

Expanding the VQQC Toolkit

Kesha Hietala¹ Liyi Li¹ Akshaj Gaur² Aaron Green¹
Robert Rand³ Xiaodi Wu¹ Michael Hicks¹

¹ University of Maryland


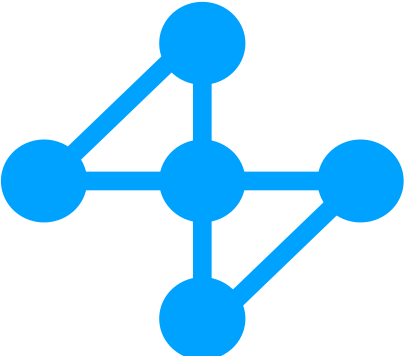

² Poolesville High School

³ University of Chicago

VOQC: Verified Optimizer for Quantum Circuits

- An optimizer for quantum circuits *formally verified* in the Coq proof assistant
 - Optimizations are proved to be *semantics preserving*, i.e., they do not change the “meaning” of the input circuit
- Circuits expressed in SQIR, a Simple Quantum Intermediate Representation
- VOQC and SQIR were presented in a distinguished paper at [POPL 2021](#)
- Followup paper to appear at [ITP 2021](#) shows how to use SQIR as a source language for verifying quantum algorithms (e.g. Grover’s, QPE)

In This Talk

- New gate sets and optimizations  Qiskit
- Better support for circuit mapping 
- Interoperability (via Python) 

“IBM” Gate Set

- Consists of the gates {U1, U2, U3, CX}
- U1, U2, U3 are parameterized by real rotation angles

$$U_1(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix}, \quad U_2(\phi, \lambda) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -e^{i\lambda} \\ e^{i\phi} & e^{i(\phi+\lambda)} \end{pmatrix}, \quad U_3(\theta, \phi, \lambda) = \begin{pmatrix} \cos(\theta/2) & -e^{i\lambda} \sin(\theta/2) \\ e^{i\phi} \sin(\theta/2) & e^{i(\phi+\lambda)} \cos(\theta/2) \end{pmatrix}$$

Qiskit's Optimize1qGates

- Finds adjacent single-qubit gates (U1, U2, U3) and combines them

- E.g. merging U1, U2
 - $U_1(\lambda_1); U_1(\lambda_2) \rightarrow U_1(\lambda_1 + \lambda_2)$
 - $U_1(\lambda_1); U_2(\phi, \lambda_2) \rightarrow U_2(\lambda_2, \lambda_1 + \phi)$

- More complicated: merging U3

$$\begin{aligned}
 U_3(\theta_1, \phi_1, \lambda_1); U_3(\theta_2, \phi_2, \lambda_2) &= R_z(\phi_2) \cdot R_y(\theta_2) \cdot R_z(\lambda_2) \cdot R_z(\phi_1) \cdot R_y(\theta_1) \cdot R_z(\lambda_1) \\
 &= R_z(\phi_2) \cdot [R_y(\theta_2) \cdot R_z(\lambda_2 + \phi_1) \cdot R_y(\theta_1)] \cdot R_z(\lambda_1) \\
 &= R_z(\phi_2) \cdot \underline{[R_z(\gamma) \cdot R_y(\beta) \cdot R_z(\alpha)]} \cdot R_z(\lambda_1) \\
 &= R_z(\phi_2 + \gamma) \cdot R_y(\beta) \cdot R_z(\alpha + \lambda_1) \\
 &= U_3(\beta, \phi_2 + \gamma, \alpha + \lambda_1)
 \end{aligned}$$

Note:

$$U_3(\theta, \phi, \lambda) = R_z(\phi) \cdot R_y(\theta) \cdot R_z(\lambda)$$

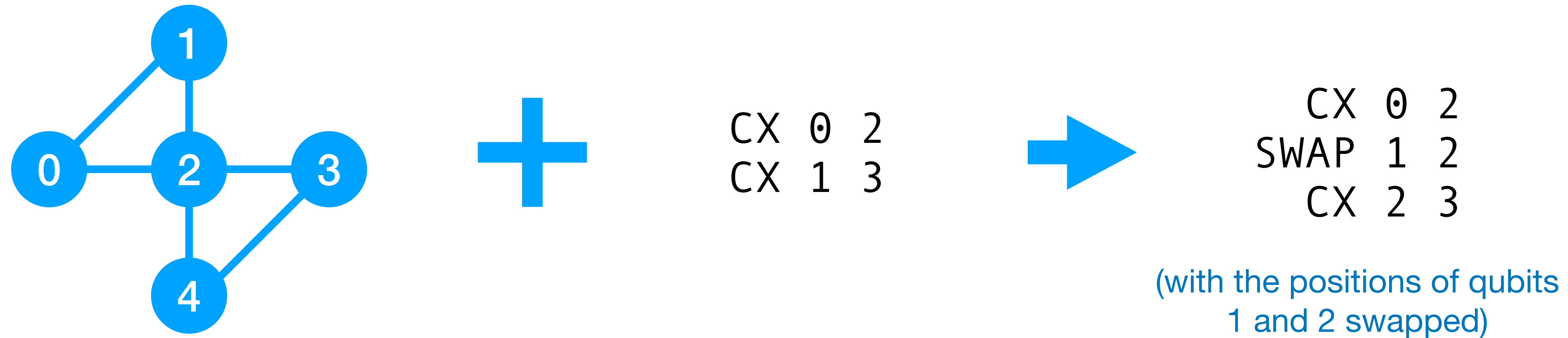
for rotations R_y, R_z

The hard part is $zyz \rightarrow yzy$ conversion

Summary of Features

- Gate sets
 - “RzQ” {X, H, Rz, CX}
 - “IBM” {U1, U2, U3, CX}
 - “Full” {I, X, Y, Z, ..., CX, CZ, SWAP, CCX, CCZ}
- Optimizations
 - Five passes from [Nam et al. \[2018\]](#) (evaluated in our POPL paper)
 - [Optimize1qGates](#) and [CXCancellation](#) from Qiskit
- Simple circuit mapping

Circuit Mapping



- We want this transformation to...
 - Be *semantics-preserving* (the two programs should be denoted by the same matrix, up to a permutation of qubits)
 - Produce an output that satisfies the architecture's constraints

Composing VOQC Transformations

- Coq program to optimize a circuit and then map it to a 10-qubit LNN architecture (OCaml syntax is similar)

```
Definition optimize_then_map c :=
  let gr := make_lnn 10 in      (* 10-qubit LNN architecture *)
  let la := trivial_layout 10 in (* trivial layout on 10 qubits *)
  if check_well_typed c 10    (* check that c is well-typed & uses ≤10 qubits *)
  then
    let c' := optimize_nam c in (* optimization #1 *)
    let c'' := optimize_ibm c' in (* optimization #2 *)
    Some (simple_map c'' la gr) (* map *)
  else None.
```

Composing VOQC Transformations

- Coq program to map a circuit to a 10-qubit LNN architecture and then perform optimization (OCaml syntax is similar)

```
Definition map_then_optimize c :=
  let gr := make_lnn 10 in          (* 10-qubit LNN architecture *)
  let la := trivial_layout 10 in   (* trivial layout on 10 qubits *)
  if check_well_typed c 10        (* check that c is well-typed & uses ≤10 qubits *)
  then
    let (c', la') := simple_map c la gr in (* map *)
    let c'' := optimize_nam c' in          (* optimization #1 *)
    Some (optimize_ibm c'', la')          (* optimization #2 *)
  else None.
```

- To support optimization after mapping, we prove that all optimizations are *mapping preserving*

(Light) Evaluation

- Is it better to optimize or map first?
- We mapped 26 arithmetic circuits to a 6x6 2D grid architecture (= 36 qubits) and compared the gate count reduction due to optimizing before vs. after

Optimize -> Map	Map -> Optimize	Optimize -> Map -> Optimize
10.4%	8.6%	10.9%

- A verified optimizer allows us to combine different transformations without worrying about correctness
- *In our POPL paper*: evaluation on a larger set of benchmarks; comparison with PyZX, Qiskit, and tket

PyVOQC

- We want to make VOQC a drop-in replacement for circuit optimizers used in frameworks like Qiskit, pytket, pyQuil, Cirq (all written in [Python](#))
- So we added Python bindings for VOQC optimizations



PyVOQC

```
from qiskit import QuantumCircuit
from pyvoqc.qiskit.voqc_pass import QiskitVOQC
from qiskit.transpiler import PassManager

# create a circuit using Qiskit's interface
circ = QuantumCircuit(2)
circ.x(0)
circ.t(0)
circ.t(1)
circ.cz(0, 1)
circ.t(0)
circ.tdg(1)
print("Before Optimization:")
print(circ)

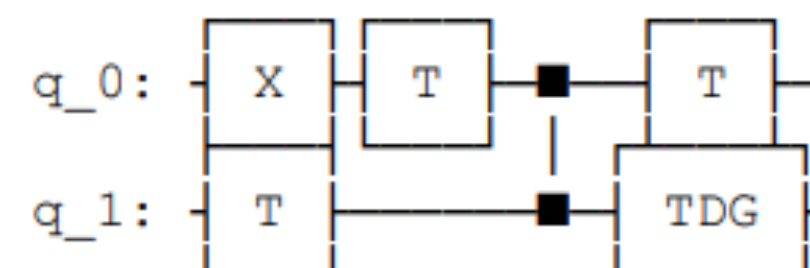
# create a Qiskit PassManager
pm = PassManager()

# decompose CZ gate
pm.append(QiskitVOQC(["decompose_to_cnot"]))
new_circ = pm.run(circ)
print("\n\nAfter 'decompose_to_cnot':")
print(new_circ)

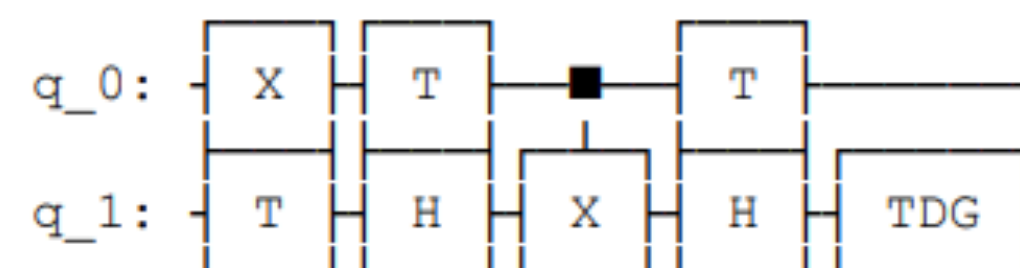
# run optimizations from Nam et al.
pm.append(QiskitVOQC(["optimize_nam", "replace_rzq"]))
new_circ = pm.run(circ)
print("\n\nAfter 'optimize_nam':")
print(new_circ)

# run IBM gate merging
pm.append(QiskitVOQC(["optimize_ibm"]))
new_circ = pm.run(circ)
print("\n\nAfter 'optimize_ibm':")
print(new_circ)
```

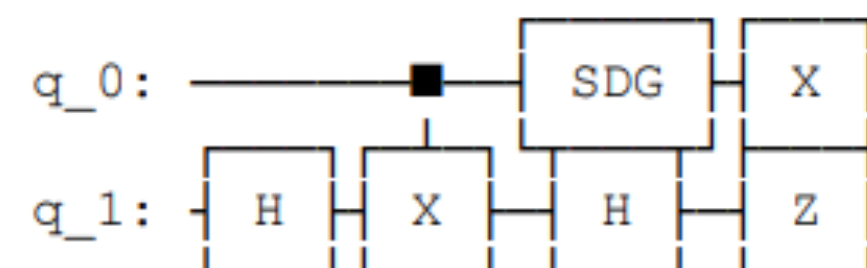
Before Optimization:



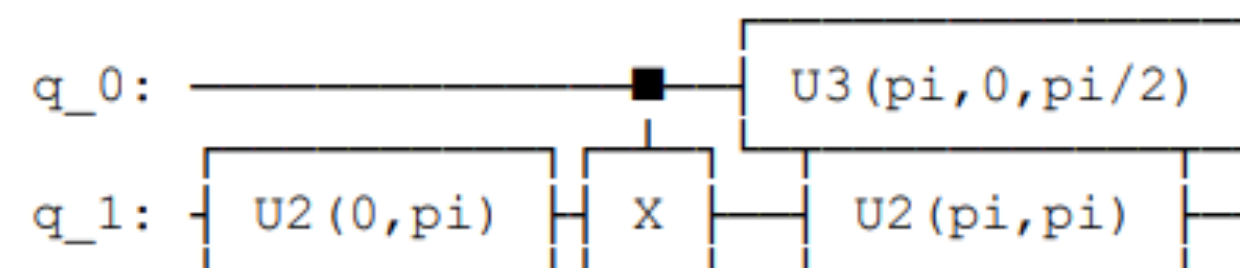
After 'decompose_to_cnot':



After 'optimize_nam':



After 'optimize_ibm':



Ongoing Work

- More thorough evaluation of VOQC (e.g. using [Arline's benchmarks](#))
- New optimizations, especially approximate
- More sophisticated mapping & mapping-aware optimizations
- *In progress*: Compilation from classical (reversible) programs to SQIR circuits

Resources

- Our Coq definitions and proofs are available at github.com/inQWIRE/SQIR.
- Our OCaml library is available at github.com/inQWIRE/mlvoqc and can be installed with “opam install voqc”.
 - Documentation on the OCaml library interface is available at <https://inqwire.github.io/mlvoqc/voqc/Voqc/index.html>.
- Our Python bindings and a tutorial are available at github.com/inQWIRE/pyvoqc.
- ***We welcome contributions!*** Feel free to file issues or pull requests.

Slides for Live Presentation

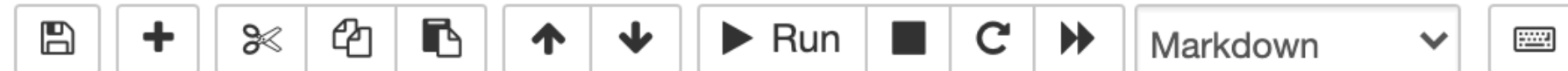
PyVOQC Tutorial

- Assuming you have working installations of pip (for Python 3) and opam

```
# install utilities  
pip install jupyter qiskit  
opam install voqc.0.2.1
```

```
# download and install pyvoqc  
git clone https://github.com/inQWIRE/pyvoqc  
cd pyvoqc  
./install.sh  
jupyter notebook planqc_tutorial.ipynb
```

- Alternatively, view the notebook on GitHub or at https://nbviewer.jupyter.org/github/inQWIRE/pyvoqc/blob/main/planqc_tutorial.ipynb



PyVOQC Tutorial (PLanQC 2021)

This tutorial introduces PyVOQC, the Python bindings for the VOQC optimizer (available at [inQWIRE/pyvoqc](https://github.com/inQWIRE/pyvoqc)). We first show how to use PyVOQC as a pass in Qiskit (our recommended method), and then show how to call PyVOQC functions directly.

Preliminaries

To run this tutorial:

1. Install our OCaml package with `opam install voqc` (requires opam)
2. Run `./install.sh` in the pyvoqc directory

For more details and troubleshooting, see the [README](#) in the pyvoqc repository.

Running PyVOQC as a Qiskit Pass

Using our `voqc_pass` wrapper, VOQC can be called just like any other optimization pass in [IBM's Qiskit framework](#). This allows us to take advantage of Qiskit's utilities for quantum programming, such as the ability to build and print circuits.

To use VOQC, simply append `QiskitVOQC([opt list])` to a Qiskit `Pass Manager` where `opt list` is an optional argument specifying one or more of the transformations in VOQC. `QiskitVOQC()` with no arguments will run all available optimizations.

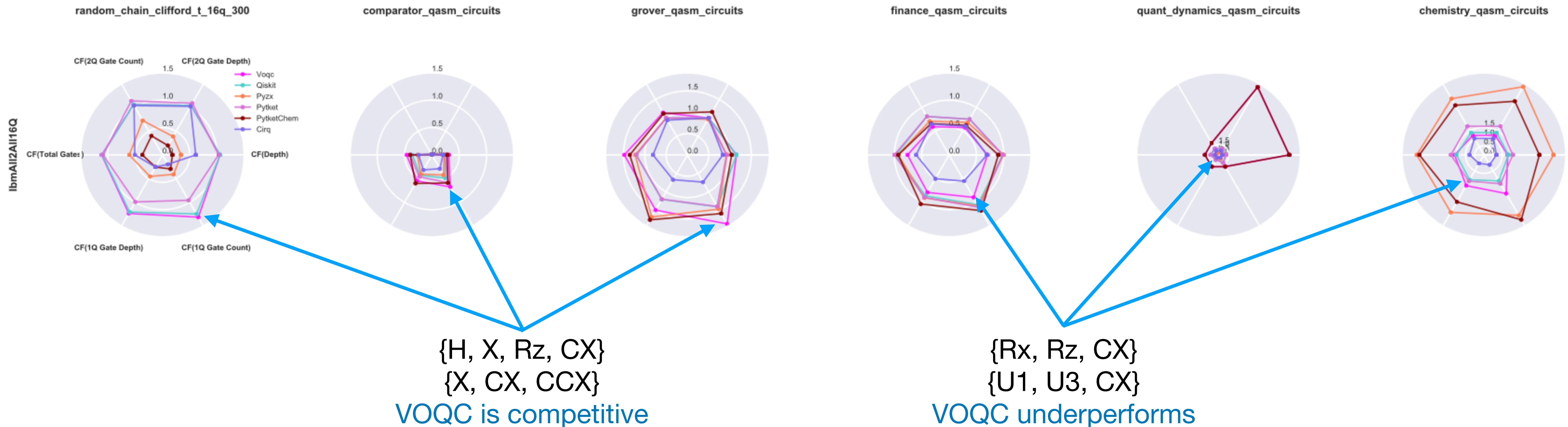
```
In [1]: from qiskit import QuantumCircuit
        from pyvoqc.qiskit.voqc_pass import QiskitVOQC
        from qiskit.transpiler import PassManager

        # create a circuit using Qiskit's interface
        circ = QuantumCircuit(2)
```

Arline Benchmarking

- [Arline Benchmarks](#) is an open-source automated benchmarking platform for quantum compilers
 - Considers a variety of benchmarks, target architectures
 - Considers total gate count & depth, single-qubit gate count & depth, two-qubit gate count & depth, execution time
- Auto-generated reports available for Cirq, pytket, PyZX, Qiskit... and soon VOQC!
- For more details see <https://www.arline.io/>

Preliminary Results



- **Takeaway:** we still have a lot to do! But we are optimistic that we can build a *formally verified* compiler that includes all the optimizations in leading compilers — without the bugs!

Resources

- Code:
 - github.com/inQWIRE/SQIR
 - github.com/inQWIRE/mlvoqc
 - github.com/inQWIRE/pyvoqc
- POPL 2021 paper: dl.acm.org/doi/10.1145/3434318
- SIGPLAN blog post: blog.sigplan.org/2021/06/02/verifying-a-quantum-compiler/
- Contact keshha@cs.umd.edu or rand@uchicago.edu with comments or questions!