ASDF: A Compiler for Qwerty, a Basis-Oriented Quantum Programming Language

CGO ’25

Austin J. Adams*, Sharjeel Khan*, Arjun S. Bhamra*, Ryan R. Abusaada*, Anthony M. Cabrera†, Cameron C. Hoechst*, Travis S. Humble†, Jeffrey S. Young*, Thomas M. Conte*

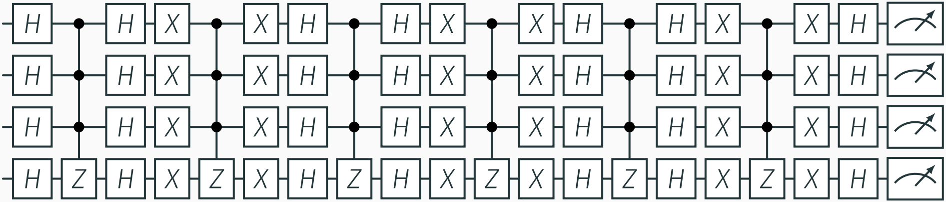
March 4th, 2025

*Georgia Tech and †Oak Ridge National Laboratory

Background: Quantum Computing

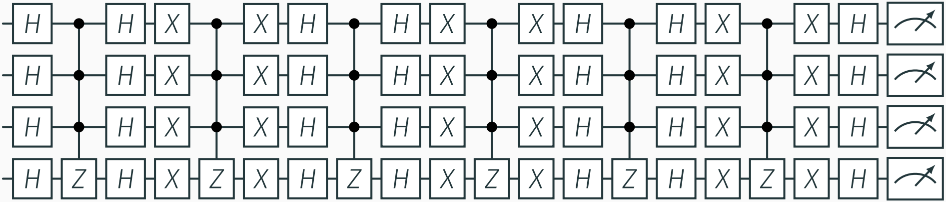
- Quantum computers promise exponential speedup for important problems (e.g., integer factoring and physics simulation)
- …but current quantum programming languages (e.g., Q# or Qiskit) require programming in low-level quantum assembly (quantum *gates* and *circuits*)

Background: Example Quantum Circuit



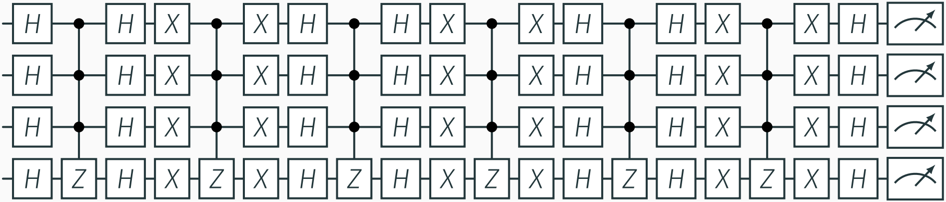
Background: Example Quantum Circuit

Unstructured search algorithm:



Background: Example Quantum Circuit

Unstructured search algorithm:



Tedious, tricky to write (like classical assembly)

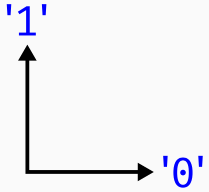
Background: The Qwerty Programming Language

- **Qwerty**: high-level quantum DSL embedded in Python
- Primitives are **basis translations** rather than quantum gates
  - Computation is a pipeline:

`x | f | g` means $g(f(x))$

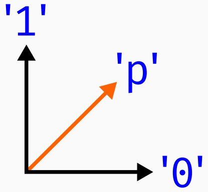
Hello World in Qwerty

Qubit literals:



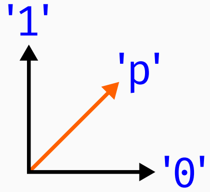
Hello World in Qwerty

Qubit literals:



Hello World in Qwerty

Qubit literals:

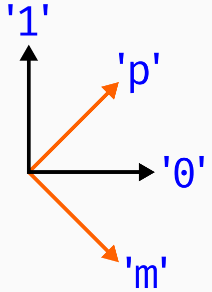


Random bit generator:

```
'p' | measure
```

Hello World in Qwerty

Qubit literals:



Random bit generator:

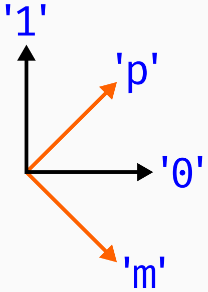
```
'p' | measure
```

Example basis literal:

```
{'p','m'}
```

Hello World in Qwerty

Qubit literals:



Random bit generator:

```
'p' | measure
```

Example basis literal:

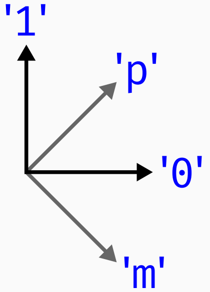
```
{'p','m'}
```

Always measures a 1:

```
'p' | {'p','m'} >> {'1','0'}
    | measure
```

Hello World in Qwerty

Qubit literals:



Random bit generator:

```
'p' | measure
```

Example basis literal:

```
{'p','m'}
```

Always measures a 1:

Basis translation

```
'p' | {'p','m'} >> {'1','0'}
    | measure
```

Realistic Qwerty Example: Grover diffuser

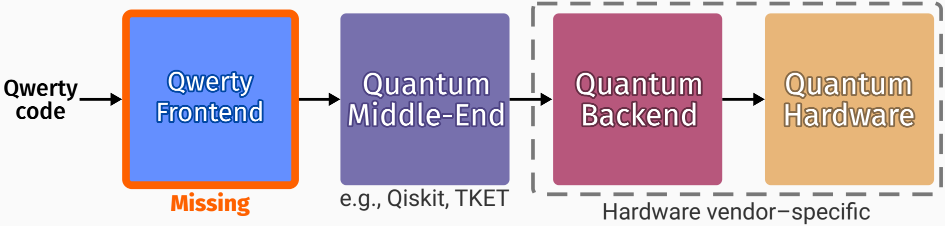
Qwerty:

```
'p'[N] >> -'p'[N]
```

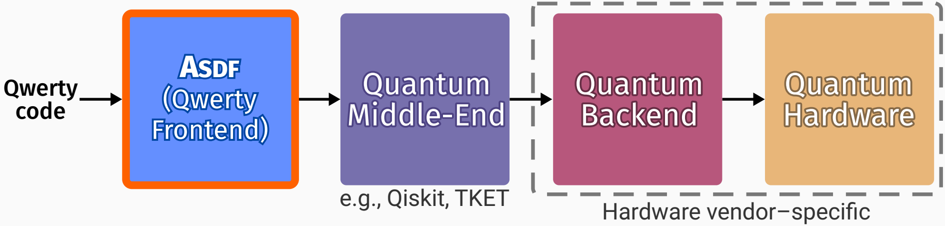
Q# (Prior work):

```
within {
    ApplyToEachA(H, q);
    ApplyToEachA(X, q);
} apply {
    Controlled Z(Most(q),
                 Tail(q));
}
```

Motivation: Qwerty Needs a Compiler

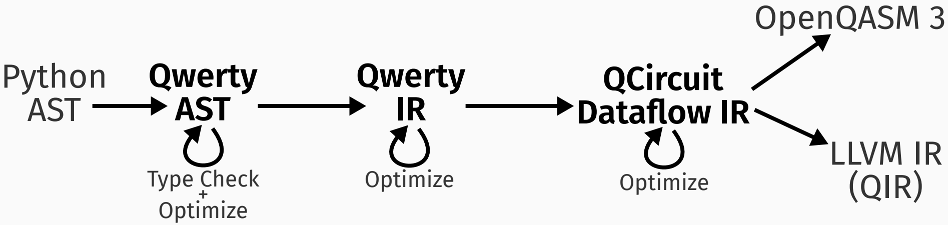


Motivation: Qwerty Needs a Compiler



We present ASDF, the first compiler for a basis-oriented quantum programming language.

Overview of ASDF



Challenges in Compiling Qwerty

① **Fast compilation** of basis-oriented operations

Challenges in Compiling Qwerty

1. **Fast compilation** of basis-oriented operations
2. **Synthesizing high-quality circuits**

Challenges in Compiling Qwerty

1. **Fast compilation** of basis-oriented operations
2. **Synthesizing high-quality circuits**
3. **Inlining code:** Qwerty is functional, quantum hardware is not

Challenges in Compiling Qwerty

1. **Fast compilation** of basis-oriented operations
2. **Synthesizing high-quality circuits**
3. **Inlining code:** Qwerty is functional, quantum hardware is not
4. **Integration** with quantum ecosystem

Our Contributions

Challenge ① — Fast compilation:

1. **Efficiently type checking** basis translations, avoiding exponential time

Our Contributions

Challenge ① — Fast compilation:

1. **Efficiently type checking** basis translations, avoiding exponential time

Challenge ② — High-quality circuits:

2. **Circuit synthesis optimized** for basis translations
   - **Empirically**, emitted code close to hand-optimized

Our Contributions

Challenge ① — Fast compilation:

1. **Efficiently type checking** basis translations, avoiding exponential time

Challenge ② — High-quality circuits:

2. **Circuit synthesis optimized** for basis translations
    - **Empirically**, emitted code close to hand-optimized

Challenge ③ — Inlining:

3. **Qwerty IR**: IR customized for Qwerty
4. **Automated reversal/predication** of quantum basic blocks

Our Contributions

Challenge ① — Fast compilation:

1. **Efficiently type checking** basis translations, avoiding exponential time

Challenge ② — High-quality circuits:

2. **Circuit synthesis optimized** for basis translations
    - **Empirically**, emitted code close to hand-optimized

Challenge ③ — Inlining:

3. **Qwerty IR**: IR customized for Qwerty
4. **Automated reversal/predication** of quantum basic blocks

Challenge ④ — Integration:

5. Embedded in **Python**, outputs **industry-standard IRs**

Span Equivalence Checking

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

Span Equivalence Checking

- Core Qwerty primitive: **basis translation** b_1 `>>` b_2, where b_1 and b_2 are bases
- Qwerty type checking requires that $\mathsf{span}(b_1) = \mathsf{span}(b_2)$
  - However, naïve type checking can take exponential time
  - **Key insight:** determine span equivalence by **factoring bases**:

```
{'00','01'} >> {'0'} + {'p','m'}
```

Span Equivalence Checking

- Core Qwerty primitive: **basis translation** b_1 `>>` b_2, where b_1 and b_2 are bases
- Qwerty type checking requires that $\mathsf{span}(b_1) = \mathsf{span}(b_2)$
  - However, naïve type checking can take exponential time
  - **Key insight:** determine span equivalence by **factoring bases**:

```
{'00','01'} >> {'0'} + {'p','m'}
                  ↓
{'0'} + {'0','1'} >> {'0'} + {'p','m'}
```

Span Equivalence Checking

- Core Qwerty primitive: **basis translation** b_1 `>>` b_2, where b_1 and b_2 are bases
- Qwerty type checking requires that $\mathsf{span}(b_1) = \mathsf{span}(b_2)$
  - However, naïve type checking can take exponential time
  - **Key insight:** determine span equivalence by **factoring bases**:

```
{'00','01'} >> {'0'} + {'p','m'}
                   ↓
{'0'} + {'0','1'} >> {'0'} + {'p','m'}
```

Identical

Span Equivalence Checking

- Core Qwerty primitive: **basis translation** b_1 `>>` b_2, where b_1 and b_2 are bases
- Qwerty type checking requires that $\mathsf{span}(b_1) = \mathsf{span}(b_2)$
  - However, naïve type checking can take exponential time
  - **Key insight:** determine span equivalence by **factoring bases**:

```
{'00','01'} >> {'0'} + {'p','m'}
```

$\downarrow$

```
{'0'} + {'0','1'} >> {'0'} + {'p','m'}
```

Identical

Both fully span

✓ Spans are equivalent

Span Equivalence Checking

- Core Qwerty primitive: **basis translation** b_1 `>>` b_2, where b_1 and b_2 are bases
- Qwerty type checking requires that $\mathsf{span}(b_1) = \mathsf{span}(b_2)$
  - However, naïve type checking can take exponential time
  - **Key insight:** determine span equivalence by **factoring bases**:

```
{'00','01'} >> {'0'} + {'p','m'}
↓
{'0'} + {'0','1'} >> {'0'} + {'p','m'}
```

Identical

Both fully span

✓ Spans are equivalent

ASDF checks span equivalence in $O(n^2 \log n)$ time instead of exponential time

Qwerty IR

- Qwerty IR is the **quantum MLIR dialect with the highest level of abstraction**
- For example, `'p'[3] >> -'p'[3]` becomes the following IR:

```
%12 = arith.constant 3.14159
%13 = qwerty.qbtrans %8 by {"ppp"} >> {exp(i*%12)*"ppp"}
```

Qwerty IR

- Qwerty IR is the **quantum MLIR dialect with the highest level of abstraction**
- For example, `'p'[3] >> -'p'[3]` becomes the following IR:

```
%12 = arith.constant 3.14159
%13 = qwerty.qbtrans %8 by {"ppp"} >> {exp(i*%12)*"ppp"}
```

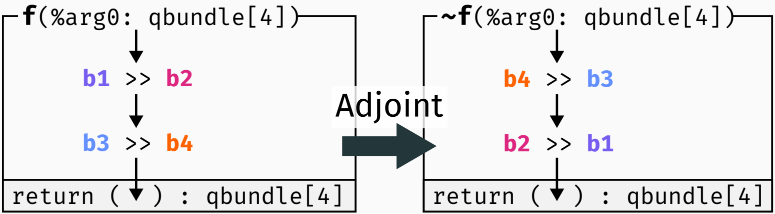
Basis-oriented ops

Bases

Qwerty IR has basis-oriented ops rather than gate ops

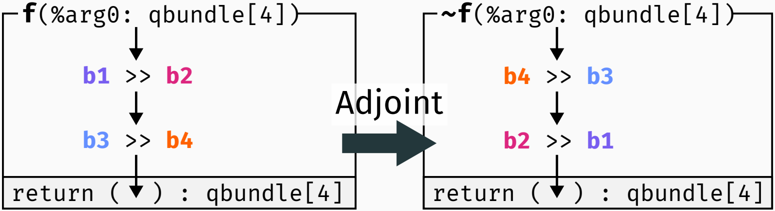
Reversing Basic Blocks

- Qwerty allows instantiating the adjoint (reversed form) of a function `f` with `~f`
- Example: ASDF taking adjoint of `f`:



Reversing Basic Blocks

- Qwerty allows instantiating the adjoint (reversed form) of a function `f` with `~f`
- Example: ASDF taking adjoint of `f`:



- Novel `Adjointable` op interface in MLIR

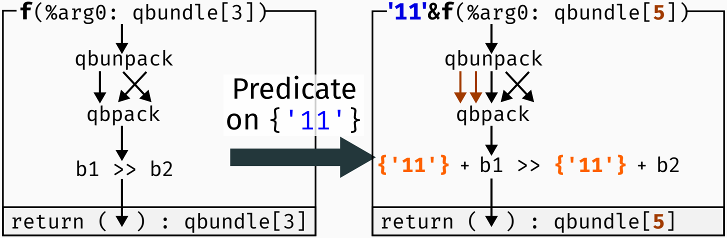
Predicating Basic Blocks

- Qwerty syntax for *predicating* a function `f` with basis `b`:

  `b & f`

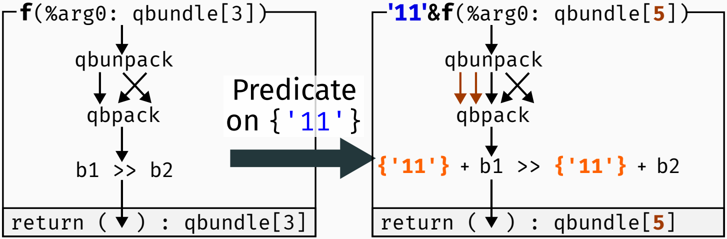
- `b & f` will run only in the proper subspace `b`

- Example:



Predicating Basic Blocks

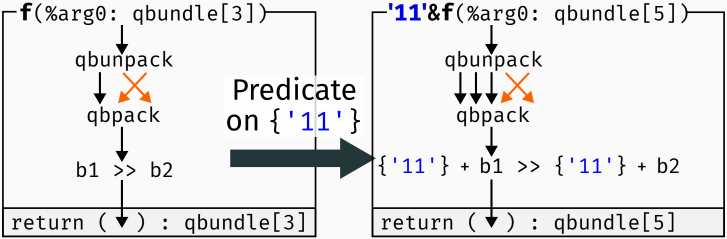
- Qwerty syntax for *predicating* a function `f` with basis `b`:
  `b & f`
- `b & f` will run only in the proper subspace `b`
- Novel **`Predicatable`** op interface in MLIR
- Example:



Predicating Basic Blocks

- Qwerty syntax for *predicating* a function `f` with basis `b`:

  `b & f`
- `b & f` will run only in the proper subspace `b`
- Novel **`Predicatable`** op interface in MLIR
- Example:



Predicating Basic Blocks

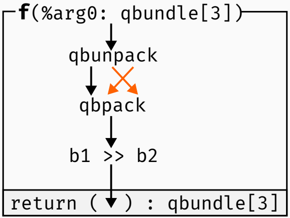
- Qwerty syntax for *predicating* a function `f` with basis `b`:

  `b & f`
- `b & f` will run only in the proper subspace `b`
- Novel **`Predicatable`** op interface in MLIR
- Example:

Qwerty Code

```
@qpu
def f(q: qubit[3]) -> qubit[3]:
    q1, q2, q3 = q
    return q1+q3+q2 | b1 >> b2
```

Qwerty IR

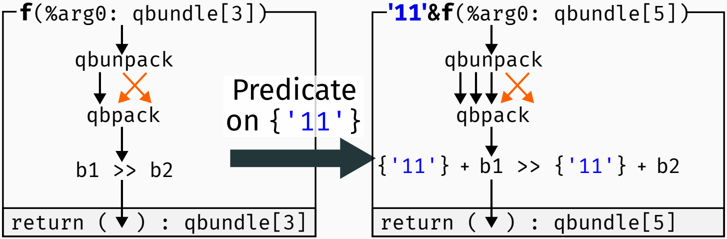


Predicating Basic Blocks

- Qwerty syntax for *predicating* a function `f` with basis `b`:

  `b & f`

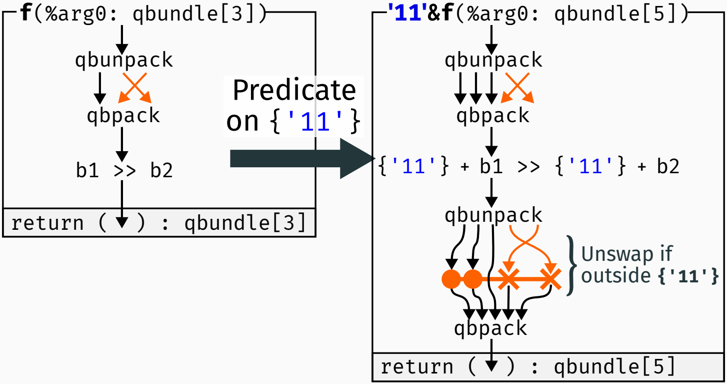
- `b & f` will run only in the proper subspace `b`
- Novel **`Predicatable`** op interface in MLIR
- Example:



Predicating Basic Blocks

- Qwerty syntax for *predicating* a function `f` with basis `b`:

  `b & f`
- `b & f` will run only in the proper subspace `b`
- Novel **`Predicatable`** op interface in MLIR
- Example:



Basis Translation Synthesis: Example 1

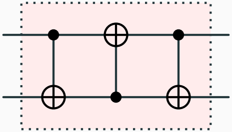
```
{'01','10'} >> {'10','01'}
```

$\downarrow$

Permutation

$|00\rangle \mapsto |00\rangle$ $\quad$ $|01\rangle \mapsto |10\rangle$

$|10\rangle \mapsto |01\rangle$ $\quad$ $|11\rangle \mapsto |11\rangle$



Basis Translation Synthesis: Example 1

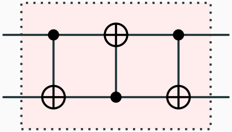
```
{'01','10'} >> {'10','01'}
```

$\downarrow$

Permutation

$|00\rangle \mapsto |00\rangle \quad |01\rangle \mapsto |10\rangle$

$|10\rangle \mapsto |01\rangle \quad |11\rangle \mapsto |11\rangle$



Permutation synthesis uses Tweedledum library from EPFL

Basis Translation Synthesis: Example 2

```
'p'[3] >> -'p'[3]
```

$\downarrow$

Standardize

$|+\rangle \mapsto |0\rangle$

$|-\rangle \mapsto |1\rangle$

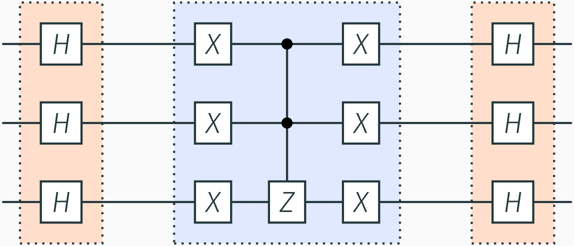
Vector phase

$|000\rangle \mapsto -|000\rangle$

Destandardize

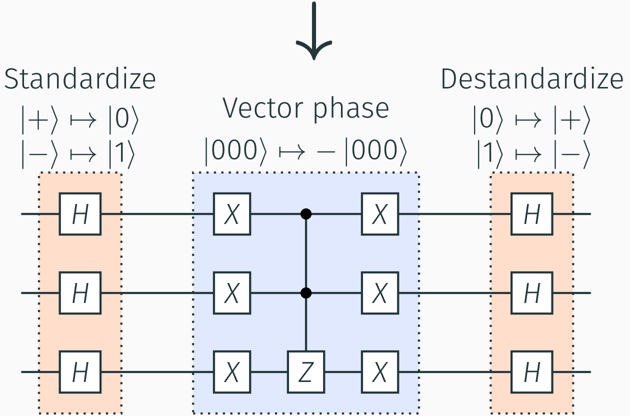
$|0\rangle \mapsto |+\rangle$

$|1\rangle \mapsto |-\rangle$



Basis Translation Synthesis: Example 2

```
'p'[3] >> -'p'[3]
```



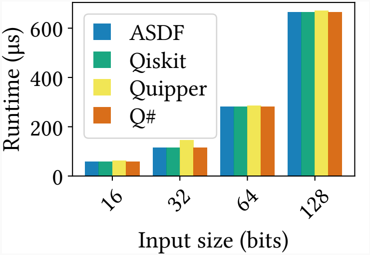
ASDF is the first compiler capable of synthesizing quantum circuits from basis translations

Evaluation

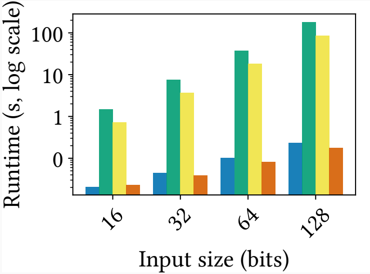
How do ASDF-synthesized circuits compare to handwritten circuits?

Evaluation: Fault-Tolerant Runtime

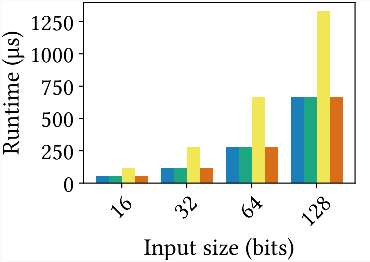
Bernstein–Vazirani



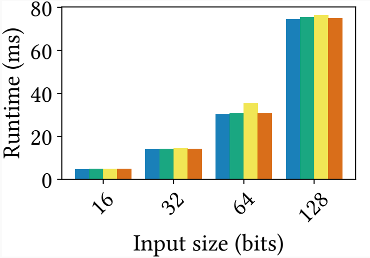
Grover's



Simon's

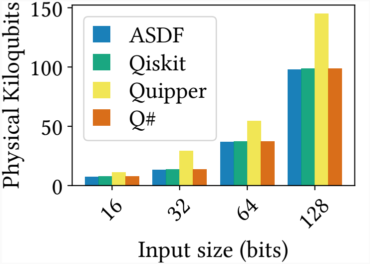


Period finding

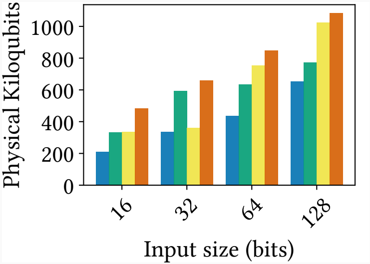


Evaluation: Fault-Tolerant Physical Qubits

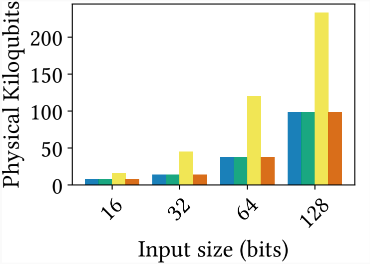
Bernstein–Vazirani



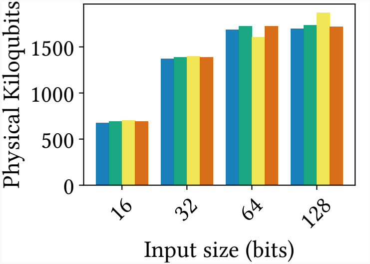
Grover's



Simon's



Period finding



Evaluation Takeaway

Overall, ASDF keeps pace with handwritten circuits compiled with gate-oriented compilers.

Conclusion

In this talk, I presented Asdf, a compiler that leverages novel basis-oriented compilation techniques to enable Qwerty's high-level quantum programming paradigm with minimal overhead.

Conclusion

In this talk, I presented Asdf, a compiler that leverages novel basis-oriented compilation techniques to enable Qwerty's high-level quantum programming paradigm with minimal overhead.

Qwerty tech report:

arXiv:2404.12603

Source code:

github.com/gt-tinker/qwerty

Backup Slides

Full Bernstein–Vazirani Example Program

```
from qwerty import *

def bv(secret_string):
    @classical[[N]](secret_string)
    def f(secret_string: bit[N], x: bit[N]) -> bit:
        return (secret_string & x).xor_reduce()

    @qpu[[N]](f)
    def kernel(f: cfunc[N,1]) -> bit[N]:
        return 'p'[N] | f.sign \
                      | pm[N] >> std[N] \
                      | measure[N]

    return kernel()

secret_string = bit.from_str('1101')
print(bv(secret_string))
```

Predication Example

Imagine `'0' & ({'0','1'} >> {'p','m'})`.

This performs the following:

`'00'` $\mapsto$ `'0p'`
`'01'` $\mapsto$ `'0m'`
`'10'` $\mapsto$ `'10'`
`'11'` $\mapsto$ `'11'`

QCirc Dialect Example

```
%q = qcirc.H %0
%q_0 = qcirc.H %1
%q_1 = qcirc.H %2
%q_2 = qcirc.X %q_1
%q_3 = qcirc.X %q_0
%q_4 = qcirc.X %q
%ctrlq:2, %q_5 = qcirc.Z [%q_4, %q_3] %q_2
%q_6 = qcirc.X %q_5
%q_7 = qcirc.X %ctrlq#1
%q_8 = qcirc.X %ctrlq#0
%q_9 = qcirc.H %q_8
%q_10 = qcirc.H %q_7
%q_11 = qcirc.H %q_6
```

Inspired by QIRO and QSSA

Qubit Index Analysis