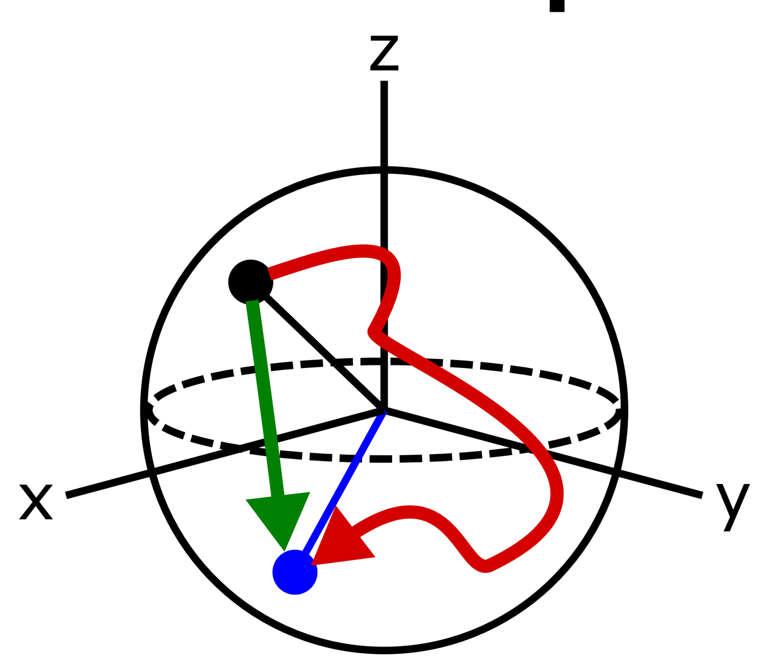
Leveraging MLIR to Augment a Python Quantum DSL

Ryan Lynch, Austin Adams, Tom Conte, Jeff Young

Georgia Tech

Background

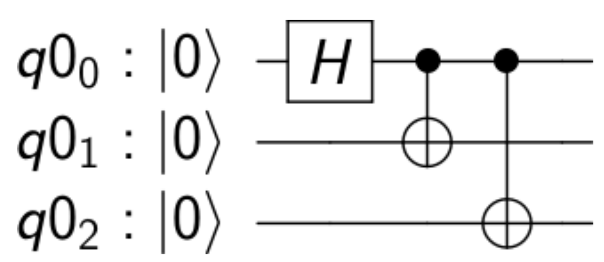
Quantum Compilation



- Qubit state can be visualized as a point on a unit Bloch sphere
  - Gates: rotations on this sphere
- Goal of quantum compilation: **"Evolve" quickly and cheaply to a useful state**
  - "Cheaply" often means to minimize number of quantum instructions

Optimization Motivation

Simple compiler optimizations can be done, but current representations are inflexible and inefficient for stronger dataflow analysis



Potential dataflow uses:
- Smarter qubit allocation
- Reorder operations to minimize decoherence
- Target-based rewriting to lower "cost"

```
qreg q[3];
h q[0];
cx q[0], q[1];
cx q[0], q[2];
```

Cannot easily find "inputs" and "outputs" of gate operations or sources/sinks

Existing Infrastructure

Qiskit

- Popular Python library for quantum programming
- Allows support for quick iteration with Python libs
- Builds list of quantum operations when Python program runs

Source Code with Qiskit

```
q = qiskit.QuantumCircuit(N)
for i in range(N):
  q.h(i)
  q.z(i)
  q.h(i)
```

Generated when Python runs

Popular Quantum IR (e.g. QASM)

```
h q[0];
z q[0];
h q[0];
h q[1];
z q[1];
h q[1];
...
h q[N];
z q[N];
h q[N];
```

MLIR

- Generalized IR plus infrastructure and tools (e.g. passes, conversions)
- IR is SSA-like and dataflow-oriented, always equivalent to an AST in memory
- Mix and match "dialects" within one file

Example MLIR file

```
func.func @example_func(%arg0 : i32) -> i32 {
  %cond = arith.constant true
  %1 = arith.constant 1 : i32
  cf.cond_br %cond, ^bb1, ^bb2(%arg0 : i32)

^bb1:
  cf.br ^bb2(%1 : i32)

^bb2(%arg : i32):
  return %arg : i32
}
```

Optimized MLIR file

```
func.func @simple_control_flow
  (%arg0: i32) -> i32 {
  %c1_i32 = arith.constant 1 : i32
  return %c1_i32 : i32
}
```

Run mlir-opt or write C++ code to run standard optimization passes

- MLIR can be agnostic when you combine different dialects
- Allows reuse of common passes and dialects

Our Goals

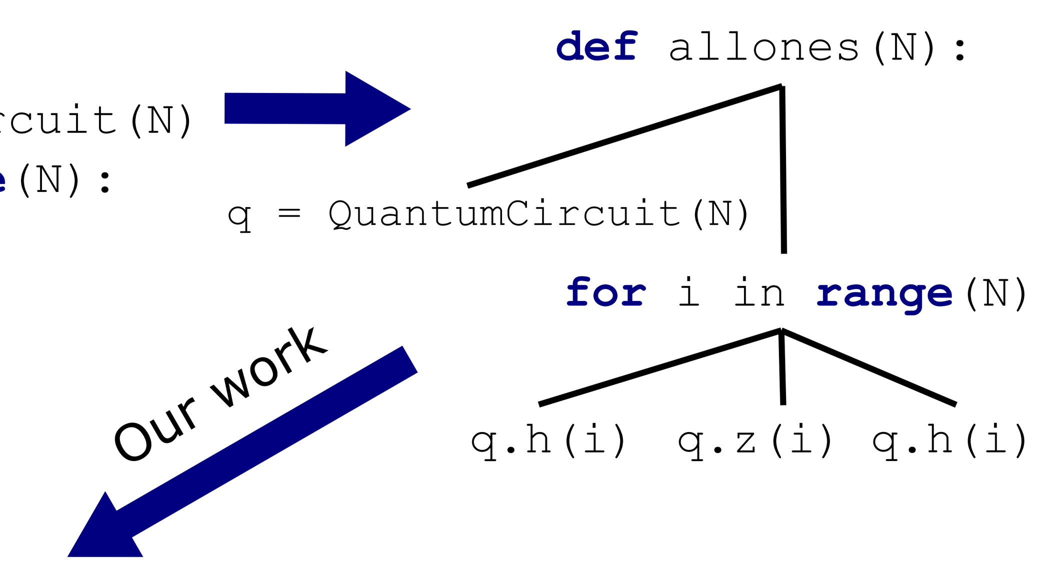
Augment Qiskit with MLIR

- Drop-in Python C extension gets Qiskit AST
- Lower AST to MLIR with our quantum dialect
- Perform dataflow analyses and clean up IR
- Perform generic and quantum optimizations
- (Ideally) Output back to Qiskit or equivalent

Python Code Using Qiskit

```
@kernel
def allones(N):
  q = QuantumCircuit(N)
  for i in range(N):
    q.h(i)
    q.z(i)
    q.h(i)
```

Python Syntax Tree



MLIR Representation

```
affine.for %i = 0 to N {
  %2 = affine.load %qreg[%i]
  %3 = quantum.h(%2)
  %4 = quantum.z(%3)
  %5 = quantum.h(%4)
  affine.store %5, %qreg[%i]
}
```

Example Optimization (HZH→X)

```
%3 = quantum.x(%2)
```

Easy Extension

- MLIR provides declarative methods to specify peephole optimizations and "rewrites"

```
def HXHOptPattern : Pat<
      (Quantum_Gate1QOp Gate<"h">, $_,
            (Quantum_Gate1QOp Gate<"x">, $_,
                  (Quantum_Gate1QOp Gate<"h">, $_, $q))),
      (Quantum_Gate1QOp Gate<"z">, (NoParams), $q)
>;
```

- MLIR can also specify "deep" transforms
- Dialects are fully customizable and extensible

Prior work (QRANE [1], QIRO [2], and QSSA [3]) can provide reusable optimizations and insight

[1] B. Gerard, T. Grosser, and M. Kong, "QRANE: lifting QASM programs to an affine IR," in Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction, New York, NY, USA, Mar. 2022, pp. 15–28. doi: 10.1145/3497776.3517775.
[2] D. Ittah, T. Häner, V. Kliuchnikov, and T. Hoefler, "QIRO: A Static Single Assignment-based Quantum Program Representation for Optimization." *ACM Transactions on Quantum Computing*, vol. 3, no. 3, Sep. 2022, pp. 1 - 32. doi: 10.1145/3491247
[3] A. Peduri, S. Bhat, and T. Grosser, "QSSA: an SSA-based IR for Quantum computing," In Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction, New York, NY, USA, 2022, pp. 2–14. doi: 10.1145/3497776.3517772

Contact: ryan.lynch@gatech.edu, aja@gatech.edu