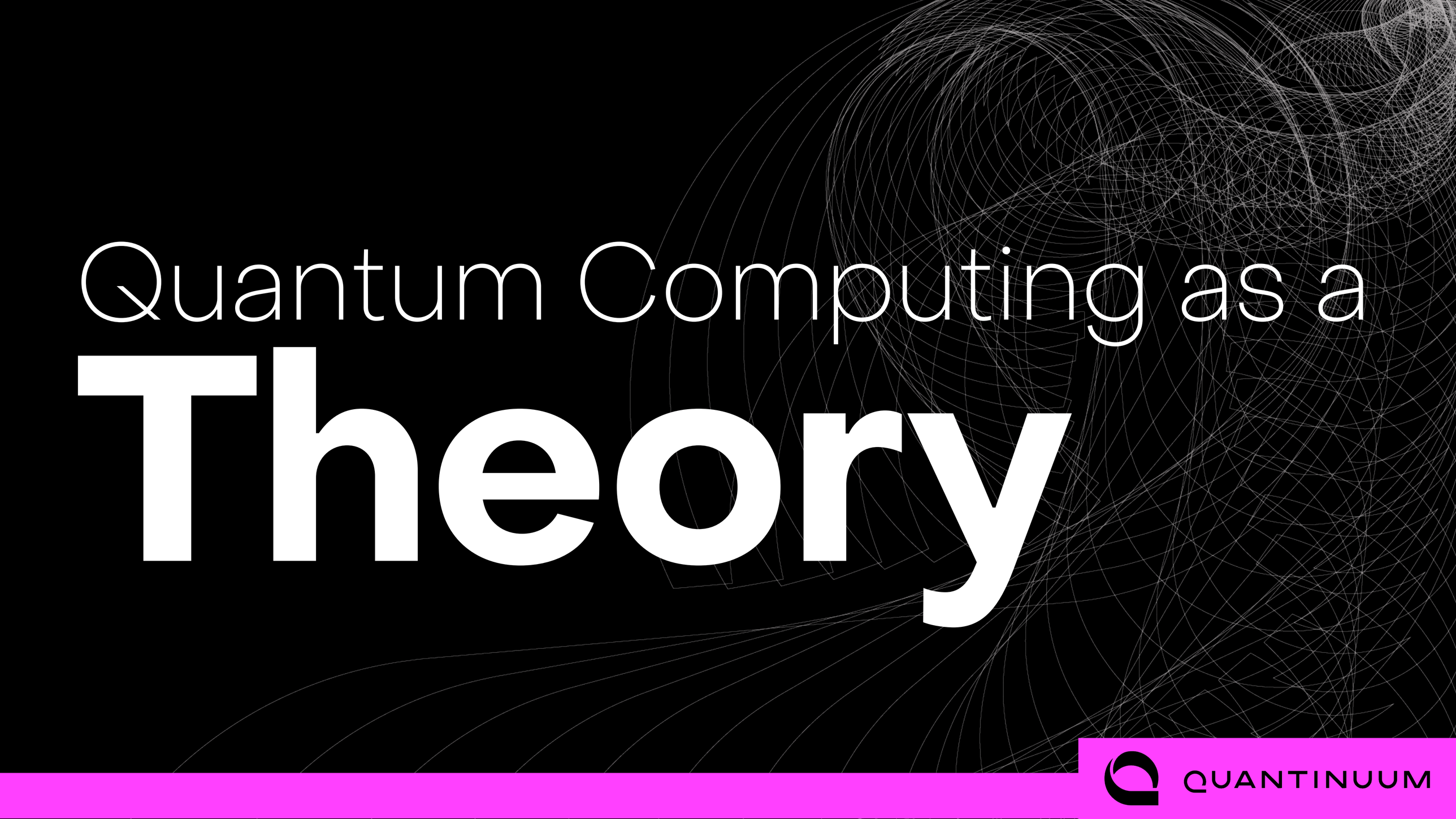
56 Qubits

All-to-all connectivity

Leading fidelity

Mid-circuit measurement

A Race Track Trapped-Ion Quantum Processor *S. A. Moses et al*
arXiv:2305.03828


Quantum Computing as a
Theory
QUANTINUUM

Quantum states are complex unit vectors

Modulo phase

- $|\psi\rangle \in \mathbb{C}^n$
- $|\langle\psi|\psi\rangle|^2 = 1$
- $|\psi\rangle \sim e^{i\alpha}|\psi\rangle$

Qubits

Atomic 2D systems

- Z-basis

$$|0\rangle, |1\rangle \in \mathbb{C}^2$$

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

Qubits

Atomic 2D systems

- Z-basis

$$|0\rangle, |1\rangle \in \mathbb{C}^2$$

$$\alpha|0\rangle + \beta|1\rangle \in \mathbb{C}^2$$

Qubits

Atomic 2D systems

- X-basis

$$|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\1\end{pmatrix}$$

$$|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = \frac{1}{\sqrt{2}}\begin{pmatrix}1\\-1\end{pmatrix}$$

Combined with the tensor product

Dimensions multiply

- $|\phi\rangle \in A, \quad |\psi\rangle \in B$
- $|\phi\rangle \otimes |\psi\rangle \in A \otimes B$
- $\dim(A \otimes B) = \dim A \times \dim B$

Combined with the tensor product

Dimensions multiply

- $|\phi\rangle \in A, \quad |\psi\rangle \in B$
- $|\phi\rangle \otimes |\psi\rangle \in A \otimes B$
- $\dim(A \otimes B) = \dim A \times \dim B$

n qubits live in 2^n dimensional space

Combined with the tensor product

Dimensions multiply

- $|0\rangle, |1\rangle \in \mathcal{Q}$
- $|00\rangle, |01\rangle, |10\rangle, |11\rangle \in \mathcal{Q}^2$
- $|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle, |111\rangle \in \mathcal{Q}^3$

Entanglement

Quantum states are more than the product of their parts

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} \neq |\psi\rangle \otimes |\phi\rangle$$

Entanglement

No need to stick with product bases

$$\frac{|00\rangle + |11\rangle}{\sqrt{2}} \qquad \frac{|01\rangle + |10\rangle}{\sqrt{2}}$$

$$\frac{|00\rangle - |11\rangle}{\sqrt{2}} \qquad \frac{|01\rangle - |10\rangle}{\sqrt{2}}$$

Unitary Dynamics

A quantum system evolves according to unitary dynamics

$$|\psi(t)\rangle = U_t |\psi(0)\rangle$$

Unitary Dynamics

A quantum system evolves according to unitary dynamics

- Unitary maps are reversible

$$U^{-1} = U^{\dagger}$$

- Unitary maps preserve the inner product

$$\langle \psi | U^{\dagger} U | \phi \rangle = \langle \psi | \phi \rangle$$

Unitary Dynamics

A quantum system evolves according to unitary dynamics

- Unitary maps are reversible

$$U^{-1} = U^{\dagger}$$

Most classical functions are *not* reversible

- Unitary maps preserve the inner product

$$\langle \psi | U^{\dagger} U | \phi \rangle = \langle \psi | \phi \rangle$$

Measurement

This is the strangest thing in science

- A quantum measurement is defined by a self-adjoint operator

$$M = \sum_i \lambda_i \, | e_i \rangle \langle e_i |$$

- The possible *outcomes* are the eigenvalues λ_i of the operator
- The *probability* of outcome λ_i is given by $| \langle e_i | \psi \rangle |^2$
- The state after the measurement is the eigenvector $| e_i \rangle$

Measurement

Makes quantum computing strange

- Quantum algorithms are intrinsically stochastic

Measurement

Makes quantum computing strange

- Quantum algorithms are intrinsically stochastic
- Can’t “read variables”

Measurement

Makes quantum computing strange

- Quantum algorithms are intrinsically stochastic
- Can’t “read variables”
- Thanks to entanglement, measuring a subsystem can have side-effects on the unmeasured parts.

No-cloning theorem

And the no-deleting theorem is similar

- There is no unitary operation such that

$$U : |0\rangle |\psi\rangle \mapsto |\psi\rangle |\psi\rangle$$
$$U : |0\rangle |\phi\rangle \mapsto |\phi\rangle |\psi\rangle$$

unless $|\psi\rangle$ and $|\phi\rangle$ are orthogonal

Simulation

Simulating quantum systems with classical computers

- Naive algorithms are just linear algebra — main issue is memory
  - 20 qubits — laptop
  - 30 qubits — cluster
  - 40+ qubits — Top 10 HPC
- 44 qubits is the current record.
- Simulating *ideal* systems is quadratically easier than noisy ones.

Simulation

Simulating quantum systems with classical computers

- **Tensor network** methods :
  - *Approximate* very large numbers of qubits
  - Require circuit depth is small or entanglement does not grow quickly
  - Good in 1D systems, 2D systems are harder, 3D???
- **Stabilizer simulations** : highly efficient at large qubit number
  - Cover only the Clifford subgroup of the unitary group
  - Still very useful for some applications
- **Hybrid techniques**

Quantum Computing as a **Computational Model**

QUANTINUUM

Quantum Gates

Basic unitaries

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

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad R_z(\alpha) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{pmatrix}$$

Quantum Gates

Basic unitaries

$$CX = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \qquad CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}$$

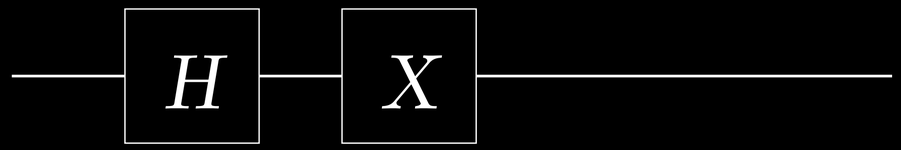
Quantum Circuits

Form a symmetric monoidal category

H

Quantum Circuits

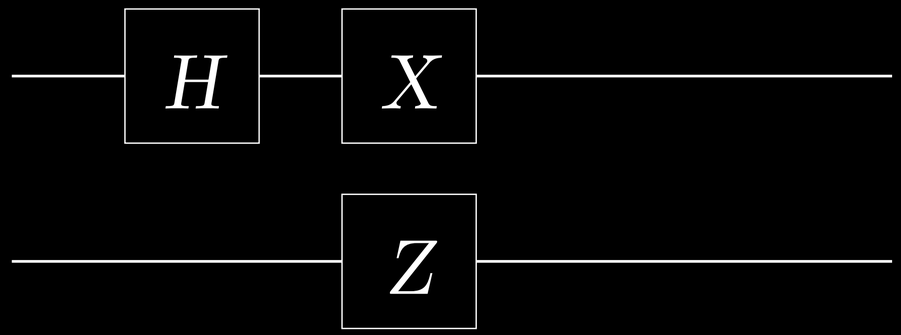
Form a symmetric monoidal category



$H; X$

Quantum Circuits

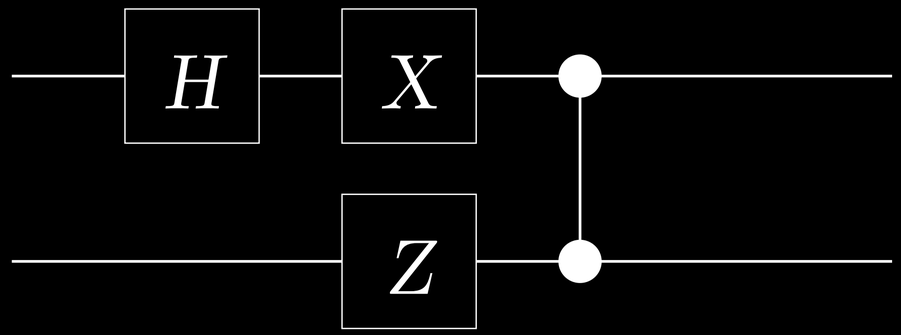
Form a symmetric monoidal category



$$(H; X) \otimes Z$$

Quantum Circuits

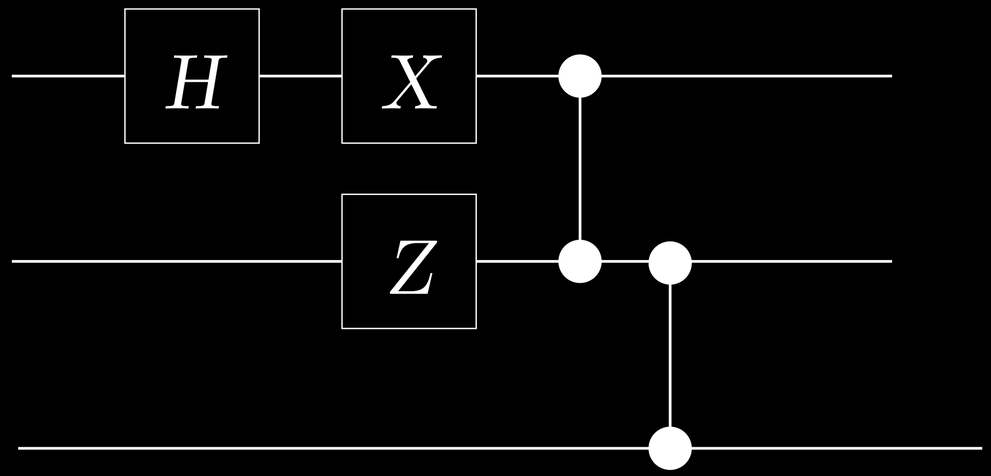
Form a symmetric monoidal category



$$((H; X) \otimes Z); CZ$$

Quantum Circuits

Form a symmetric monoidal category



$$((((H;X)\otimes Z);CZ)\otimes I);(I\otimes CZ)$$

Universality

- A gate set is called universal if its circuits can generate all the unitaries

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

$$R_z(\alpha) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{pmatrix}$$

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

Equations

An infinitude of theories

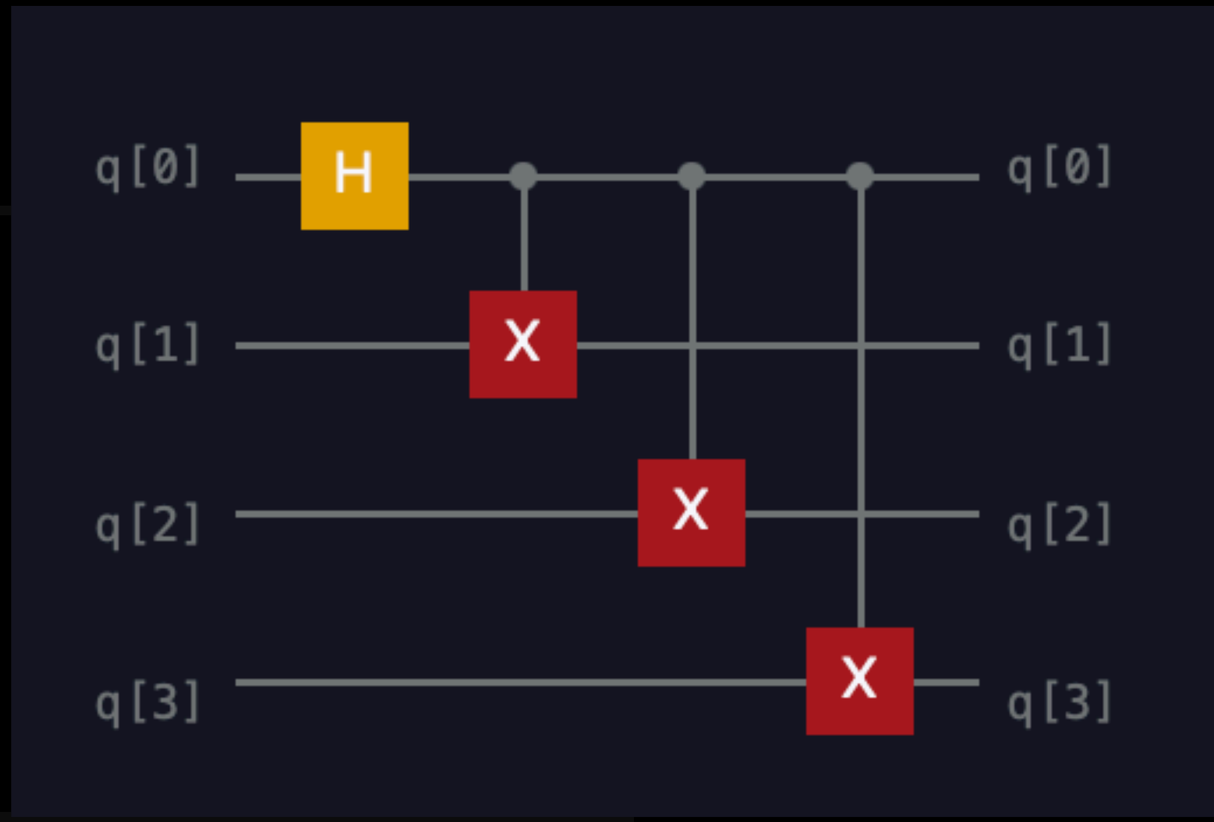
- Every unitary has infinitely many circuit representations in any universal gate set.
- “Optimising circuits” means to finding “better” representatives of the circuit of interest
  - Fewer gates
  - Lower depth
  - Fewer qubits

Programming circuits

It’s usually in python

```
[16]: from pytket import Circuit

      ghz_circ = Circuit(4)
      ghz_circ.H(0)
      ghz_circ.CX(0, 1)
      ghz_circ.CX(0, 2)
      ghz_circ.CX(0, 3)
```




```
[16]: [H q[0]; CX q[0], q[1]; CX q[0], q[2]; CX q[0], q[3]; ]
```

General Paradigm

This is the old way

- Use a classical host program to generate the circuit you want.
- Run the circuit and measure the qubits at the end
- Repeat until you have enough samples.

Gup•py [ˈgʌpi] *noun.*

A high-level quantum PL
that looks like Python *and*
is embedded into Python.

Realtime Classical Logic

Arbitrary Control-Flow

Dynamic Qubit Allocation

Type Safety

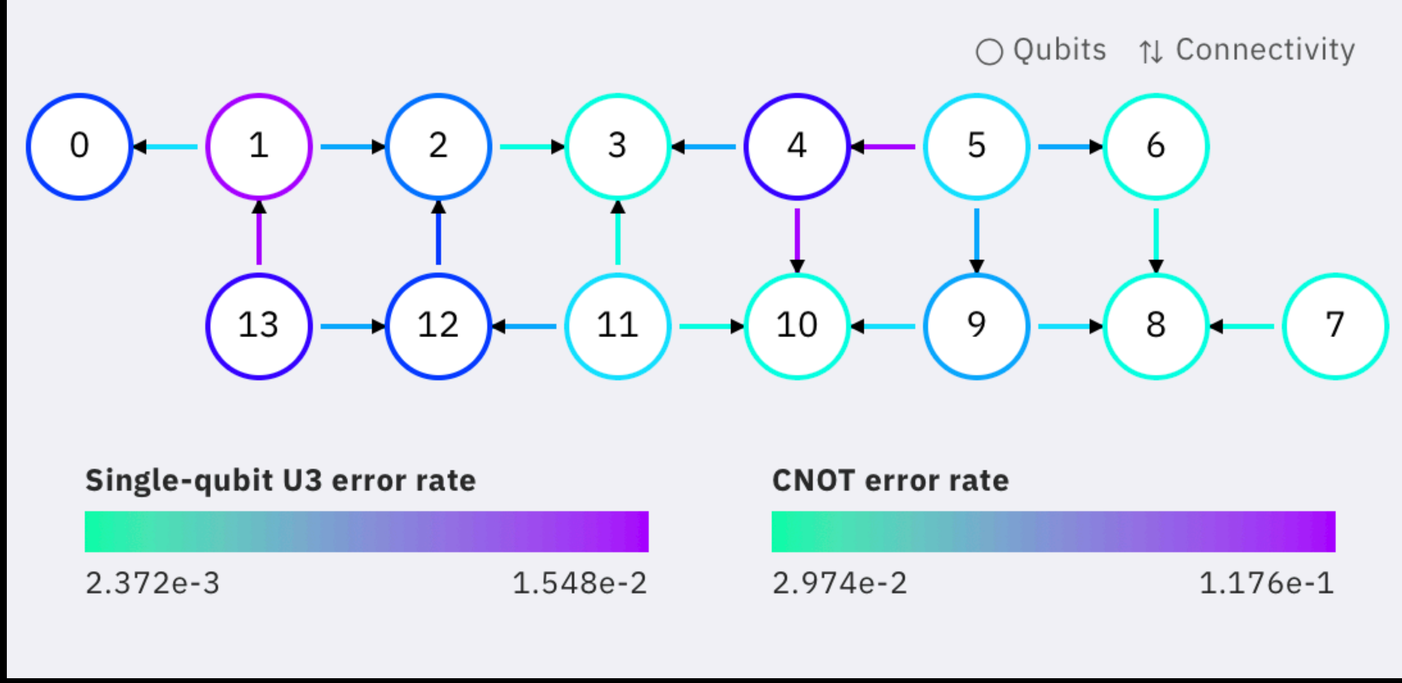
Guppy is open source and available now

https://github.com/CQCL/guppylang

```
> pip install guppylang
```

Noise

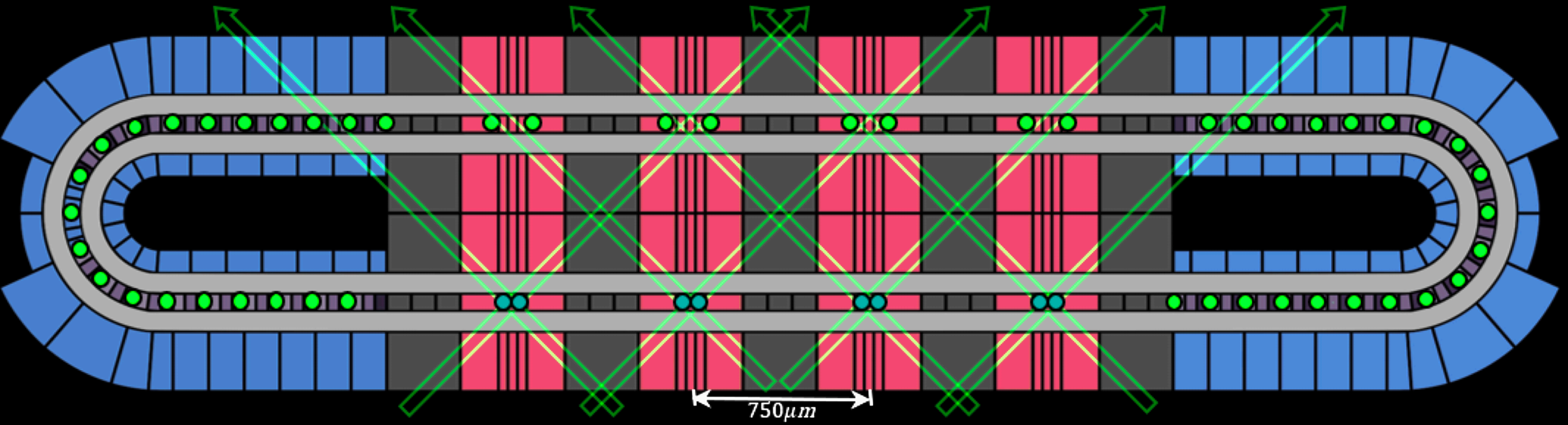
The bane of existence



- In real hardware you get *noise*
  - State initialisation
  - Gates
  - Measurements
  - Just doing nothing…
- Everything has a non-zero probability to deviate from the desired operation.

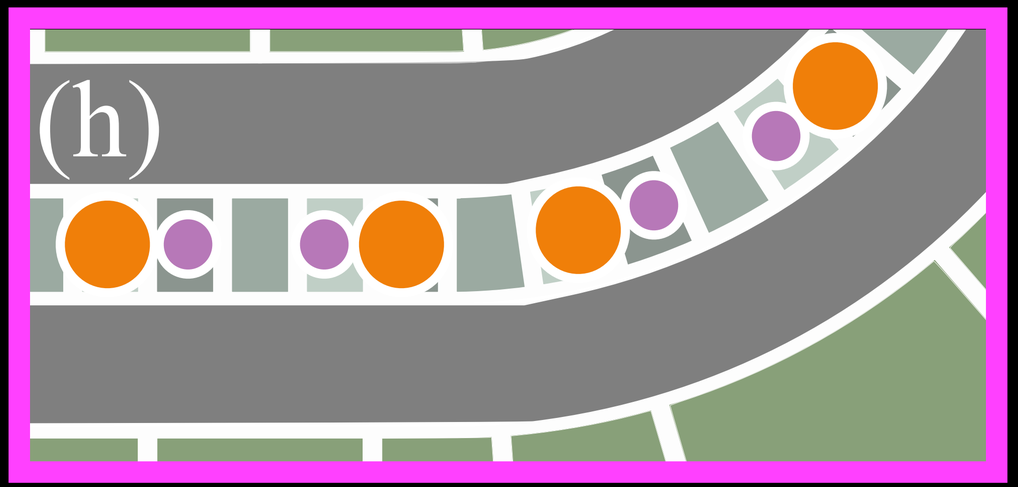
Quantum Computing as a **System**

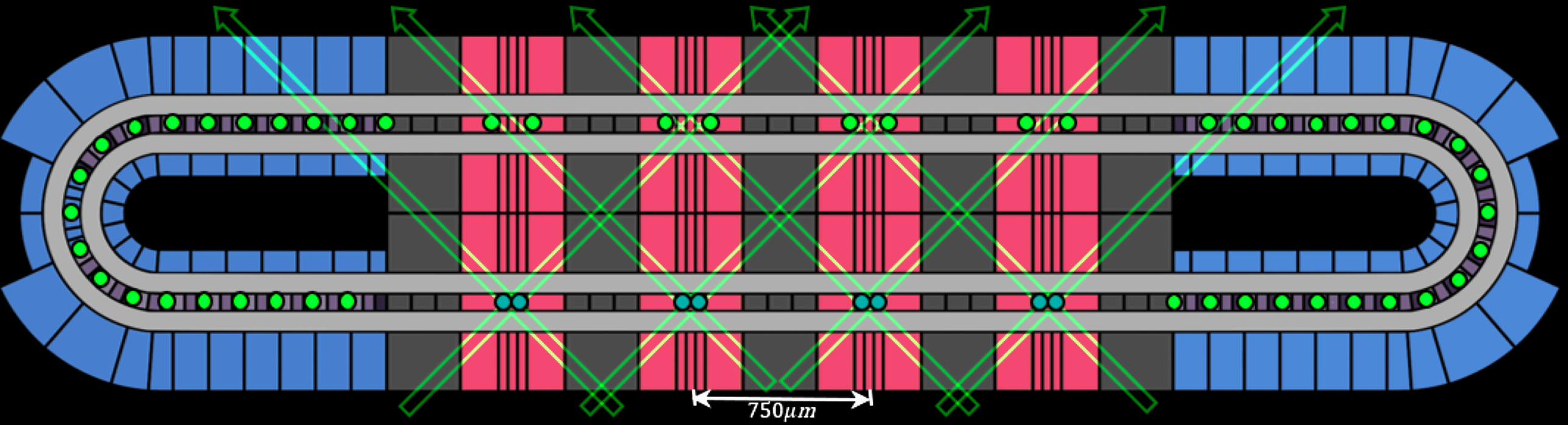
QUANTINUUM

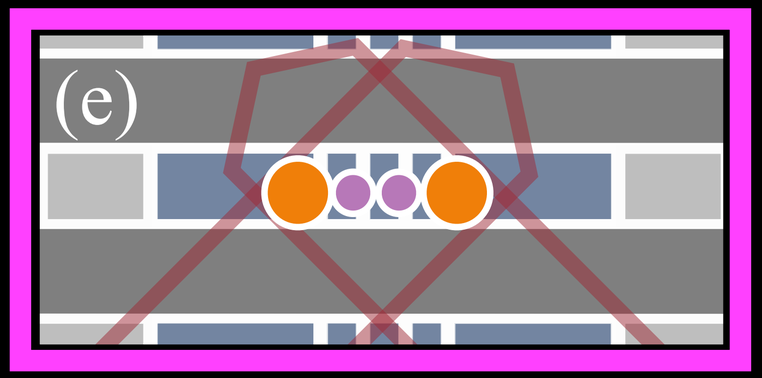


$^{171}Yb^{+}$ $^{138}Ba^{+}$

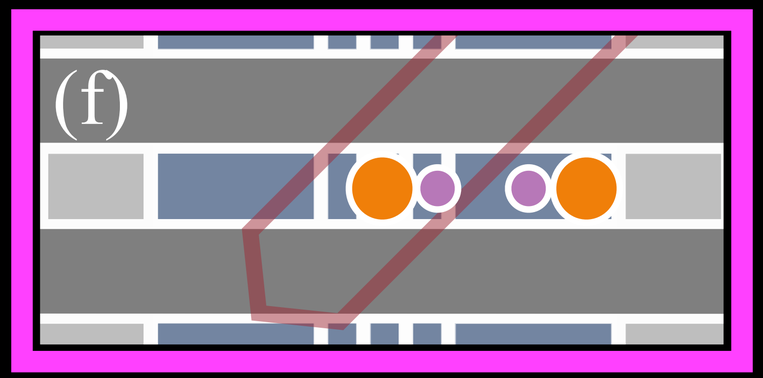
“Data” “Cooling”



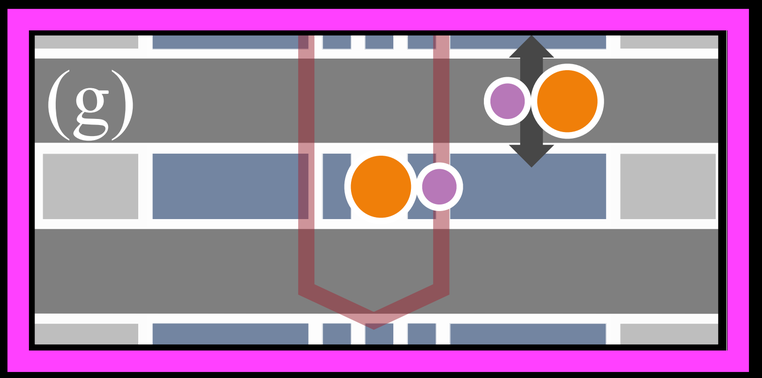




2 qubit gate



1 qubit gate

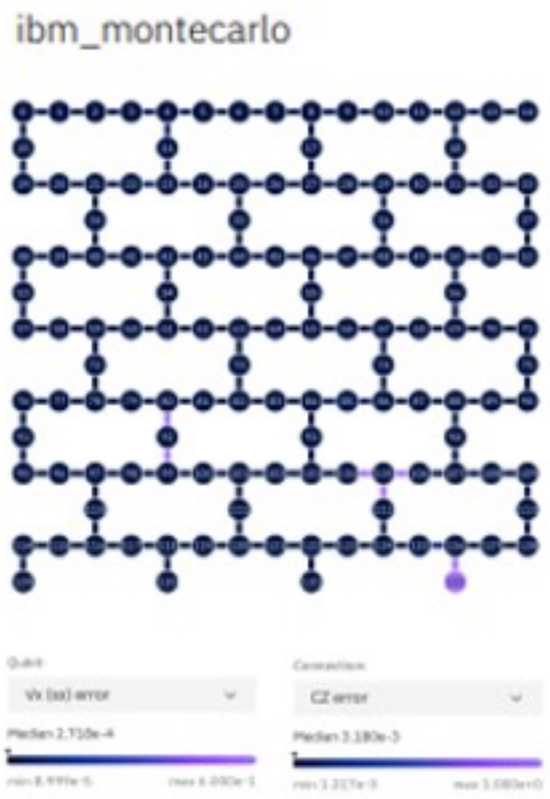


Measure

Heron

133 qubit systems

Tunable coupler architecture

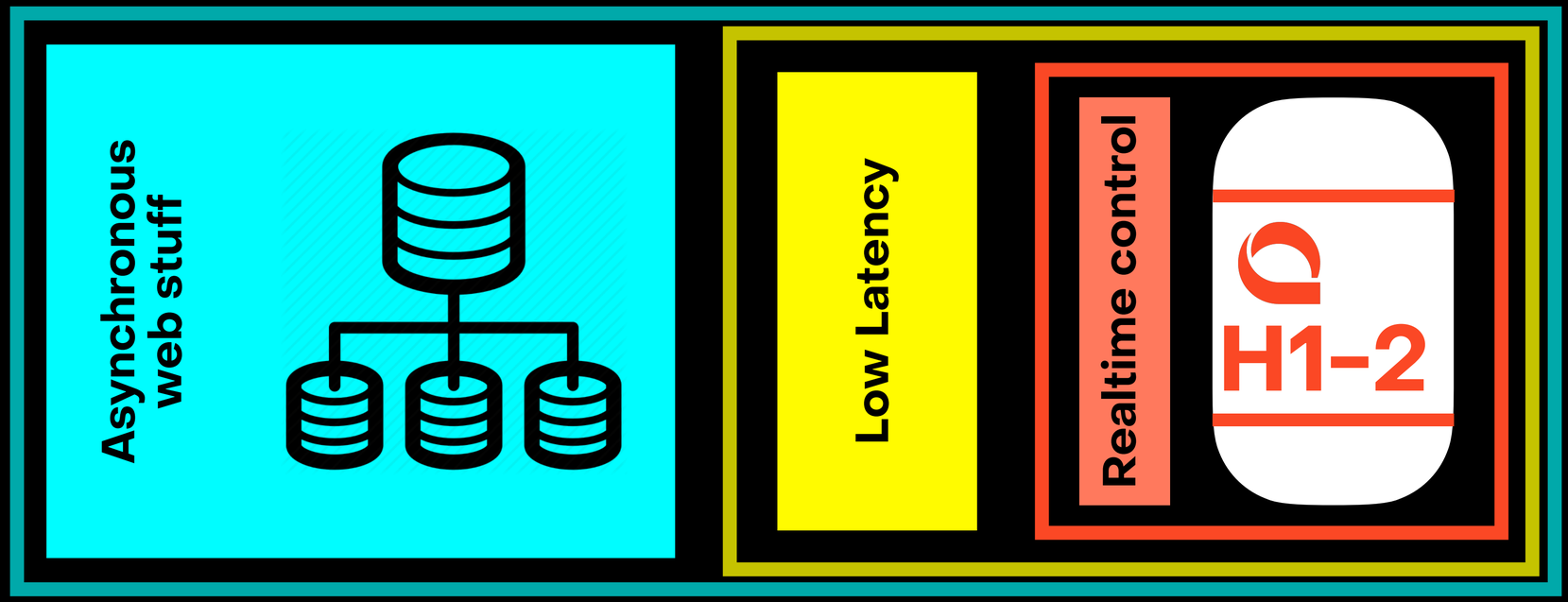


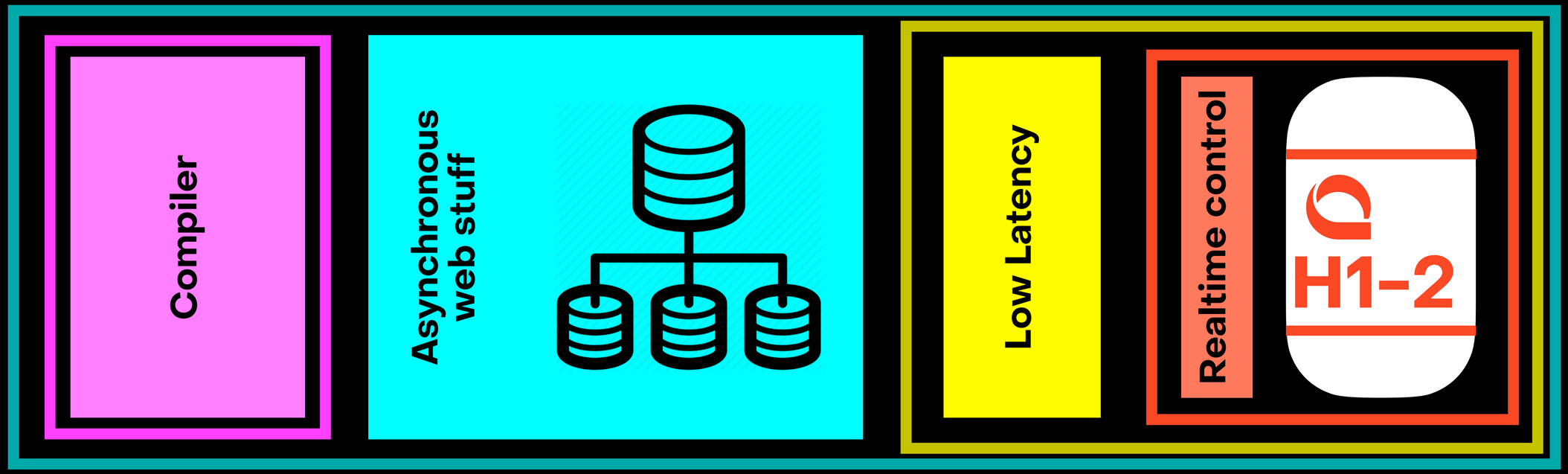
	ibm_sherbrooke Eagle	*ibm_montecarlo (Heron)*
Gate Error (best system)	0.6-0.7%	0.3%
Crosstalk	High (qubit-qubit collisions)	Almost zero!
Gate time	500-600ns	90-100ns

Hard Real Time

Near Time

Hard Real Time

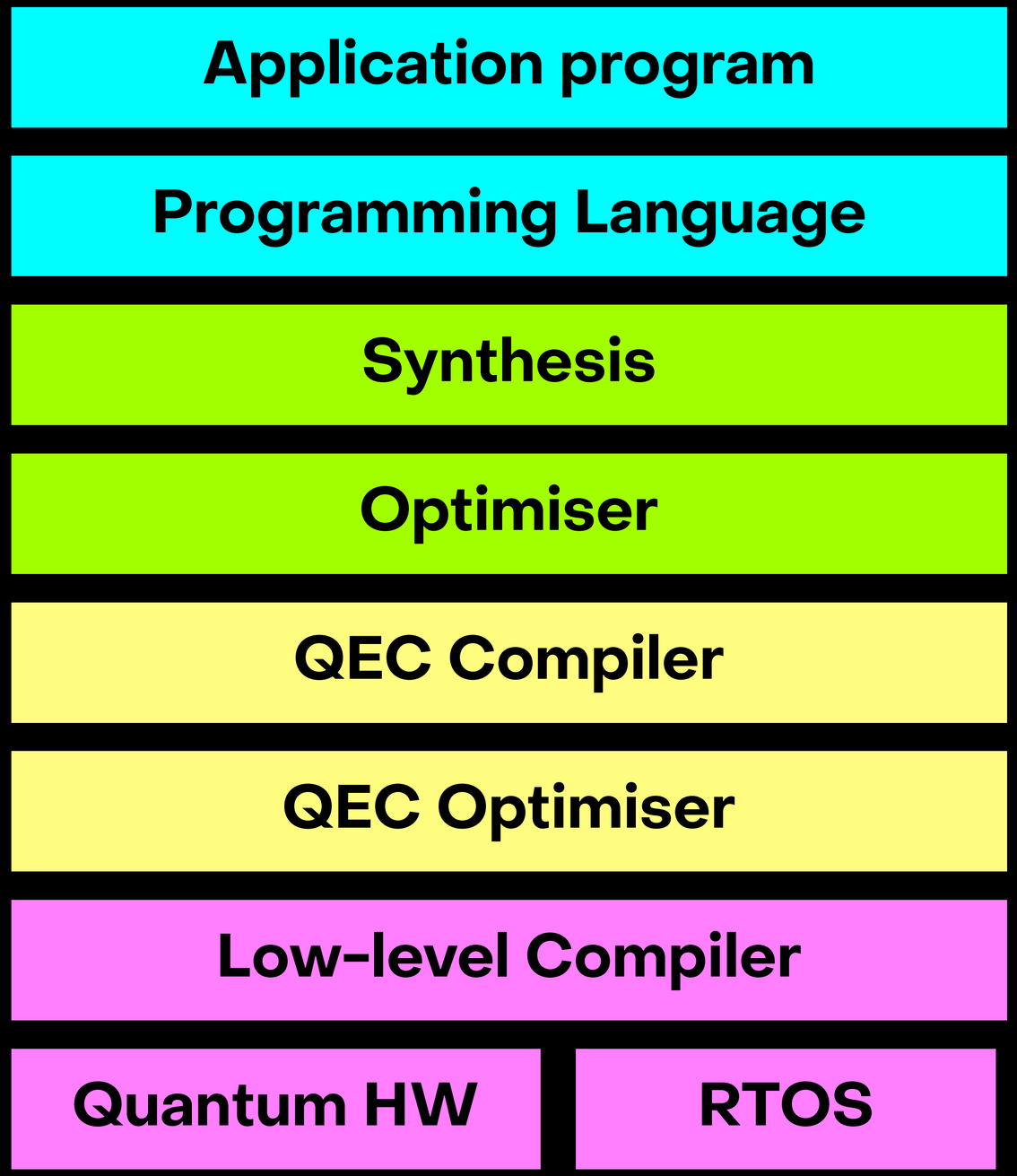

Asynchronous
web stuff
Low Latency
Realtime control
H1-2
Slow runtime
Near Time
Hard Real Time



Each box is :

- a different time regime
- a different set of capabilities
- a different scope

The program as a whole must be correctly distributed between them


Application program
Programming Language
Synthesis
Optimiser
QEC Compiler
QEC Optimiser
Low-level Compiler
Quantum HW
RTOS

QUANTINUUM

What does “Verify” mean?

At least three things

- Verify the performance of the system
  - minimally to check that it is a functioning quantum computer in the relevant regime
- Verify the computer is executing the program the user asked for
  - Often in adversarial scenarios, cf. blind quantum computing

What does “Verify” mean?

At least three things

- Verify that the program is free from errors
  - Either bugs introduced by the application programmer
  - Or invalid program transformations in the stack

Physicists don’t write bugs

At least that is what they tell me

- Verify that the program is free from errors
  - Either bugs introduced by the **application programmer**
  - Or invalid program transformations in the stack

“I just want the compiler not to mess up my code”

What's a good definition of "correct" ?

And what's a good way to formalise it?

- Strong correctness (whole system):
  - The output distribution is not distinguishable from the intended one
- Weak correctness (compiler):
  - The program that runs is equivalent to the one that was written.

What's a good definition of "correct" ?

And what's a good way to formalise it?

- Strong correctness (whole system):
  - The output distribution is not distinguishable from the intended one
- Weak correctness (compiler):
  - The program that runs is equivalent to the one that was written.

The impossibility of testing

- Since quantum computers are intractable to simulate, you need a quantum computer to test on
  - *Extant quantum computers are few, expensive, and difficult to access*
- No-cloning forbids inspecting the state in a running program
  - *Can only test subroutines in isolation, can't debug whole programs*
- A single measurement doesn't reveal the quantum state
  - *Worst case, exponentially many shots are needed for tomography*

0. Useful static analysis on a practical PL

Or IR

- A lot of work is being done on toy languages and experiements.
- Can we prove useful things about some popular toolkit?

1. Provably correct IR rewrites

For HUGR specifically

- Circuits are not terms — the tree structure is your enemy
- Modern IRs (like HUGR) are graph structured
- Want a provably correct graph rewriting *kernel* for working with circuits in their natural form.
- Still want freedom to choose wild rewrite rules

2. Synthesis / Rewriting for fault-tolerance

Code are going to get big and complex

- Given a code, synthesise fault-tolerant state-preparations
  - This is already possible using SMT-solvers, but scaling is not great
- Same as above, but for gates
- Given a circuit prove that it is fault-tolerant for a given distance
- Given a fault-tolerant circuit optimise it while maintaining fault-tolerance
- Find a fault-tolerant equivalent of a given non-FT circuit.

3. Whole Program Error Budget

Even approximately

- Many synthesis techniques involve *approximating* the desired operator
  - More accuracy means more gates…
  - More gates mean more hardware error…
  - More hardware error means less accuracy…

- BQSKit circuit synthesis can now track the approximation error (somewhat)
  - Can we balance this against the hardware error channels?
  - Can we do this with control flow?

4. Proof Lowering

Can program transforms be also proof transforms?

- Compilation usually involves lowering from high-level program description to more fully specified version.
  - If I have a proof of some correctness property upstairs can I take it downstairs?

Which verification challenges do you think matter most?

Thanks!

We're always hiring

QUANTINUUM