Applying and Expanding the VOQC Toolkit

KESHA HIETALA, University of Maryland, USA
LIYI LI, University of Maryland, USA
AKSHAJ GAUR, Poolesville High School, USA
AARON GREEN, University of Maryland, USA
ROBERT RAND, University of Chicago, USA
XIAODI WU, University of Maryland, USA
MICHAEL HICKS, University of Maryland, USA

This abstract presents recent extensions to VOQC, a *verified optimizer for quantum circuits*, first presented at POPL 2021 [Hietala et al. 2021b]. All code described in this abstract is freely available online.[1]

1 OVERVIEW

VOQC [Hietala et al. 2021b] (pronounced "vox") is a compiler for quantum circuits, in the style of tools like Qiskit [Aleksandrowicz et al. 2019], tket [Cambridge Quantum Computing Ltd 2019], Quilc [Rigetti Computing 2019], and Cirq [Developers 2021]. What makes VOQC different from these tools is that it has been *formally verified* in the Coq proof assistant [Coq Development Team 2019]. VOQC source programs are expressed in SQIR, a *simple quantum intermediate representation*, which has a precise mathematical semantics. We use Gallina, Coq's programming language, to implement VOQC transformations over SQIR programs, and use Coq to prove the source program's semantics are preserved. We then *extract* these Gallina definitions to OCaml, and compile the OCaml code to a library that can operate on standard-formatted circuits.

VOQC, and SQIR, were built to be general-purpose. For example, while we originally designed SQIR for use in verified optimizations, we subsequently found SQIR could also be suitable for writing, and proving correct, source programs [Hietala et al. 2021a]. We have continued to develop the VOQC codebase to expand its reach and utility.

In this abstract, we present new extensions to VOQC as an illustration of its flexibility. These include support for calling VOQC transformations from Python, added support for new gate sets and optimizations, and the extension of our notion of correctness to include *mapping-preservation*, which allows us to apply optimizations after mapping, reducing the cost introduced by making a program conform to hardware constraints.

2 PYVOQC

In order to make VOQC compatible with existing Python-based frameworks for compiling quantum programs (e.g. Qiskit, pytket, Quilc, Cirq), we provide a Python wrapper (dubbed `pyvoqc`) around the VOQC OCaml library. To interface between Python and OCaml, we wrap the OCaml code in a C library (following standard conventions [INRIA 2021]) and call to this C library using Python's

[1] Software links:

- Our Coq definitions and proofs are available at https://github.com/inQWIRE/SQIR.
- Our OCaml library is available at https://github.com/inQWIRE/mlvoqc and can be installed with "opam install voqc".
- Documentation on the OCaml library interface is available at https://inqwire.github.io/mlvoqc/voqc/Voqc/index.html.
- Our Python bindings and tutorials are available at https://github.com/inQWIRE/pyvoqc.

Authors' addresses: Kesha Hietala, University of Maryland, USA, kesha@cs.umd.edu; Liyi Li, University of Maryland, USA, liyili2@umd.edu; Akshaj Gaur, Poolesville High School, USA, akshajgaur@gmail.com; Aaron Green, University of Maryland, USA, agreen19@umd.edu; Robert Rand, University of Chicago, USA, rand@uchicago.edu; Xiaodi Wu, University of Maryland, USA, xwu@cs.umd.edu; Michael Hicks, University of Maryland, USA, mwh@cs.umd.edu.

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

Standard	Single-qubit gates:	I, X, Y, Z, H, S, T, Sdg, Tdg, Rx(r), Ry(r), Rz(r), Rzq(q), U1(r), U2(r, r), U3(r, r, r)
	Two-qubit gates:	CX, CZ, SWAP
	Three-qubit gates:	CCX, CCZ
RzQ	Single-qubit gates:	X, H, Rzq(q)
	Two-qubit gates:	CX
IBM	Single-qubit gates:	U1(r), U2(r, r), U3(r, r, r)
	Two-qubit gates:	CX

`ctypes` [Python Software Foundation 2021]. For convenience, we have written Python code that makes voqc look like an optimization pass in IBM's Qiskit or Google's Cirq, allowing us to take advantage of these frameworks' utilities for quantum programming (e.g. constructing and printing circuits, unverified optimizations and mapping routines). We show an example of using voqc as a Qiskit pass in Section 3.

3 SUPPORT FOR ADDITIONAL GATE SETS

When we first presented voqc we defined our optimizations over the "RzQ" gate set [Hietala et al. 2021b, §4.2], but argued that our work could be applied to other gate sets as well since many of our utility definitions and proofs are gate-set independent. Here, we demonstrate this extensibility for two new gate sets: the "standard" gate set and the "IBM" gate set, both shown in Table 1. The standard gate set is used for parsing and aims for completeness: Instead of having to translate a T gate in the source program to the semantically equivalent `Rz(PI/4)`, we can translate it directly to `T`. Likewise, we can translate the three-qubit CCX gate directly to `CCX`, rather than decomposing it into a series of one- and two-qubit gates (potentially incorrectly).

The IBM gate set is the default basis for the Qiskit compiler.[2] It includes the two-qubit controlled-NOT (CX) gate, along with three parameterized single-qubit gates:

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

One interesting property of this gate set (which does not hold of our original RzQ gate set) is that any sequence of single-qubit gates can be combined into a single gate. This allows us to implement an optimization (which Qiskit calls `Optimize1qGates`) that merges all adjacent single-qubit gate by applying rules like $U_1(\lambda_1); U_1(\lambda_2) \to U_1(\lambda_1 + \lambda_2)$ and $U_1(\lambda_1); U_2(\phi, \lambda_2) \to U_2(\lambda_2, \lambda_1 + \phi)$.

The most complicated rule for merging gates is the one for combining a U_2 and U_3 gate or two U_3 gates. In this case, the two gates are first converted into a sequence of Euler rotations [Euler 1776] about the y- and z-axes, e.g. $U_3(\theta,\phi,\lambda) = R_z(\phi) \cdot R_y(\theta) \cdot R_z(\lambda)$. We will 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,

[2] Actually, the U_1, U_2, and U_3 gates are used in many quantum compilers. We called this the "IBM" gate set because, at the time, we were aiming to verify a Qiskit optimization.

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

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

# create a Qiskit PassManager
pm = PassManager()

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

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

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

```
Before Optimization:
q_0: ┤ X ├┤ T ├──■──┤ T ├
     ├───┤└───┘  │  └───┘
q_1: ┤ T ├───────■──┤TDG├
     └───┘          └───┘

After 'decompose_to_cnot':
q_0: ┤ X ├┤ T ├──■──┤ T ├──────
     ├───┤└───┘  │  └───┘
q_1: ┤ T ├┤ H ├──■──┤ H ├┤TDG├
     └───┘└───┘     └───┘└───┘

After 'optimize_nam':
q_0: ──────■──┤SDG├┤ X ├
           │  └───┘└───┘
q_1: ┤ H ├──■──┤ H ├┤ Z ├
     └───┘     └───┘└───┘

After 'optimize_ibm':
q_0: ──────────■──┤U3(pi,0,pi/2)├
     ┤U2(0,pi)├──■──┤U2(pi,pi)├
q_1: └────────┘     └─────────┘
```

Fig. 1. Example of using VOQC as a Qiskit pass. The output from the script on the left is shown on the right.

here is the process for combining two U_3 gates:

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

where α, β, γ satisfy $R_y(\theta_2)\cdot R_z(\lambda_2+\phi_1)\cdot R_y(\theta_1) = R_z(\gamma)\cdot R_y(\beta)\cdot R_z(\alpha)$.

Importantly, we were able to define and verify this optimization using the same infrastructure we had developed for our original RzQ gate set, showing that our framework is indeed extensible. The most difficult part of the proof was showing the correctness of the gate combination rules. In particular, the method for converting from a YZY to ZYZ rotation (shown in Figure 2) was challenging to verify because it contains many cases, all of which involve complicated trigonometric expressions. To our knowledge, we are the first to formally verify this method in a proof assistant.

In total, VOQC supports the 9 optimizations listed in Table 2. We show the effect of applying `optimize_nam` and `optimize_ibm` (as Qiskit passes) on an example circuit in Figure 1. In addition, VOQC provides a `simple_map` function that takes as input a circuit, a description of the underlying architecture connectivity, and an initial mapping from the circuit's qubits to machine qubits, and returns a program that respects the constraints of the architecture (see Section 4 for more details). Our `simple_map` routine is effectively the same as Qiskit's `BasicSwap` pass [Qiskit Development Team 2021].

```
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_minus (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_minus 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. 2. Code for converting a YZY rotation to a ZYZ rotation.

4 INTERLEAVING MAPPING AND OPTIMIZATION

Near-term machines only allow two-qubit gates to be applied between certain pairs of qubits and in particular orientations. For example, in IBM's 5-qubit Tenerife machine (shown on the right), a CX gate may be applied with Q4 as the control and Q2 as the target, but not the reverse. No two-qubit gate is possible between physical qubits Q4 and Q1. So the program `CX Q4 Q1` will need to be transformed to, e.g., `SWAP Q2 Q4; CX Q2 Q1` in order to be executed on the machine.[3]

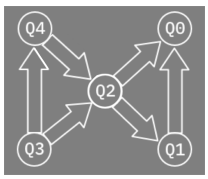

Fig. 3. Connectivity on IBM's 5-qubit Tenerife machine.

Circuit mapping automates this process, taking as input a circuit and architecture connectivity graph, and returning a transformed circuit that respects the constraints of the architecture [Saeedi et al. 2011; Zulehner et al. 2017]. Circuit mapping increases the number of gates, typically adding many CX and H gates to perform SWAPs between qubits.

[3]The SWAP gate will be decomposed to `CX Q4 Q2; CX Q2 Q4; CX Q4 Q2`, which is transformed into `CX Q4 Q2; H Q2; H Q4; CX Q4 Q2; H Q2; H Q4; CX Q4 Q2` to respect architecture constraints.

Table 2. Optimizations available in VOQC.

Name	Description	Gate Set
`not_propagation`	[Hietala et al. 2021b, §4.3]	RzQ
`hadamard_reduction`	[Hietala et al. 2021b, §4.4]	RzQ
`cancel_single_qubit_gates`	[Hietala et al. 2021b, §4.3]	RzQ
`cancel_two_qubit_gates`	[Hietala et al. 2021b, §4.3]	RzQ
`merge_rotations`	[Hietala et al. 2021b, §4.4]	RzQ
`optimize_nam`	Applies all RzQ optimizations in the ordering described in Hietala et al. [2021b, §4.6]	RzQ
`optimize_1q_gates`	Implementation of Qiskit's `Optimize1qGates` [Qiskit Development Team 2021]	IBM
`cx_cancellation`	Implementation of Qiskit's `CXCancellation` [Qiskit Development Team 2021]	IBM
`optimize_ibm`	Applies `optimize_1q_gates` followed by `cx_cancellation`	IBM

It is desirable to reduce this overhead by applying optimization after mapping. However, this is only worthwhile if the optimization preserves the guarantee from mapping that all CX gates are allowed by the connectivity graph. We have verified that all of the optimizations in Table 2, except `hadamard_reduction`, preserve connectivity guarantees. We call this property *mapping-preservation*, in contrast to the standard property that we prove, *semantics-preservation*, which that says that an optimization does not change the behavior ("semantics") of the input program.

5 ONGOING WORK

We are working to extend VOQC with more gates, optimizations, and mapping routines, taking inspiration from frameworks like Qiskit, tket, Quilc, and Cirq. We are especially interested in implementing and verifying more sophisticated circuit mappers and adding support for *approximate* optimizations that do not preserve semantics exactly, but instead return a lower-cost circuit with similar behavior (e.g. [Peterson 2021]). Our current mapping routine is quite simple compared to state-of-the-art mappers, which involve complex subroutines like A* search [Zulehner et al. 2017] or Steiner tree approximations [Nash et al. 2020]. To avoid verifying the entirety of these algorithms, we are exploring approaches to verified translation validation of their components.

One major limitation of our current work with the IBM gate set is that we have proved optimizations correct for gates that use Coq real numbers as parameters. Because Coq reals are axiomatized, there is no way to extract our Coq definitions to OCaml without providing an implementation of real arithmetic. For simplicity (and compatibility with existing frameworks), we have chosen to extract Coq reals to OCaml floats. This is not ideal because it allows for the possibility of rounding error not accounted for in our proofs. We previously avoided this issue by using rational gate parameters (which can be extracted to OCaml multi-precision rationals), but this is not sufficient for the IBM single-qubit gate optimization, which involves trigonometric functions that are not defined over rationals (Figure 2). One possible solution is to verify our optimizations over gates that use Coq float parameters (e.g. using the Floqc library [Boldo and Melquiond 2011]).

ACKNOWLEDGMENTS

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 Grant No. FA95502110051.

REFERENCES

Gadi Aleksandrowicz, Thomas Alexander, Panagiotis Barkoutsos, Luciano Bello, Yael Ben-Haim, David Bucher, Francisco Jose Cabrera-Hernández, Jorge Carballo-Franquis, Adrian Chen, Chun-Fu Chen, Jerry M. Chow, Antonio D. Córcoles-Gonzales, Abigail J. Cross, Andrew Cross, Juan Cruz-Benito, Chris Culver, Salvador De La Puente González, Enrique De La Torre, Delton Ding, Eugene Dumitrescu, Ivan Duran, Pieter Eendebak, Mark Everitt, Ismael Faro Sertage, Albert Frisch, Andreas Fuhrer, Jay Gambetta, Borja Godoy Gago, Juan Gomez-Mosquera, Donny Greenberg, Ikko Hamamura, Vojtech Havlicek, Joe Hellmers, Łukasz Herok, Hiroshi Horii, Shaohan Hu, Takashi Imamichi, Toshinari Itoko, Ali Javadi-Abhari, Naoki Kanazawa, Anton Karazeev, Kevin Krsulich, Peng Liu, Yang Luh, Yunho Maeng, Manoel Marques, Francisco Jose Martín-Fernández, Douglas T. McClure, David McKay, Srujan Meesala, Antonio Mezzacapo, Nikolaj Moll, Diego Moreda Rodríguez, Giacomo Nannicini, Paul Nation, Pauline Ollitrault, Lee James O'Riordan, Hanhee Paik, Jesús Pérez, Anna Phan, Marco Pistoia, Viktor Prutyanov, Max Reuter, Julia Rice, Abdón Rodríguez Davila, Raymond Harry Putra Rudy, Mingi Ryu, Ninad Sathaye, Chris Schnabel, Eddie Schoute, Kanav Setia, Yunong Shi, Adenilton Silva, Yukio Siraichi, Seyon Sivarajah, John A. Smolin, Mathias Soeken, Hitomi Takahashi, Ivano Tavernelli, Charles Taylor, Pete Taylour, Kenso Trabing, Matthew Treinish, Wes Turner, Desiree Vogt-Lee, Christophe Vuillot, Jonathan A. Wildstrom, Jessica Wilson, Erick Winston, Christopher Wood, Stephen Wood, Stefan Wörner, Ismail Yunus Akhalwaya, and Christa Zoufal. 2019. Qiskit: An open-source framework for quantum computing. https://doi.org/10.5281/zenodo.2562110

S. Boldo and G. Melquiond. 2011. Flocq: A Unified Library for Proving Floating-Point Algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic*. 243–252. https://doi.org/10.1109/ARITH.2011.40

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

The Coq Development Team. 2019. The Coq proof assistant, version 8.10.0. https://doi.org/10.5281/zenodo.3476303

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.

Leonhard Euler. 1776. *Formulae generales pro translatione quacunque corporum rigidorum*. Novi Commentarii academiae scientiarum Petropolitanae 20, 189–207.

Kesha Hietala, Robert Rand, Shih-Han Hung, Liyi Li, and Michael Hicks. 2021a. Proving Quantum Programs Correct. In *Proceedings of The International Conference on Interactive Theorem Proving (ITP 2021)*. to appear.

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021b. A Verified Optimizer for Quantum Circuits. *Proceedings of the ACM on Programming Languages* 5, 37 (2021).

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

Beatrice Nash, Vlad Gheorghiu, and Michele Mosca. 2020. Quantum circuit optimizations for NISQ architectures. *Quantum Science and Technology* 5, 2 (mar 2020), 025010. https://doi.org/10.1088/2058-9565/ab79b1

Eric Peterson. 2021. quilc approx.lisp. https://github.com/rigetti/quilc/blob/master/src/compilers/approx.lisp. Accessed: 2021-04-09.

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

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

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

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

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]