A Verified Optimizer for Quantum Circuits

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, Michael Hicks

&

Verified translation between low-level quantum languages

Kartik Singhal, Robert Rand, Michael Hicks
Verified Compiler Stack

- End goal: verified compiler stack for quantum programs

High-level Language
E.g. QWIRE, Quipper, Q#

Unoptimized IR
E.g. OpenQASM, Quil

Optimized IR
E.g. OpenQASM, Quil

Hardware Instructions
Veriﬁed Compiler Stack

• End goal: veriﬁed compiler stack for quantum programs

High-level Language
E.g. QWIRE, Quipper, Q#

Unoptimized IR
E.g. OpenQASM, Quil

Optimized IR
E.g. OpenQASM, Quil

Hardware Instructions

Optimization
Circuit synthesis
Circuit mapping
...
Verified Compiler Stack

- End goal: verified compiler stack for quantum programs

Means that we’ve formally verified that the transformation is semantics-preserving
Verified Compiler Stack

- We present **VOQC**, our Verified Optimizer for Quantum Circuits, which is built on top of **SQIR**, our Small Quantum Intermediate Representation.

- Implemented in 8000 lines of Coq code, with 1500 for core SQIR and the rest for program transformations.

- 400 lines of standalone OCaml code for parsing and translating OpenQASM.

- We extract VOQC to OCaml and compile it to a binary, so using VOQC doesn’t require knowledge of Coq or OCaml.
Verified Compiler Stack

- End goal: verified compiler stack for quantum programs

High-level Language
E.g. QWIRE, Quipper, Q#

Unoptimized IR
E.g. OpenQASM, Quil

Optimized IR
E.g. OpenQASM, Quil

Hardware Instructions

Unoptimized SQIR

Optimized SQIR

VOQC
SQIR

• Syntax

\[
U := U_1; U_2 \mid G q \mid G q_1 q_2 \\
P := \text{skip} \mid P_1; P_2 \mid U \mid \text{meas } q P_1 P_2
\]

• Semantics assumes a \textbf{global register} of size \(d\)
  • A unitary program corresponds to a unitary matrix of size \(2^d \times 2^d\)
  • A non-unitary program corresponds to a function between density matrices of size \(2^d \times 2^d\)


**SQIR**

- **Syntax**

\[
U := U_1; U_2 \mid G q \mid G q_1 q_2
\]

\[
P := \text{skip} \mid P_1; P_2 \mid U \mid \text{meas } q P_1 P_2
\]

- **Semantics** assumes a **global register** of size \(d\)

<table>
<thead>
<tr>
<th>Unitary semantics</th>
<th>Non-unitary semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\left[ U_1; U_2 \right]_d = \left[ U_2 \right]_d \times \left[ U_1 \right]_d)</td>
<td>({\text{skip}}_d(\rho) = \rho)</td>
</tr>
<tr>
<td>(\left[ G_1 q \right]_d = \begin{cases} \text{apply}<em>1(G_1, q, d) &amp; \text{well-typed} \ 0</em>{2^d} &amp; \text{otherwise} \end{cases})</td>
<td>({P_1; P_2}_d(\rho) = ({P_2}_d \circ {P_1}_d)(\rho))</td>
</tr>
<tr>
<td>(\left[ G_2 q_1 q_2 \right]_d = \begin{cases} \text{apply}<em>2(G_2, q_1, q_2, d) &amp; \text{well-typed} \ 0</em>{2^d} &amp; \text{otherwise} \end{cases})</td>
<td>({U}_d(\rho) = \left[ U \right]_d \times \rho \times \left[ U \right]_d^+)</td>
</tr>
<tr>
<td></td>
<td>({\text{meas } q P_1 P_2}_d(\rho) = {P_2}_d(</td>
</tr>
<tr>
<td></td>
<td>(+ {P_1}_d(</td>
</tr>
</tbody>
</table>
VOQC Transformations

- Unitary optimizations - inspired by Nam et al.\textsuperscript{1}
  - Gate propagation and cancellation
  - Rotation merging

- Non-unitary optimizations
  - Classical state propagation
  - Removing z-rotations before measurement

- Circuit mapping
  - Naive mapping for arbitrary connected graph

\textsuperscript{1}Nam et al. *Automated Optimization of Large Quantum Circuits with Continuous Parameters*. 2018.
Example: X/Z Propagation

- Simplified code:

```plaintext
let propagate_X q lst = match lst with
  | []               → [X q]
  | X q :: t         → t
  | H q :: t         → H q ; propagate_Z q t
  | Rz q :: t        → Rz† q ; propagate_X q t
  ...
```
Example: X/Z Propagation

---

- **Simplified code:**

```haskell
let propagate_X q lst = match lst with
  | []          → [X q]
  | X q :: t    → t
  | H q :: t    → H q ; propagate_Z q t
  | Rz q :: t   → Rz† q ; propagate_X q t
  ...```

---
Example: X/Z Propagation

- Simplified code:

```haskell
let propagate_X q lst = match lst with
  | []       → [X q]  B
  | X q :: t  → t
  | H q :: t  → H q ; propagate_Z q t
  | Rz q :: t → Rz† q ; propagate_X q t
  ...
```
Proof Overview

• For a transformation \( T \), we want to prove that
  \[ \forall P, \llbracket T(P) \rrbracket = \llbracket P \rrbracket \] (up to a global phase)
  
  • For complex optimizations, rather than proving this equality directly we may prove that \( T(P) \) and \( P \) have the same output on every basis state

  • For circuit mapping we also prove that for every \( P, T(P) \) respects the provided connectivity constraints

  • Proofs proceed by induction on \( P \)
Example: X/Z Propagation

- We will want to prove that for any instruction list `lst`, `(propagate_X q lst)` has the same denotation as `(X q ; lst)`
  - `propagate_X q lst ≡ X q ; lst`

- Proof proceeds by induction on `lst`

```latex
let propagate_X q lst = match lst with
  | [] → [X q]
  | X q :: t → t
  | H q :: t → H q ; propagate_Z q t
  | Rz q :: t → Rz^† q ; propagate_X q t
  ...
```
Example: X/Z Propagation

- We will want to prove that for any instruction list lst, (propagate_X q lst) has the same denotation as (X q ; lst)
  - propagate_X q lst ≡ X q ; lst

- Proof proceeds by induction on lst

```plaintext
let propagate_X q lst = match lst with
  | []     → [X q]  propagate_X q [] →[X q] ≡ X q ; []
  | X q :: t → t
  | H q :: t → H q ; propagate_Z q t
  | Rz q :: t → Rz† q ; propagate_X q t
  ...
```
Example: X/Z Propagation

- We will want to prove that for any instruction list lst, 
  (propagate_X q lst) has the same denotation as (X q ; lst)
- \( \text{propagate}_X q \text{ lst} \equiv X q ; \text{ lst} \)

- Proof proceeds by induction on lst

```latex
let \text{propagate}_X q \text{ lst} = \text{match lst with}
| [] \rightarrow [X q]
| X q :: t \rightarrow t \quad \text{propagate}_X q (X q :: t) \rightarrow t \equiv X q ; X q ; t
| H q :: t \rightarrow H q ; \text{propagate}_Z q t
| Rz q :: t \rightarrow Rz^\dagger q ; \text{propagate}_X q t
...```


Example: X/Z Propagation

- We will want to prove that for any instruction list lst, (propagate_X q lst) has the same denotation as (X q ; lst)
  - propagate_X q lst ≡ X q ; lst

- Proof proceeds by induction on lst

let propagate_X q lst = match lst with
  | [] → [X q]
  | X q :: t → t
  | H q :: t → H q ; propagate_Z q t
  | Rz q :: t → Rz† q ; propagate_X q t
  ...
Verifying Matrix Equivalences

- Proving matrix equivalences in Coq is tedious
- E.g. \( X \ n; \ CNOT \ m \ n \equiv \ CNOT \ m \ n; \ X \ n \)

\[
apply_1(X, n, d) \times apply_2(CNOT, m, n, d) = apply_2(CNOT, m, n, d) \times apply_1(X, n, d).
\]

\( Rz \)
Verifying Matrix Equivalences

- Proving matrix equivalences in Coq is tedious

- E.g. $X n; \text{CNOT} m n \equiv \text{CNOT} m n; X n$

\[ \text{apply}_1(X, n, d) \times \text{apply}_2(\text{CNOT}, m, n, d) = \text{apply}_2(\text{CNOT}, m, n, d) \times \text{apply}_1(X, n, d). \]

\[ \text{apply}_1(X, n, d) = I_{2^n} \otimes \sigma_x \otimes I_{2^q} \]

\[ \text{apply}_2(\text{CNOT}, m, n, d) = I_{2^m} \otimes |1\rangle\langle 1| \otimes I_{2^p} \otimes \sigma_x \otimes I_{2^q} + I_{2^m} \otimes |0\rangle\langle 0| \otimes I_{2^p} \otimes I_2 \otimes I_{2^q} \]
Verifying Matrix Equivalences

- Proving matrix equivalences in Coq is tedious

E.g. \( X n; \; CNOT \; m \; n \equiv CNOT \; m \; n; \; X \; n \)

\[
\text{apply}_1(X, n, d) \times \text{apply}_2(CNOT, m, n, d) = \text{apply}_2(CNOT, m, n, d) \times \text{apply}_1(X, n, d).
\]

\[
I_{2m} \otimes |1\rangle\langle 1| \otimes I_{2p} \otimes I_2 \otimes I_{2q} + I_{2m} \otimes |0\rangle\langle 0| \otimes I_{2p} \otimes \sigma_x \otimes I_{2q}.
\]
Verifying Matrix Equivalences

- Proving matrix equivalences in Coq is tedious

- E.g. \( X \ n; \ \text{CNOT} \ m \ n \equiv \text{CNOT} \ m \ n; \ X \ n \)

\[
\text{apply}_1(X, n, d) \times \text{apply}_2(\text{CNOT}, m, n, d) = \text{apply}_2(\text{CNOT}, m, n, d) \times \text{apply}_1(X, n, d).
\]

- Fortunately, this is mostly automated in our development
Experiment

• Evaluated unitary optimizations

• Compared against Qiskit, tket, PyZX, Nam et al. and Amy et al.²

• Benchmark of 29 programs from Amy et al., ranging from 45 to 61629 gates and 5 to 192 qubits

• Considered reduction in total gate count and T-gate count

## Results

- **Average gate count reduction**

<table>
<thead>
<tr>
<th>Nam et al.</th>
<th>Qiskit</th>
<th>ℏket</th>
<th>VOQC ✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>26.5%</td>
<td>10.7%</td>
<td>11.2%</td>
<td>18.4%</td>
</tr>
</tbody>
</table>

- **Average T-count reduction**

<table>
<thead>
<tr>
<th>Amy et al.</th>
<th>Nam et al.</th>
<th>PyZX</th>
<th>VOQC ✓</th>
</tr>
</thead>
<tbody>
<tr>
<td>40.9%</td>
<td>41.0%</td>
<td>42.6%</td>
<td>39.4%</td>
</tr>
</tbody>
</table>
Is the translation between industry IRs and SQIR correct?

High-level Language
E.g. QWIRE, Quipper, Q#

Unoptimized IR
E.g. OpenQASM, Quil

Optimized IR
E.g. OpenQASM, Quil

Hardware Instructions

Unoptimized SQIR

Optimized SQIR

VOQC
Is the translation between industry IRs and SQIR correct?

- **High-level Language**
  - E.g. QWIRE, Quipper, Q#

- **Unoptimized IR**
  - E.g. OpenQASM, Quil

- **Optimized IR**
  - E.g. OpenQASM, Quil

- **Unoptimized SQIR**

- **Optimized SQIR**

- **Hardware Instructions**

- **VOQC**

The diagram shows the flow of translation from high-level languages to hardware instructions, with unoptimized and optimized IRs and SQIRs involved.
We verify translation between OpenQASM and SQIR

A feature-complete parser for OpenQASM

Translation between unitary fragments of the two languages

A denotational semantics for unitary OpenQASM

Semantic preservation property of translation
Unitary SQIR and OpenQASM
have fairly similar abstract syntax

SQIR

\[ U ::= U_1; U_2 | G \ q | G \ q_1 \ q_2 \]

\[ G ::= H | CNOT \]

Qubits are indices into a global register

OpenQASM

Expression \[ E ::= x | x[i] \]

Unitary Statement \[ U ::= H(E) | C(X(E_1, E_2) | E(E_1, \ldots, E_n) | U_1; U_2 \]

Command \[ C ::= \text{qreg } x[i] \ | \ \text{gate } x(x_1, \ldots, x_n) \{ U \} | U | C_1; C_2 \]

OpenQASM is a larger language
that supports declaring qubit registers
and user-defined gates
Unitary SQIR and OpenQASM have fairly similar abstract syntax

**SQIR**

\[ U ::= U_1; U_2 | G q | G q_1 q_2 \]
\[ G ::= H | CNOT \]

Qubits are indices into a global register

**OpenQASM**

**Expression**

\[ E ::= x | x[i] \]

**Unitary Statement**

\[ U ::= H(E) | CX(E_1, E_2) | E(E_1, \ldots, E_n) | U_1; U_2 \]

**Command**

\[ C ::= qreg x[i] | gate x(x_1, \ldots, x_n) \{ U \} | U | C_1; C_2 \]

OpenQASM is a larger language that supports declaring qubit registers and user-defined gates
Unitary SQIR and OpenQASM have fairly similar abstract syntax

**SQIR**

\[
U ::= U_1; U_2 \mid G q \mid G q_1 q_2 \\
G ::= H \mid \text{CNOT}
\]

Qubits are indices into a global register

**OpenQASM**

Expression \( E ::= \) [\( x \mid x[i] \)]

Unitary Statement \( U ::= \) [\( H(E) \mid \text{CX}(E_1, E_2) \mid E(E_1, \ldots, E_n) \)\( \mid U_1; U_2 \)]

Command \( C ::= \) qreg \( x[i] \) \( \mid \) gate \( x(x_1, \ldots, x_n) \) \{ \( U \) \} \( \mid \) U \( \mid C_1; C_2 \)

OpenQASM is a larger language that supports declaring qubit registers and user-defined gates
Unitary SQIR and OpenQASM have fairly similar abstract syntax

SQIR

\[ U ::= U_1; U_2 \mid G \select{q \in q_1 \ q_2} \]
\[ G ::= H \mid \text{CNOT} \]

Qubits are indices into a global register

OpenQASM\[^a\]

Expression \[ E ::= x \mid x[i] \]
Unitary Statement \[ U ::= H(E) \mid \text{CX}(E_1, E_2) \mid E(E_1, \ldots, E_n) \mid U_1; U_2 \]
Command \[ C ::= \text{qreg } x[i] \mid \text{gate } x(x_1, \ldots, x_n) \{ U \} \mid U \mid C_1; C_2 \]

OpenQASM is a larger language that supports declaring qubit registers and user-defined gates

Only shown a sample gate set of Hadamard (H) and controlled NOT (\text{CNOT} or \text{CX})
Unitary SQIR and OpenQASM have fairly similar abstract syntax

**SQIR**

\[
U ::= U_1; U_2 \mid G q \mid G q_1 q_2 \\
G ::= H \mid CNOT
\]

Qubits are indices into a global register

**OpenQASM\textsuperscript{a}**

**Expression**

\[
E ::= x \mid x[i]
\]

**Unitary Statement**

\[
U ::= H(E) \mid CX(E_1, E_2) \mid E(E_1, \ldots, E_n) \mid U_1; U_2
\]

**Command**

\[
C ::= \text{qreg } x[i] \mid \text{gate } x(x_1, \ldots, x_n) \{ U \} \mid U \mid C_1; C_2
\]

OpenQASM is a larger language that supports declaring qubit registers and user-defined gates

\textsuperscript{a}Amy M. Sized Types for Low-Level Quantum Metaprogramming. Reversible Computation. RC 2019.
Denotational semantics of SQIR and OpenQASM correspond

**SQIR**

\[ [U_1; U_2]_d = [U_2]_d \times [U_1]_d \]

\[ [G_1 q]_d = \begin{cases} \text{apply}_1(G_1, q, d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases} \]

\[ [G_2 q_1 q_2]_d = \begin{cases} \text{apply}_2(G_2, q_1, q_2, d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases} \]

A SQIR unitary program denotes a \(2^d \times 2^d\) unitary matrix

**OpenQASM**

Value \(V\) = Location, \(l + \text{Loc. Array}, (l_j, \ldots, l_k) + \text{Unitary Gate}, \lambda(x_1, \ldots, x_n).U\)

Environment \(\sigma\) = Identifier \(\rightarrow\) Value

Quantum State \(|\psi\rangle\) = \(2^d\)-dimension complex vector

Expressions need environment and return bound values

Unitary statements modify quantum state

Commands modify both environment and quantum state

\(|-|)_E : E \times \sigma \rightarrow V

\(|-|)_U : U \times \sigma \times |\psi\rangle \rightarrow |\psi'\rangle

\(|-|)_C : C \times \sigma \times |\psi\rangle \rightarrow \sigma' \times |\psi'\rangle

Details elided
Semantic preservation properties are maintained during translation

\[ f : \mathcal{L}(\text{sQIR}) \rightarrow \mathcal{L}(\text{OpenQASM}) \]
\[ g : \mathcal{L}(\text{OpenQASM}) \rightarrow \mathcal{L}(\text{sQIR}) \]

For all valid programs in SQIR of dimension \( d \), their denotation is equivalent to the denotation of their translation, and vice versa.

\[ \forall x \in \mathcal{L}(\text{sQIR}), \ [x]_d = |f(x)| \]
\[ \forall y \in \mathcal{L}(\text{OpenQASM}), \ |y| = [g(y)]_d \]

Further, converting from SQIR to OpenQASM and back recovers the original program.

\[ \forall x \in \mathcal{L}(\text{sQIR}), \ g(f(x)) = x \]
Semantic preservation properties are maintained during translation

\[ f : \mathcal{L}(\text{SQIR}) \rightarrow \mathcal{L}(\text{OpenQASM}) \]
\[ g : \mathcal{L}(\text{OpenQASM}) \rightarrow \mathcal{L}(\text{SQIR}) \]

For all valid programs in SQIR of dimension \( d \), their denotation is equivalent to the denotation of their translation, and vice versa.

\[ \forall x \in \mathcal{L}(\text{SQIR}), \quad [x]_d = |f(x)| \]
\[ \forall y \in \mathcal{L}(\text{OpenQASM}), \quad |y| = [g(y)]_d \]

Further, converting from SQIR to OpenQASM and back recovers the original program.

\[ \forall x \in \mathcal{L}(\text{SQIR}), \quad g(f(x)) = x \]

But the reverse direction does not hold.
OpenQASM parser written in OCaml programming language

Conforms to OpenQASM spec\(^b\)

Uses OCamllex and Menhir parser generator

Available now as an OCaml library on OPAM package repository:

```
  opam install openQASM
```

\(^b\)Cross et al. Open Quantum Assembly Language. arXiv:1707.03429
Ongoing and Future Work

• We’re always looking for more transformations to verify
  • Optimization from other compilers (incl. error aware)
  • More sophisticated circuit mapping
  • Compilation of classical circuits

• Performance improvements; evaluations on larger sets of benchmarks

• Larger verified toolchain
  • Translation and verification of non-unitary fragments
  • Validate our OpenQASM parser using Menhir’s Coq backend
  • Verify translation from high-level languages such as QWIRE
Conclusions

• We have developed a compiler **VOQC** and an IR **SQIR**, both implemented and verified in Coq
  • Performance is comparable to state-of-the-art compilers

• We have also taken steps to ease interoperability with industry toolchains with translation from and to OpenQASM

• Lots of ongoing work, let us know if you’re interested!

• Code:
  • github.com/inQWIRE/SQIR
  • github.com/inQWIRE/openqasm-parser

• Draft of VOQC paper: cs.umd.edu/~mwh/papers/voqc-draft.pdf