A Verified Optimizer for Quantum Circuits

KESHA HIETALA, University of Maryland, USA
ROBERT RAND, University of Chicago, USA
LIYI LI, University of Maryland, USA
SHIH-HAN HUNG, University of Texas at Austin, USA
XIAODI WU, University of Maryland, USA
MICHAEL HICKS*, University of Maryland, USA and Amazon, USA

We present VOQC, the first fully *verified optimizer for quantum circuits*, written using the Coq proof assistant. Quantum circuits are expressed as programs in a simple, low-level language called SQIR, a *small quantum intermediate representation*, which is deeply embedded in Coq. Optimizations and other transformations are expressed as Coq functions, which are proved correct with respect to a semantics of SQIR programs. SQIR programs denote complex-valued matrices, as is standard in quantum computation, but we treat matrices symbolically in order to reason about programs that use an arbitrary number of quantum bits. SQIR's careful design and our provided automation make it possible to write and verify a broad range of optimizations in VOQC, including full-circuit transformations from cutting-edge optimizers.

CCS Concepts: • **Hardware** → *Quantum computation*; *Circuit optimization*; • **Software and its engineering** → **Formal software verification**.

Additional Key Words and Phrases: Formal Verification, Quantum Computing, Circuit Optimization, Certified Compilation

ACM Reference Format:
Kesha Hietala, Robert Rand, Liyi Li, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2022. A Verified Optimizer for Quantum Circuits. 1, 1 (July 2022), 35 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Programming quantum computers will be challenging, at least in the near term. Qubits will be scarce and gate pipelines will need to be short to prevent decoherence. Fortunately, optimizing compilers can transform a source algorithm to work with fewer resources. Where compilers fall short, programmers can optimize their algorithms by hand.

Of course, both compiler and by-hand optimizations will inevitably have bugs. As evidence of the former, Kissinger and van de Wetering [26] discovered mistakes in the optimized outputs produced by the circuit optimizer of Nam et al. [33], and Nam et al. themselves found that the optimization library they compared against (Amy et al. [3]) sometimes

[*] Work done prior to starting at Amazon

Authors' addresses: Kesha Hietala, University of Maryland, USA, kesha@cs.umd.edu; Robert Rand, University of Chicago, USA, rand@uchicago.edu; Liyi Li, University of Maryland, USA, liyili2@umd.edu; Shih-Han Hung, University of Texas at Austin, USA, shung@cs.utexas.edu; Xiaodi Wu, University of Maryland, USA, xwu@cs.umd.edu; Michael Hicks, University of Maryland, USA and Amazon, USA, mwh@cs.umd.edu.

Manuscript submitted to ACM

produced incorrect results. Making mistakes when optimizing by hand is also to be expected: quantum computing can be frustratingly unintuitive.

Unfortunately, the very factors that motivate optimizing quantum compilers make it difficult to test their correctness. Comparing runs of a source program to those of its optimized version is often impractical due to the indeterminacy of typical quantum algorithms and the substantial expense involved in executing or simulating them. Indeed, resources may be too scarce, or the qubit connectivity too constrained, to run the program without optimization!

An appealing solution to this problem is to apply rigorous *formal methods* to prove that an optimization or algorithm always does what it is intended to do. For example, CompCert [28] is a compiler for C programs that is written and proved correct using the Coq proof assistant [11]. CompCert includes sophisticated optimizations whose proofs of correctness are validated by Coq's type checker.

In this paper, we apply CompCert's approach to the quantum setting. We present VOQC (pronounced "vox"), a *verified optimizer for quantum circuits*. VOQC takes as input a quantum program written in a language we call SQIR ("squire"). SQIR is designed to be a *small quantum intermediate representation*, but it is suitable for source-level programming too: It is not very different from languages such as Quil [45] or OpenQASM 2.0 [12], which describe quantum programs as circuits. SQIR is deeply embedded in Coq, similar to how Quil is embedded in Python via PyQuil [40], allowing us to write sophisticated quantum programs. VOQC applies a series of optimizations to SQIR programs, ultimately producing a result that is compatible with a specified quantum architecture. For added convenience, VOQC provides translators between SQIR and OpenQASM 2.0.

At the core of VOQC is a framework for writing transformations of SQIR programs and verifying their correctness (Section 4). To ensure that the framework is suitably expressive, we have used it to develop verified versions of a variety of optimizations (Section 5). Many are based on those used in recent, state-of-the-art circuit optimizer developed by Nam et al. [33]. We abstract these optimizations into a couple of different classes, and provide library functions, lemmas, and automation to simplify their construction and proof. We have also verified circuit mapping routines that transform SQIR programs to satisfy constraints on how qubits may interact on a specified target architecture (Section 6).

We evaluated the quality of the optimizations we verified in VOQC, and by extension the quality of our framework, by measuring how well it optimizes a set of benchmark programs, compared to several other optimizing compilers (Section 7). The results are encouraging. On a benchmark of 35 circuit programs developed by Amy and Gheorghiu [2] we find that VOQC reduces total gate count on average by 28.5% compared to 14.4% for IBM's Qiskit compiler [37], and 18.5% for CQC's t|ket⟩ [43]. On the same benchmarks, VOQC reduces T-gate count (an important measure when considering fault tolerance) on average by 43.1% compared to 45.4% by Amy and Gheorghiu [2] and 47.1% by the PyZX optimizer [26], although VOQC outperforms both in terms of total gate count reduction. Experiments on an even larger benchmark suite (detailed in Appendix A) yield similar results. In sum, VOQC is expressive enough to verify a range of useful optimizations, yielding performance competitive with leading unverified compilers.

VOQC is the first fully verified optimizer for general quantum programs (Section 8). Amy et al. [4] developed a verified optimizing compiler from source Boolean expressions to reversible circuits and Fagan and Duncan [15] verified an optimizer for ZX-diagrams representing Clifford circuits; however, neither of these tools handle general quantum programs. Tao et al. [48] developed Giallar, which uses symbolic execution and SMT solving to automatically verify circuit transformations in the Qiskit compiler; however, Giallar is limited to verifying correct application of local equivalences and does not provide a way to describe general quantum states, which limits the types of optimizations that it can reason about. Smith and Thornton [44] presented a compiler with built-in translation validation via QMDD equivalence checking [32]. However, QMDDs represent quantum state concretely, which means that the validation time will increase exponentially with the number of qubits in the compiled program. Burgholzer et al. [6] improved on this model by taking advantage of the fact that the identity matrix (which should be the result of composing a circuit with its optimized adjoint) can be efficiently represented using a decision diagram [7], allowing them to perform equivalence checking on circuits that use tens of thousands of operations. However, like any other translation validation technique, this approach adds extra compile time overhead, and has so far only been used to validate applications of local rewrite rules. In contrast to these, sqir denotes quantum programs as *symbolical complex-valued matrices*, which allows us to reason about arbitrary quantum computation and verify once-and-for-all interesting, non-local optimizations, independently of the number of qubits in the optimized program.

Our work[1] on voqc and sqir are steps toward a broader goal of developing a full-scale verified compiler toolchain. Next steps include developing certified transformations from higher-level quantum languages to sqir and implementing optimizations with different objectives, e.g., that aim to reduce the probability that a result is corrupted by quantum noise. All code is freely available online.[2]

2 OVERVIEW

We begin with a brief background on quantum programs, focusing on the challenges related to formal verification. We then provide an overview of voqc and sqir, summarizing how they address these challenges.

2.1 Preliminaries

Quantum programs operate over *quantum states*, which consist of one or more *quantum bits* (a.k.a. *qubits*). A single qubit is represented as a vector of complex numbers $\langle \alpha, \beta \rangle$ such that $|\alpha|^2 + |\beta|^2 = 1$. The vector $\langle 1, 0 \rangle$ represents the state $|0\rangle$ while vector $\langle 0, 1 \rangle$ represents the state $|1\rangle$. A state written $|\psi\rangle$ is called a *ket*, following Dirac's notation. We say a qubit is in a *superposition* of $|0\rangle$ and $|1\rangle$ when both α and β are non-zero. Just as Schrodinger's cat is both dead and alive until the box is opened, a qubit is only in superposition until it is *measured*, at which point the outcome will be 0 with probability $|\alpha|^2$ and 1 with probability $|\beta|^2$. Measurement is not passive: it has the effect of collapsing the state to match the measured outcome, i.e., either $|0\rangle$ or $|1\rangle$. As a result, all subsequent measurements return the same answer.

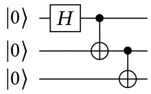
Operators on quantum states are linear mappings. These mappings can be expressed as matrices, and their application to a state expressed as matrix multiplication. For example, the *Hadamard* operator H is expressed as a matrix $\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$. Applying H to state $|0\rangle$ yields state $\langle \frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}} \rangle$, also written as $|+\rangle$. Many quantum operators are not only linear, they are also *unitary*—the conjugate transpose (or adjoint) of their matrix is its inverse. This ensures that multiplying a qubit by the operator preserves the qubit's sum of norms squared. Since a Hadamard is its own adjoint, it is also its own inverse: hence $H\,|+\rangle = |0\rangle$.

A quantum state with n qubits is represented as vector of length 2^n. For example, a 2-qubit state is represented as a vector $\langle \alpha, \beta, \gamma, \delta \rangle$ where each component corresponds to (the square root of) the probability of measuring $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$, respectively. Because of the exponential size of the complex quantum state space, it is not possible to simulate a 100-qubit quantum computer using even the most powerful classical computer!

[1] sqir and voqc were first presented in Hietala et al. [19]. This paper is an expanded version of that paper with new content including: (a) support for additional gate sets, (b) expanded circuit mapping and mapping validation, (c) extended evaluation results, and (d) improvements to presentation.

[2] Software links:

- The sqir and voqc Coq definitions and proofs are available at https://github.com/inQWIRE/SQIR.
- The voqc OCaml library is available at https://github.com/inQWIRE/mlvoqc and can be installed with "opam install voqc".
- The voqc Python bindings and tutorials are available at https://github.com/inQWIRE/pyvoqc.
- The voqc benchmarks and benchmarking scripts are available at https://github.com/inQWIRE/VOQC-benchmarks.



(a) Quantum Circuit

```
H 0
CNOT 0 1
CNOT 1 2
```

(b) Quil

```
def ghz_state(qubits):
  program = Program()
  program += H(qubits[0])
  for q1,q2 in zip(qubits, qubits[1:]):
    program += CNOT(q1, q2)
  return program
```

(c) PyQuil (arbitrary number of qubits)

Fig. 1. Example quantum program: GHZ state preparation

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

n-qubit operators are represented as $2^n \times 2^n$ matrices. For example, the CX (or $CNOT$) operator over two qubits is expressed as the matrix shown at the right. It expresses a *controlled not* operation—if the first qubit (called the *control*) is $|0\rangle$ then both qubits are mapped to themselves, but if the first qubit is $|1\rangle$ then the second qubit (called the *target*) is negated, e.g., $CX\,|00\rangle = |00\rangle$ while $CX\,|10\rangle = |11\rangle$.

n-qubit operators can be used to create *entanglement*, which is a situation where two qubits cannot be described independently. For example, while the vector $\langle 1, 0, 0, 0\rangle$ can be written as $\langle 1, 0\rangle \otimes \langle 1, 0\rangle$ where $\otimes$ is the tensor product, the state $\langle \frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}}\rangle$ cannot be similarly decomposed. We say that $\langle \frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}}\rangle$ is an entangled state.

2.2 Quantum Circuits

Quantum programs are typically expressed as circuits, as shown in Figure 1(a). In these circuits, each horizontal wire represents a *qubit* and boxes on these wires indicate quantum operators, or *gates*. Gates can either be unitary operators (e.g., H, CX) or non-unitary ones (e.g., measurement). In software, quantum circuit programs are often represented using lists of instructions that describe the different gate applications. For example, Figure 1(b) is the Quil [45] representation of the circuit in Figure 1(a).

In the *QRAM model*, quantum computers are used as co-processors to classical computers. The classical computer generates descriptions of circuits to send to the quantum computer and then processes the measurement results. High-level quantum programming languages are designed to follow this model. For example, Figure 1(c) shows a program in PyQuil [40], a quantum programming framework embedded in Python. The `ghz_state` function takes an array `qubits` and constructs a circuit that prepares the Greenberger-Horne-Zeilinger (GHZ) state [16], which is an n-qubit entangled quantum state of the form

$$|\mathrm{GHZ}^n\rangle = \frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle^{\otimes n}).$$

Calling `ghz_state([0,1,2])` returns the Quil program in Figure 1(b), which produces the quantum state $\frac{1}{\sqrt{2}}(|000\rangle+|111\rangle)$. The high-level language may provide facilities to optimize constructed circuits, e.g., to reduce gate count, circuit depth, and qubit usage. It may also perform transformations to account for hardware-specific details like the number of qubits, available gates, or connectivity between physical qubits.

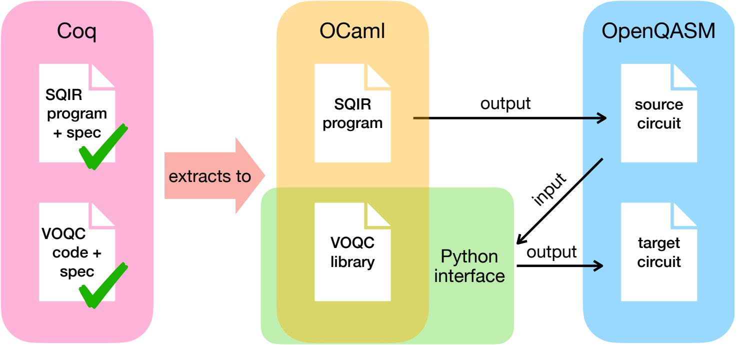


Fig. 2. The voqc architecture.

2.3 sqir: A Small Quantum Intermediate Representation Supporting Verification

sqir is a simple circuit-oriented language deeply embedded in the Coq proof assistant in a manner similar to how Quil is embedded in Python via PyQuil. We use sqir's host language, Coq, to define the syntax and semantics of sqir programs and to express properties about quantum states. We developed a library of lemmas and tactic-based automation to assist in writing proofs about quantum programs; such proofs make heavy use of complex numbers and linear algebra. These proofs are aided by isolating sqir's *unitary core* from primitives for measurement, which require consideration of probability distributions of outcomes (represented as *density matrices*); this means that (sub-)programs that lack measurement can have simpler proofs. Either way, in sqir we perform reasoning *symbolically*. For example, we can prove that every circuit generated by the sqir-equivalent of `ghz_state` produces the expected state $|\mathrm{GHZ}^n\rangle$ when applied to input lists of length n, *for any* n. We present sqir's syntax and semantics along with an example program and verified property of correctness in Section 3; for more details see Hietala et al. [18].

2.4 voqc: A Verified Optimizer for Quantum Circuits

While sqir is suitable for proving correctness properties about source programs like `ghz_state`, its primary use has been as the intermediate representation of voqc, our verified optimizer for quantum circuits, and the signature achievement of this paper. An *optimizer* is a function from programs to programs, with the intention that the output program has the same semantics as the input. In voqc, we prove this is always the case: a voqc optimization f is a Coq function over sqir circuit C, and we prove that the semantics of the input circuit C is always equivalent to the semantics of the output $f(C)$.

The voqc approach stands in contrast to prior work that relies on translation validation [6, 26, 44], which may fail to identify latent bugs in the optimizer, while adding compile-time overhead. By proving correctness with respect to an explicit semantics for input/output programs (i.e., that of sqir), voqc optimizations are flexible in their expression. Prior work has been limited to peephole optimizations [48], leaving highly effective, full-circuit optimizations we have proved correct in voqc out of reach.

The structure of voqc is summarized in Figure 2. sqir programs and voqc transformations are defined and formally verified in Coq (left). We use Coq's standard code extraction mechanism [23] to extract sqir programs and voqc transformations into OCaml (middle). sqir programs are extracted to OCaml code that generates OpenQASM 2.0 [12], a standard representation for quantum circuits, and voqc is extracted to a standalone OCaml library that takes as input

OpenQASM circuits (right). Since a number of quantum programming frameworks, including Qiskit [37], t|ket⟩ [43], Quil [41], Project Q [46] and Cirq [13], can output OpenQASM, this allows us to run VOQC on a variety of generated circuits, without requiring the user to program in OCaml or Coq. We also provide a Python wrapper around our OCaml library to make VOQC compatible with the many Python-based frameworks for compiling quantum programs (e.g., Qiskit, t|ket⟩, Quil, Cirq).

VOQC is implemented in about 15K lines of Coq, with roughly 6K lines for general-purpose SQIR program manipulation, 4K lines for program optimizations, and 5K lines for circuit mapping.

3 SQIR: A SMALL QUANTUM INTERMEDIATE REPRESENTATION

Here we present the syntax and semantics of SQIR, a *small quantum intermediate representation*. The SQIR language is composed of two parts: a core language of unitary operators and a full language that incorporates measurement. This paper only uses the former; see Hietala et al. [18, §3.3] for a discussion of the latter.

The semantics of a unitary SQIR program is expressed directly as a matrix, in contrast to the full SQIR, which treats programs as functions over density matrices. This matrix semantics greatly simplifies proofs, both of the correctness of unitary optimizations (the focus of VOQC) and of source programs, many of which are essentially unitary (measurement is the very last step).

3.1 Syntax

A unitary SQIR program U is a sequence of applications of gates G to qubits q.

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

Qubits are referred to by natural numbers that index into a *global register* of quantum bits. Each SQIR program is parameterized by a set of unitary one- and two-qubit gates (from which G is drawn) and the dimension of the global register (i.e., the number of available qubits). In Coq, a unitary SQIR program `U` has type `ucom g n`, where `g` identifies the gate set and `n` is the size of the global register.

As an example, consider the program to the right, which is equivalent to PyQuil's `ghz_state` from Figure 1(c). The Coq function `ghz` recursively constructs a SQIR program, in this case a Coq value of type `ucom base n`, which uses gate set `base`. When run, this program prepares the GHZ state. When n is 0, `ghz` produces a SQIR program that is just the identity gate I applied to qubit 0. When n is 1, the result is the Hadamard gate H applied to qubit 0. When n is greater than 1, `ghz` constructs the program $U_1; U_2$, where U_1 is the `ghz` circuit on `n'` (i.e., $n-1$) qubits, and U_2 is the appropriate CX gate. The result of `ghz 3` is equivalent to the circuit shown in Figure 1(a).

```
Fixpoint ghz (n : ℕ) : ucom base n :=
  match n with
  | 0 ⇒ I 0
  | 1 ⇒ H 0
  | S n' ⇒ ghz n'; CX (n'−1) n'
  end.
```

3.2 Semantics

Suppose that M_1 and M_2 are the matrices corresponding to unitary gates U_1 and U_2, which we want to apply to a quantum state vector $|\psi\rangle$. Matrix multiplication is associative, so $M_2(M_1\,|\psi\rangle)$ is equivalent to $(M_2M_1)\,|\psi\rangle$. Moreover, multiplying two unitary matrices yields a unitary matrix. As such, the semantics of SQIR program $U_1;\ U_2$ is naturally described by the unitary matrix M_2M_1.

$$
\begin{aligned}
[\![U_1;\ U_2]\!]_d &= [\![U_2]\!]_d \times [\![U_1]\!]_d \\
[\![G_1\ q]\!]_d &= \begin{cases} apply_1(G_1,\ q,\ d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases} \\
[\![G_2\ q_1\ q_2]\!]_d &= \begin{cases} apply_2(G_2,\ q_1,\ q_2,\ d) & \text{well-typed} \\ 0_{2^d} & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 3. Semantics of unitary sqir programs, assuming a global register of dimension d. The $apply_k$ function maps a gate name to its corresponding unitary matrix and extends the intended operation to the given dimension by applying an identity operation on every other qubit in the system.

This semantics is shown in Figure 3. If a program is not *well-typed* its denotation is the zero matrix (of size $2^d \times 2^d$). A program U is well-typed if every gate application is *valid*, meaning that its index arguments are within the bounds of the global register, and no index is repeated. The latter requirement enforces linearity and thereby quantum mechanics' *no-cloning theorem*, which says that it is impossible to create a copy of an arbitrary quantum state.

Otherwise, the program's denotation is the composition of the matrices corresponding to its unitary gates. The only wrinkle is that a full program consists of many gates, each operating on only 1 or 2 of the total qubits; thus, a gate application's matrix needs to apply the identity operation to the qubits not being operated on. This is what $apply_1$ and $apply_2$ do. For example, $apply_1(G_u,\ q,\ d) = I_{2^q} \otimes u \otimes I_{2^{(d-q-1)}}$ where u is the matrix interpretation of the gate G_u and I_k is the $k \times k$ identity matrix. The $apply_2$ function requires us to decompose the two-qubit unitary into a sum of tensor products: For instance, CX can be written as $|0\rangle\langle 0| \otimes I_2 + |1\rangle\langle 1| \otimes \sigma_x$ where $\sigma_x = \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$. We then have

$$apply_2(CX,\ q_1,\ q_2,\ d) = I_{2^{q_1}} \otimes |0\rangle\langle 0| \otimes I_{2^r} \otimes I_2 \otimes I_{2^s} + I_{2^{q_1}} \otimes |1\rangle\langle 1| \otimes I_{2^r} \otimes \sigma_x \otimes I_{2^s}$$

where $r = q_2 - q_1 - 1$ and $s = d - q_2 - 1$, assuming $q_1 < q_2$.

In our development we define the semantics of sqir programs over gate set $G \in \{R_{\theta,\phi,\lambda},\ CX\}$ where $R_{\theta,\phi,\lambda}$ is a general single-qubit rotation parameterized by three real-valued rotation angles and CX is the standard two-qubit controlled-not gate. This is our *base set* of gates. It is the same as the underlying set used by OpenQASM [12] and is *universal*, meaning that it can approximate any unitary operation to within arbitrary error. The matrix interpretation of the single-qubit $R_{\theta,\phi,\lambda}$ gate is

$$\begin{pmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i(\phi+\lambda)}\cos(\theta/2) \end{pmatrix}$$

and the matrix interpretation of the CX gate is given in Section 2.1.

Common single-qubit gates can be defined in terms of $R_{\theta,\phi,\lambda}$. For example, the two single-qubit gates used in our GHZ example—identity I and Hadamard H—are respectively defined as $R_{0,0,0}$ and $R_{\pi/2,0,\pi}$. The Pauli X ("NOT") gate is $R_{\pi,0,\pi}$ and the Pauli Z gate is $R_{0,0,\pi}$. We can also define more complex operations as sqir programs. For example, the $SWAP$ operation, which swaps two qubits, can be defined as a sequence of three CX gates.

3.3 Source-Program Proofs

sqir is part of a *general purpose* framework for reasoning about quantum programs. While this paper focuses on sqir's use in proving circuit optimizations correct (by proving sqir program transformations are semantics preserving), the framework can be used to prove correctness properties of sqir programs, too. As an illustration, we present a sqir proof of correctness for *GHZ state preparation*. We close with discussion of ongoing efforts to prove more sophisticated algorithms correct in sqir.

GHZ Proof. As an example of a proof we can carry out using sqir, we show that ghz, sqir's Greenberger-Horne-Zeilinger (GHZ) state [16] preparation circuit given in Section 3.1, correctly produces the mathematical GHZ state. The GHZ state is an n-qubit entangled quantum state of the form $\frac{1}{\sqrt{2}}(|0\rangle^{\otimes n} + |1\rangle^{\otimes n})$. This vector can be defined in Coq as shown to the right.

```
Definition GHZ (n : ℕ) : Vector (2 ^ n) :=
  match n with
  | 0    ⇒ I 1
  | S n' ⇒ $\frac{1}{\sqrt{2}} * |0\rangle^{\otimes n} + \frac{1}{\sqrt{2}} * |1\rangle^{\otimes n}$
  end.
```

Our goal is to show that for any n > 0 the circuit generated by ghz n produces the corresponding GHZ n vector when applied to $|0\rangle^{\otimes n}$:

```
Lemma ghz_correct : ∀ n : ℕ, n > 0 → $[\![\text{ghz n}]\!]_n \times |0\rangle^{\otimes n}$ = GHZ n.
```

The proof proceeds by induction on n. The $n = 0$ case is trivial as it contradicts our hypothesis. For $n = 1$ we show that H applied to $|0\rangle$ produces the $|+\rangle$ state. In the inductive step, the induction hypothesis says that the result of applying ghz n' to the input state nket n' $|0\rangle$ is the state $(\frac{1}{\sqrt{2}} * |0\rangle^{\otimes n'} + \frac{1}{\sqrt{2}} * |1\rangle^{\otimes n'}) \otimes |0\rangle$. By applying CX (n' − 1) n' to this state, we show that ghz (n' + 1) = GHZ (n' + 1).

Further Proofs. It turns out that with the right abstractions, sqir is capable of verifying a range of quantum algorithms, from Grover's search algorithm to quantum phase estimation. Most recently, sqir has been used for a proof of an end-to-end implementation of Shor's factorization algorithm [34]. All in all, the sqir development includes implementations and proofs of GHZ state preparation, superdense coding, quantum teleportation, the Deutsch-Jozsa algorithm, Simon's algorithm, Grover's algorithm, quantum phase estimation, and Shor's algorithm. As this paper's focus is voqc, we refer the interested reader to separate papers [18, 34] for detailed discussion of these source-program proofs and proof techniques.

4 VOQC: A VERIFIED FRAMEWORK FOR OPTIMIZING QUANTUM PROGRAMS

This section introduces general features of voqc's design. We discuss specific optimizations in Section 5 and circuit mapping routines in Section 6.

4.1 voqc Program Representation

To ease the implementation of and proofs about sqir program transformations, we developed a framework of supporting library functions that operate on sqir programs as lists of gate applications, rather than on the native sqir representation. The conversion code takes a sequence of gate applications in the original sqir program and *flattens* it so that a program like $(G_1\ p; G_2\ q); G_3\ r$ is represented as the Coq list $[G_1\ p;\ G_2\ q;\ G_3\ r]$. The denotation of the list representation is the denotation of its corresponding sqir program. Examples of the list operations voqc provides include:

- Finding the next gate acting on a qubit that satisfies some predicate f.
- Propagating a gate using a set of cancellation and commutation rules (see Section 5.1).
- Replacing a sub-program with an equivalent program (see Section 5.2).
- Computing the maximal matching prefix of two programs.

We verify that these functions have the intended behavior (e.g., in the last example, that the returned sub-program is indeed a prefix of both input programs).

Table 1. Gate sets used in VOQC. r is a real parameter and q is a rational parameter.

Full	Single-qubit gates:	$I,\ X,\ Y,\ Z\ ,H,\ S,\ T,\ S^\dagger,\ T^\dagger,\ R_x(r),\ R_y(r),$ $R_z(r),\ R_zQ(q), U_1(r),\ U_2(r,r),\ U_3(r,r,r)$
	Two-qubit gates:	$CX,\ CZ,\ SWAP$
	Three-qubit gates:	$CCX,\ CCZ$
RzQ	Single-qubit gates:	$X,\ H,\ R_zQ(q)$
	Two-qubit gates:	CX
IBM	Single-qubit gates:	$U_1(r),\ U_2(r,r),\ U_3(r,r,r)$
	Two-qubit gates:	CX
Mapping	Single-qubit gates:	∞
	Two-qubit gates:	$CX,\ SWAP$

4.2 Program Equivalence

The VOQC optimizer takes as input a SQIR program and attempts to reduce its total gate count by applying a series of optimizations. For each optimization, we verify that it is *semantics preserving* (or *sound*), meaning that the output program is guaranteed to be equivalent to the input program. We say that two unitary programs of dimension d are equivalent, written $U_1 \equiv U_2$, if their denotation is the same, i.e., $[\![U_1]\!]_d = [\![U_2]\!]_d$. We can then write our soundness condition for optimization function `optimize` as follows.

`Definition` sound {G} (`optimize` : ∀ {d : ℕ}, ucom G d → ucom G d) := ∀ (d : ℕ) (u : ucom G d), $[\![$`optimize` u$]\!]_d \equiv [\![$u$]\!]_d$.

This property is quantified over G, d, and u, meaning that the property holds for *any program that uses any set of gates and any number of qubits*. The optimizations in our development are defined over particular gate sets, described below, but still apply to programs that use any number of qubits. Our statements of soundness also occasionally have an additional precondition that requires program u to be well typed.

We also support two more general versions of equivalence: We say that two circuits are *equivalent up to a global phase*, written $U_1 \cong U_2$, when there exists a θ such that $[\![U_1]\!]_d = e^{i\theta}[\![U_2]\!]_d$; We say that two circuits are *equivalent up to permutation* if there exist permutation matrices P_1, P_2 such that $[\![U_1]\!]_d = P_1 \times [\![U_2]\!]_d \times P_2$.[3] Equivalence up to a global phase is useful in the quantum setting because $|\psi\rangle$ and $e^{i\theta}\,|\psi\rangle$ (for $\theta \in \mathbb{R}$) represent the same physical state. Equivalence up to permutation is useful in the context of circuit mapping (Section 6) where inserted *SWAP* gates may change the positions of qubits in the system.

4.3 Supported Gate Sets

The VOQC framework supports arbitrary gate sets; the utility functions and properties described above are all parameterized by choice of gate set. However, the program transformations in Sections 5 and 6 are defined over the particular gate sets listed in Table 1. Using a custom gate set for a transformation makes writing the transformation cleaner and simplifies the proof of soundness (typically, each gate corresponds to one case in the proof). We summarize which transformations are defined over which gate sets in Figure 4.

The *full gate set* is used for parsing and consists of a variety of standard quantum gates. It aims for completeness: Instead of having to translate a T gate in the source OpenQASM program to the semantically equivalent $U_1(\pi/4)$, we can translate it directly to T. Likewise, we can translate the three-qubit CCX gate directly to CCX, rather than

[3] A permutation matrix is a square binary matrix with a single 1 entry in each row and column and 0s elsewhere. Left-multiplying a matrix A by a permutation matrix P (i.e., PA) permutes A's rows, and right-multiplying (AP) permutes A's columns.

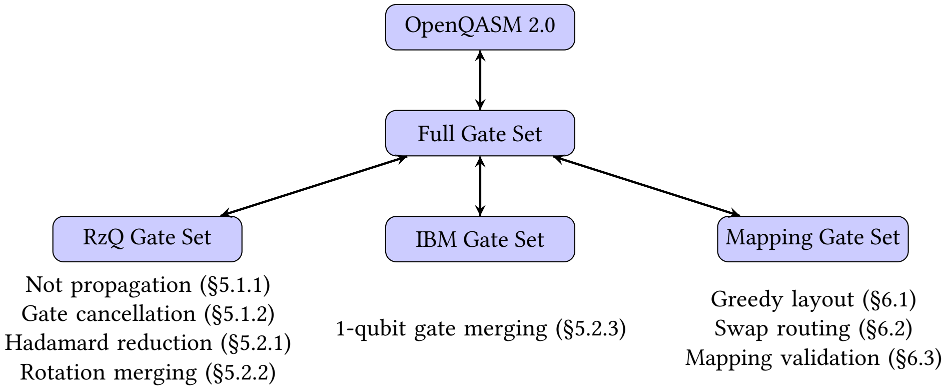

OpenQASM 2.0
Full Gate Set
RzQ Gate Set
IBM Gate Set
Mapping Gate Set
Not propagation (§5.1.1)
Gate cancellation (§5.1.2)
Hadamard reduction (§5.2.1)
Rotation merging (§5.2.2)
1-qubit gate merging (§5.2.3)
Greedy layout (§6.1)
Swap routing (§6.2)
Mapping validation (§6.3)

Fig. 4. Summary of features available in VOQC.

Table 2. Decompositions of multi-qubit gates in the full gate set into simpler gates in the full gate set. Decomposition to the RzQ or IBM gate sets can be performed by further applying the rules in Table 3. Note that CX is primitive in every gate set we support.

Input Gate	**Decomposition**
$CX\ a\ b$	$CX\ a\ b$
$CZ\ a\ b$	$H\ b;\ CX\ a\ b;\ H\ b$
$SWAP\ a\ b$	$CX\ a\ b;\ CX\ b\ a;\ CX\ a\ b$
$CCZ\ a\ b\ c$	$CX\ b\ c;\ T^\dagger\ c;\ CX\ a\ c\ ;\ \ T\ c\ ;\ CX\ b\ c\ ;\ T^\dagger\ c;$ $CX\ a\ c;\ CX\ a\ b\ ;\ T^\dagger\ b;\ CX\ a\ b\ ;\ T\ a;\ T\ b;\ T\ c$
$CCX\ a\ b\ c$	$H\ c;\ CCZ\ a\ b\ c;\ H\ c$

decomposing it into a series of one- and two-qubit gates (potentially incorrectly). As shown in Figure 4, although optimizations are defined over different gate sets internally, in the interface we expose, all functions are defined over the full gate set. We convert between the different gate sets using the rules in Tables 2 and 3.

The *RzQ gate set*, inspired by the one used by Nam et al. [33], consists of $\{H,\ X,\ R_zQ,\ CX\}$ where $R_zQ(q)$ describes rotation about the z-axis by $q\pi$ for $q \in \mathbb{Q}$. We use a rational parameter for the R_zQ gate instead of a real parameter in an effort to avoid unsound extraction to OCaml. Our Coq formalization relies on an axiomatized definition of real numbers [22], so there is no way to extract Coq definitions using reals to OCaml without providing an implementation of real arithmetic. One option (used for the IBM gate set below) is to extract Coq reals to OCaml floats, although this leads to the possibility of *floating-point error* not accounted for in our proofs.

The *IBM gate set* is the default basis for the Qiskit compiler, and is supported in many quantum compilers. It includes the two-qubit CX gate, along with three parameterized single-qubit gates:

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

U_3 gates are the most general,[4] and require two quantum "pulses" to implement on hardware. U_2 and U_1 gates are more specialized, but require one and zero pulses, respectively. One interesting property of this gate set (which is not true of the RzQ gate set) is that any sequence of single-qubit gates can be combined into a single gate (see Section 5.2.3).

[4] U_1 and U_2 gates can both be written in terms of U_3: $U_1(\lambda) = U_3(0,0,\lambda)$ and $U_2(\phi,\lambda) = U_3(\frac{\pi}{2},\phi,\lambda)$.

Table 3. Decompositions of single-qubit gates in the full gate set into gates in the RzQ and IBM gate sets. When needed, we perform implict coercion from real expressions (e.g., $\frac{r}{\pi}$) to rational numbers.

Input	**RzQ Decomp.**	**IBM Decomp.**
I	$R_zQ(0)$	$U_1(0)$
X	X	$U_3(\pi, 0, \pi)$
Y	$R_zQ(\frac{3}{2});\ X;\ R_zQ(\frac{1}{2})$	$U_3(\pi, \frac{\pi}{2}, \frac{\pi}{2})$
Z	$R_zQ(1)$	$U_1(\pi)$
H	H	$U_2(0, \pi)$
S	$R_zQ(\frac{1}{2})$	$U_1(\frac{\pi}{2})$
T	$R_zQ(\frac{1}{4})$	$U_1(\frac{\pi}{4})$
$S^\dagger$	$R_zQ(\frac{3}{2})$	$U_1(-\frac{\pi}{2})$
$T^\dagger$	$R_zQ(\frac{7}{4})$	$U_1(-\frac{\pi}{4})$
$R_x(r)$	$H;\ R_zQ(\frac{r}{\pi});\ H$	$U_3(r, -\frac{\pi}{2}, \frac{\pi}{2})$
$R_y(r)$	$R_zQ(\frac{3}{2});\ H;\ R_zQ(\frac{r}{\pi});\ H;\ R_zQ(\frac{1}{2})$	$U_3(r, 0, 0)$
$R_z(r)$	$R_zQ(\frac{r}{\pi})$	$U_1(r)$
$R_zQ(q)$	$R_zQ(q)$	$U_1(q\pi)$
$U_1(r)$	$R_zQ(\frac{r}{\pi})$	$U_1(r)$
$U_2(r_1, r_2)$	$R_zQ(\frac{r_2}{\pi} - 1);\ H;\ R_zQ(\frac{r_1}{\pi})$	$U_2(r_1, r_2)$
$U_3(r_1, r_2, r_3)$	$R_zQ(\frac{r_3}{\pi} - 1/2);\ H;\ R_zQ(\frac{r_1}{\pi});\ H;\ R_zQ(\frac{r_2}{\pi} + 1/2)$	$U_3(r_1, r_2, r_3)$

However, during combination, it is not possible to stay in the domain of rational numbers, which forces us to change the parameter type to be real, leading to potential unsoundness in extraction (discussed below).

The *mapping gate set* is used for circuit mapping (Section 6). It is parameterized by a set of single-qubit gates and includes multi-qubit gates CX and $SWAP$. In our implementation, we instantiate the mapping gate set using the single-qubit gates from the full gate set.

To compute a program's denotation, voqc's gates must be translated into the CX and $R_{\theta,\phi,\lambda}$ gates in sqir's base set. In the RzQ set, H, X, and $RzQ(q)$ are translated into $R_{\pi/2,0,\pi}$, $R_{\pi,0,\pi}$, and $R_{0,0,q\pi}$ respectively. In the IBM set, $U_3(\theta, \phi, \lambda)$ is translated into $R_{\theta,\phi,\lambda}$. We compute the denotation of a program in the full gate set using the rules in Tables 2 and 3.

4.4 Extraction to Executable Code

We use Coq's standard code extraction mechanism [23] to extract voqc into a standalone OCaml library. For performance, our library uses OCaml primitives for describing multi-precision rational numbers, maps and sets, rather than the code generated from Coq. We thus implicitly trust that the OCaml implementation of these data types is consistent with Coq's; we believe that this is a reasonable assumption. A more problematic assumption is that the behavior of OCaml's 64-bit float type matches the behavior of Coq's mathematical reals. As mentioned above, we extract the real parameters of the IBM and full gate sets to floats, which may allow for floating-point error not accounted for in our soundness proofs. We would prefer to use a full-precision datatype, like rationals, but the trigonometric functions used to optimize U_2 and U_3 gates are not defined over rationals. We note that existing quantum compilers also use float parameters, so they are equally susceptible to floating-point errors. They may even enforce precision limitations internally: For example, Qiskit's `CommutativeCancellation` optimization pass [38] uses a cutoff precision of 10^{-5}, below which rotations are treated as identities.

$$\begin{array}{rcl} X\ q;\ H\ q & \equiv & H\ q;\ Z\ q \\ X\ q;\ R_zQ(k)\ q & \cong & R_zQ(2-k)\ q;\ X\ q \\ X\ q_1;\ CX\ q_1\ q_2 & \equiv & CX\ q_1\ q_2;\ X\ q_1;\ X\ q_2 \\ X\ q_2;\ CX\ q_1\ q_2 & \equiv & CX\ q_1\ q_2;\ X\ q_2 \end{array}$$

Fig. 5. Equivalences used in not propagation.

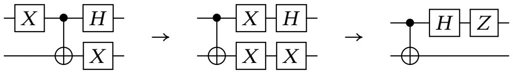


Fig. 6. An example of not propagation. In the first step the leftmost X gate propagates through the CX gate to become two X gates. In the second step the upper X gate propagates through the H gate and the lower X gates cancel.

In order to make voqc compatible with existing Python-based frameworks for compiling quantum programs (e.g., Qiskit [37], pytket [8], Quilc [41], Cirq [13]), we provide a Python wrapper around the voqc OCaml library. To interface between Python and OCaml, we wrap the OCaml code in a C library (following standard conventions [21]) and call to this C library using Python's `ctypes` [36]. For convenience, we have written Python code that makes voqc look like an optimization pass in IBM's Qiskit, allowing us to take advantage of this framework's utilities for quantum programming (e.g., constructing and printing circuits, unverified optimizations and mapping routines). We use the voqc Qiskit pass in our evaluation in Section 7.2.

5 OPTIMIZATIONS

voqc primarily implements optimizations inspired by the state-of-the-art circuit optimizer by Nam et al. [33]. As such, we do not claim credit for the optimizations themselves. Rather, our contribution is a framework that is sufficiently flexible that it can be used to prove such state-of-the-art optimizations correct.

voqc implements two basic kinds of optimizations: *replacement* and *propagate-cancel*. The former simply identifies a pattern of gates and replaces it with an equivalent pattern. The latter works by commuting sets of gates when doing so produces an equivalent quantum program—often with the effect of "propagating" a particular gate rightward in the program—until two adjacent gates can be removed because they cancel out.

5.1 Optimization by Propagation and Cancellation

Our *propagate-cancel* optimizations have two steps. First we localize a set of gates by repeatedly applying commutation rules. Then we apply a circuit equivalence to replace that set of gates. In voqc, most optimizations of this form use a library of code patterns, but one—*not propagation*—is slightly different, so we discuss it first.

5.1.1 Not Propagation. The goal of not propagation is to remove cancelling X ("not") gates. Two X gates cancel when they are adjacent or they are separated by a circuit that commutes with X. We find X gates separated by commuting circuits by repeatedly applying the propagation rules in Figure 5. An example application of the not propagation algorithm is shown in Figure 6.

This implementation may introduce extra X gates at the end of a circuit or extra Z gates in the interior of the circuit. Extra Z gates are likely to be cancelled by the gate cancellation and rotation merging passes that follow, and moving X gates to the end of a circuit makes the rotation merging optimization more likely to succeed. We note that our

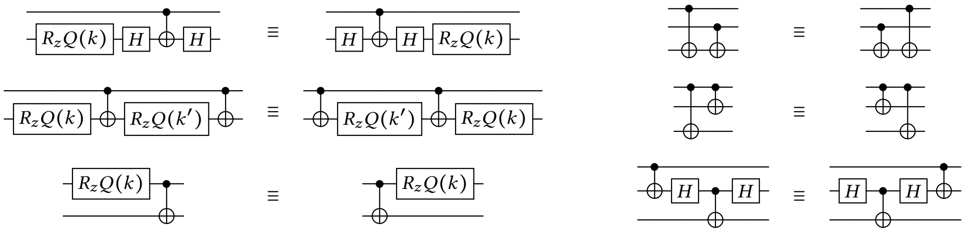


Fig. 7. Commutation equivalences for single- and two-qubit gates adapted from Nam et al. [33, Figure 5]. We use the second and third rules for propagating both single- and two-qubit gates.

version of this optimization is a simplification of Nam et al.'s, which supports the three-qubit CCX gate; this gate can be decomposed into a $\{H, R_zQ, CX\}$ program per Table 2. In our experiments, we did not observe any difference in performance between VOQC and Nam et al. due to this simplification.

5.1.2 Gate Cancellation. The single- and two-qubit gate cancellation optimizations rely on the same propagate-cancel pattern used in not propagation, except that gates are returned to their original location if they fail to cancel. To support this pattern, we provide a general `propagate` function in VOQC. This function takes as inputs (i) an instruction list, (ii) a gate to propagate, and (iii) a set of rules for commuting and cancelling that gate. At each iteration, `propagate` performs the following actions:

(1) Check if a cancellation rule applies. If so, apply that rule and return the modified list.
(2) Check if a commutation rule applies. If so, commute the gate and recursively call `propagate` on the remainder of the list.
(3) Otherwise, return the gate to its original position.

We have proved that our `propagate` function is sound when provided with valid commutation and cancellation rules.

Each commutation or cancellation rule is implemented as a partial Coq function from an input circuit to an output circuit. A common pattern in these rules is to identify one gate (e.g., an X gate), and then to look for an adjacent gate it might commute with (e.g., CX) or cancel with (e.g., X). For commutation rules, we use the rewrite rules shown Figure 7. For cancellation rules, we use the fact that H, X, and CX are all self-cancelling and $R_zQ(k)$ and $R_zQ(k')$ combine to become $R_zQ(k + k')$.

5.2 Circuit Replacement

We have implemented three optimizations that work by replacing one pattern of gates with an equivalent one; no preliminary propagation is necessary. These aim either to reduce the gate count directly, or to set the stage for additional optimizations.

5.2.1 Hadamard Reduction. The Hadamard reduction routine employs the equivalences shown in Figure 8 to reduce the number of H gates in the program. Removing H gates is useful because H gates limit the size of the $\{R_zQ, CX\}$ subcircuits used in the rotation merging optimization.

5.2.2 Rotation Merging. The rotation merging optimization allows for combining R_zQ gates that are not physically adjacent in the circuit. This optimization is more sophisticated than the previous optimizations because it does not rely on small structural patterns (e.g., that adjacent X gates cancel), but rather on more general (and non-local) circuit

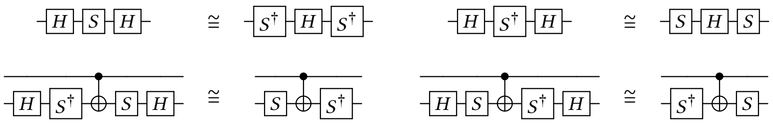


Fig. 8. Equivalences for removing Hadamard gates adapted from Nam et al. [33, Figure 4]. S is the phase gate $R_zQ(1/2)$ and $S^\dagger$ is its inverse $R_zQ(3/2)$.

behavior. The basic idea behind rotation merging is to (i) identify subcircuits consisting of only CX and R_zQ gates and (ii) merge R_zQ gates within those subcircuits that are applied to qubits in the same logical state.

The argument for the correctness of this optimization relies on the *phase polynomial* representation of a circuit. Let C be a circuit consisting of CX gates and rotations about the z-axis. Then on basis state $|x_1, ..., x_n\rangle$ for $x_i \in \{0, 1\}$, C will produce the state

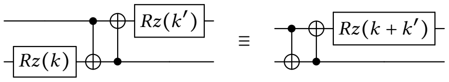
$$e^{ip(x_1,...,x_n)}\,|h(x_1, ..., x_n)\rangle$$

where $h : \{0, 1\}^n \to \{0, 1\}^n$ is an affine reversible function and

$$p(x_1, ..., x_n) = \sum_{i=1}^{l} (\theta_i \mod 2\pi)\ f_i(x_1, ..., x_n)$$

is a linear combination of affine boolean functions. $p(x_1, ..., x_n)$ is called the phase polynomial of circuit C. Each rotation gate in the circuit is associated with one term of the sum and if two terms of the phase polynomial satisfy $f_i(x_1, ..., x_n) = f_j(x_1, ..., x_n)$ for some $i \neq j$, then the corresponding i and j rotations can be merged.

As an example, consider the two circuits shown below.



To prove that these circuits are equivalent, we can consider their behavior on basis state $|x_1, x_2\rangle$. Applying $R_zQ(k)$ to the basis state $|x\rangle$ produces the state $e^{ik\pi x}\,|x\rangle$ and $CX\,|x, y\rangle$ produces the state $|x, x \oplus y\rangle$ where $\oplus$ is the xor operation. Thus evaluation of the left-hand circuit proceeds as follows:

$$|x_1, x_2\rangle \to e^{ik\pi x_2}\,|x_1, x_2\rangle \to e^{ik\pi x_2}\,|x_1, x_1 \oplus x_2\rangle \to e^{ik\pi x_2}\,|x_2, x_1 \oplus x_2\rangle \to e^{ik\pi x_2} e^{ik'\pi x_2}\,|x_2, x_1 \oplus x_2\rangle\,.$$

Whereas evaluation of the right-hand circuit produces

$$|x_1, x_2\rangle \to |x_1, x_1 \oplus x_2\rangle \to |x_2, x_1 \oplus x_2\rangle \to e^{i(k+k')\pi x_2}\,|x_2, x_1 \oplus x_2\rangle\,.$$

The two resulting states are equal because $e^{ik\pi x_2} e^{ik'\pi x_2} = e^{i(k+k')\pi x_2}$. This implies that the unitary matrices corresponding to the two circuits are the same. We can therefore replace the circuit on the left with the one on the right, removing one gate from the circuit.

Our rotation merging optimization follows the reasoning above for arbitrary $\{R_zQ, CX\}$ circuits. For every gate in the program, it tracks the Boolean function associated with every qubit (the Boolean functions above are x_1, x_2, $x_1 \oplus x_2$), and merges R_zQ rotations when they are applied to qubits associated with the same Boolean function. To prove equivalence over $\{R_zQ, CX\}$ circuits, we show that the original and optimized circuits produce the same output on every basis state. We have found evaluating behavior on basis states to be useful for proving equivalences that are not as direct as those listed in Figures 7 and 8.

$$\begin{array}{rclrcl}
U_1(\lambda_1)\ ;\ U_1(\lambda_2) & = & U_1(\lambda_1+\lambda_2) & U_1(\lambda_1)\ ;\ U_2(\phi_2,\ \lambda_2) & = & U_2(\phi_2,\ \lambda_1+\lambda_2) \\
U_1(\lambda_1)\ ;\ U_3(\theta_2,\ \phi_2,\ \lambda_2) & = & U_3(\theta_2,\ \phi_2,\ \lambda_1+\lambda_2) & U_2(\phi_1,\ \lambda_1)\ ;\ U_1(\lambda_2) & = & U_2(\phi_1+\lambda_2,\ \lambda_1) \\
U_2(\phi_1,\ \lambda_1)\ ;\ U_2(\phi_2,\ \lambda_2) & = & U_3(\pi-\phi_1-\lambda_2,\ \phi_2+\frac{\pi}{2},\ \lambda_1+\frac{\pi}{2}) & U_3(\theta_1,\ \phi_1,\ \lambda_1)\ ;\ U_1(\lambda_2) & = & U_3(\theta_1,\ \phi_1+\lambda_2,\ \lambda_1)
\end{array}$$

Fig. 9. Rules for single-qubit gate merging.

Although our merge operation is identical to Nam et al.'s, our approach to constructing $\{R_zQ, CX\}$ subcircuits differs. We construct a $\{R_zQ, CX\}$ subcircuit beginning from a R_zQ gate whereas Nam et al. begin from a CX gate. The result of this simplification is that our subcircuits may be smaller than Nam et al.'s, causing us to miss some opportunities for merging. However, in our experiments (Appendix A) we found that this choice impacted only one benchmark.

5.2.3 Single-qubit Gate Merging. In the IBM gate set, any two single-qubit gates can be combined into one gate. This allows us to implement an optimization over programs in the IBM gate set (which Qiskit calls `Optimize1qGates` [38]) that merges all adjacent single-qubit gate by applying the rules in Figure 9, along with a more complex rule for combining a U_2 and U_3 gate or two U_3 gates.

In the more complex rule, the two gates are first converted into a sequence of Euler rotations about the y- and z-axes: $U_3(\theta,\phi,\lambda) \longrightarrow R_z(\phi)\cdot R_y(\theta)\cdot R_z(\lambda)$. Call this a ZYZ rotation. Next, local identities are applied to combine the two ZYZ rotations into a single ZYZYZ rotation. Then the interior YZY rotation is converted to a new ZYZ rotation, yielding a ZZYZZ rotation. Finally, this is simplified to a ZYZ rotation, which can be represented as a U_3 gate. For example, here is the process for combining two U_3 gates:

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

where α, β, γ satisfy $R_y(\theta_2)\cdot R_z(\lambda_2+\phi_1)\cdot R_y(\theta_1) = R_z(\gamma)\cdot R_y(\beta)\cdot R_z(\alpha)$. The angles α, β, γ can be generated using arithmetic over trignometric functions sin, cos, arccos, and arctan [14], as shown in Figure 10. Proving the generation of α, β, γ correct was the most difficult part of verifying soundness for this optimization; to our knowledge, we are the first to formally verify this method in a proof assistant.

5.3 Scheduling

The VOQC `optimize` function applies each of the optimizations we have discussed one after the other, in the following order (due to Nam et al.):

$$0, 1, 3, 2, 3, 1, 2, 4, 3, 2$$

where 0 is not propagation, 1 is Hadamard reduction, 2 is single-qubit gate cancellation, 3 is two-qubit gate cancellation, and 4 is rotation merging. Nam et al. justify this ordering at length, though they do not prove that it is optimal. In brief, removing X and H gates (0,1) allows for more effective application of the gate cancellation (2,3) and rotation merging (4) optimizations. In our experiments (Section 7), we observed that single-qubit gate cancellation and rotation merging were the most effective at reducing gate count.

We (optionally) conclude by converting gates to the IBM gate set and performing single-qubit gate merging in order to produce output in the $\{U_1, U_2, U_3, CX\}$ gate set for fair comparison with other tools.

```
Definition rm02 (x y z : R) : R := sin x * cos z + cos x * cos y * sin z.
Definition rm12 (x y z : R) : R := sin y * sin z.
Definition rm22 (x y z : R) : R := cos x * cos z − sin x * cos y * sin z.
Definition rm10 (x y z : R) : R := sin y * cos z.
Definition rm11 (x y z: R) : R := cos y.
Definition rm20_min (x y z : R) : R := cos x * sin z + sin x * cos y * cos z.
Definition rm21 (x y z : R) : R := sin x * sin y.

Definition atan2 (y x : R) : R :=
 if 0 <? x then atan (y/x)
 else if x <? 0 then if negb (y <? 0) then atan (y/x) + PI else atan (y/x) − PI
      else if 0 <? y then PI/2 else if y <? 0 then −PI/2 else 0.

Definition yzy_to_zyz (x y z : R) : R * R * R :=
 if rm22 x y z <? 1
 then if −1 <? rm22 x y z
      then (atan2 (rm12 x y z) (rm02 x y z), acos (rm22 x y z), atan2 (rm21 x y z) (rm20_min x y z))
      else (− atan2 (rm10 x y z) (rm11 x y z), PI, 0)
 else (atan2 (rm10 x y z) (rm11 x y z), 0, 0).

(* Correctness property: *)
Lemma yzy_to_zyz_correct : ∀ θ1 ξ θ2 ξ1 θ ξ2,
  yzy_to_zyz θ1 ξ θ2 = (ξ1, θ, ξ2) →
  y_rotation θ2 × phase_shift ξ × y_rotation θ1
     ∝ phase_shift ξ2 × y_rotation θ × phase_shift ξ1.
```

Fig. 10. Code for converting a YZY rotation to a ZYZ rotation.

5.4 Verifying Low-Level Circuit Equivalences

voqc optimizations make heavy use of circuit equivalences such as those shown in Figures 5 and 7 to 9. To prove that voqc optimizations are sound, we must formally verify these equivalences are correct. Such proofs require showing equality between two matrix expressions, which can be tedious in the case where the matrix size is left symbolic. For example, consider the following equivalence used in *not propagation*:

$$X\ n;\ CX\ m\ n \equiv CX\ m\ n;\ X\ n$$

for arbitrary n, m and dimension d. Applying our definition of equivalence, this amounts to proving

$$apply_1(X, n, d)\ \times\ apply_2(CX, m, n, d) = apply_2(CX, m, n, d)\ \times\ apply_1(X, n, d), \tag{1}$$

per the semantics in Figure 3. Suppose both sides of the equation are well typed ($m < d$ and $n < d$ and $m \neq n$), and consider the case where $m < n$ (the $n < m$ case is similar). We expand $apply_1$ and $apply_2$ as follows with $p = n - m - 1$ and $q = d - n - 1$:

$$\begin{aligned} apply_1(X, n, d) &= I_{2^n} \otimes \sigma_x \otimes I_{2^q} \\ apply_2(CX, m, n, d) &= I_{2^m} \otimes |1\rangle\langle 1| \otimes I_{2^p} \otimes \sigma_x \otimes I_{2^q} + I_{2^m} \otimes |0\rangle\langle 0| \otimes I_{2^p} \otimes I_2 \otimes I_{2^q} \end{aligned}$$

Here, σ_x is the matrix interpretation of the X gate and $|1\rangle\langle 1| \otimes \sigma_x + |0\rangle\langle 0| \otimes I_2$ is the matrix interpretation of the CX gate (in Dirac notation). We can complete the proof of equivalence by normalizing and simplifying each side of Equation (1), showing both sides to be the same.

We address the tedium of such proofs in VOQC by automating the matrix normalization and simplification steps. We provide a Coq tactic called `gridify` for proving general equivalences correct. Rather than assuming $m < n < d$ as above, the `gridify` tactic does case analysis, immediately solving all cases where the circuit is ill-typed (e.g., $m = n$ or $d \leq m$) and thus has the zero matrix as its denotation. In the remaining cases ($m < n$ and $n < m$ above), it puts the expressions into a form we call *grid normal* and applies a set of matrix identities.

In grid normal form, each arithmetic expression has addition on the outside, followed by tensor product, with multiplication on the inside, i.e., $((.. \times ..) \otimes (.. \times ..)) + ((.. \times ..) \otimes (.. \times ..))$. The `gridify` tactic rewrites an expression into this form by using the following rules of matrix arithmetic:

- $I_{mn} = I_m \otimes I_n$
- $A \times (B + C) = A \times B + A \times C$
- $(A + B) \times C = A \times C + B \times C$
- $A \otimes (B + C) = A \otimes B + A \otimes C$
- $(A + B) \otimes C = A \otimes C + B \otimes C$
- $(A \otimes B) \times (C \otimes D) = (A \times C) \otimes (B \times D)$

The first rule is applied to facilitate application of the other rules. (For instance, in the example above, I_{2^n} would be replaced by $I_{2^m} \otimes I_2 \otimes I_{2^p}$ to match the structure of the $apply_2$ term.) After expressions are in grid normal form, `gridify` simplifies them by removing multiplication by the identity matrix and rewriting simple matrix products (e.g. $\sigma_x \sigma_x = I_2$).

In our example, normalization and simplification by `gridify` rewrites each side of the equality in Equation (1) to be the following

$$I_{2^m} \otimes |1\rangle\langle 1| \otimes I_{2^p} \otimes I_2 \otimes I_{2^q} + I_{2^m} \otimes |0\rangle\langle 0| \otimes I_{2^p} \otimes \sigma_x \otimes I_{2^q},$$

thus proving that the two expressions are equal.

We use `gridify` to verify most of the equivalences used in the optimizations given in Sections 5.1 and 5.2. The tactic is most effective when equivalences are small: The equivalences used in *gate cancellation* and *Hadamard reduction* apply to patterns of at most five gates applied to up to three qubits within an arbitrary circuit. For equivalences over large sets of qubits, like the one used in *rotation merging*, we do not use `gridify` directly, but still rely on our automation for matrix simplification.

6 CIRCUIT MAPPING

While optimization aims to reduce qubit and gate usage to make programs more feasible to run on near-term machines, *circuit mapping* addresses the connectivity constraints of near-term machines, transforming a program so that it is able to run on a machine [42, 53]. Circuit mapping algorithms take as input an arbitrary circuit and output a circuit that respects the connectivity constraints of the underlying architecture. Consider the connectivity constraints of IBMs's 16-qubit Guadalupe machine [20], shown in Figure 11. This is a representative example of a superconducting qubit system, where qubits are laid out in a 2-dimensional grid and possible interactions are described by edges between qubits. For instance, a CX gate may be applied between qubits 1 and 4, but not between qubits 1 and 3.

Circuit mapping typically consists of two stages: *layout* (or placement), which associates each logical qubit in the program with some physical qubit on the machine; and *routing*, which, given an initial layout, transforms a program to satisfy connectivity constraints. Routing is often performed by inserting *SWAP* gates to "move" qubits to compatible locations when they are used together in a two-qubit gate. This approach is used by the routing routines in Qiskit [37]

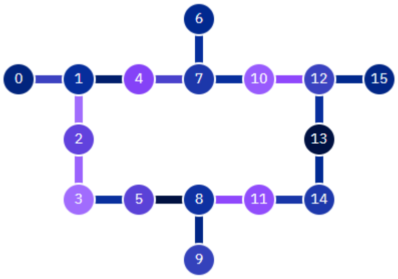
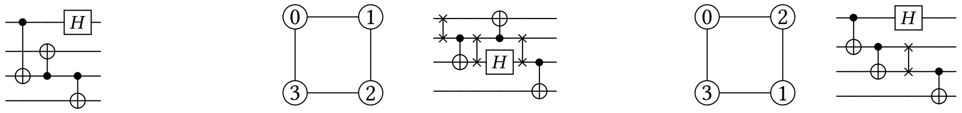


Fig. 11. IBM's Guadalupe machine [20]. The different colors on nodes and connections reflect different error rates (darker means lower error and lighter means higher error).



P1 = CX 0 2; CX 2 1; H 0; CX 2 3

(a) Input program

P2 = SWAP 0 2; CX 1 2; SWAP 1 2;
CX 1 0; H 2; SWAP 1 2; CX 2 3

(b) Trivial layout

P3 = CX 0 1; CX 1 2; H 0; SWAP 1 2; CX 2 3

(c) Greedy layout

Fig. 12. Circuit mapping example

and t|ket⟩ [43]. Another approach is to compute the unitary matrix corresponding to the input (sub)circuit and then *resynthesize* the circuit in a way that satisfies connectivity constraints; Staq [2] implements this approach.

6.1 Verified Layout

We provide two layout functions in voqc: *trivial layout*, which maps logical qubit i to physical qubit i, and *greedy layout*, which takes into account the program and architecture characteristics. Greedy layout scans through the program and allocates logical qubits to adjacent physical qubits when they are used together in a CX gate, until all logical qubits are allocated. Figure 12(a) shows an example quantum circuit that uses four qubits. Imagine that we would like to map this circuit to an architecture with four qubits, connected in a ring. Figure 12(b) shows the result of arranging the circuit's qubits according to a trivial layout, and Figure 12(c) shows the result of arranging the qubits according to our greedy layout routine. The greedy layout allocates logical qubits 0 and 2 (and 1 and 2) to adjacent physical qubits since they are used together in a CX gate.

We verify that both the trivial and greedy layout functions produce well-formed layouts, i.e., one-to-one mappings between logical and physical qubits. We also provide a function to convert a list l to a layout where physical qubit i maps to logial qubit $l[i]$; we prove that this function produces a well-formed layout if the input list has no duplicates and every element in the list is less than the length of the list. We use this function to generate safe layouts for our translation validation routine in Section 6.3.

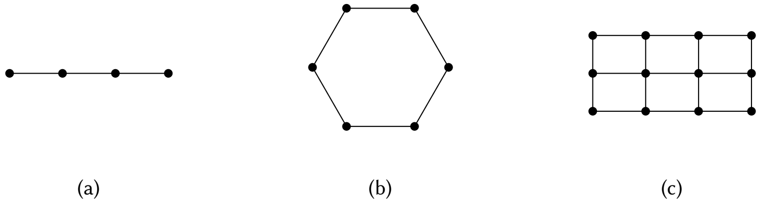


Fig. 13. Architectures supported in voqc. From left to right: LNN, LNN ring, and 2D grid. Each architecture is shown with a fixed number vertices, but in our implementation the number of vertices is a parameter.

6.2 Verified Routing

We have implemented a simple *SWAP*-based routing method for unitary sqir programs and verified that it is sound (up to a permutation of qubits) and produces programs that satisfy the relevant hardware constraints. Our routing method is parameterized by a description of the connectivity of an architecture, which includes a function to check whether an edge is in the connectivity graph and a function to find a path between two nodes. Given an initial layout, our implementation iterates through the gates of the input program and, every time a *CX* occurs between two logical qubits whose corresponding physical qubits are not adjacent in the underlying architecture, inserts *SWAP*s to move the control adjacent to the target.

Figure 12(b-c) show the result of applying our routing routine to the circuit in Figure 12(a) starting from the trivial and greedy layouts. In Figure 12(b), three *SWAP* gates are inserted to ensure that the circuit can execute on the target hardware, while in Figure 12(c) only one *SWAP* is inserted. In both produced circuits, the wires are ordered: top left physical qubit, top right, lower right, lower left.

Although our routing algorithm is simple (it is equivalent to Qiskit's `BasicSwap` pass [38]), it allows for some flexibility in design because we do not specify the method for finding paths in the connectivity graph, which allows, for example, strategies that take into account error characteristics of the machine [47]. We have built-in connectivity graphs for the linear nearest neighbor (LNN), LNN ring, and 2D nearest neighbor architectures pictured in Figure 13.

6.3 Verified Translation Validation

There are a wide variety of proposed layout and routing techniques, many of which involve complex search algorithms and heuristic techniques [10, 29, 42, 47, 53]. Rather than aiming to verify all of these different approaches in Coq, we provide a *verified translation validation* function for checking the correctness of circuit mapping on particular inputs. Unlike our verified optimizations and mapping routines, the verified translation validator may fail at runtime, indicating a potential bug in the circuit mapper. However, if translation validation succeeds, then our proofs guarantee that the input and output programs are mathematically equivalent up to permutation. Our translation validator works by removing *SWAP* gates, performing a logical relabelling of qubits, and then checking for equality modulo reordering of gates that respects gate dependencies.

This approach to equivalence checking will only work for routing routines that insert *SWAP* gates while leaving the rest of the program's structure unchanged (e.g., those in Qiskit and t|ket⟩[5] It will not be able to validate circuits generated using resynthesis (e.g., using Staq's `steiner` routing). However, it is possible to develop polynomial-time

[5] t|ket⟩ may choose to compile a distance-2 *CX* gate to a distributed *CX*, rather that a *SWAP* followed by a *CX*, if the heuristics find that this improves the final result [43, Section 7.2]. This optimization will cause our translation validator to fail; a simple fix would be to decompose *SWAP* gates before validation and then remove select *CX* gates from the mapped circuits. We did not implement this because it is not needed for Qiskit's routing methods.

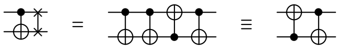
translation validation functions for these cases too since the input and output circuits have a restricted form (e.g., Staq performs routing for $\{CX, X, R_z\}$ sub-circuits, which can be easily analyzed using phase polynomials per our discussion of rotation merging in Section 5.2).

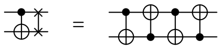
Translation validation is popular for providing assurance for quantum compilers: Amy [1] checks for equivalence of optimized and unoptimized programs using the path-sums semantics; PyZX [26] performs translation validation by checking if the result of optimizing a circuit followed by its optimized adjoint produces an identity program; and Smith and Thornton [44] provide a compiler with built-in translation validation via QMDD equivalence checking [32]. Most recently, Burgholzer et al. [6] presented a technique for equivalence checking, specialized to validating results of the Qiskit compiler, that relies on the fact that the identity matrix (which should be the result of composing a circuit with its optimized adjoint) can be efficiently represented using decision diagrams [7]; this observation allows them to perform equivalence checking on circuit that use tens of thousands of operations. But none of these other tools has been formally verified, and all aside from Burgholzer et al. [6] are more computationally heavy than our mapping validation, which simply requires a linear scan through the input, because they aim to detect equivalence between a more general class of programs.

6.4 Mapping with Optimization

Circuit mapping increases the size of the program, typically adding many CX gates to perform $SWAP$s between qubits. It is desirable to reduce this overhead by applying optimization after mapping, but this is only worthwhile if optimization preserves the guarantee from mapping that all CX gates are allowed by the connectivity graph. We have verified that all the optimizations in Section 5 preserve connectivity guarantees, allowing us to apply optimization before and/or after mapping.

We also apply some light mapping-specific optimizations. For example, after mapping we carefully decompose $SWAP$ gates to enable further optimization. $SWAP$ gates have two natural decompositions in terms of CX gates: $SWAP\ a\ b\ =\ CX\ a\ b;\ CX\ b\ a;\ CX\ a\ b$ and $SWAP\ a\ b\ =\ CX\ b\ a;\ CX\ a\ b;\ CX\ b\ a$. We choose the decomposition that will allow CX gates to be removed during gate cancellation (Section 5.1). For example, the subcircuit in Figure 12(b) and Figure 12(c) that consists of CX followed by $SWAP$ will be decomposed as shown on the left below, rather than the right, since this will save two gates in the optimized form.





7 EXPERIMENTAL EVALUATION

The value of VOQC is determined by the quality of the verified optimizations we can write with it. We can judge optimization quality empirically by running VOQC on a benchmark of circuit programs to see how well it optimizes those programs, compared to (non-verified) state-of-the-art compilers.

To this end, in Section 7.1, we compare the performance of VOQC's verified optimizations against IBM's Qiskit compiler [37], CQC's t|ket⟩ [43], PyZX [26], and Staq [2] on a set of benchmarks developed for Staq. We find that VOQC has comparable performance to all of these: it generally beats these tools in terms of total gate count reduction, and often matches reduction of T gate count. In Section 7.2, we evaluate VOQC's optimization and mapping routines by running them via a pass in the Qiskit transpiler on a set of benchmarks used to evaluate prior work on mapping algorithms [50, 53]. Compared to Qiskit's default settings, we find that VOQC's optimizations provide an advantage, even for mapped circuits, and our verified translation validation does not add undue overhead. Finally, in Appendix A,

Table 4. Summary of optimizations used in evaluation.

`qiskit-terra` 0.19.1	
Optimize1qGatesDecomposition	✓
CommutativeCancellation	✓*
ConsolidateBlocks w/ UnitarySynthesis	
`pytket` 0.19.2	
RemoveRedundancies	✓
FullPeepholeOptimise	
`pystaq` 2.1	
simplify	✓
rotation_fold	✓*
cnot_resynth	
`pyzx` 0.7.0	
full_optimize	✓*
full_reduce	

we provide a detailed comparison of the performance of VOQC and Nam et al., showing that our verified implementation is mostly faithful to its inspiration.

Note that the aim of this section is not to claim superiority over existing tools (after all, we have implemented a subset of the optimizations available in Nam et al. [33] and Qiskit), but to demonstrate that the optimizations we have implemented in VOQC are on par with existing *unverified* tools.

7.1 Evaluation on Staq Benchmarks

Benchmarks. The benchmark used to evaluate Staq [2] consists of 35 programs written in the "Clifford+T" gate set (CX, H, S and T, where S and T are z-axis rotations by $\pi/2$ and $\pi/4$, respectively). The benchmark programs contain arithmetic circuits, implementations of multiple-control X gates, Galois field multiplier circuits, and some small quantum algorithm components. We exclude programs with more than 10^4 gates, following the precedent of prior work [43, Section 9.1.1].

We measure reduction in total gate count, two-qubit gate count, and T-gate count (when appropriate). Total and two-qubit gate counts are useful metrics for near-term quantum computing, where the length of the computation must be minimized to reduce error and two-qubit gates have particularly high error rates. T-gate count is relevant in the *fault-tolerant regime* where qubits are encoded using quantum error correcting codes and operations are performed fault-tolerantly. In this regime, the standard method for making Clifford+T circuits fault tolerant produces particularly expensive translations for T gates, so reducing T-count is a common optimization goal. The Clifford+T set is a subset of VOQC's RzQ gate set where each z-axis rotation is restricted to be a multiple of $\pi/4$.

Baselines. We first compare VOQC's performance with that of Qiskit Terra version 0.19.1 (release date December 10, 2021) and t|ket⟩ version 0.19.2 (February 18, 2022). Both of these tools produce output using the $\{U_1, U_2, U_3, CX\}$ gate set, so we configure VOQC to produce output using the IBM gate set. We then compare VOQC against Staq version 2.1 (January 17, 2022) and PyZX version 0.7.0 (February 19, 2022). Since these tools produce output using the Clifford+T gate set, we use use VOQC's RzQ gate set.

Table 4 lists the optimizations we include in our evaluation. For every tool except t|ket⟩, we evaluate all available (unitary) optimizations; we exclude t|ket⟩'s OptimisePhaseGadgets and PauliSimp as they hurt improve performance on

Reduction of:	**Qiskit**	**t|ket⟩**	**voqc**
Total gate count	14.4%	18.5%	28.5%
Two-qubit gate count	2.3%	3.8%	9.8%

(a) IBM gate set

Reduction of:	**Staq**	**PyZX**	**voqc**
Total gate count	15.4%	-27.8%	23.2%
Two-qubit gate count	0.6%	-91.7%	9.8%
T-gate count	45.4%	47.1%	43.1%

(b) RzQ gate set

Qiskit	**t|ket⟩**	**Staq**	**PyZX**	**voqc**
0.60s	1.53s	0.02s	14.58s	0.01s

(c) Running times

Fig. 14. Geometric mean gate count reductions (a,b) and running times (c) on the Staq benchmarks. The full results are in Appendix A.

our benchmarks. voqc provides the complete and verified functionality of the routines marked with ✓; we write ✓* to indicate that voqc contains a verified optimization with similar, although not identical, behavior. voqc's gate cancellation routines generalize Qiskit's Optimize1qGatesDecomposition, t|ket⟩'s RemoveRedundancies, and Staq's simplify; voqc's gate cancellation is also similar to Qiskit's CommutativeCancellation, but Qiskit uses matrix multiplication to determine whether gates commute while we use a rule-based approach. voqc's rotation merging is similar to Staq's rotation_fold and (when combined with gate cancellation) PyZX's full_optimize.

As far as the optimizations voqc does not support: Qiskit's UnitarySynthesis and t|ket⟩'s FullPeepholeOptimize resynthesize two-qubit gate sequences (e.g., using KAK decomposition [49]), Staq's cnot_synthesis resynthesizes arbitrary $\{CX, X, Rz\}$ subcircuits (as discussed in Section 6), and PyZX's full_reduce applies the ZX-calculus rewrite rules described in Kissinger and van de Wetering [27].

Results. The results are summarized in Figure 14; the full results are given in Appendix A. Overall, voqc is the most effective at reducing total and two-qubit gate count for both the IBM and RzQ gate sets, while it is slightly less effective than Staq and PyZX at eliminating T gates.

On all 35 programs, voqc is more effective at reducing total gate count than Qiskit. On 33/35 programs it is more effective at reducing total gate count than t|ket⟩. This gap in performance is primarily due to voqc's rotation merging optimization, which has no analogue in Qiskit or t|ket⟩. On 9/35 programs, Qiskit, t|ket⟩, and voqc provide no reduction in two-qubit gate count, suggesting that these tools are not particularly effective for this type of gate. Of the remaining 23/35 programs, voqc provides a higher reduction in two-qubit gate count, despite the fact that Qiskit and t|ket⟩ both support two-qubit circuit resynthesis, which should give them an advantage. This suggests that small circuit resynthesis optimizations are not particularly effective for this class of circuits.

voqc is more effective than (or equally effective as) Staq and PyZX at reducing total gate count on 29/35 programs. On 4/35 programs, Staq is the most effective, and on the remaining 2/35 PyZX performs best. However, on 26/35 programs PyZX actually *increases* the total gate count.[6] The reason for this is that PyZX converts a circuit to a ZX-diagram, performs optimization, and then converts back; this conversion process can introduce many additional gates. However, PyZX is the best overall at reducing T-gate count. On 28/35 programs, PyZX is more effective than (or equally effective as) Staq and voqc at reducing T-gate count. On 7/35 programs, Staq performs best. Surprisingly, on 13/35 benchmarks, all optimizers produce the same T-count. This is unexpected since, although all these optimizers rely on some form of

[6] To account for this, when computing the "geometric mean reduction" for each type of gate, we actually compute the geometric mean *proportion* of the final to original gate count, which is a strictly positive number. We then subtract this number from 1, which produces a negative value in cases where gate count increased on average.

rotation merging, their implementations differ substantially. Kissinger and van de Wetering [27] posit that this indicates a local optimum in the ancilla-free case for some of the benchmarks (in particular the tof benchmarks, whose T-count is not reduced by applying additional techniques [17]). As with the IBM gate set experiments, voqc is not particularly effective at reducing two-qubit gate count. However, unlike PyZX and Staq, it never increases the number of two-qubit gates, which leads to significantly better performance overall.

To compare the running times of the different tools, we ran 11 trials of voqc, Qiskit, t|ket⟩, Staq, and PyZX (taking the median time for each benchmark) on a standard laptop with a 2.9 GHz Intel Core i5 processor and 16 GB of 1867 MHz DDR3 memory, running macOS Monterey. We then took the geometric mean over the median runtimes. Qiskit, Staq, and voqc all have mean runtimes of under one second. t|ket⟩ is slightly slower, but the mean is still under two seconds. PyZX was consistently the slowest; the full_optimize routine in particular scaled poorly with increasing qubit and gate counts. We set a time limit of 10 minutes for each run of PyZX, defaulting to applying full_reduce (but not full_optimize) in case of a timeout. In the case of a timeout, we consider only the time required to run full_reduce. In addition, full_optimize is not deterministic, and may produce different output circuits for different runs. The gate counts we report are from the trial that produced the lowest T-gate count.

Overall, these results are encouraging evidence that voqc supports useful and interesting verified optimizations. Furthermore, despite having been written with verification in mind, voqc's running times are not significantly worse than (and often better than) that of current tools.

7.2 Evaluation of Mapping Validation

Benchmarks. To evaluate voqc's support for circuit mapping, we used the test suite from the MQT quantum circuit mapping tool [9], subsets of which have been used in the evaluations of Zulehner et al. [53] and Wille et al. [50]. The benchmark programs all use at most 16 qubits, and, like before, we include only those circuits with less than 10^4 gates. The circuits are primarily taken from the RevLib [51] suite of classical reversible circuits, but the benchmark also includes some truly quantum circuits including quantum Fourier transforms and circuits for quantum chemistry. In total, we consider 126 circuit programs.

Baselines. We developed a custom pass for the Qiskit compiler that applies the following sequence of transformations: voqc optimization (using the RzQ gate set), Qiskit circuit mapping, voqc mapping validation, and voqc optimization (using the IBM gate set). Our verified translation validation allows us to use the sophisticated mapping routines available in Qiskit without sacrificing soundness. We can run voqc optimizations both before and after mapping because we have proved that our optimizations preserve mapping guarantees.

We compare our pass against the default Qiskit pass with optimization level 3, which applies mapping followed by optimization. For both passes, we use Qiskit's default layout and routing routines which are based on Li et al. [29]. This routing algorithm is non-deterministic, so we run multiple (11) trials, storing the result of the run that produced the circuit with the lowest total gate count. We additionally compare against an OCaml program that applies voqc optimization (using the RzQ gate set), greedy layout and swap routing (Section 6), and optimization (using the IBM gate set). We record change in total gate count and two-qubit gate count. Note that unlike Section 7.1, where we expected gate count reduction, for this experiment we expect gate count to *increase* since mapping introduces many additional CX gates, some of which will be removed by optimization. All circuits are mapped to a 16-qubit ring architecture (Figure 13(b)). Like before, we use Qiskit Terra version 0.19.1 and run on a standard laptop with a 2.9 GHz Intel Core i5

Table 5. Geometric mean gate count increases and running times for the mapping experiment over all 126 benchmark circuits.

	Qiskit	**voqc**	**Qiskit+voqc**
Total gate count increase	80.9%	73.9%	38.9%
Two-qubit gate count increase	113.5%	204.1%	126.0%
Running time	1.07s	<0.01s	1.01s

processor and 16 GB of 1867 MHz DDR3 memory, running macOS Monterey. When recording timing data, we take the median running time over 11 trials.

Results. The results are summarized in Table 5. The first column shows the result of the Qiskit default pass, the second column shows the result of the voqc OCaml program, and the last column shows the result of the voqc pass within Qiskit. Compared to the Python-based Qiskit passes, the OCaml program is significantly faster; however, it also introduces the most two-qubit gates due to its simple $SWAP$-insertion strategy. Despite this, the OCaml program results in a lower total gate count overhead than Qiskit—this can be explained by voqc's superior optimizations, as per Section 7.1. Overall, Qiskit's default pass provides the lowest overhead in two-qubit gate count. This is due to Qiskit's two-qubit circuit resynthesis optimization, which is especially effective at reducing two-qubit gate count post-mapping. The Qiskit+voqc pass has the best performance overall: it has running time and two-qubit gate count overhead comparable with Qiskit, but provides a much lower total gate count overhead and *proven guarantees* that the output circuit is semantically equivalent to the input. During our trials, we found that translation validation accounted for <1% of the Qiskit+voqc pass running time and we never encountered a validation failure, which means that Qiskit's mapping routines assuredly preserved the semantics of the programs we tested (although this says nothing about semantics-preservation of Qiskit's optimizations).

8 RELATED WORK

In addition to the compilers Qiskit, t|ket⟩, Staq, PyZX, and Nam et al. (discussed in Section 7), other recent efforts include quilc [41], Cirq [13], ScaffCC [25], and Project Q [46]. Due to resource limits on near-term quantum computers, most compilers for quantum programs contain some degree of optimization, and nearly all place an emphasis on satisfying architectural requirements, like mapping to a particular gate set or qubit topology. None of the optimization or mapping code in these compilers is formally verified.

Previously, formal verification has been applied to parts of the quantum compiler stack, but has not supported general quantum programs. Amy et al. [4] and Rand et al. [39] developed certified compilers from source Boolean expressions to reversible circuits. Fagan and Duncan [15] verified an optimizer for ZX diagrams representing Clifford circuits (which use the non-universal gate set $\{CX, H, S\}$). Work on translation validation [6, 26, 44] (discussed in Section 6.3) supports general quantum programs, but adds compile-time overhead and allows for the possibility of compile-time failure due to input/output inequivalence.

Tao et al. [48] developed Giallar, a verification toolkit used to verify transformations in the Qiskit compiler. Their approach has two steps. First, like voqc, they use Coq to prove that a circuit equivalence is valid. Second, they use symbolic execution to generate verification conditions for parts of the program that manipulate circuits. These are given to an SMT solver to verify that pattern equivalences are applied correctly according to programmer-provided function specifications and invariants. Compared to voqc, Giallar is easier for non-experts to use to develop verified software, and simpler to integrate into existing workflows. However, more complex optimizations like rotation merging (which provides a significant benefit over the optimizations in Qiskit, per Section 7) cannot be implemented by applying small local rewrites. Giallar may also fail to prove an optimization correct, e.g., because of complicated control code. We have tried to alleviate proof burden in VOQC by providing a library of verified functions and optimization "templates" (Section 4.1). In particular, the optimizations implemented in Giallar could be implemented using our propagate-cancel template.

Finally, Xu et al. [52] presented Quartz, a *superoptimizer* for quantum circuits with SMT-based verification of circuit equivalences. Quartz begins by applying (unverified implementations of) Toffoli decomposition and rotation merging, as described in Nam et al. [33]. Similar to our results, they find that these passes are a significant source of gate count reduction. After that, having generated a complete set of small (verified) circuit rewrite rules, they use (unverified) cost-based backtracking search to find the optimal sequence of rewrites. This is in contrast to VOQC, which applies only the fixed set of rules described in Section 5. Excitingly, this leads to higher gate count reduction than VOQC [52, Section 7], but it does so at a higher cost: their experiments are run on a 128-core CPU with 512GB RAM, with a search timeout of 24 hours (compared to VOQC, which typically has a running time of under a second on a standard 2015 laptop). It would be interesting to analyze the results of Quartz's optimization to see which rewrite rules are most often triggered and implement and verify these inside of VOQC, improving VOQC's gate count reduction while avoiding the expense of backtracking search.

9 CONCLUSIONS AND FUTURE WORK

This paper has presented VOQC, the first fully verified optimizer for quantum circuits. A key component of VOQC is SQIR, a simple, low-level quantum language deeply embedded in the the Coq proof assistant, which gives a semantics to quantum programs that is amenable to proof. Optimization passes are expressed as Coq functions which are proved to preserve the semantics of their input SQIR programs. VOQC's optimizations are mostly based on local circuit equivalences, implemented by replacing one pattern of gates with another, or commuting a gate rightward until it can be cancelled. Others, like rotation merging, are more complex. These were inspired by, and in some cases generalize, optimizations in industrial compilers, but in VOQC are proved correct. When applied to a benchmark suite of 35 circuit programs, we found VOQC performed comparably to state-of-the-art compilers, reducing gate count on average by 28.5% compared to 14.4% for IBM's Qiskit compiler [37], and 18.5% for CQC's t|ket⟩ [43]. On the same benchmarks, VOQC reduced T-gate count on average by 43.1% compared to 45.4% by Amy and Gheorghiu [2] and 47.1% by the PyZX optimizer [26], although VOQC outperforms both in terms of total gate count reduction.

We are confident that additional circuit-level optimizations and mapping routines can be implemented and verified within the VOQC framework. Some interesting problems not solved by SQIR and VOQC include verified compilation from high-level languages (e.g., Silq [5]) to SQIR circuits, and verified optimizations that operate on high-level programs (e.g., Li et al. [30] and Ittah et al. [24]). We have started work in this direction with VQO [31], a verified compiler from a high-level classical language to SQIR circuits, but there is still much to be done to support verified optimized compilation of general-purpose high-level quantum languages.

ACKNOWLEDGMENTS

We thank Akshaj Gaur for work on the Python bindings, Aaron Green for work on proving mapping preservation, Jake Zweifler for work on the QuantumLib library, and Le Chang for preliminary work on implementing a verified layout routine. We are also grateful to Leonidas Lampropoulos, Kartik Singhal, and anonymous reviewers for helpful comments on drafts of this paper. This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Quantum Testbed Pathfinder Program under Award Number DE-SC0019040, and the Air Force Office of Scientific Research under award number FA95502110051. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of these agencies.

REFERENCES

[1] Matthew Amy. 2018. Towards large-scale functional verification of universal quantum circuits. https://www.mathstat.dal.ca/qpl2018/papers/QPL_2018_paper_30.pdf. Presented at QPL 2018..

[2] Matthew Amy and Vlad Gheorghiu. 2020. staq – A full-stack quantum processing toolkit. *Quantum Science and Technology* 5, 3, Article 034016 (June 2020), 21 pages. https://doi.org/10.1088/2058-9565/ab9359 arXiv:1912.06070

[3] Matthew Amy, Dmitri Maslov, and Michele Mosca. 2013. Polynomial-time T-depth optimization of Clifford+T circuits via matroid partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33 (03 2013). https://doi.org/10.1109/TCAD.2014.2341953

[4] Matthew Amy, Martin Roetteler, and Krysta M. Svore. 2017. Verified compilation of space-efficient reversible circuits. In *Proceedings of the 28th International Conference on Computer Aided Verification (CAV 2017)*. Springer. https://www.microsoft.com/en-us/research/publication/verified-compilation-of-space-efficient-reversible-circuits/

[5] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: A High-Level Quantum Language with Safe Uncomputation and Intuitive Semantics. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. Association for Computing Machinery, New York, NY, USA, 286–300. https://doi.org/10.1145/3385412.3386007

[6] Lukas Burgholzer, Rudy Raymond, and Robert Wille. 2020. Verifying results of the IBM Qiskit quantum circuit compilation flow. In *International Conference on Quantum Computing and Engineering*.

[7] Lukas Burgholzer and Robert Wille. 2020. Advanced Equivalence Checking for Quantum Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 9 (10 2020). https://doi.org/10.1109/TCAD.2020.3032630

[8] Cambridge Quantum Computing Ltd. 2019. pytket. https://cqcl.github.io/pytket/build/html/index.html

[9] Chair for Design Automation at the Technical University of Munich. 2022. MQT QMAP - A tool for Quantum Circuit Mapping written in C++. https://github.com/cda-tum/qmap

[10] Andrew M. Childs, Eddie Schoute, and Cem M. Unsal. 2019. Circuit Transformations for Quantum Architectures. In *14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 135)*, Wim van Dam and Laura Mancinska (Eds.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 3:1–3:24. https://doi.org/10.4230/LIPIcs.TQC.2019.3

[11] The Coq Development Team. 2019. The Coq Proof Assistant, version 8.10.0. https://doi.org/10.5281/zenodo.3476303

[12] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 2017. Open Quantum Assembly Language. arXiv:1707.03429 https://github.com/Qiskit/openqasm/tree/OpenQASM2.x

[13] Cirq Developers. 2021. Cirq. https://doi.org/10.5281/zenodo.4586899 See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors.

[14] David Eberly. 1999. Euler Angle Formulas. https://www.geometrictools.com/Documentation/EulerAngles.pdf Included in documentation for Geometric Tools.

[15] Andrew Fagan and Ross Duncan. 2018. Optimising Clifford Circuits with Quantomatic. In *Proceedings of the 15th International Conference on Quantum Physics and Logic, QPL 2018, Halifax, Nova Scotia, 3-7 June 2018*.

[16] Daniel M. Greenberger, Michael A. Horne, and Anton Zeilinger. 1989. *Going Beyond Bell's Theorem.* Springer Netherlands, Dordrecht, 69–72. https://doi.org/10.1007/978-94-017-0849-4_10

[17] Luke Heyfron and Earl T. Campbell. 2018. An efficient quantum compiler that reduces T count. *Quantum Science and Technology* 4 (2018). https://doi.org/10.1088/2058-9565/aad604

[18] Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. 2021. Proving Quantum Programs Correct. In *12th International Conference on Interactive Theorem Proving (ITP 2021) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 193)*, Liron Cohen and Cezary Kaliszyk (Eds.). Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, Article 21, 21:1–21:19 pages. https://doi.org/10.4230/LIPIcs.ITP.2021.21

[19] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A Verified Optimizer for Quantum Circuits. *PACM on Programming Languages* 5, POPL, Article 37 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434318 arXiv:1912.02250

[20] IBM. 2022. IBM Quantum Processor Types. https://quantum-computing.ibm.com/lab/docs/iql/manage/systems/processors

[21] INRIA. 2021. Interfacing C with OCaml. https://ocaml.org/manual/intfc.html. Accessed: 2021-04-09.

[22] INRIA. 2022. Library Coq.Reals.Reals. https://coq.inria.fr/library/Coq.Reals.Reals.html Accessed: 2022-03-23.

[23] Inria, CNRS and contributors. 2021. Program Extraction. https://coq.inria.fr/refman/addendum/extraction.html. Accessed: 2021-09-24.

[24] David Ittah, Thomas Häner, Vadym Kliuchnikov, and Torsten Hoefler. 2022. QIRO: A Static Single Assignment-Based Quantum Program Representation for Optimization. *ACM Transactions on Quantum Computing* 3, 2 (Feb. 2022). https://doi.org/10.1145/3491247

[25] Ali Javadi-Abhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2015. ScaffCC: Scalable Compilation and Analysis of Quantum Programs. *Parallel Comput.* 45 (June 2015), 2–17. https://doi.org/10.1016/j.parco.2014.12.001 arXiv:1507.01902

[26] Aleks Kissinger and John van de Wetering. 2019. PyZX: Large Scale Automated Diagrammatic Reasoning. In *Proceedings of the 16th International Conference on Quantum Physics and Logic, QPL 2019*.

[27] Aleks Kissinger and John van de Wetering. 2019. Reducing T-count with the ZX-calculus. *arXiv e-prints* (2019). arXiv:1903.10477 [quant-ph]

[28] Xavier Leroy. 2009. Formal verification of a realistic compiler. *Commun. ACM* 52, 7 (July 2009), 107–115. https://doi.org/10/c9sb7q

[29] Gushu Li, Yufei Ding, and Yuan Xie. 2019. Tackling the Qubit Mapping Problem for NISQ-Era Quantum Devices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 1001–1014. https://doi.org/10.1145/3297858.3304023

[30] Gushu Li, Anbang Wu, Yunong Shi, Ali Javadi-Abhari, Yufei Ding, and Yuan Xie. 2022. Paulihedral: A Generalized Block-Wise Compiler Optimization Framework for Quantum Simulation Kernels. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS 2022)*. Association for Computing Machinery, New York, NY, USA, 554–569. https://doi.org/10.1145/3503222.3507715

[31] Liyi Li, Finn Voichick, Kesha Hietala, Yuxiang Peng, Xiaodi Wu, and Michael Hicks. 2021. Verified Compilation of Quantum Oracles. *arXiv e-prints*, Article arXiv:2112.06700 (Dec. 2021), arXiv:2112.06700 pages. arXiv:2112.06700 [quant-ph]

[32] D. M. Miller and M. A. Thornton. 2006. QMDD: A Decision Diagram Structure for Reversible and Quantum Circuits. In *36th International Symposium on Multiple-Valued Logic (ISMVL'06)*.

[33] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. 2018. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information* 4, 1 (2018), 23. https://doi.org/10.1038/s41534-018-0072-4

[34] Yuxiang Peng, Kesha Hietala, Runzhou Tao, Liyi Li, Robert Rand, Michael Hicks, and Xiaodi Wu. 2022. A Formally Certified End-to-End Implementation of Shor's Factorization Algorithm. *arXiv e-prints*, Article arXiv:2204.07112 (April 2022), arXiv:2204.07112 pages. arXiv:2204.07112 [cs.PL]

[35] John Preskill. 2018. Quantum computing in the NISQ era and beyond. *Quantum* 2 (Aug. 2018), 79. https://doi.org/10.22331/q-2018-08-06-79

[36] Python Software Foundation. 2021. ctypes – A foreign function library for Python. https://docs.python.org/3/library/ctypes.html. Accessed: 2021-04-09.

[37] Qiskit Community. 2017. Qiskit: An Open-Source Framework for Quantum Computing. https://doi.org/10.5281/zenodo.2562110

[38] Qiskit Development Team. 2021. Transpiler Passes (`qiskit.transpiler.passes`). https://qiskit.org/documentation/apidoc/transpiler_passes.html. Accessed: 2021-04-05.

[39] Robert Rand, Jennifer Paykin, and Steve Zdancewic. 2018. QWIRE Practice: Formal Verification of Quantum Circuits in Coq. In *Proceedings of the 14th International Conference on Quantum Physics and Logic (QPL), Nijmegen, the Netherlands, July 3–7, 2017 (Electronic Proceedings in Theoretical Computer Science, Vol. 266)*, Bob Coecke and Aleks Kissinger (Eds.). Open Publishing Association, Waterloo, NSW, Australia, 119–132. https://doi.org/10.4204/EPTCS.266.8

[40] Rigetti Computing. 2019. Pyquil Documentation. http://pyquil.readthedocs.io/en/latest/

[41] Rigetti Computing. 2019. The @rigetti optimizing Quil compiler. https://github.com/rigetti/quilc

[42] Mehdi Saeedi, Robert Wille, and Rolf Drechsler. 2011. Synthesis of quantum circuits for linear nearest neighbor architectures. *Quantum Information Processing* 10, 3 (01 June 2011), 355–377. https://doi.org/10.1007/s11128-010-0201-2

[43] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. t|ket⟩: a retargetable compiler for NISQ devices. *Quantum Science and Technology* 6, 1, Article 014003 (Nov. 2020), 014003 pages. https://doi.org/10.1088/2058-9565/ab8e92 arXiv:2003.10611

[44] Kaitlin N. Smith and Mitchell A. Thornton. 2019. A Quantum Computational Compiler and Design Tool for Technology-specific Targets. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA '19)*.

[45] Robert S. Smith, Michael J. Curtis, and William J. Zeng. 2016. A Practical Quantum Instruction Set Architecture. arXiv:1608.03355 https://github.com/rigetti/quil

[46] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: An Open Source Software Framework for Quantum Computing. *Quantum* 2, Article 49 (Jan. 2018), 49 pages. https://doi.org/10.22331/q-2018-01-31-49

[47] Swamit S. Tannu and Moinuddin K. Qureshi. 2019. Not all qubits are created equal: A case for variability-aware policies for NISQ-era quantum computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. https://doi.org/10.1145/3297858.3304007

[48] Runzhou Tao, Yunong Shi, Jianan Yao, Xupeng Li, Ali Javadi-Abhari, Andrew W. Cross, Frederic T. Chong, and Ronghui Gu. 2022. Giallar: Push-Button Verification for the Qiskit Quantum Compiler. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3519939.3523431 arXiv:2205.00661 To appear at PLDI 2022.

[49] Farrokh Vatan and Colin Williams. 2004. Optimal quantum circuits for general two-qubit gates. *Phys. Rev. A* 69 (March 2004), 032315. Issue 3. https://doi.org/10.1103/PhysRevA.69.032315

[50] Robert Wille, Lukas Burgholzer, and Alwin Zulehner. 2019. Mapping Quantum Circuits to IBM QX Architectures Using the Minimal Number of SWAP and H Operations. In *Design Automation Conference*.

[51] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler. 2008. RevLib: An Online Resource for Reversible Functions and Reversible Circuits. In *Int'l Symp. on Multi-Valued Logic*. 220–225. RevLib is available at http://www.revlib.org.

[52] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. 2022. Quartz: Superoptimization of Quantum Circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '22)*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/3519939.3523433 arXiv:2204.09033 To appear at PLDI 2022.

[53] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2017. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *arXiv e-prints* (Dec. 2017). arXiv:1712.04722 [quant-ph]

Table 6. Summary of optimizations available in Nam et al..

Nam et al.	
Not propagation (P)	✓*
Hadamard gate reduction (L, H)	✓
Single-qubit gate cancellation (L, H)	✓
Two-qubit gate cancellation (L, H)	✓
Rotation merging using phase polynomials (L)	✓*
Floating R_z gates (H)	
Special-purpose optimizations (L, H)	
• LCR optimizer	✓
• Toffoli decomposition	

A ADDITIONAL BENCHMARK RESULTS

This section contains the full results of the experiments described in Section 7.

A.1 Staq Benchmarks

Section 7 summarized the results of running Qiskit [37], t|ket⟩ [43], PyZX [26], and Staq [2] on 35 circuit programs used to evaluate Staq. Tables 7 to 9 show the full results. In each row, for each type of gate, we shade the cell of the best-performing optimizer. The geometric mean reduction for each gate type is given in the last row. Cases where PyZX hit our timeout (so we ran full_reduce, but not full_optimize) are marked with stars.

A.2 Nam et al. Benchmarks

In this section, we evaluate voqc's performance on all 99 benchmark programs considered by Nam et al. [33], confirming that voqc is a faithful implementation of a subset of the optimizations present in Nam et al. (along with being proved correct!). The benchmarks are divided into three categories, as described below. Our versions of the benchmarks are available online.[7] All results were obtained using a laptop with a 2.9 GHz Intel Core i5 processor and 16 GB of 1867 MHz DDR3 memory, running macOS Catalina. For timings, we take the median of three trials. We do not re-run Nam et al. (which is proprietary software), but instead report the results from Nam et al. [33]; they are from a similar machine with 8 GB RAM running OS X El Capitan. Their implementation is written in Fortran.

We summarize the optimizations available in Nam et al. in Table 6. P stands for "preprocessing" and L and H indicate whether the routine is in the "light" or "heavy" versions of the optimizer. voqc provides the complete and verified functionality of the routines marked with ✓; we write ✓* to indicate that voqc contains a verified optimization with similar, although not identical, behavior. We have not yet implemented "Toffoli decomposition" and "Floating R_z gates" and compared to Nam et al.'s rotation merging, voqc performs a slightly less powerful optimization (as discussed in Section 5.2).

In cases where Toffoli decomposition and heavy optimization are not used (the QFT, QFT-based adder, and product formula circuits), voqc's results are identical to Nam et al.'s. In the other cases, voqc is slightly less effective. In the worst case, voqc's running time is four orders of magnitude worse than Nam et al.'s. However, voqc's running time is often less than a second. We view this performance as acceptable, given that benchmarks with more than 1000 two-qubit

[7] https://github.com/inQWIRE/VOQC-benchmarks

gates (the only programs for which VOQC optimization takes longer than one second) are well out of reach of current quantum hardware [35]. We are optimistic that VOQC's performance can be improved through more careful engineering.

Arithmetic and Toffoli. These benchmarks overlap with the Staq benchmarks discussed in Section 7 (both originate from an earlier paper by Amy et al. [3]). The programs range from 45 to 346,533 gates and 5 to 489 qubits. The total gate count reduction and timing results for all 32 benchmarks are given in Table 10. In 12 out of 32 cases VOQC outperforms Nam et al., but VOQC has a lower average reduction. This is primarily due to Nam et al.'s "special-purpose Toffoli decomposition," which affects how *CCX* gates are decomposed. Their decomposition enables rotation merging and single-qubit gate cancellation to cancel two gates (e.g. cancel T and $T^\dagger$) where we instead combine two gates into one (e.g. T and T becomes S). Interestingly, the cases where VOQC outperforms Nam et al. can also be attributed to their Toffoli decomposition heuristic, which sometimes result in fewer cancellations than the naïve decomposition that we use. We do not expect adding and verifying this form of Toffoli decomposition to pose a challenge in VOQC. Reduction in T-gate count is not shown, but VOQC matches Nam et al. (both L and H) on all benchmarks but two. The first case (qcla_adder_10) is due to our simplification in rotation merging. In the second case (qcla_mod_7), Nam et al.'s optimized circuit was later found to be inequivalent to the original circuit [27, Section 2] (as marked by the red in Table 10), so the lower T-count is spurious.

QFT and Adders. These benchmarks consist of components of Shor's integer factoring algorithm, in particular the quantum Fourier transform (QFT) and integer adders. Two types of adders are considered: an in-place modulo $2q$ adder implemented in the Quipper library and an in-place adder based on the QFT. These benchmarks range from 148 to 381,806 gates and 8 to 4096 qubits. Results on all 27 benchmarks are given in Table 11, Table 12, and Table 13. The Quipper adder programs use similar gates to the arithmetic and Toffoli circuits, so the results are similar—VOQC is close to Nam et al., but under-performs due to our simplified Toffoli decomposition. The QFT circuits use rotations parameterized by $\pi/2^n$ for varying $n \in \mathbb{N}$ (and no Toffoli gates) so VOQC's results are identical to Nam et al.'s. For consistency with Nam et al., on the QFT and QFT-based adder circuits we run a simplified version of our optimizer that does not include rotation merging.

Product Formula. These benchmarks implement product formula algorithms for simulating Hamiltonian dynamics. The benchmarks range from 260 to 127,500 gates and 10 to 100 qubits; they use rotations parameterized by floating point numbers, which we convert to OCaml rationals at parse time. The product formula circuits are intended to be repeated for a fixed number of iterations, and our resource estimates account for this. VOQC applies Nam et al.'s "LCR" optimization routine to optimize programs across loop iterations. On all 40 product formula benchmarks, our results are the same as those reported by Nam et al. [33, Table 3]. H gate reductions range from 62.5% to 75%. Reductions in Clifford z-axis rotations (i.e. rotations by multiples of $\pi/2$) range from 75% to 87.5% while reductions in non-Clifford z-axis rotations range from 0% to 28.6%. CX gate reductions range from 0% to 33%. Running times range from 0.01s to 406.93s. By comparison, Nam et al.'s running times range from 0.004s to 0.137s.

Table 7. Reduced gate counts for the IBM gate set on the Staq [2] benchmarks. Shaded cells mark the best performance.

	Total Gate Count				2-qubit Gate Count			
Name	**Original**	**Qiskit**	**t|ket⟩**	**voqc**	**Original**	**Qiskit**	**t|ket⟩**	**voqc**
adder_8	934	820	806	643	409	385	383	337
barenco_tof_3	60	53	52	46	24	24	24	22
barenco_tof_4	120	104	100	89	48	48	46	44
barenco_tof_5	180	155	148	132	72	72	68	66
barenco_tof_10	480	410	388	347	192	192	178	176
csla_mux_3	170	154	141	146	80	69	69	72
csum_mux_9	448	403	366	308	168	168	168	168
cycle_17_3	9738	8397	7753	5963	3915	3903	3723	3001
gf2^4_mult	243	206	206	190	99	99	99	99
gf2^5_mult	379	318	319	289	154	154	154	154
gf2^6_mult	545	454	454	408	221	221	221	221
gf2^7_mult	741	614	614	547	300	300	300	300
gf2^8_mult	981	804	806	703	405	405	402	405
gf2^9_mult	1223	1006	1009	882	494	494	494	494
gf2^10_mult	1509	1238	1240	1080	609	609	609	609
gf2^16_mult	3885	3148	3150	2691	1581	1581	1581	1581
grover_5	831	669	605	526	288	288	288	248
ham15-low	443	406	390	351	236	236	226	220
ham15-med	1272	1105	1015	820	534	533	498	434
ham15-high	5308	4589	4371	3532	2149	2143	2110	1853
hwb6	257	236	223	205	116	115	111	108
mod_adder_1024	4285	3721	3542	2832	1720	1702	1702	1402
mod_mult_55	119	110	100	83	48	48	48	40
mod_red_21	272	237	226	191	105	105	105	93
mod5_4	65	56	56	53	28	28	28	28
qcla_adder_10	539	477	441	408	233	213	213	207
qcla_com_7	463	407	363	292	186	174	174	148
qcla_mod_7	920	813	752	666	382	366	366	338
qft_4	179	97	81	94	46	44	38	46
rc_adder_6	200	177	159	141	93	81	81	73
tof_3	45	41	40	36	18	18	18	16
tof_4	75	68	66	58	30	30	30	26
tof_5	105	95	92	80	42	42	42	36
tof_10	255	230	222	190	102	102	102	86
vbe_adder_3	160	133	121	100	70	58	58	54
Geo. Mean Red.	–	14.4%	18.5%	28.5%	–	2.3%	3.8%	9.8%

Table 8. Reduced gate counts for the RzQ gate set on the Staq [2] benchmarks. Shaded cells mark the best performance. PyZX often increases both total and two-qubit gate count (hence the negative reduction for both). Some PyZX results are marked with a star* to indicate that full_optimize exceeded our time limit, so we only used full_reduce.

	Total Gate Count				*T* Gate Count				2-qubit Gate Count			
Name	**Original**	**Staq**	**PyZX**	**voqc**	**Original**	**Staq**	**PyZX**	**voqc**	**Original**	**Staq**	**PyZX**	**voqc**
adder_8	934	756	911	682	399	179	167	215	409	382	609	337
barenco_tof_3	60	48	53	50	28	16	16	16	24	20	28	22
barenco_tof_4	120	90	88	95	56	28	28	28	48	38	44	44
barenco_tof_5	180	132	184	140	84	40	40	40	72	56	116	66
barenco_tof_10	480	342	460	365	224	100	100	100	192	146	299	176
csla_mux_3	170	174	291	156	70	62	47	64	80	88	201	72
csum_mux_9	448	294	508	308	196	84	76	84	168	126	371	168
cycle_17_3	9738	7143	11157^*	6314	4529	1821	1821^*	1821	3915	3351	6229^*	3001
gf2^4_mult	243	242	277	192	112	66	50	68	99	141	211	99
gf2^5_mult	379	366	650	291	175	113	92	115	154	209	526	154
gf2^6_mult	545	540	1309^*	410	252	148	150^*	150	221	327	789^*	221
gf2^7_mult	741	714	1843^*	549	343	215	217^*	217	300	423	1162^*	300
gf2^8_mult	981	968	2610^*	705	448	262	264^*	264	405	603	1745^*	405
gf2^9_mult	1223	1178	3162^*	885	567	348	351^*	351	494	713	2065^*	494
gf2^10_mult	1509	1484	4118^*	1084	700	406	410^*	410	609	927	2779^*	609
gf2^16_mult	3885	3800	11627^*	2695	1792	1032	1040^*	1040	1581	2427	8368^*	1581
grover_5	831	678	696	586	336	166	166	172	288	263	395	248
ham15-low	443	417	650	373	161	97	97	97	236	252	477	220
ham15-med	1272	989	1205	880	574	242	211	248	534	483	821	434
ham15-high	5308	3967	5084^*	3739	2457	1021	1019^*	1049	2149	1870	2343^*	1853
hwb6	257	237	274	221	105	75	74	75	116	115	163	108
mod_adder_1024	4285	3401	5572^*	3039	1995	1011	1011^*	1011	1720	1584	2813^*	1402
mod_mult_55	119	107	123	90	49	37	28	35	48	42	82	40
mod_red_21	272	235	340	214	119	73	72	73	105	94	211	93
mod5_4	65	53	27	56	28	8	8	16	28	28	14	28
qcla_adder_10	539	445	894^*	438	238	162	162^*	164	233	209	382^*	207
qcla_com_7	463	329	473	314	203	95	92	95	186	146	312	148
qcla_mod_7	920	725	1426	723	413	237	226	249	382	334	1012	338
qft_4	179	170	191	156	91	50	66	67	46	56	72	46
rc_adder_6	200	173	256	157	77	47	47	47	93	81	162	73
tof_3	45	40	55	40	21	15	15	15	18	16	31	16
tof_4	75	65	83	65	35	23	23	23	30	26	48	26
tof_5	105	90	121	90	49	31	31	31	42	36	68	36
tof_10	255	215	338	215	119	71	71	71	102	86	213	86
vbe_adder_3	160	101	155	101	70	24	24	24	70	54	105	54
Geo. Mean Red.	–	15.4%	-27.8%	23.2%	–	45.4%	47.1%	43.1%	–	0.6%	-91.7%	9.8%

Table 9. Median running times (in seconds) on Staq benchmarks.

Name	# qubits	# gates	Qiskit	t|ket⟩	Staq	PyZX	voqc
adder_8	24	934	1.43	4.47	0.07	19.16	0.05
barenco_tof_3	5	60	0.09	0.18	<0.01	2.24	<0.01
barenco_tof_4	7	120	0.17	0.38	<0.01	2.54	<0.01
barenco_tof_5	9	180	0.26	0.58	<0.01	0.58	<0.01
barenco_tof_10	19	480	0.63	1.49	0.02	63.04	0.02
csla_mux_3	15	170	0.29	0.75	0.01	72.74	<0.01
csum_mux_9	30	448	0.58	1.52	0.02	45.87	0.01
cycle_17_3	35	9738	13.08	41.56	1.63	498.51*	5.74
gf2^4_mult	12	243	0.32	1.04	0.01	57.25	0.01
gf2^5_mult	15	379	0.50	1.65	0.02	0.23	0.01
gf2^6_mult	18	545	0.70	2.22	0.04	0.43*	0.02
gf2^7_mult	21	741	1.13	3.03	0.08	0.73*	0.04
gf2^8_mult	24	981	1.28	4.16	0.14	1.48*	0.07
gf2^9_mult	27	1223	1.62	5.04	0.22	1.94*	0.11
gf2^10_mult	30	1509	2.09	6.22	0.35	2.95*	0.17
gf2^16_mult	48	3885	5.28	17.37	3.30	25.41*	1.24
grover_5	9	831	1.01	2.37	0.03	34.30	0.01
ham15-low	17	443	0.67	2.69	0.02	119.85	0.01
ham15-med	17	1272	1.88	4.53	0.07	261.71	0.05
ham15-high	20	5308	7.53	20.59	0.33	23.31*	0.66
hwb6	7	257	0.38	1.04	0.01	58.54	<0.01
mod_adder_1024	28	4285	5.95	15.56	0.37	171.43*	1.31
mod_mult_55	9	119	0.18	0.47	<0.01	5.62	<0.01
mod_red_21	11	272	0.37	0.90	0.01	54.76	0.01
mod5_4	5	65	0.10	0.18	<0.01	2.17	<0.01
qcla_adder_10	36	539	0.80	1.97	0.04	1.86*	0.02
qcla_com_7	24	463	0.65	1.50	0.02	79.69	0.01
qcla_mod_7	26	920	1.42	3.33	0.08	411.36	0.06
qft_4	5	179	0.19	0.30	<0.01	14.83	<0.01
rc_adder_6	14	200	0.31	0.92	0.01	14.38	<0.01
tof_3	5	45	0.07	0.15	<0.01	8.68	<0.01
tof_4	7	75	0.11	0.24	<0.01	13.22	<0.01
tof_5	9	105	0.15	0.33	<0.01	25.43	<0.01
tof_10	19	255	0.35	0.81	0.01	75.79	0.01
vbe_adder_3	10	160	0.24	0.51	<0.01	19.21	<0.01
Geo. Mean	–	–	0.60	1.53	0.02	14.58	0.01

Table 10. Total gate count reduction on the “Arithmetic and Toffoli” circuits. Nam (H) results were not available for the large benchmarks. Red cells indicate programs optimized incorrectly. Bold results mark the best performance.

	Orig.	**Nam (L)**		**Nam (H)**		**VOQC**	
Name	**Total**	**Total**	**t (s)**	**Total**	**t (s)**	**Total**	**t (s)**
adder_8	900	646	0.004	**606**	0.101	682	0.048
barenco_tof_3	58	42	<0.001	**40**	0.001	50	0.001
barenco_tof_4	114	78	<0.001	**72**	0.001	95	0.002
barenco_tof_5	170	114	<0.001	**104**	0.003	140	0.003
barenco_tof_10	450	294	0.001	**264**	0.012	365	0.019
csla_mux_3	170	161	<0.001	**155**	0.009	158	0.003
csum_mux_9	420	294	<0.001	**266**	0.009	308	0.006
gf2^4_mult	225	**187**	0.001	**187**	0.009	192	0.006
gf2^5_mult	347	296	0.001	296	0.020	**291**	0.012
gf2^6_mult	495	**403**	0.003	**403**	0.047	410	0.025
gf2^7_mult	669	555	0.004	555	0.105	**549**	0.045
gf2^8_mult	883	712	0.006	712	0.192	**705**	0.070
gf2^9_mult	1095	891	0.010	891	0.347	**885**	0.119
gf2^10_mult	1347	**1070**	0.009	**1070**	0.429	1084	0.183
gf2^16_mult	3435	2707	0.065	2707	5.566	**2695**	1.347
gf2^32_mult	13593	10601	1.834	10601	275.698	**10577**	26.808
gf2^64_mult	53691	41563	58.341	–	–	**41515**	546.887
gf2^128_mult	213883	165051	1744.746	–	–	**164955**	9841.797
gf2^131_mult	224265	173370	1953.353	–	–	**173273**	10877.112
gf2^163_mult	346533	267558	4955.927	–	–	**267437**	27612.565
mod5_4	63	**51**	<0.001	**51**	0.001	56	<0.001
mod_mult_55	119	91	<0.001	91	0.002	**90**	0.002
mod_red_21	278	184	<0.001	**180**	0.008	214	0.005
qcla_adder_10	521	411	0.002	**399**	0.044	438	0.018
qcla_com_7	443	**284**	0.001	**284**	0.016	314	0.013
qcla_mod_7	884	636	0.004	624	0.077	**723**	0.058
rc_adder_6	200	142	<0.001	**140**	0.004	157	0.003
tof_3	45	**35**	<0.001	**35**	<0.001	40	<0.001
tof_4	75	**55**	<0.001	**55**	<0.001	65	0.001
tof_5	105	**75**	<0.001	**75**	0.001	90	0.002
tof_10	255	**175**	<0.001	**175**	0.004	215	0.006
vbe_adder_3	150	**89**	<0.001	**89**	0.001	101	0.002
Geo. Mean Red.	–	24.6%		26.4%		19.2%	

Table 11. Total gate count reduction on Quipper adder circuits. voqc's H and T counts are identical to Nam (L) and (H), but the total Rz and CX counts are higher due to Nam et al.'s specialized Toffoli decomposition. The difference between Nam (L) and Nam (H) is entirely due to CX count. Our initial gate counts are higher than those reported by Nam et al. because we do not have special handling for +/- control Toffoli gates; we simply consider the standard Toffoli gate conjugated by additional X gates.

	Original	**Nam (L)**		**Nam (H)**		**voqc**	
n	**Total**	**Total**	**t (s)**	**Total**	**t (s)**	**Total**	**t (s)**
8	585	239	0.001	190	0.006	352	0.02
16	1321	527	0.003	414	0.018	784	0.12
32	2793	1103	0.014	862	0.066	1648	0.63
64	5737	2255	0.057	1758	0.598	3376	3.30
128	11625	4559	0.244	3550	4.697	6832	16.37
256	23401	9167	1.099	7134	34.431	13744	79.74
512	46953	18383	5.292	14302	307.141	27568	394.74
1024	94057	36815	25.987	28638	2446.336	55216	1894.41
2048	188265	73679	145.972	57310	23886.841	110512	9307.36
Avg. Red.	–	63.7%		71.6%		45.7%	

Table 12. Results on QFT circuits. Exact timings and gate counts are not available for Nam (L) or Nam (H), but our gate counts are consistent with those reported in Nam et al. [33, Figure 1].

	Original			**voqc**			
n	CX	R_z	H	CX	R_z	H	**t (s)**
8	56	84	8	56	42	8	<0.01
16	228	342	16	228	144	16	<0.01
32	612	918	32	612	368	32	0.01
64	1380	2070	64	1380	816	64	0.07
128	2916	4374	128	2916	1712	128	0.39
256	5988	8982	256	5988	3504	256	2.34
512	12132	18198	512	12132	7088	512	15.69
1024	24420	36630	1024	24420	14256	1024	106.71
2048	48996	73494	2048	48996	28592	2048	674.11
Avg. Red.	–	–	–	0%	59.3%	0%	

Table 13. Results on QFT-based adder circuits. Final gate counts are identical for voqc and Nam (L).

	Original			**voqc**				**Nam (L)**
n	CX	R_z	H	CX	R_z	H	**t (s)**	**t (s)**
8	184	276	16	184	122	16	<0.01	<0.001
16	716	1074	32	716	420	32	0.02	0.001
32	1900	2850	64	1900	1076	64	0.13	0.002
64	4268	6402	128	4268	2388	128	0.90	0.004
128	9004	13506	256	9004	5012	256	5.52	0.08
256	18476	27714	512	18476	10260	512	36.80	0.018
512	37420	56130	1024	37420	20756	1024	255.20	0.045
1024	75308	112962	2048	75308	41748	2048	1695.65	0.115
2048	151084	226626	4096	151084	83732	4096	8481.66	0.215
Avg. Red.	–	–	–	0%	61.8%	0%		