# Transpiling OpenQASM 3.0 Programs to CUDA-Q Kernels

ALVAN CALEB ARULANDU, Harvard University, USA

We present a Python transpiler that generates CUDA-Q kernels from OpenQASM 3.0 source programs. Leveraging existing OpenQASM parsing infrastructure, we support a significant portion of the OpenQASM 3.0 specification including custom gates, control/adjoint modifiers, binary expression arguments, and more. Our transpiler passes a custom unit test suite in addition to randomized testing on Clifford circuits of varying input size and depth. Through the course of development, we also contribute to open source tooling for circuit optimization, semantic analysis, and the unrolling of OpenQASM programs.

CCS Concepts: • **Computer systems organization** → **Quantum computing**; • **Software and its engineering** → *Compilers*.

Additional Key Words and Phrases: Quantum Computing, Transpilers, Circuit Optimization, Unrolling

**ACM Reference Format:**
Alvan Caleb Arulandu. 2024. Transpiling OpenQASM 3.0 Programs to CUDA-Q Kernels. *J. ACM* ??, ?, Article ??? (December 2024), 8 pages. https://doi.org/XXXXXXX.XXXXXXX

## 1 INTRODUCTION

Three days prior to the submission of this work, Google Quantum AI announced Willow [Neven 2024], a state-of-the-art quantum chip achieving record performance for random circuit sampling (RCS), a widely used classically hard benchmark. A few years earlier, Google's formerly claimed supremacy result [Arute et al. 2019] was met with criticism after random circuit sampling in the presence of a constant gate noise rate was shown to admit a polynomial-time classical algorithm via low-degree approximation [Aharonov et al. 2023]. Now, Google's claims that RCS for depth 40 circuits takes $10^{25}$ years on SoTA classical supercomputers [Acharya et al. 2024] have been met with skepticism. Nevertheless, regardless of the precision of the figure, the point stands that there is reason to believe that certain computational tasks, arbitrarily contrived for the time being, admit *physically-realizable* efficient quantum algorithms but can not be classically solved efficiently, that is in polynomial time.

Realizability aside, the complexity class BQP captures all decision problems solvable by a quantum computer in polynomial time with bounded error. While $\mathrm{P} \subseteq \mathrm{BQP}$, strict inclusion remains unproven, as is natural with a plethora of complexity theoretic questions including $\mathrm{P} = \mathrm{NP}$. Besides the seminal result [Shor 1999] showing polynomial quantum algorithms for factoring, the discrete logarithm, and the Fast Fourier Transform, modern complexity theory simultaneously conjectures that certain oracle problems are in BQP yet outside the polynomial hierarchy and many NP-complete problems, such as SAT and the traveling salesman, are not in BQP. Thus, common belief is that BQP and $\mathrm{NP} \subseteq \mathrm{PH}$ are incomparable and hope for meaningful quantum supremacy lies in $\mathrm{NP} \setminus (\mathrm{P} \cup \text{NP-complete})$, which contains problems like factoring.

Author's address: Alvan Caleb Arulandu, aarulandu@college.harvard.edu, Harvard University, Cambridge, Massachusetts, USA.


ACM 0004-5411/2024/12-ART???
https://doi.org/XXXXXXX.XXXXXXX

## 2 RECENT WORK

As experimentalists progress toward realizing noisy intermediate-scale quantum (NISQ) devices and theorists work toward error correction goals and resource-efficient algorithms, in the last decade, quantum software infrastructure has exploded to accommodate. High-level Python-embedded domain specific languages (DSL) are most popular, including IBM's Qiskit [Javadi-Abhari et al. 2024], Xanadu's Pennylane [Bergholm et al. 2018], Google's Cirq [Heim et al. 2020], and Rigetti's PyQuil [Computing 2019], though Microsoft's Q#, syntactically related to C# and F#, has also gained traction.

The primary use of such DSLs is to synthesize and manipulate quantum circuits. Quantum circuits specified in supported DSLs are then evaluated by either classical simulators or quantum hardware providers. While ideal state vector and unitary simulators exactly evolve a quantum state according to a quantum circuit, such simulations are limited to low qubit counts as the description of a quantum system grows exponentially in the number of qubits. Various noisy simulators surpass this using established classical techniques such as tensor networks [Yuan et al. 2021].

These development frameworks have also evolved to support practical research. Google's standalone project OpenFermion [McClean et al. 2020] specializes in quantum chemistry, with Qiskit [Sharkey et al. 2022] and Pennylane [Arrazola et al. 2021] offering their own respective chemistry modules. Recent years have seen an increased focus in quantum machine learning (QML). Google's standalone project TensorFlow Quantum [Broughton et al. 2020] delivers QML features to Cirq, with Qiskit and Pennylane integrating with existing ML frameworks. Most notably, Xanadu's Catalyst project [Ittah et al. 2024] brings JIT compilation with JAX for auto-differentiable hybrid quantum programs and is scheduled to be up-streamed into Pennylane in a future release.

Quantum hardware is even more fragmented. Physical constraints result in different hardware topologies, where each device only permits certain interactions for particular sets of qubits. Particular hardware providers only support a specific low-level assembly language for interacting with their devices such as DWave's [Pakin 2016].

Naturally, there have been multiple attempts to unify quantum hardware and software around a standard specification and transpiler suite. With Qiskit being the dominant DSL of choice for many researchers, IBM open-sourced Qiskit's intermediate representation (IR) and quantum assembly language (QASM) with the moniker OpenQASM [A. W. Cross et al. 2017]. Not intended for general-purpose computation, OpenQASM represents quantum circuits, custom gates, and measurements with straight-line programs, and to this day, the majority of compilers and DSLs support interopability with OpenQASM 2.0. While OpenQASM 3.0 extended the specification to gate modifiers, timing operations, loops, and more [A. Cross et al. 2022], at the time of release, the project suffered from a severe lack of tooling, offering simply a parser. While Qiskit's team began work on a public semantic analyzer this year, adoption of OpenQASM 3.0 beyond Qiskit is lackluster, though any maintainer of a frequented quantum DSL is working towards support [Wesley 2024].

In the last year, the quantum community has seen a large push toward unification around the Quantum Intermediate Representation (QIR), an IR built on top of LLVM IR by Microsoft for Q#. The effort is led by the QIR Alliance formed under the Linux Foundation's Joint Development Foundation for the development of Open Standards. Backed by Microsoft, NVIDIA, Rigetti, etc. and a large community base, the QIR Alliance seeks to unify quantum software with QIR, a suite of transpilers, and strong tooling [Alliance 2024b].

## 3 BACKGROUND

### 3.1 Quantum Circuits

For the sake of brevity, we defer the full introduction of quantum information and computation to the standard text of Nielsen and Chuang [2010]. For the following, we assume that the reader possesses a working knowledge of classical computation and linear algebra.

The theory of quantum computation begins with the qubit, the smallest unit of quantum computation, whose state we denote as some vector $|\psi\rangle \in \mathbb{C}^2$ such that $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$, $|a|^2 + |b|^2 = 1$, and $|0\rangle, |1\rangle$ are called the *computational basis states*. Our convention is that states are invariant under global phase: $e^{i\theta}\,|\psi\rangle \equiv |\psi\rangle$. Multi-qubit states are represented via tensor product such that, for example, $|\psi\rangle \otimes |\phi\rangle = |\psi\rangle\,|\phi\rangle = |\psi\phi\rangle$ denotes that the first qubit is in state $|\psi\rangle$ while the second is in state $|\phi\rangle$. Valid operations on a $n$-qubit state $|\psi\rangle \in \mathbb{C}^{2^n}$ are simply *unitary* matrices $U \in \mathbb{C}^{2^n \times 2^n}$ such that $UU^\dagger = I$. This unitary condition enforces the fact that all operations must be reversible and all reversible computations on our basis states can be represented as some unitary.

Naturally, in computation, we express most unitaries of interest via a decomposition of gates acting on a small number of qubits. Note that specifying a unitary on the computational basis states fixes its operation on any state. For example, the $X$ gate, the quantum analog of NOT, behaves on a single qubit such that $X\,|i\rangle = |\bar{i}\rangle$, while the $CX$ gate, the *controlled* not, behaves on two qubits such that $CX\,|0a\rangle = |0i\rangle$ and $CX\,|1i\rangle = |1\bar{i}\rangle$. In general, $CU$ is the controlled unitary operation which performs unitary $U$ on the target qubits if the control qubit is 1. While the number of unitaries is uncountable, quantum circuits in practice are restricted to a gateset. There are a number of gatesets are *universal*, meaning any unitary can be approximated to arbitrary precision by some finite sequence of gates from the set. The most common is the *Clifford+T* gateset consisting of $\{H, S, CX, T\}$. Since specific devices only support certain gates in practice, modern transpilers use gate decompositions to derive a valid circuit on the intended gate set. However, optimal circuit optimization is rich open problem [Xu et al. 2023].

Finally, besides operations, quantum circuits may contain measurements, during which the quantum state $|\psi\rangle = \sum_i c_i\,|i\rangle$ collapses to a classical value such that the value $i$ appears with probability $|c_i|^2$.

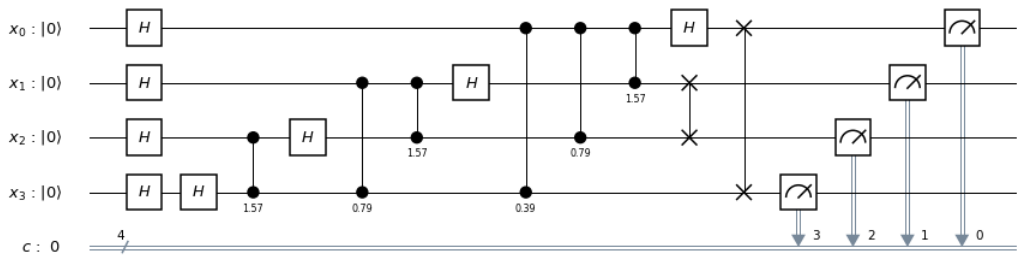


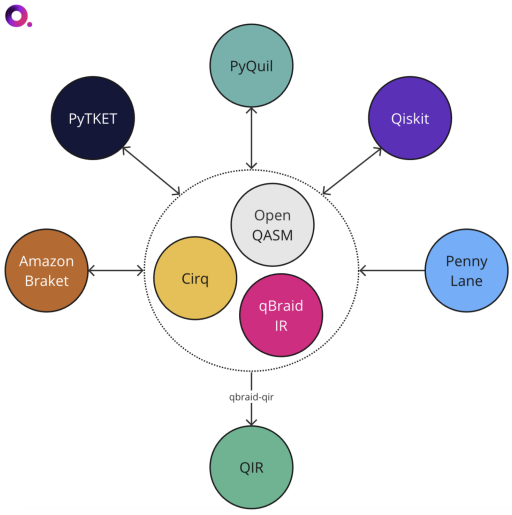

Fig. 1. Qiskit Circuit for the Quantum Fourier Transform on 4 Bits

General quantum algorithms interleave unitary operations, classical computation, and measurements aim to exploit the large $C^{2^n}$ state space to perform expedited computation, the caveat being that measurement destroys the amplitude information of a state irreversibly. For concreteness, Figure 1 describes a circuit for the quantum Fourier transform, which underpins the hallmark factoring algorithm.

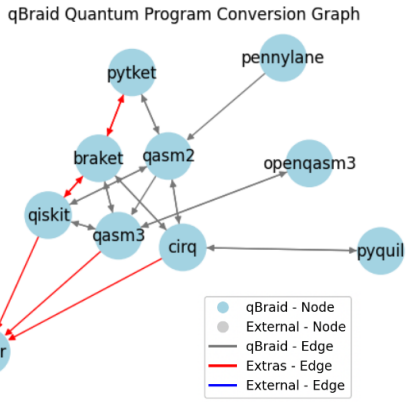
Beyond meeting application needs, research in quantum programming languages aims to ease such operations while building rigorous semantic models for these quantum objects. For example, Silq [Bichsel et al. 2020] is a high-level language for safe automatic uncomputation, with machinery to manage additional the quantum overhead of ensuring reversible operations while allowing the programmer to use classical control flow freely. Google's latest DSL, Qualtran [Harrigan et al. 2024], provides semantics for analyzing the quantum resource requirements of programs. Finally, Jia et al. [2022] make progress toward semantics for hybrid quantum-classical probabilistic effects.

### 3.2 Quantum Software Ecosystem

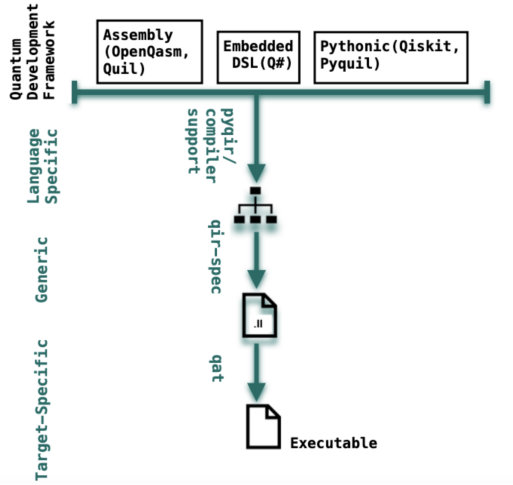
Unifying quantum software around QIR requires strong open-source tooling for IR itself as well as a transpiler network for common DSLs. While the former is a direct effort by the alliance, the latter involves a number of 3rd parties, including qBraid, a cloud-based platform for provider-agnostic quantum computing [Louamri et al. 2024].



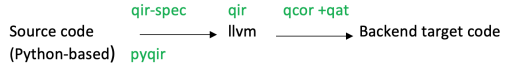

(a) Ecosystem




(b) Conversion Graph




(c) Layers of Abstraction




(d) QIR Compilation

Fig. 2. The Unification of Quantum Software around QIR

qBraid contributes to the unification of transpilers for quantum software. As in Figures 2a and 2b, the qBraid-qir project [Gupta, Jain, et al. 2024] leverages existing transpilers from established DSLs as well as custom transpilers to traverse the directed acyclic conversion graph. Doing so, a wide variety of industry standard DSLs can be translated to QIR as specified by “qir-spec"[*QIR Specification* 2021]. Valid QIR is then translated to executable target code on the hardware backend of interest via optimization passes and tooling by the “qat" project Alliance [2024a].

### 3.3 cuQuantum & CUDA-Q

While quantum hardware improves, high-performance quantum simulation software on CPUs, GPUs, and HPC clusters is crucial to research progress. While Pennylane’s Lightning [Asadi et al. 2024] and Qulacs [Suzuki et al. 2021] have grown in popularity, NVIDIA’s recent joining of the QIR Alliance [Alliance 2023] and dominance in high-performance has brought traction to their cuQuantum SDK [Bayraktar et al. 2023]. With support for density matrix, statevector, and tensor network simulation, NVIDIA’s SDK has accelerated quantum emulation by orders of magnitude and is built on CUDA-Q [Philippidis 2024], a quantum programming model in modern C++ engineered for performance.

CUDA-Q kernels, specified via the C++ API, are compiled to a set of internal MLIR dialects known as the Quantum Kernel Execution (Quake) dialect for quantum computing abstractions and CC for classical computing abstractions. These are then lowered to LLVM adherent to the QIR specification [The CUDA-Q development team n.d.].

## 4 IMPLEMENTATION

While CUDA-Q does provide Python bindings for their C++ API with a builder-based syntax, or the newer annotation-based syntax, documentation for CUDA-Q is sparse. Granted, some of Qiskit’s simulators do support acceleration via NVIDIA CUDA-Q, but community libraries and tooling for CUDA-Q programs are still lacking.

We contribute to the larger qBraid-qir project and QIR Alliance mission by presenting a transpiler from OpenQASM 3.0 to CUDA-Q kernels, as an addition to the qBraid-SDK [Hill et al. 2024]. This culminates in the following PR which closes this issue. Given the existing partial support of OpenQASM 3.0 and remaining compiler infrastructure, this suffices to deliver CUDA-Q transpilation support to a broad class of DSLs via Figure 2b, widening access to performant simulators for quantum programmers.

### 4.1 Transpiler

Our transpiler begins by leveraging the “pyqasm" [Gupta and Hill 2024] project, a semantic analyzer for OpenQASM 3 built on the OpenQASM parser, to construct an AST from well-formed OpenQASM 3 strings, gracefully handling syntax errors. Using pyqasm, we also perform relevant gate decompositions, unroll custom gate definitions, simplify control flow, and inline constant expressions. We then imperatively construct a CUDA-Q kernel with builder syntax by traversing the OpenQASM AST while managing context and Quake references.

At the time of writing, our transpiler supports the following gates: {I, X, Y, Z, H, S, $S^\dagger$, CX, CY, CZ, SWAP, T, $T^\dagger$}. Notably, our transpiler supports all OpenQASM 3.0 gate modifiers: “pow", “inv", “ctrl". While most usages of modifiers are unrolled, we handle the residual “inv" and “ctrl" modifiers by constructing sub-kernels for the target operation and embedding them in the primary kernel via CUDA-Q’s support for arbitrary controlled and adjoint kernels.


(a) Input (b) Unrolled (c) CUDA-Q IR Dialect

Fig. 3. Side-by-Side Example of Transpilation Passes

### 4.2 Testing

Due to the non-determinism of MLIR UUIDs in symbols, implementing a test suite with strong coverage via standard string comparison is near impossible. With some work, we've employed the following testing strategies.

(1) **Circuit Translation**: While CUDA-Q typically gives the kernel in the Quake/CC dialect, it also allows direct access to the final QIR dialect as well as transpilation capabilities to OpenQASM 2.0. Since OpenQASM 3.0 is backwards compatible with OpenQASM 2.0, we can transform our CUDA-Q kernel into a corresponding OpenQASM 3.0 program that is semantically correct, but syntactically different. We then proceed by transpiling both the input program and resulting program into Qiskit circuits via the OpenQASM 3.0 transpiler provided by Qiskit. These circuits can then be compared via standard Qiskit testing tooling.
(2) **Statevector Testing**: Since OpenQASM 2.0 does not support all OpenQASM 3.0 language features, for certain tests, specifically regarding gate modifiers, CUDA-Q to OpenQASM 2.0 translation fails, preventing the above approach. Thus, for circuits without measurement, we use CUDA-Q to get the state vector of the produced kernel and Qiskit-Aer's StatevectorSimulator to get the statevector of the input OpenQASM 3.0 program. We then compare these statevectors up to global phase.
(3) **Randomized Testing**: Using Qiskit's random circuit tooling, we sample a Clifford circuit from our supported gateset without measurement use the existing Qiskit to OpenQASM 3.0 transpiler to generate an input string. We test on circuits of various input sizes and depths.

In the process of developing these testing solutions, we've also discovered a bug in CUDA-Q's OpenQASM 2.0 transpiler regarding handling adjoint modifiers on kernels.

### 4.3 Ecosystem Contributions

As a thank you, we also contribute to the following existing projects.

(1) **pyqasm**: This PR, closing this issue, implements branch unrolling for multi-bit comparison on classical registers. Supported binary operations include "==", ">=", "<=", ">", and "<".
(2) **qBraid-SDK**: This PR, closing this issue, fixes a testing bug regarding optional dependencies.
(3) **qBraid-qir**: This PR, closing this issue, replaces the naive sequential circuit decomposer with Cirq's gate decomposition and optimization tooling.

Note that all PRs mentioned in this work are in-review and should be merged soon.

## 5 FUTURE WORK

Most immediately, we plan to support conditionals, loops, and eventually, the full OpenQASM 3.0 specification, with the exception of timing operations and a few other features that are impossible in a performance-first DSL like CUDA-Q. This not only means extending the transpiler but also requires contributions to the language support of pyqasm. Broadly, we are excited by programming language challenges in quantum computing and are motivated to continue contributing open-source quantum software that furthers modern research.

## 6 CODE AVAILABILITY

Please refer to the aforementioned PRs for source code, implementation details, discussions with the qBraid team, and future work. Executing the relevant artifacts requires an environment that can run the qBraid test suite, and evidence of passing tests can be seen in the GitHub workflows of the respective PRs. We use a private Dockerfile which can be made available upon request, but installation instructions are present through the respective repositories.

## 7 ACKNOWLEDGEMENT

We would like to thank Professor Nada Amin and TF Raffi Sanna for their support. While this work began in November as a final project for Harvard's CS 2520R, it has evolved into a rewarding intellectual pursuit, and we are looking forward to continue working on quantum software in the future.

## REFERENCES

Rajeev Acharya et al.. Dec. 2024. "Quantum error correction below the surface code threshold." *Nature*, (Dec. 2024). DOI: 10.1038/s41586-024-08449-y.

Dorit Aharonov, Xun Gao, Zeph Landau, Yunchao Liu, and Umesh Vazirani. 2023. "A polynomial-time classical algorithm for noisy random circuit sampling." In: *Proceedings of the 55th Annual ACM Symposium on Theory of Computing*, 945–957.

QIR Alliance. 2023. *NVIDIA Joins the QIR Alliance as the Effort Enters Year Two.* https://www.qir-alliance.org/posts/year$_o ne_i n_r eview$/. (2023).

QIR Alliance. 2024a. *qat.* https://github.com/qir-alliance/qat. (2024).

QIR Alliance. 2024b. *QIR Ecosystem.* https://www.qir-alliance.org/projects/. (2024).

Juan Miguel Arrazola et al.. 2021. "Differentiable quantum computational chemistry with PennyLane." *arXiv preprint arXiv:2111.09967*.

Frank Arute et al.. 2019. "Quantum supremacy using a programmable superconducting processor." *Nature*, 574, 7779, 505–510.

Ali Asadi, Amintor Dusko, Chae-Yeun Park, Vincent Michaud-Rioux, Isidor Schoch, Shuli Shu, Trevor Vincent, and Lee James O'Riordan. 2024. "Hybrid quantum programming with PennyLane Lightning on HPC platforms." *arXiv preprint arXiv:2403.02512*.

Harun Bayraktar et al.. 2023. "cuQuantum SDK: A high-performance library for accelerating quantum science." In: *2023 IEEE International Conference on Quantum Computing and Engineering (QCE)*. Vol. 1. IEEE, 1050–1061.

Ville Bergholm et al.. 2018. "Pennylane: Automatic differentiation of hybrid quantum-classical computations." *arXiv preprint arXiv:1811.04968*.

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*, 286–300.

Michael Broughton et al.. 2020. "Tensorflow quantum: A software framework for quantum machine learning." *arXiv preprint arXiv:2003.02989*.

Rigetti Computing. 2019. "Pyquil documentation." *URL http://pyquil. readthedocs. io/en/latest*, 64.

Andrew Cross et al.. 2022. "OpenQASM 3: A broader and deeper quantum assembly language." *ACM Transactions on Quantum Computing*, 3, 3, 1–50.

Andrew W Cross, Lev S Bishop, John A Smolin, and Jay M Gambetta. 2017. "Open quantum assembly language." *arXiv preprint arXiv:1707.03429*.

[SW] Harshit Gupta and Ryan James Hill, *PyQASM: Python toolkit for OpenQASM program analysis and compilation.* Version 0.1.0, Dec. 2024. URL: https://github.com/qBraid/pyqasm.

[SW] Harshit Gupta, Rohan Jain, Samuel Kushnir, Priyansh Parakh, and Ryan James Hill, *qBraid-QIR: Python package for QIR conversions, integrations, and utilities*. Version 0.2.3, Sept. 2024. URL: https://github.com/qBraid/qbraid-qir.

Matthew P Harrigan, Tanuj Khattar, Charles Yuan, Anurudh Peduri, Noureldin Yosri, Fionn D Malone, Ryan Babbush, and Nicholas C Rubin. 2024. "Expressing and analyzing quantum algorithms with qualtran." *arXiv preprint arXiv:2409.04643*.

Bettina Heim, Mathias Soeken, Sarah Marshall, Chris Granade, Martin Roetteler, Alan Geller, Matthias Troyer, and Krysta Svore. 2020. "Quantum programming languages." *Nature Reviews Physics*, 2, 12, 709–722.

[SW] Ryan James Hill et al., *qBraid-SDK: Platform-agnostic quantum runtime framework*. Version 0.8.9, Dec. 2024. DOI: 10.5281/zenodo.12627596, URL: https://github.com/qBraid/qBraid.

David Ittah, Ali Asadi, Erick Ochoa Lopez, Sergei Mironov, Samuel Banning, Romain Moyard, Mai Jacob Peng, and Josh Izaac. 2024. "Catalyst: a Python JIT compiler for auto-differentiable hybrid quantum programs." *Journal of Open Source Software*, 9, 99, 6720.

Ali Javadi-Abhari et al.. 2024. "Quantum computing with Qiskit." *arXiv preprint arXiv:2405.08810*.

Xiaodong Jia, Andre Kornell, Bert Lindenhovius, Michael Mislove, and Vladimir Zamdzhiev. 2022. "Semantics for variational quantum programming." *Proceedings of the ACM on Programming Languages*, 6, POPL, 1–31.

Mohamed Messaoud Louamri, Abdellah Tounsi, Mohamed Taha Rouabah, et al.. 2024. "Comparative Study of Quantum Transpilers: Evaluating the Performance of qiskit-braket-provider, qBraid-SDK, and Pytket Extensions." *arXiv preprint arXiv:2406.06836*.

Jarrod R McClean et al.. 2020. "OpenFermion: the electronic structure package for quantum computers." *Quantum Science and Technology*, 5, 3, 034014.

Hartmut Neven. Dec. 2024. *Meet Willow, our state-of-the-art quantum chip*. https://blog.google/technology/research/google-willow-quantum-chip. (Dec. 2024).

Michael A Nielsen and Isaac L Chuang. 2010. *Quantum computation and quantum information*. Cambridge university press.

Scott Pakin. 2016. "A quantum macro assembler." In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 1–8.

Alex Philippidis. 2024. "GTC 2024: Nvidia's Quantum Expansion Runs through the Cloud." *GEN Edge*, 6, 1, 249–254.

QIR Alliance: https://qir-alliance.org . 2021. *QIR Specification*. Version 0.1. QIR Alliance: https://qir-alliance.org. https://github.com/qir-alliance/qir-spec.

Keeper L Sharkey, Alain Chancé, and Alex Khan. 2022. *Quantum Chemistry and Computing for the Curious: Illustrated with Python and Qiskit® code*. Packt Publishing Ltd.

Peter W Shor. 1999. "Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer." *SIAM review*, 41, 2, 303–332.

Yasunari Suzuki et al.. 2021. "Qulacs: a fast and versatile quantum circuit simulator for research purpose." *Quantum*, 5, 559.

[SW] The CUDA-Q development team, *CUDA-Q*. URL: https://github.com/NVIDIA/cuda-quantum.

Scott Wesley. 2024. "LinguaQuanta: Towards a Quantum Transpiler Between OpenQASM and Quipper (Extended)." *arXiv preprint arXiv:2404.08147*.

Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. 2023. "Synthesizing quantum-circuit optimizers." *Proceedings of the ACM on Programming Languages*, 7, PLDI, 835–859.

Xiao Yuan, Jinzhao Sun, Junyu Liu, Qi Zhao, and You Zhou. 2021. "Quantum simulation with hybrid tensor networks." *Physical Review Letters*, 127, 4, 040501.

Received 12 December 2024