Leveraging MLIR to Compile a Basis-Oriented Quantum Programming Language

2025 US LLVM Developers' Meeting

Austin Adams

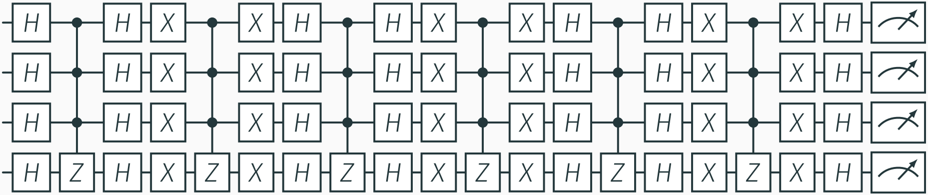
October 28th, 2025

Georgia Tech

Background: Quantum Computing

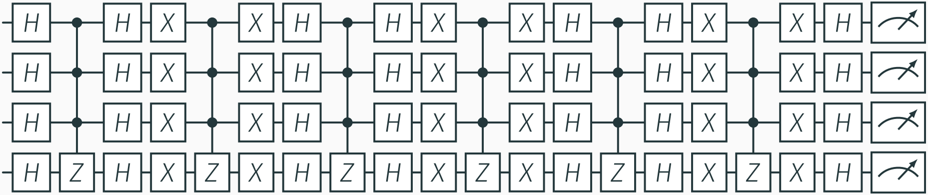
- Quantum computers promise exponential speedup for important problems (e.g., integer factoring and physics simulation)
- …but current quantum programming languages (e.g., Q# or Qiskit) require programming in low-level quantum assembly (quantum *gates* and *circuits*)

Background: Example Quantum Circuit



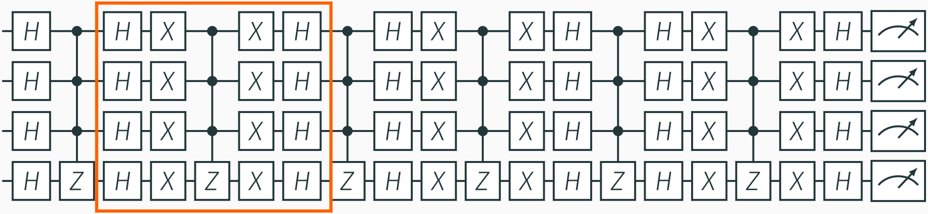
Background: Example Quantum Circuit

Search algorithm:



Background: Example Quantum Circuit

Search algorithm:



QCL (2000)

```
operator diffuse(qureg q) {
  H(q);
  Not(q);
  CPhase(pi,q);
  !Not(q);
  !H(q);
}
```

Quantum Programming Today

QCL (2000)

```
operator diffuse(qureg q) {
  H(q);
  Not(q);
  CPhase(pi,q);
  !Not(q);
  !H(q);
}
```

Q# (2025)

```
operation Diffuse(q : Qubit[])
                  : Unit {
  within {
    ApplyToEachA(H, q);
    ApplyToEachA(X, q);
  } apply {
    Controlled Z(Most(q),
                 Tail(q));
  }
}
```

```
'p'**4 >> -'p'**4
```

Basis translation

```
'p'**4 >> -'p'**4
```

```
@classical
def oracle(x: bit[4]) -> bit:
    return x.and_reduce()

@qpu
def grover_iter(q):
    return (q | oracle.sign
              | 'p'**4 >> -'p'**4)
@qpu
def grover():
    return ('p'**4 | grover_iter
                   | grover_iter
                   | grover_iter
                   | measure**4)
```

Basis translation

```python
@classical
def oracle(x: bit[4]) -> bit:
    return x.and_reduce()

@qpu
def grover_iter(q):
    return (q | oracle.sign
              | 'p'**4 >> -'p'**4)
@qpu
def grover():
    return ('p'**4 | grover_iter
                   | grover_iter
                   | grover_iter
                   | measure**4)
```

Classical oracle

Basis translation

Qwerty: High-Level Quantum DSL Embedded in Python

```
@classical
def oracle(x: bit[4]) -> bit:
    return x.and_reduce()
```

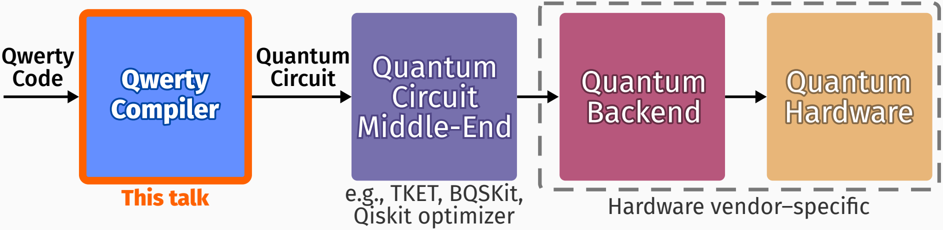
Classical oracle

```
@qpu
def grover_iter(q):
    return (q | oracle.sign
              | 'p'**4 >> -'p'**4)
@qpu
def grover():
    return ('p'**4 | grover_iter
                   | grover_iter
                   | grover_iter
                   | measure**4)
```

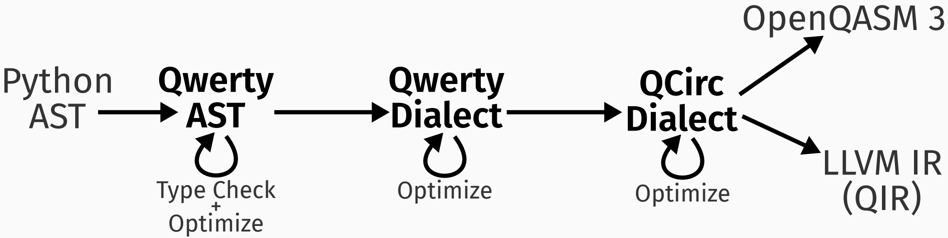
Basis translation

Qubit literal

Motivation: Qwerty Needs a Compiler



Overview of the Qwerty Compiler



Qwerty MLIR dialect

- Our Qwerty dialect is the **quantum MLIR dialect with the highest known level of abstraction**
- Example: `'p'*4 >> -'p'**4` becomes

```
%12 = arith.constant 3.14159
%13 = qwerty.btrans %8 by {"pppp"} >> {"pppp"@(%12)}
```

Qwerty MLIR dialect

- Our Qwerty dialect is the **quantum MLIR dialect with the highest known level of abstraction**
- Example: `'p'*4 >> -'p'**4` becomes

```
%12 = arith.constant 3.14159
%13 = qwerty.btrans %8 by {"pppp"} >> {"pppp"@(%12)}
```

Basis-oriented ops

Bases

Qwerty IR has basis-oriented ops rather than gate ops

Qwerty Dialect

Two kinds of `ops`:

1. Basis ops: `btrans`, `measure`
2. Function ops: `func`, `call`, etc.

Calling Functions in Qwerty

Three ways to call a Qwerty function `f`:

Calling Functions in Qwerty

Three ways to call a Qwerty function `f`:

1. Run `f` forward: `f(arg)`

Calling Functions in Qwerty

Three ways to call a Qwerty function `f`:

1. Run `f` forward: `f(arg)`
2. Run `f` backward: `(~f)(arg)`

Calling Functions in Qwerty

Three ways to call a Qwerty function `f`:

1. Run `f` forward: `f(arg)`
2. Run `f` backward: `(~f)(arg)`
3. Run `f` in a proper subspace (*predicate*): `(f if '1_1' else id)(arg)`

Calling Functions in Qwerty

Three ways to call a Qwerty function value `f`:

1. Run `f` forward: `f(arg)`
2. Run `f` backward: `(~f)(arg)`
3. Run `f` in a proper subspace (*predicate*):
   `(f if '1_1' else id)(arg)`

↑
Function value

Calling Functions in Qwerty

Three ways to call a Qwerty function value `f`:

1. Run `f` forward: `f(arg)`
2. Run `f` backward: `(~f)(arg)`
3. Run `f` in a proper subspace (*predicate*):
   `(f if '1_1' else id)(arg)`

Function value

✓ `(~(f if '1_1' else id))(arg)`

Handling Reverse Calls

```
(~f)(arg)
```

Handling Reverse Calls

```
(~f)(arg)
```

$\downarrow$ Lower from AST

```
%0 = qwerty.func_const @f
%1 = qwerty.func_rev %0
%2 = qwerty.call_indirect %1(%arg)
```

Handling Reverse Calls

```
(~f)(arg)
```

$\downarrow$ Lower from AST

```
%0 = qwerty.func_const @f
%1 = qwerty.func_rev %0
%2 = qwerty.call_indirect %1(%arg)
```

$\downarrow$ Canonicalize

```
%1 = qwerty.call rev @f(%arg)
```

Handling Reverse Calls

```
(~f)(arg)
```

$\downarrow$ Lower from AST

```
%0 = qwerty.func_const @f
%1 = qwerty.func_rev %0
%2 = qwerty.call_indirect %1(%arg)
```

$\downarrow$ Canonicalize

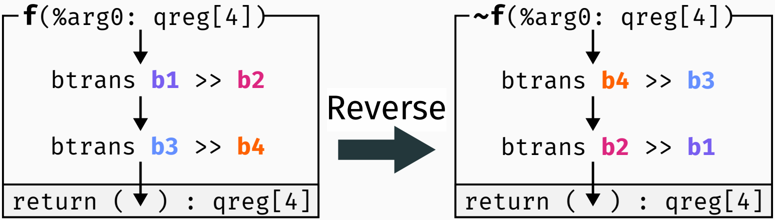
```
%1 = qwerty.call rev @f(%arg)
```

$\downarrow$ Inline

?

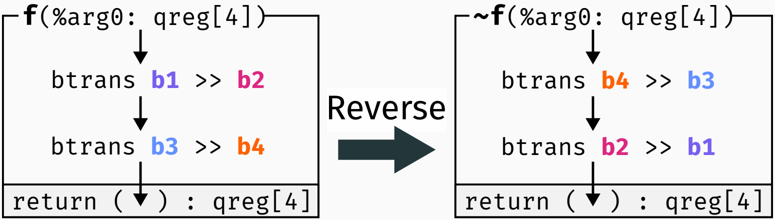
Reversing Basic Blocks

- Qwerty allows getting the reversed form of a function `f` with `~f`
- Example:



Reversing Basic Blocks

- Qwerty allows getting the reversed form of a function `f` with `~f`
- Example:



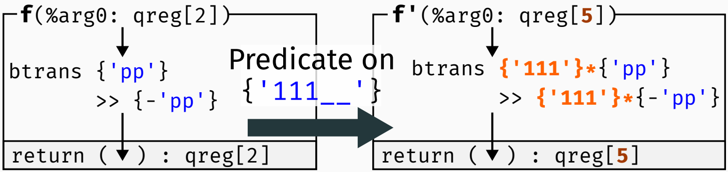
- Novel `Reversible` op interface

Predicating Basic Blocks

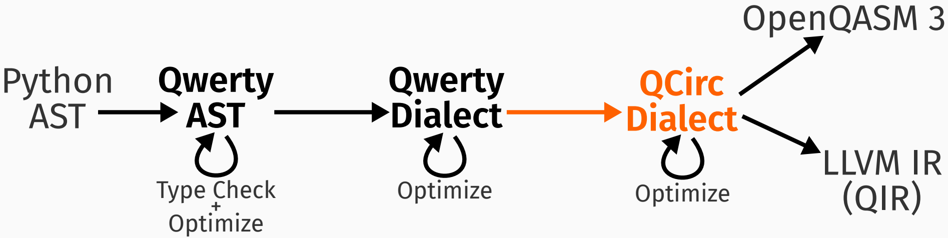
- Qwerty syntax for *predicating* a function `f` on basis pattern `'111__'`:

```
f if '111__' else id
```

- Novel `Predicatable` op interface
- Example:



Next: Quantum Circuit Synthesis



QCirc Dialect

General quantum circuit dialect

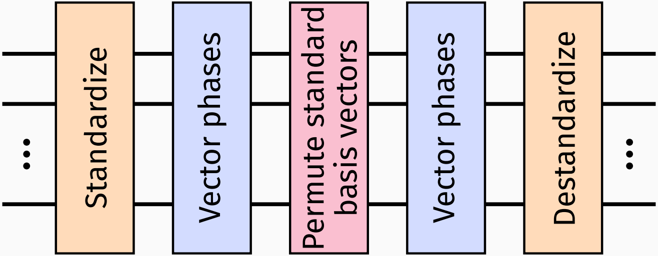
Example:

```
%c3, %c4, %t5 = gate Z [%c0, %c1](%t2)
```

Similar dialects:

1. McCaskey and Nguyen: MLIR dialect for QIR
2. QSSA, QIRO: Quantum SSA
3. Commercial: IBM (qe-compiler), Xanadu (Catalyst), Nvidia (Quake)

Basis Translation Synthesis



Basis Translation Synthesis: Example

```
%0 = arith.constant 3.14159
%out_reg = btrans {'pp'} >> {'pp'@(%0)} %in_reg
```

$\big\downarrow$ Lower

```
%0 = arith.constant 3.14159
%1:2 = unpack %in_reg
```

Basis Translation Synthesis: Example

```
%0 = arith.constant 3.14159
%out_reg = btrans {'pp'} >> {'pp'@(%0)} %in_reg
```

↓ Lower

```
%0 = arith.constant 3.14159
%1:2 = unpack %in_reg
%t0 = gate H [](%1#0)  // Standardize
%t1 = gate H [](%1#1)
```

Basis Translation Synthesis: Example

```
%0 = arith.constant 3.14159
%out_reg = btrans {'pp'} >> {'pp'@(%0)} %in_reg
```

↓ Lower

```
%0 = arith.constant 3.14159
%1:2 = unpack %in_reg
%t0 = gate H [](%1#0)  // Standardize
%t1 = gate H [](%1#1)
%t2 = gate X [](%t0)   // Vector phase
%t3 = gate X [](%t1)
%c4, %t5 = gate P(%0) [%t2](%t3)
%t6 = gate X [](%c4)
%t7 = gate X [](%t5)
```

Basis Translation Synthesis: Example

```
%0 = arith.constant 3.14159
%out_reg = btrans {'pp'} >> {'pp'@(%0)} %in_reg
```

↓ Lower

```
%0 = arith.constant 3.14159
%1:2 = unpack %in_reg
%t0 = gate H [](%1#0)  // Standardize
%t1 = gate H [](%1#1)
%t2 = gate X [](%t0)   // Vector phase
%t3 = gate X [](%t1)
%c4, %t5 = gate P(%0) [%t2](%t3)
%t6 = gate X [](%c4)
%t7 = gate X [](%t5)
%t8 = gate H [](%t6)   // Destandardize
%t9 = gate H [](%t7)
```

Basis Translation Synthesis: Example

```
%0 = arith.constant 3.14159
%out_reg = btrans {'pp'} >> {'pp'@(%0)} %in_reg
```

↓ Lower

```
%0 = arith.constant 3.14159
%1:2 = unpack %in_reg
%t0 = gate H [](%1#0)  // Standardize
%t1 = gate H [](%1#1)
%t2 = gate X [](%t0)   // Vector phase
%t3 = gate X [](%t1)
%c4, %t5 = gate P(%0) [%t2](%t3)
%t6 = gate X [](%c4)
%t7 = gate X [](%t5)
%t8 = gate H [](%t6)   // Destandardize
%t9 = gate H [](%t7)
%out_reg = pack %t8, %t9
```

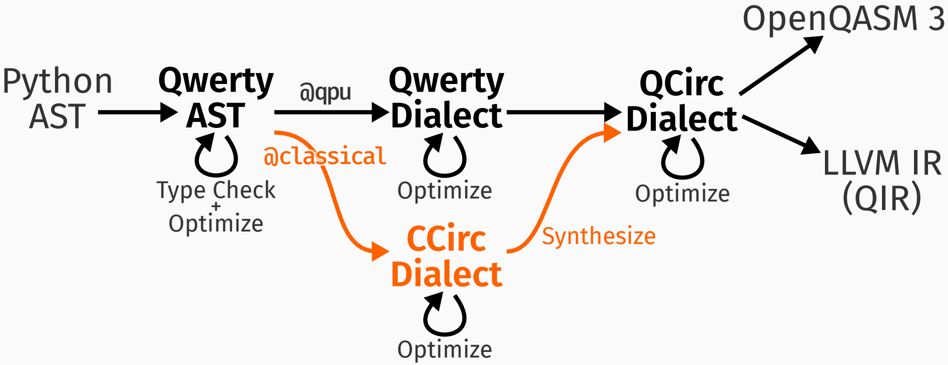
Reminder: Classical Functions

```
@classical
def oracle(x: bit[4]) -> bit:
    return x.and_reduce()

@qpu
def grover_iter(q):
    return (q | oracle.sign
              | 'p'**4 >> -'p'**4)
@qpu
def grover():
    return ('p'**4 | grover_iter
                   | grover_iter
                   | grover_iter
                   | measure**4)
```

Classical oracle

Classical to Quantum Synthesis

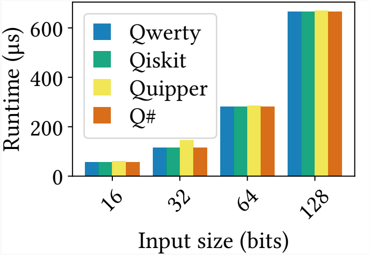


Evaluation

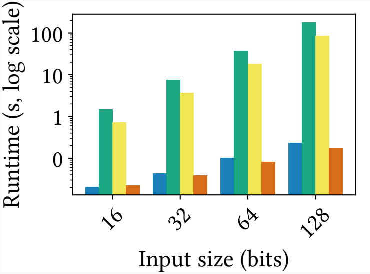
How do circuits we synthesize compare to handwritten circuits?

Evaluation: Fault-Tolerant Runtime

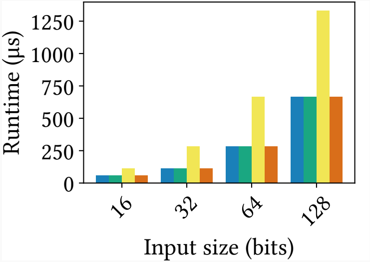
Bernstein–Vazirani



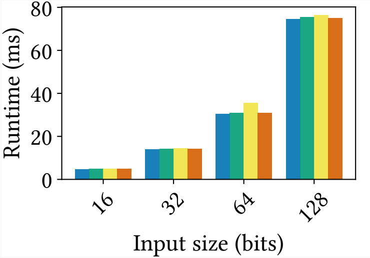
Grover's



Simon's



Period finding



Evaluation Takeaway

Overall, the Qwerty compiler keeps pace with handwritten circuits compiled with gate-oriented compilers.

Conclusion

In this talk, I presented the Qwerty compiler, which leverages both MLIR and novel basis-oriented compilation techniques to enable Qwerty's high-level quantum programming paradigm with minimal overhead.

Conclusion

In this talk, I presented the Qwerty compiler, which leverages both MLIR and novel basis-oriented compilation techniques to enable Qwerty's high-level quantum programming paradigm with minimal overhead.

Qwerty paper: QCE '25

Compiler paper: CGO '25

GitHub: gt-tinker/qwerty

Thank you to: Sharjeel Khan, Arjun Bhamra, Ryan Abusaada, Cameron Hoechst, Jaehun Baek, Owen Sigg, Tom Conte, and more

Backup Slides

Span Equivalence Checking

- Core Qwerty primitive: basis translation b_1 `>>` b_2, where b_1 and b_2 are bases
- Qwerty type checking requires $\mathsf{span}(b_1) = \mathsf{span}(b_2)$

Span Equivalence Checking

- Core Qwerty primitive: **basis translation** b_1 `>>` b_2, where b_1 and b_2 are bases
- Qwerty type checking requires $\mathsf{span}(b_1) = \mathsf{span}(b_2)$
- Examples:
  ✓ `{'0','1'} >> {'0','1'@90}`

Span Equivalence Checking

- Core Qwerty primitive: **basis translation** b_1 `>>` b_2, where b_1 and b_2 are bases
- Qwerty type checking requires $\textsf{span}(b_1) = \textsf{span}(b_2)$
- Examples:

✓ `{'0','1'} >> {'0','1'@90}`

✗ `{'0'} >> {'1'}`

Span Equivalence Checking

- Core Qwerty primitive: **basis translation** b_1 `>>` b_2, where b_1 and b_2 are bases
- Qwerty type checking requires $\mathsf{span}(b_1) = \mathsf{span}(b_2)$
- Examples:

✓ `{'0','1'} >> {'0','1'@90}`

✗ `{'0'} >> {'1'}`

Efficient! (*Not* exponential time)

Basis Translation Synthesis: Permutation

```
%out_reg = btrans {'01','10'}
                 >> {'10','01'} %in_reg
```

$\big\downarrow$ Lower

```
%0:2 = unpack %in_reg
```

Basis Translation Synthesis: Permutation

```
%out_reg = btrans {'01','10'}
                  >> {'10','01'} %in_reg
```

$$\Big\downarrow \text{Lower}$$

```
%0:2 = unpack %in_reg
%c0, %t1 = gate X [%0#0](%0#1)  // Permutation
%c2, %t3 = gate X [%t1](%c0)
%c4, %t5 = gate X [%t3](%c2)
```

Basis Translation Synthesis: Permutation

```
%out_reg = btrans {'01','10'}
                   >> {'10','01'} %in_reg
```

↓ Lower

```
%0:2 = unpack %in_reg
%c0, %t1 = gate X [%0#0](%0#1)  // Permutation
%c2, %t3 = gate X [%t1](%c0)
%c4, %t5 = gate X [%t3](%c2)
%out_reg = pack %c4, %t5
```

Basis Translation Synthesis: Permutation

```
%out_reg = btrans {'01','10'}
                  >> {'10','01'} %in_reg
```

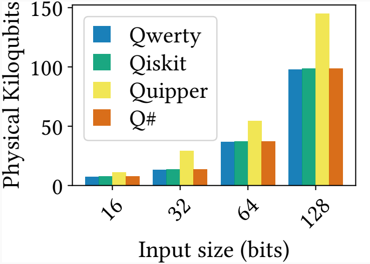
↓ Lower

```
%0:2 = unpack %in_reg
%c0, %t1 = gate X [%0#0](%0#1)  // Permutation
%c2, %t3 = gate X [%t1](%c0)
%c4, %t5 = gate X [%t3](%c2)
%out_reg = pack %c4, %t5
```

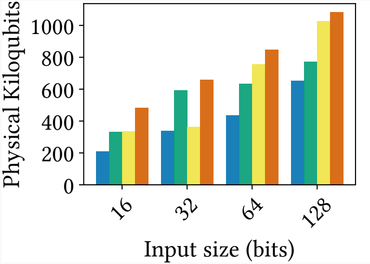
Permutation uses Tweedledum from EPFL

Evaluation: Fault-Tolerant Physical Qubits

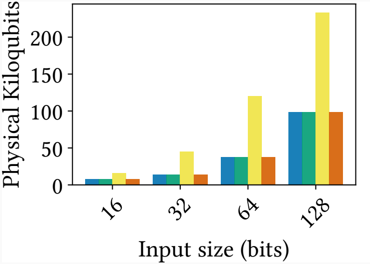
Bernstein–Vazirani



Grover's



Simon's



Period finding