QUANTINUUM

Programming Quantum Computers with Guppy

Callum Macpherson

https://guppylang.org

What is Guppy?

A language for the next generation of quantum programs

1. Embedded in Python – intuitive Python syntax
2. Fully fledged programming language – functions, loops, conditional logic, recursion
3. Focus on safety – statically compiled, catch errors early

Docs and tutorials – https://guppylang.org

Open source! - https://github.com/CQCL/guppylang

pip install guppylang

```python
from guppylang import guppy
from guppylang.std.quantum import qubit, toffoli, s, measure
from guppylang.std.quantum.functional import h

@guppy
def repeat_until_success(q: qubit, attempts: int) -> bool:
    """
    Repeat-until-success circuit for Rz(acos(3/5))
    from Nielsen and Chuang, Fig. 4.17.
    """
    for i in range(attempts):
        a, b = h(qubit()), h(qubit())
        toffoli(a, b, q)
        s(q)
        toffoli(a, b, q)
        if not (measure(h(a)) | measure(h(b))):
            result("rus_attempts", i)
            return True
    return False
repeat_until_success.check() # type check
```

QUANTINUUM

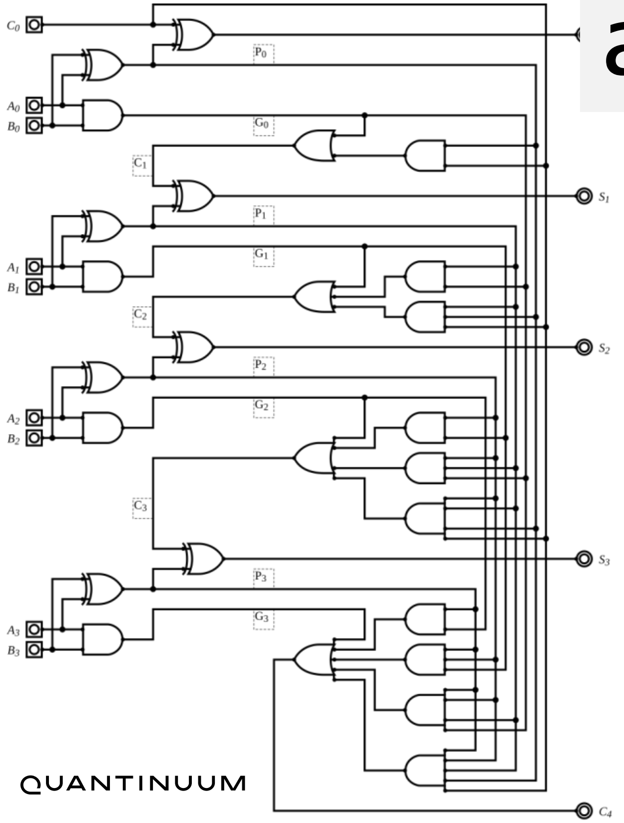
Recap: Classical and Quantum logic circuits

Classical computers → logic gates

Goal: implement all Boolean functions using a restricted set of logic gates, e.g., { AND, NOT } or { NOR }

Example: four-bit (carry) adder

Image: Wikipedia CC0 license



a + b

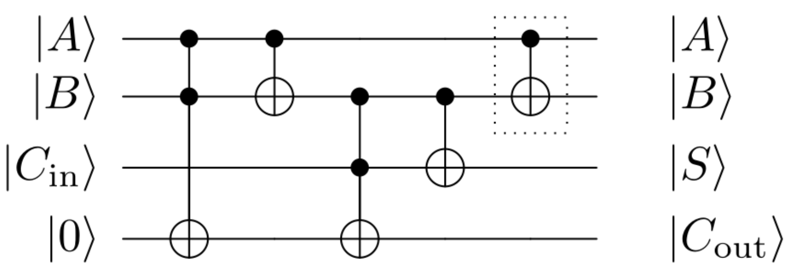
Quantum computers → quantum logic gates

Goal: implement all unitary operators using a restricted set of few-qubit gates, e.g., { CNOT, H, S, T } or { Toffoli + H }

Example: two-qubit (carry) adder: $|S\rangle = |A \oplus B \oplus C_{\text{in}}\rangle$

Feynman, Foundations of Physics, 16, 6, (1986)

Image: Wikipedia CC BY-SA 4.0 license

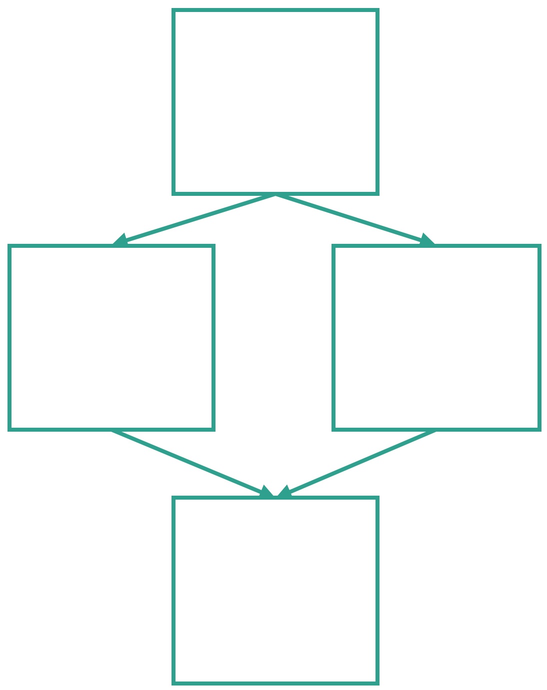


(Toffoli + CNOT gates)

QUANTINUUM

Beyond the Circuit

- Standard Quantum Frameworks vs. Guppy
  - Standard: Build a static list of gates
  - Programming: Conditional logic, control flow, functions, abstraction
- New Device Capabilities
  - Dynamic qubit allocation
  - Mid-circuit measurement
  - Real-time branching (if/else)
  - Loops (while) based on measurement results



Quantinuum roadmap

We are here

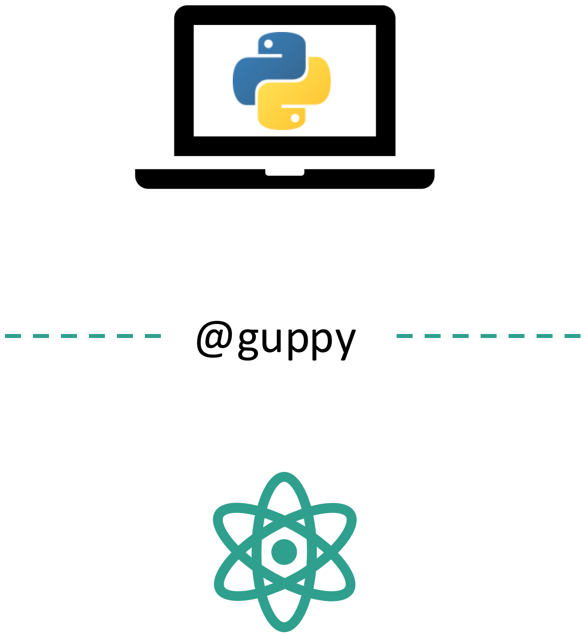
	2020	2023	2025	2027	2029	2030+
SYSTEMS	QUANTINUUM H1	QUANTINUUM H2	QUANTINUUM HELIOS	QUANTINUUM SOL	QUANTINUUM APOLLO	QUANTINUUM LUMOS
PHYSICAL QUBITS	20	56	98	192	1,000s	
PHYSICAL 2-QUBIT GATE ERROR	1×10^{-3}	1×10^{-3}	$< 5 \times 10^{-4}$	$< 2 \times 10^{-4}$	1×10^{-4}	
LOGICAL QUBITS		> 12	50	~ 100	100s	1,000,000+ QUBITS
LOGICAL ERROR RATES		1×10^{-3}	$< 10^{-4}$	$\sim 10^{-5}$	1×10^{-5} to 1×10^{-10}*	
NVIDIA GPU INTEGRATION			QEC Decoding ~1 PFLOP	QEC Decoding ~10 PFLOP	Control + QEC Decoding ~100 PFLOP	

*analysis based on recent literature in new, novel error correcting codes predict that error could be as low as 1E-10 in Apollo (ref: arXiv:2403.16054, arXiv:2308.07915).

QUANTINUUM

Quantum “Kernel"

- The @guppy decorator
  - Marks the boundary between 'Host' and 'Device' code
  - Code inside runs at quantum runtime
- The Standard Library
  - guppylang.std.quantum (Qubits, Gates)
  - guppylang.std.builtins (Arrays, Results)

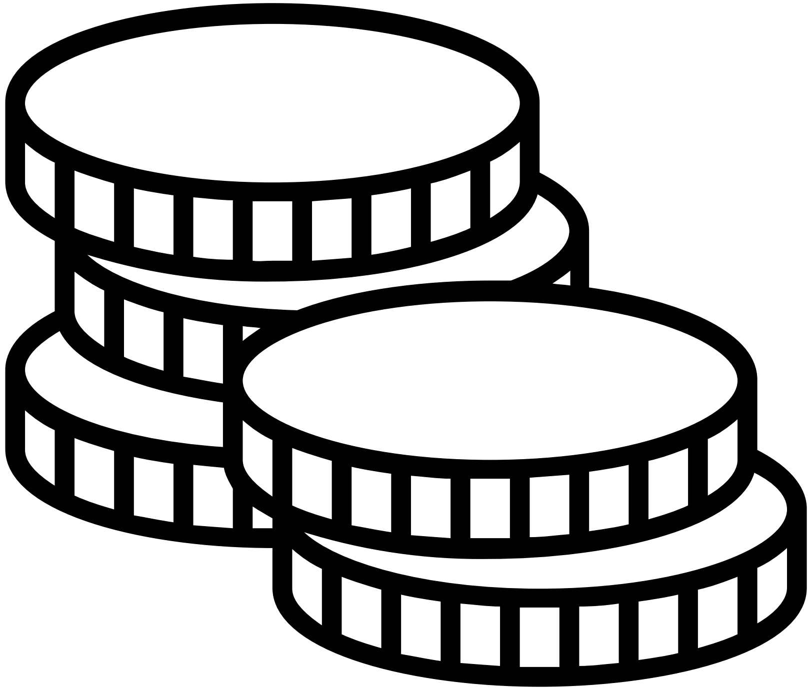


Types: Safety & Predictability

- Type annotations on function inputs and outputs
- Type inference for all values
- Prevent expensive bugs
  - Error before submission
  - Fewer surprises
- Improve performance – optimize using type knowledge

Linear Types

- Qubits are resources
  - Can’t copy (No-Cloning Theorem)
  - Can’t 'drop' (Must be measured/discarded)
- Linear types catch errors early
  - Compiler error if you use a qubit twice
  - Compiler error if you forget to measure



comptime

- Use standard Python logic during compilation
- Use Cases:
  - Calculating rotation angles
  - Unrolling loops with static bounds
  - Generating complex circuit parameters
  - Circuit structure pre-computation (e.g. graph states)
- **Inline:** *x* = *comptime(…)* for simple constants.
- **Functions:** *@guppy.comptime* for complex logic (ansatz patterns, metaprogramming).

Execution with Selene

Framework for executing hybrid quantum programs

- New on-demand execution model
- Framework for high fidelity emulation of the Helios device
- New on-demand execution model
- Multiple simulation modes – Statevector and Stabilizer
- Support for error models

Core functionality is open source - https://github.com/CQCL/selene

QUANTINUUM