Unqomp: Synthesizing uncomputation in Quantum circuits

Author(s):
Paradis, Anouk; Bichsel, Benjamin; Steffen, Samuel; Vechev, Martin

Publication date:
2021-06

Permanent link:
https://doi.org/10.3929/ethz-b-000493352

Originally published in:
https://doi.org/10.1145/3453483.3454040

Unqomp: Synthesizing Uncomputation in Quantum Circuits

Anouk Paradis
ETH Zurich, Switzerland
anouk.paradis@inf.ethz.ch

Benjamin Bichsel
ETH Zurich, Switzerland
benjamin.bichsel@inf.ethz.ch

Samuel Steffen
ETH Zurich, Switzerland
samuel.steffen@inf.ethz.ch

Martin Vechev
ETH Zurich, Switzerland
martin.vechev@inf.ethz.ch

Abstract

A key challenge when writing quantum programs is the need for *uncomputation*: temporary values produced during the computation must be reset to zero before they can be safely discarded. Unfortunately, most existing quantum languages require tedious manual uncomputation, often leading to inefficient and error-prone programs.

We present Unqomp, the first procedure to automatically synthesize uncomputation in a given quantum circuit. Unqomp can be readily integrated into popular quantum languages, allowing the programmer to allocate and use temporary values analogously to classical computation, knowing they will be uncomputed by Unqomp.

Our evaluation shows that programs leveraging Unqomp are not only shorter (-19% on average), but also generate more efficient circuits (-71% gates and -19% qubits on average).

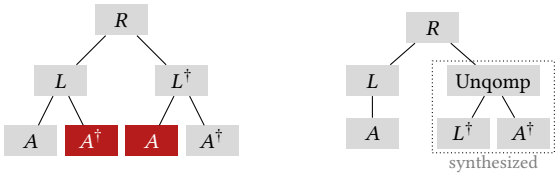
CCS Concepts: • **Hardware → Logic synthesis**; • **Computer systems organization → Quantum computing**.

Keywords: Quantum Circuits, Uncomputation, Synthesis

ACM Reference Format:
Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. 2021. Unqomp: Synthesizing Uncomputation in Quantum Circuits. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3453483.3454040

1 Introduction

Quantum programs often produce temporary values during execution. However, in contrast to classical values, the mere existence of temporary quantum values can lead to unexpected side effects on the remainder of the program state due to the phenomenon of quantum entanglement. Preventing such side effects typically requires resetting temporary quantum values to zero before discarding them, in a process called *uncomputation* [5].



(a) Modular Uncomputation. **(b)** Efficient Uncomputation.

Figure 1. Manual yet modular uncomputation is inefficient.

Synthesizing Uncomputation. This need for uncomputation is a major roadblock preventing programmers from writing correct, efficient, and intuitive quantum programs.

Such programs construct quantum circuits to be run on a quantum computer. Ideally, uncomputation would be synthesized automatically during circuit construction, allowing the programmer to simply omit it. Unfortunately, existing uncomputation synthesizers are restricted to quantum programs consisting exclusively of classical operations (e.g., [3, 13, 17, 21]).

Recently, Silq [6] addressed uncomputation in quantum programs by introducing a type system which statically checks that temporary values can be uncomputed. However, Silq does not explicitly synthesize uncomputation as it does not include a compiler. Likewise, ReQWire [21] can verify that manually provided uncomputation is safe, but cannot synthesize it.

Consequently, most quantum languages require tedious manual uncomputation by explicitly reversing all operations applied to temporary values, sometimes aided by (unsafe) convenience functions (e.g., `ApplyWith`, `with_computed`, discussed in §8). However, this manual approach leads to tension between modularity and efficiency, discussed next.

PLDI '21, June 20–25, 2021, Virtual, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454040

Modular Programs. Generally, writing complex quantum programs requires a modular approach, in particular when developing libraries. Indeed, a quantum library function L typically uncomputes all its internal temporary values, without exposing them to the caller. The programmer can then use L's inverse $L^\dagger$ to uncompute the result of L. If L computes a temporary value using an auxiliary function A, L can in turn leverage $A^\dagger$ to uncompute the result of A.

We visualize a call tree resulting from this modular approach in Fig. 1a, where a function R uses the library L and later uncomputes its result using $L^\dagger$, which internally recomputes A. Note that if L did not encapsulate the uncomputation of A, it would have to expose the output of A to be uncomputed by the user of L, thus breaking modularity.

Inefficient Circuits. While the above modular construction facilitates correct uncomputation, it often results in inefficient circuits. This is a critical problem, as near-future quantum computers only support a limited number of qubits and are subject to noise limiting the number of gates [19].

In the example of Fig. 1a, A is uncomputed in L only to be recomputed again in $L^\dagger$. This redundant work (highlighted as ■) increases exponentially with the depth of the call tree, becoming prohibitive for complex programs.

In contrast, Fig. 1b shows the call tree of an equivalent but more efficient computation, which avoids recomputing A. Achieving this without exposing the output of A and thus breaking modularity of L's implementation is only possible if uncomputation is synthesized during circuit construction.

Sacrificing Modularity. In some cases, programmers sacrifice modularity for efficiency and manually build call trees similar to Fig. 1b. However, the resulting code is error-prone and may introduce other sources of inefficiencies (see §7).

These downsides are exacerbated by the fact that erroneous uncomputation is particularly hard to detect. For example, programmers often reuse the same physical qubit to first hold temporary value a and later temporary value b, in which case incorrectly uncomputing a may corrupt the computation involving b.

This Work: Unqomp. To enable writing modular yet efficient quantum programs, we introduce Unqomp, the first procedure to automatically synthesize uncomputation.

Technically, Unqomp relies on the fundamental insight that a temporary value can be safely uncomputed by inverting the operation that computed it, if the original computation can be described classically and depends on values that can be reused for uncomputation [6]. Unfortunately, whether these values are available for uncomputation can depend on the exact order in which operations are applied, even though these operations often commute. To address this challenge, Unqomp operates on *circuit graphs*, a representation of quantum circuits which does not enforce unnecessary ordering constraints among operations.

Evaluation Results. Unqomp is designed such that it can be readily integrated into existing quantum languages currently requiring manual uncomputation (see §4).

Our evaluation demonstrates that integrating Unqomp into Qiskit [2] allows writing code that is shorter (19% on average), more modular (preventing bugs existing in current implementations), and often significantly more efficient (71% fewer gates and 19% fewer qubits on average). We reported a set of efficiency issues revealed by our evaluation to the Qiskit developers, who have since addressed them. Even compared to the resulting enhanced version of Qiskit, Unqomp allows for significant improvements (57% for gates and 19% for qubits). Furthermore, when used on purely classical examples, Unqomp significantly outperforms other approaches, saving 40% of gates and 41% of qubits on average when compared to Quipper.

Main Contributions. Our main contributions are:

- Unqomp, a procedure synthesizing automatic uncomputation for quantum circuits (§4).
- A formalization of circuit graphs, Unqomp's internal representation of quantum circuits (§5).
- A correctness theorem for Unqomp and its proof (§6).
- An end-to-end implementation[1] and a thorough evaluation of Unqomp on common quantum algorithms (§7).

2 Background

We now introduce basic concepts of quantum computation used in this work. We refer readers unfamiliar with quantum computation to [16] for an excellent in-depth introduction.

2.1 Quantum States

Qubit. The basic unit of information in quantum computing is a qubit. The state φ of a single qubit x is described by a linear combination (called *superposition*) $\varphi = \gamma_0 \left|0\right\rangle_x + \gamma_1 \left|1\right\rangle_x$, with complex coefficients $\gamma_0, \gamma_1 \in \mathbb{C}$. We say φ lies in the Hilbert space $\mathcal{H}_x(\{0,1\})$ with basis $\left|0\right\rangle_x, \left|1\right\rangle_x$. In the following, we will omit the subscript x indicating the qubit name whenever that name is not relevant.

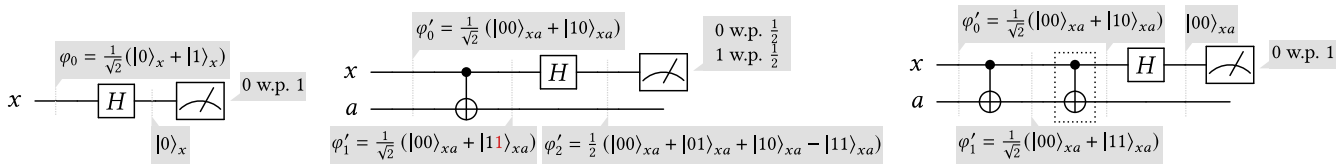
Tensor Product and Entanglement. The state of a system with multiple qubits is described using the tensor product $\otimes$. For instance, the state φ of a system with two qubits x, y is in $\mathcal{H}_x(\{0,1\}) \otimes \mathcal{H}_y(\{0,1\}) = \mathcal{H}_{xy}(\{0,1\}^2)$. We write

$$\varphi = \gamma_0 \left|0\right\rangle_x \otimes \left|0\right\rangle_y + \gamma_1 \left|0\right\rangle_x \otimes \left|1\right\rangle_y + \gamma_2 \left|1\right\rangle_x \otimes \left|0\right\rangle_y + \gamma_3 \left|1\right\rangle_x \otimes \left|1\right\rangle_y$$

with $\gamma_i \in \mathbb{C}$. For readability, we often abbreviate $\left|a\right\rangle_x \otimes \left|b\right\rangle_y$ by $\left|a\right\rangle_x \left|b\right\rangle_y$ or $\left|ab\right\rangle_{xy}$.

The state of a system with multiple qubits is *unentangled* if it can be factorized into the tensor product of the state of each qubit, and *entangled* otherwise. For instance, the unentangled state $\varphi_1 = \frac{1}{\sqrt{2}} \left|00\right\rangle + \frac{1}{\sqrt{2}} \left|10\right\rangle$ can be written as

[1] https://github.com/eth-sri/Unqomp/tree/pldi2021



(a) Example circuit. **(b)** Example circuit storing a temporary value into a. **(c)** Uncomputing temporary value.

Figure 2. Uncomputation in an example quantum circuit, inspired by [24].

$\varphi_1 = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \otimes |0\rangle$. In contrast, $\varphi_2 = \frac{1}{\sqrt{2}}|00\rangle + \frac{1}{\sqrt{2}}|11\rangle$ is entangled as it cannot be factorized.

Measurement. While the state of a qubit cannot be directly observed, we can gain information about it by performing a *measurement*. The outcome of a measurement is probabilistic: measuring qubit x in state $\varphi = \sum_{v\in\{0,1\}} \gamma_v |v\rangle_x$ yields $v' \in \{0,1\}$ with probability $\|\gamma_{v'}\|^2$. To ensure the probabilities of all possible outcomes sum up to one, we make the standard assumption that φ is normalized, meaning that $\|\varphi\|^2 = \sum_{v\in\{0,1\}} \|\gamma_v\|^2 = 1$.

Importantly, measurement affects the state: whenever a measurement of qubit x yields v', the state *collapses* to $|v'\rangle_x$. Due to entanglement, this collapse can lead to side effects on the state of other qubits that are *not* measured. Specifically, consider the state $\sum_{v\in\{0,1\}} \gamma_v |v\rangle_x \otimes \varphi_v$ over qubit x and additional qubits incorporated in the (normalized) term φ_v. When measuring qubit x in this state yields v', the state collapses to $|v'\rangle_x \otimes \varphi_{v'}$, also affecting the state of the remaining qubits. For example, if measuring x in state $\varphi = \frac{1}{\sqrt{2}}|00\rangle_{xy} + \frac{1}{\sqrt{2}}|11\rangle_{xy}$ yields 1, the state after measurement collapses to $|11\rangle_{xy}$.

2.2 Quantum Circuits

Quantum programs construct quantum circuits (see Fig. 2b) to be run on a quantum computer.

Wires. Quantum circuits represent each qubit as a wire, depicted as horizontal lines named x and a in Fig. 2b. For example, initializing x to $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and a to $|0\rangle$ yields the initial state φ'_0 in Fig. 2b.

Gates. Circuits manipulate qubits using linear unitary operators called *gates*, which may span multiple wires. For example, the first gate in Fig. 2b is the controlled NOT gate, called CX. It flips the second qubit ($\oplus$) if the first qubit ($\bullet$) is 1. More formally: $CX_{xy}|ab\rangle_{xy} = |a\rangle_x \otimes |a \oplus b\rangle_y$, where $\oplus$ is the XOR operation. Like every gate, CX is linear and this definition hence extends naturally to arbitrary superpositions. For example, in Fig. 2b, $CX_{xa}(\varphi'_0) = \varphi'_1$. The second gate applied in Fig. 2b is the Hadamard transform H, which maps $|0\rangle$ to $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$ and $|1\rangle$ to $\frac{|0\rangle-|1\rangle}{\sqrt{2}}$. The second qubit a is not modified by H_x. For example, $H_x(\varphi'_1) = \varphi'_2$. Finally, Fig. 2b applies a measurement to the qubit x, shown as .

Controls and Targets. The qubits involved in a gate can generally be divided into two groups. First, the *controls* (depicted as $\bullet$ in circuits) are preserved by the gate (see also [16, §4.3]). The other qubits, which may be modified by the gate depending on the controls, are called *targets*.

Formally, a gate U controlled by one[2] qubit x maps state

$$\sum_{a\in\{0,1\}} \gamma_a |a\rangle_x \otimes \varphi_a \quad \text{to} \quad \sum_{a\in\{0,1\}} \gamma_a |a\rangle_x \otimes \varphi'_a ,$$

preserving the coefficients γ_a of control x, and only modifying the remainder of the state φ_a to φ'_a, where the mapping $\varphi_a \mapsto \varphi'_a$ may depend on a.

In this work, we only consider gates with exactly one target and zero or more controls. For example, CX is controlled by its first qubit ($\bullet$) and targets the second qubit ($\oplus$). The gate H targets only one qubit and has no controls. As single qubit gates and CX are universal for quantum computation, considering only one target is not a restriction [16, §4.5.2].

Qfree Gates. Silq [6] introduced the concept of *qfree* gates, called *classical* in ReQWire [21]. Qfree gates are particularly relevant as these can be automatically uncomputed (see §6.3).

Intuitively, a gate U is qfree if it can be expressed on classical bits. More precisely, for a gate U with one[2] control c and target t, U is qfree if its semantics can be described in terms of a function $f\colon \{0,1\} \times \{0,1\} \to \{0,1\}$ as

$$|i\rangle_c |k\rangle_t \overset{U_{ct}}{\longmapsto} |i\rangle_c |f_i(k)\rangle_t , \tag{1}$$

writing $f_i(k)$ for $f(i,k)$ and $a \overset{U}{\mapsto} b$ when $U(a) = b$. For a gate U with no controls, the above definition simplifies to

$$|k\rangle_t \overset{U}{\mapsto} |f(k)\rangle_t , \tag{2}$$

for some $f\colon \{0,1\} \to \{0,1\}$.

Examples of qfree gates include the identity I with semantics $I|a\rangle = |g(a)\rangle$ where g is the identity and CX with semantics $CX|a\rangle|b\rangle = |a\rangle|f_a(b)\rangle$ for $f_a(b) := a \oplus b$. In contrast, the Hadamard transform H is not qfree.

[2]The definition generalizes naturally to multiple controls.

3 Problem Statement

Next, we motivate why uncomputation is critical and formally define the problem statement addressed by this work.

Effect of Temporary Values. To observe the effect of temporary values that are not uncomputed, first consider Fig. 2a, which applies H to x in initial state φ_0 and measures x, yielding 0 with probability 1. This circuit is extended to Fig. 2b, which copies[3] x into an additional temporary qubit a (called *ancilla*). To this end, Fig. 2b initializes a to $|0\rangle$ yielding $\varphi_0' = \varphi_0 \otimes |0\rangle_a$, and applies CX to flip the value of a if x is one. The resulting state φ_1' highlights the flipped value in red (see Fig. 2b). At a high level, because x is not modified by CX (x is a control), the two circuits should not differ in their effect on x. However, measuring x yields different results, as we mathematically demonstrate in Fig. 2b: the measurement now returns 0 or 1 with probability $\frac{1}{2}$. This difference is caused by the existence of a, which is entangled with x due to the CX gate (see φ_1').

We note that in this toy example, copying x into a is pointless, as the copy is never used. However, we can easily imagine this copy being required in the remainder of the computation (not shown). This is a common pattern in practice, where ancillae store intermediate computation results.

Uncomputation. If we want to avoid the side effect of a onto x, we need to disentangle a from the remainder of the state before measuring x. This can be achieved by uncomputing a, which resets a to its initial, unentangled state $|0\rangle$. Mathematically, this amounts to transforming φ_2' to

$$\tfrac{1}{2}\left(|00\rangle_{xa} + |00\rangle_{xa} + |10\rangle_{xa} - |10\rangle_{xa}\right) = |00\rangle_{xa}\,. \tag{3}$$

To this end, we can insert another CX gate (the self-inverse of CX) as shown in Fig. 2c (dashed box). This gate reverts the original CX gate, thus uncomputing ancilla a. Then, the result of the measurement is again 0 with probability 1, as expected.

Goal: Synthesizing Uncomputation. The goal of this work is to automate the process of uncomputation. Given a (quantum) circuit C and a list of ancilla qubits A, our goal is to create a new circuit with the same effect as C, except that ancilla qubits are brought back to $|0\rangle$, as in Eq. (3).

The procedure Unqomp presented in this work achieves this goal, formalized in Thm. 3.1 below. Thm. 3.1 represents circuit C as a *circuit graph* G (discussed shortly), and describes the effect of G on an initial state by the semantics $[\![G]\!]$.

[3]Note that copying using CX does not violate the *no-cloning* theorem.

Theorem 3.1 (Correctness). *Let* $\textsc{Unqomp}(G, A) = \mathcal{G}$ *for circuit graph* G *with* n *qubits of which* m *are ancilla qubits. Without loss of generality, assume that those ancillae* $A = \left(a^{(1)}, \ldots, a^{(m)}\right)$ *are the first* m *qubits of* G. *If*

$$|0\cdots0\rangle_A \otimes \varphi \overset{[\![G]\!]}{\longmapsto} \sum_{k\in\{0,1\}^m} \gamma_k\,|k\rangle_A \qquad \otimes \phi_k, \text{ then} \tag{4}$$

$$|0\cdots0\rangle_A \otimes \varphi \overset{[\![\mathcal{G}]\!]}{\longmapsto} \sum_{k\in\{0,1\}^m} \gamma_k\,|0\cdots0\rangle_A \otimes \phi_k. \tag{5}$$

Because $\mathcal{G}$ resets ancillae to state $|0\cdots0\rangle$, they are unentangled with the remainder of the state, and can hence be safely discarded without unexpected side effects.

We note that Thm. 3.1 implicitly assumes that G contains no measurement. In particular, in Fig. 2, G would correspond to Fig. 2b without the measurement, and $\mathcal{G}$ would correspond to Fig. 2c without the measurement.

4 Overview

We now provide an overview of Unqomp, following Fig. 3.

Unqomp for Circuit-Based Languages. At a high level, Fig. 3 shows how Unqomp can be readily integrated into circuit-based programming languages such as Qiskit [2] (in the example), Cirq [23], Q# [22], or Quipper [13]. Such languages describe quantum programs (see Fig. 3a) which are then compiled to quantum circuits (see Fig. 3b).

Relying on Unqomp, we can extend Qiskit to Qiskit++, which allows declaring ancilla qubits at allocation time (see Lin. 2 in Fig. 3a). Qiskit++ (i) constructs the circuit without uncomputation, (ii) transforms the circuit to a circuit graph (§5.2), (iii) runs Unqomp to uncompute the ancilla qubits (§6), and (iv) compiles the result back to a circuit (§5.4). We note that the resulting circuit may be subject to post-processing such as decomposing the circuit into universal gates.

Next, we walk through the steps in Fig. 3 in more detail.

Adder Circuit without Uncomputation. The circuit in Fig. 3b (constructed from Fig. 3a) is an adder circuit which takes as input two qubits x and y representing the binary encoding of the number $x + 2y$. The circuit adds to this number the value of qubit b using a temporary carry qubit c.

First, the circuit computes the value of c, which is initialized with $|0\rangle$, using the CCX gate  (a natural generalization of the CX gate to two controls) to change c to 1 iff both x and b are 1. Next, the circuit flips the value of x if b is 1, correctly determining the least significant qubit of the result. Finally, the circuit flips the value of y if the carry c is 1. Note that this circuit does not perform uncomputation of c.

Finding the Uncomputation Position. In order to uncompute c, we have to revert the CCX gate computing c. As a naive attempt, we could try to append the inverse gate of CCX (which is CCX again) at the end of the circuit in Fig. 3b.

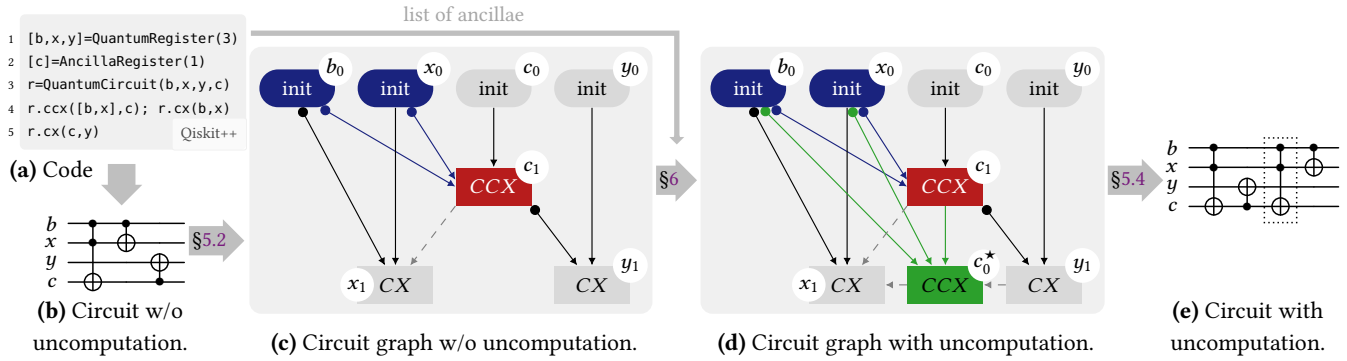


Figure 3. Overview of Unqomp: A circuit incrementing $x + 2y$ by b, with carry c.

Unfortunately, this does not correctly uncompute c: the computation of c is controlled by x, whose value may change by the end of the circuit due to the CX gate targeting x.

A key challenge of uncomputation is therefore finding the position in the circuit to insert the inverse gate $g^\dagger$ uncomputing a gate g. This position must be (i) after all gates involving the computed value (here, after the CX gate controlled by c), but (ii) before any other gates targeting any qubit involved in $g^\dagger$ (here, before the CX gate targeting x). In Fig. 3b, satisfying (i–ii) is only possible when reordering the two CX gates. In Fig. 3e, we have reordered the CX gates, and inserted the uncomputation gate CCX in-between.

Crucially, reordering the CX gates in this example yields an equivalent circuit with the same semantics—on the same input state, the circuit produces the same output state.

4.1 Circuit Graphs

To avoid the need for gate reorderings, we introduce an alternative circuit representation called *circuit graph*, which abstracts different gate orderings with the same semantics.

Fig. 3c shows the circuit graph G corresponding to Fig. 3b.

Nodes and Edges. For every qubit in the circuit, the circuit graph contains an *init node* indicating the circuit's input (here: b_0, x_0, y_0, and c_0). The remaining nodes represent gates. For example, c_1 represents the CCX gate from Fig. 3b, which targets c (indicated by a *target edge* $\longrightarrow$) and is controlled by b and x (indicated by *control edges* $\bullet\!\!\longrightarrow$).

Anti-dependency Edges. Any linearization of gate nodes in G can be interpreted as a quantum circuit applying the corresponding gates in the specified order. In order to ensure that all such circuits have equivalent semantics, we introduce additional ordering constraints using *anti-dependency edges* $\dashrightarrow$.

For instance, the anti-dependency edge $c_1 \dashrightarrow x_1$ in Fig. 3c indicates that the CCX gate must be applied before the CX gate targeting x. This is critical, because the former uses the value of its control qubit x, which is modified by the latter.

Note that G does not enforce an ordering between gate nodes x_1 and y_1, implicitly accounting for the fact that we can swap these without affecting the semantics of the circuit.

4.2 Uncomputation

We now show how Unqomp leverages the circuit graph in Fig. 3c to uncompute the carry qubit c, yielding Fig. 3d.

One Step of Unqomp. First, Unqomp determines the last gate targeting c, which is the CCX gate in c_1 (■). Second, Unqomp checks that CCX is qfree (otherwise, it returns an error). We discuss this necessity in §6.2.

Next, Unqomp inserts a node applying the inverse of CCX (which is again CCX) into the graph (■). We refer to this new node as $c_0^\star$ because it resets the state of c to its state after c_0. We control $c_0^\star$ by the same controls as c_1 (■).

Finally, Unqomp checks that the resulting graph does not contain any cycles (otherwise, it would return an error). This check takes into account anti-dependency edges, which are also updated in Fig. 3d. In particular, edge $y_1 \dashrightarrow c_0^\star$ ensures that the uncomputation node $c_0^\star$ comes after gate node y_1 controlled by c_1, while edge $c_0^\star \dashrightarrow x_1$ ensures that uncomputation node $c_0^\star$ comes before gate node x_1 targeting the control x_0 of $c_0^\star$.

Multiple Uncomputation Steps. If more than one gate was applied to c, Unqomp would execute multiple uncomputation steps as described above, one for each gate targeting c. For instance, assume that two gates U_1, U_2 are applied to c. The circuit graph then contains three nodes for this qubit:

Unqomp steps through all gates applied to ancilla qubits, in reverse order. To process c_2, it checks that U_2 is qfree, inserts $c_1^\star$ controlled by the same nodes as c_2, and links it to the latest node operating on c, yielding $c_2 \longrightarrow c_1^\star$. Next, Unqomp processes c_1, checking that it is qfree, inserting $c_0^\star$ controlled by the same nodes as c_1, and adding an edge from the latest node operating on c, yielding $c_1^\star \longrightarrow c_0^\star$ as shown below:

5 Circuit Graphs

We now provide a more formal introduction to circuit graphs. In particular, we discuss their motivation and definition (§5.1), show how a circuit is transformed to its graph representation (§5.2), provide semantics for circuit graphs (§5.3), and show how a circuit graph is compiled back to a circuit (§5.4).

5.1 Motivation and Definition

We start by motivating the need for circuit graphs and presenting their formal definition.

Gate Ordering. As discussed in §4, quantum circuits typically enforce an unnecessarily restrictive order on their gates. In contrast, circuit graphs abstract away irrelevant ordering constraints: instead of enforcing a total order, circuit graphs use edges to record only the relevant ordering constraints between gate nodes, inducing a partial order on gates.

Specifically, circuit graphs reflect the fact that any two adjacent gates can be reordered, unless the qubit targeted by one of them is involved in the other gate (as a control or as a target). We illustrate this in Fig. 4, where we show two equivalent circuits and their shared circuit graph representation. In particular, the circuit graph does not contain an edge between gate nodes U and V, allowing them to be ordered arbitrarily. Formally, this equivalence can be derived from the properties of control qubits (see §2.2).

Circuit Graphs. A circuit graph is a directed acyclic graph $G = (V, E)$. Its nodes $V = V_{\text{init}} \cup V_{\text{gates}}$ consist of init nodes V_{init} and gate nodes V_{gates}. The set V_{init} contains an init node for each qubit accessed during the computation. For $v \in V_{\text{init}}$, we define $\text{qbit}(v)$ to be the qubit modeled by v. The set V_{gates} contains one node for each gate in the circuit. We define $\text{gate}(v)$ to be the gate represented by $v \in V_{\text{gates}}$, and $\text{qbit}(v)$ to be the qubit targeted by $\text{gate}(v)$. For example, in Fig. 3c, the init node b_0 models the qubit b in Fig. 3b, and gate node c_1 models the CCX gate targeting c.

The edges E are divided into target (visualized as $\longrightarrow$), control ($\bullet\!\!\longrightarrow$), and anti-dependency edges ($\dashrightarrow$). Target edges represent input-output relationships, control edges represent additional dependencies, and anti-dependency edges specify implicit ordering constraints on gate nodes. In general, anti-dependency edges can always be reconstructed from target and control edges. Specifically, circuit graphs contain an anti-dependency edge $t \dashrightarrow c$ whenever there exists a node c' such that $c' \bullet\!\!\longrightarrow t$ and $c' \longrightarrow c$. This anti-dependency edge models the ordering constraint that t must be applied before c.

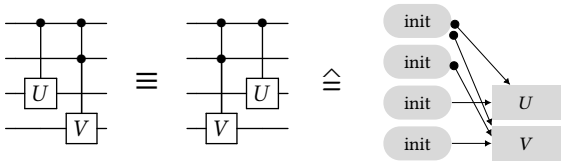


Figure 4. Valid gate reordering.

Valid Circuit Graphs. A circuit graph is valid if it represents an actual quantum circuit. More precisely, $G = (V, E)$ is valid iff (i) its init nodes have no incoming target edge while gate nodes have exactly one, (ii) all its nodes have at most one outgoing target edge, (iii) its anti-dependency edges can be reconstructed from its control and target edges according to the rule discussed above, (iv) the number of incoming control edges of each gate node $v \in V_{\text{gates}}$ equals the number of controls of $\text{gate}(v)$, and (v) G is acyclic. Consequently, the target edges should form disjoint paths starting at init nodes.

In the following, we only consider valid circuit graphs. All operations discussed in this work preserve validity.

Naming Convention. We usually respect the following naming convention: for qubit a, a_0 is the init node and a_i the i^{th} gate node targeting a. For example, in Fig. 3c, node c_1 is the first gate targeting c. Further, we refer to gate nodes inserted by Unqomp as $a_i^\star$, indicating that $a_i^\star$ restores the state of qubit a after node a_i.

5.2 From Circuits to Circuit Graphs

We now describe how to transform a given circuit into its circuit graph representation.

Building a Circuit Graph. To construct a circuit graph from a circuit, we first create an init node for each qubit.

Then, we process the gates in order. When processing a gate U, we add a fresh node u representing U to V_{gates}. To determine the incoming target edge at u, let q be the qubit targeted by U. We then introduce a target edge $v \longrightarrow u$, where v is the node corresponding to the latest gate targeting q, or the init node for q if we are processing the first gate targeting q. Similarly, to determine incoming control edges at u, we consider each control qubit c of the gate. We introduce a control edge $v \bullet\!\!\longrightarrow u$, where v is the node corresponding to the latest gate targeting c, or the init node for c.

Finally, we introduce anti-dependency edges based on the inserted target and control edges (see §5.1).

Example. When processing the first CX gate in Fig. 3b, we introduce gate node x_1 and add edge $x_0 \longrightarrow x_1$. Because this

gate is controlled by qubit b represented by init node b_0, we further add edge $b_0 \bullet\!\!\rightarrow x_1$ to the graph. The anti-dependency edge $c_1 \dashrightarrow x_1$ exists due to $x_0 \bullet\!\!\rightarrow c_1$ and $x_0 \rightarrow x_1$.

5.3 Circuit Graph Semantics

We now define the semantics of circuit graphs. We first define the semantics of individual nodes, and then extend this semantics to whole circuit graphs.

States. The init nodes $V_{\text{init}} = \{v_1, \ldots, v_n\}$ of a circuit graph specify the qubits of the system. The state of this system is in $\mathcal{H}_{q_1,\ldots,q_n}(\{0,1\}^n)$, where $q_i := \text{qbit}(v_i)$.

Semantics of Gate Nodes. The semantics $[\![v]\!]$ of a gate node $v \in V_{\text{gates}}$ is defined according to the semantics of $\text{gate}(v)$, where the target and control edges ending at v determine the involved target and control qubits, respectively. For example, the semantics of x_1 in Fig. 3c is $[\![x_1]\!] = CX_{bx}$.

Semantics of Circuit Graphs. The semantics of a circuit graph $G = (V, E)$ is the composition of the semantics of its gate nodes, according to the partial order specified by its edges. More precisely, we first select an arbitrary linearization $\mathcal{L}(G)$ of the gate nodes $V_{\text{gates}} \subseteq V$ consistent with the partial order induced by E. Then, we compose the node semantics in this order by function composition. Importantly, the resulting semantics is independent of the choice of $\mathcal{L}(G)$.

For example, for the circuit graph G in Fig. 3b, we can select $\mathcal{L}(G) = (c_1, x_1, y_1)$, yielding the semantics:

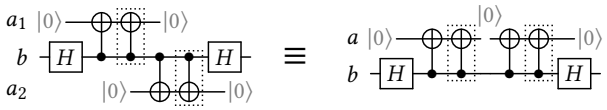
$$[\![G]\!] = [\![y_1]\!] \circ [\![x_1]\!] \circ [\![c_1]\!] .$$

5.4 Compilation to Circuit

We now discuss how circuit graphs are compiled to circuits. As visualized in Fig. 3, this step is applied after introducing uncomputation in the circuit graph (discussed in §6). As such, the steps performed here are not part of the transformation covered by Thm. 3.1, which for instance assumes a constant number of ancillae.

CCX Gates Optimization. Before the actual compilation, we run a simple optimization pass suggested in [4, §6]: we replace every *CCX* gate targeting an ancilla qubit by a more efficient Margolus gate, which has the same semantics as *CCX*, except that it maps $|111\rangle$ to $-|110\rangle$ instead of $|110\rangle$. This so-called phase change does not affect the semantics of the whole circuit, as Unqomp ensures that all replaced *CCX* gates are paired with a gate uncomputing it, which, when also replaced, reverts the phase change.

This optimization is already selectively leveraged by experts. For instance, Qiskit uses Margolus gates in its library implementation of the MCX gate. In our case, by leveraging the uncomputation information available in the circuit graph, we can effortlessly extend this optimization to *all* uncomputed *CCX* gates in a circuit.



(a) Naive compilation. **(b)** Efficient compilation.

Figure 5. Reusing qubits.

Linearization. To compile $G = (V, E)$ to a circuit after applying the optimization above, we first prepare a wire for each init node and then select a linearization $\mathcal{L}(G)$ yielding $(v_1, \ldots, v_n)$. Next, we insert $\text{gate}(v_1), \ldots, \text{gate}(v_n)$ into the circuit according to this linearization, determining the target and control qubits involved in $\text{gate}(v_i)$ by the incoming target and control edges at v_i.

For example, compiling Fig. 3d using the linearization $(c_1, y_1, c_0^\star, x_1)$ yields the circuit in Fig. 3e.

Reusing Qubits. Unfortunately, this strategy allocates a wire for each ancilla qubit in G. This is unnecessarily wasteful, as a wire holding a correctly uncomputed ancilla is in unentangled state $|0\rangle$ and can be safely reused to hold another ancilla without introducing unexpected side effects.

For example, Fig. 5a shows the result of uncomputing qubits a_1 and a_2 in a toy circuit, where dotted boxes indicate gates inserted by Unqomp. The resulting circuit requires 3 wires in total. In contrast, Fig. 5b shows a more efficient compilation which only requires 2 wires by reusing the same wire to hold both ancillae. This is possible because the lifetimes of the two ancillae do not overlap.

Final Nodes. To track the lifetime of ancillae, we introduce *final* nodes, which mark the end of the computation involving a qubit. Specifically, for every node $u \in V$ with no outgoing target edge, we add to G a final node v, a target edge $u \rightarrow v$, and any induced anti-dependency edges. Additionally, we extend the linearization to $\mathcal{L}^+(G)$, which includes init, gate, and final nodes, respecting the partial order induced by E. As a result, in $\mathcal{L}^+(G)$, the qubit corresponding to a final node is not involved in any gate at any later position.

Greedy Ancilla Allocation. To reduce the number of wires, we employ a simple but effective greedy strategy reusing wires whenever possible. Specifically, we process the nodes in V in the order $\mathcal{L}^+(G)$. The init and final nodes allow us to track the start and end of a qubit's lifetime. Upon visiting an ancilla init node, we greedily try to allocate it on a wire that holds an ancilla qubit which is no longer alive and thus must have been uncomputed by Unqomp. If no such wire exists, we allocate a fresh wire.

This approach is an instantiation of linear scan register allocation [18], which is provably optimal for a fixed linearization [8, §17]. As finding an optimal linearization is computationally expensive, we next introduce a greedy heuristic to select a linearization that performs well empirically (see §7).

Linearization Heuristic. To select a linearization $\mathcal{L}^+(G)$, we slightly modified Kahn's algorithm [14]—a standard algorithm which creates a linear order of G's vertices by iteratively removing vertices that have no incoming edges.

Specifically, when selecting the next node to be removed from G, we de-prioritize ancilla init nodes and only select them if no other choice exists. As a consequence, $\mathcal{L}^+(G)$ greedily completes computations involving ancillae before allocating new ancillae, whenever possible.

Discussion: Graph Coloring. Unfortunately, graph coloring allocation (e.g., [1, §8]) cannot be readily adapted for circuit graphs as they enforce fewer ordering constraints between operations than control flow graphs, meaning that we cannot definitively determine if two given ancillae are live simultaneously. Hence, recording pairwise conflicts is insufficient: even if each pair of ancillae (a, b), (b, c) and (c, a) can be allocated on the same wire, allocating all three of them on the same wire may be impossible.

6 Synthesizing Uncomputation

In this section, we formalize Unqomp (§6.1), discuss the conditions it checks to ensure uncomputation is possible (§6.2), and sketch Unqomp's correctness proof (§6.3).

6.1 Unqomp

Alg. 1 formalizes Unqomp, which takes a circuit graph G and a list of ancilla qubits to be uncomputed (Lin. 1), and returns, if possible, G extended by gate nodes uncomputing the ancilla qubits. The following discussion complements our informal presentation of Unqomp on the example in §4.

All Steps. To uncompute a given list of ancillae on circuit graph $G = (V, E)$, Unqomp iterates over a linearization $\mathcal{L}(G)$ in reverse order (Lin. 3–5), introducing an uncomputation step for every gate node targeting an ancilla.

Single Step. The core of Unqomp is the procedure UncomputeStep (Lin. 6), which takes a circuit graph $G = (V, E)$ and a gate node a_n to be uncomputed. It first checks that the gate of a_n is qfree (Lin. 7). It then determines the last gate node $a_n^\star$ targeting qbit(a_n) (Lin. 8), which restores the state of qubit a after a_n and will be the target of the inserted uncomputation. We note that if this is the first uncomputation step on qubit a, then $a_n^\star = a_n$, as in our overview example (Fig. 3). Otherwise, $a_n^\star$ was inserted by Unqomp in a previous step.

In Lin. 9, UncomputeStep determines the nodes controlling a_n, which are required for controlling the uncomputation. Note that some of those control nodes c may have already been uncomputed by a previous step of Unqomp. In this case, as c and $c^\star$ can be used interchangeably, UncomputeStep replaces the former by the latter whenever possible (Lin. 11). This is helpful as using $c^\star$ is more likely to result in a cycle-free graph than using c (see App. B).

Algorithm 1 Unqomp: Synthesizing uncomputation.

```
1:  procedure Unqomp(G, a^(1), ..., a^(n): qubits)
2:      (v_1, ..., v_n) ← L(G)                          ▷ linearization of gate nodes
3:      for all v in (v_n, ..., v_1) do                  ▷ iterate in reverse order
4:          if qbit(v) ∈ {a^(1), ..., a^(n)} then        ▷ gates operates on ancilla
5:              G ← UncomputeStep(G, v)
        return G
6:  procedure UncomputeStep(G, a_n: gate node)           ▷ G = (V, E)
7:      assert gate(a_n) is qfree
8:      a_n^⋆ ← last gate node targeting qbit(a_n)
9:      ctrls ← {c ∈ V | c •⟶ a_n ∈ E}                   ▷ controls of a_n
10:     for all c ∈ ctrls do    ▷ in ctrls, replace c by c^⋆ wherever possible
11:         if c^⋆ ∈ V then ctrls ← ctrls \ {c} ∪ {c^⋆}
12:     a_{n-1}^⋆ ← Inverse(a_n)                         ▷ fresh node with inverse gate
13:     E_u ← {c •⟶ a_{n-1}^⋆ | c ∈ ctrls} ∪ {a_n^⋆ → a_{n-1}^⋆}   ▷ fresh edges
14:     E_a ← {v ⇢ a_{n-1}^⋆ | a_n^⋆ •⟶ v ∈ E, v ∈ V} ∪   ▷ anti-dependencies
15:           {a_{n-1}^⋆ ⇢ v | c → v ∈ E, c ∈ ctrls}
16:     G_u ← (V ∪ {a_{n-1}^⋆}, E ∪ E_u ∪ E_a)            ▷ adding uncomputation
17:     assert G_u has no cycles
18:     return G_u
```

Next, UncomputeStep constructs the gate node $a_{n-1}^\star$ and edges E_u to be inserted into G. Specifically, the gate node $a_{n-1}^\star$ applies the inverse gate of a_n (Lin. 12), targets $a_n^\star$, and is controlled by *ctrls* (Lin. 13). UncomputeStep further generates the anti-dependency edges induced by the new edges (Lin. 14), and constructs the new graph G_u by inserting all new gates and edges into G (Lin. 16).

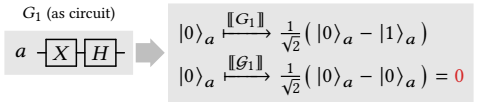
Finally, UncomputeStep reports an error if G_u contains a cycle (Lin. 17), and returns G_u otherwise (Lin. 18).

6.2 Incompleteness

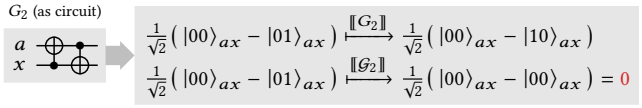
Unfortunately, uncomputation is mathematically impossible in some cases. Fig. 6 exemplifies the two fundamental reasons for this. In both examples, satisfying Thm. 3.1 would require constructing a circuit graph $\mathcal{G}$ which produces the invalid state $0 = \sum_{k\in\{0,1\}^n} 0\cdot|k\rangle$. As no quantum circuit can produce this state, uncomputation is impossible in these cases, forcing Unqomp to return an error.

Non-qfree. In Fig. 6a, the underlying problem is that gate H applied to ancilla a is not qfree and therefore mixes basis states: it turns the basis state $|1\rangle_a$ (the result of applying X to the initial state $|0\rangle_a$) into superposition $\frac{1}{\sqrt{2}}\left(|0\rangle_a - |1\rangle_a\right)$. Replacing $|1\rangle_a$ by $|0\rangle_a$ in this state (as required by Thm. 3.1) then yields the invalid state 0. Therefore, Unqomp only uncomputes qfree gates, as asserted in Lin. 7 of Alg. 1.

Cyclic Dependency. Fig. 6b demonstrates that even for circuits containing only qfree gates, uncomputation may not be possible. The underlying problem in the example is that inserting an uncomputation gate would result in a cyclic



(a) Ancilla a is modified by the non-qfree gate H.



(b) Uncomputing ancilla a would result in a cyclic dependency.

Figure 6. Ancilla qubits that cannot be uncomputed.

dependency: the uncomputation node for a would have to be (i) before the CX gate targeting x, as it uses the initial value of x, but also (ii) after it, as this gate uses the updated value of a. Therefore, Unqomp asserts that the generated circuit graph contains no cycles (Lin. 17 in Alg. 1).

Conservative Criteria. We note that Unqomp may return an error even though uncomputation would be possible in principle, as the criteria it checks (Lin. 7 and 17 in Alg. 1) are conservative. For example, applying gate H to an ancilla twice has no effect on its state (as H is self-inverse), but triggers an error in Unqomp as H is not qfree. In such rare cases, the programmer may revert to manual uncomputation, at the costs discussed in Fig. 1.

We note that this is not a concern in practice: in our evaluation (§7), Unqomp was able to uncompute all temporary values, except when they involved temporary changes of controls. This problem appeared for MCX gates with negated controls, present in two of our examples (Adder and WeightedAdder, see §7). To resolve this issue, we ensured that Unqomp treats these problematic gates as atomic qfree gates.

6.3 Correctness of Unqomp

Next, we discuss the key insights in our proof of Thm. 3.1.

Theorem 3.1 (Correctness). *Let* $\textsc{Unqomp}(G, A) = \mathcal{G}$ *for circuit graph* G *with* n *qubits of which* m *are ancilla qubits. Without loss of generality, assume that those ancillae* $A = (a^{(1)}, \ldots, a^{(m)})$ *are the first* m *qubits of* G*. If*

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![G]\!]} \sum_{k\in\{0,1\}^m} \gamma_k |k\rangle_A \otimes \phi_k, \text{ then} \quad (4)$$

$$|0\cdots0\rangle_A \otimes \varphi \xmapsto{[\![\mathcal{G}]\!]} \sum_{k\in\{0,1\}^m} \gamma_k |0\cdots0\rangle_A \otimes \phi_k. \quad (5)$$

Outline. We first prove Thm. 3.1 in a restricted setting, where (i) there is only a single ancilla a targeted by a single gate U, (ii) the controls c of U are in a basis state $|i\rangle$, and (iii) the gate U occurs first in the computation, while (iv) the gate $U^\dagger$ uncomputing U occurs last. Then, we discuss how our full proof avoids these restrictions.

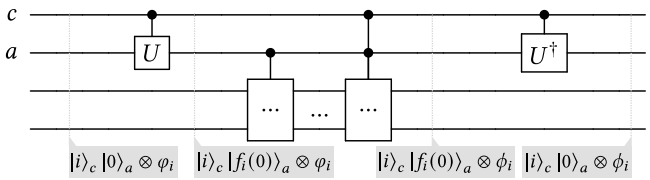


Figure 7. Effect of uncomputation on quantum state.

Proof Sketch. The key insight of our proof is that if an ancilla is computed using a qfree gate U, it can be uncomputed as long as its controls are still available.

Fig. 7 shows how the qfree gate U and its uncomputation affect the quantum state. First, the effect of U follows Eq. (1) in §2, updating qubit a to $|f_i(0)\rangle$. Then, any remaining gates preserve both a (as U is the only gate targeting a) and c (as any gate targeting the control c of $U^\dagger$ must be after $U^\dagger$ due to anti-dependency edges). Finally, applying $U^\dagger$ restores the state of ancilla qubit a to $\left|f_i^{-1}(f_i(0))\right\rangle = |0\rangle$.

This concludes our proof as the first, second to last, and last states in Fig. 7 correspond to the left hand side of Eq. (4), the right hand side of Eq. (4), and the right hand side of Eq. (5), respectively.

Full Proof. We provide a full proof in App. B. It (i) handles multiple ancillae and multiple gates by induction on the number of gate nodes in G, (ii) naturally extends to controls in superposition, (iii) accounts for gates before U and after $U^\dagger$, and (iv) takes into account that nodes c and $c^\star$ can be used interchangeably (see Lin. 11 in Alg. 1).

7 Evaluation

We now present an extensive experimental evaluation of Unqomp on common quantum algorithms.

Implementation. We implemented our approach as a language extension called Qiskit++, which integrates Unqomp into Qiskit as visualized in Fig. 3. Qiskit++ allows annotating ancilla qubits in a Qiskit program at allocation time (e.g., see Lin. 2 in Fig. 3a) and uncomputes these automatically. Like Qiskit, Qiskit++ is written in Python.

Research Questions. Our evaluation addresses the following research questions, where Q1 and Q2 analyze the input Qiskit++ code and Q3 evaluates the compilation result.

- **Q1** Code Length: Does Unqomp reduce the amount of code, compared to manual uncomputation?
- **Q2** Modularity: Does Unqomp allow writing more modular and hence less complex code?
- **Q3** Efficiency: Does Unqomp yield more efficient circuits in terms of gates and qubits?

Table 1. Comparing code complexity with and without Unqomp. The table shows the lines of code (incl. relative difference), whether ancilla allocation is modular (M), and whether uncomputation is implicit (U). Algorithms marked with [c] are from Cirq [7], while all other algorithms are from Qiskit [20].

Algorithm	Original lines	M	U	with Unqomp lines	diff	M	U
Adder [c]	31	✓		28	−10%	✓	✓
Deutsch-Jozsa	13	✓	✓	12	−8%	✓	✓
Grover	46	✓	✓	31	−33%	✓	✓
IntegerComparator	45			38	−16%	✓	✓
MCRY	7			4	−43%	✓	✓
MCX	16	✓		13	−19%	✓	✓
Multiplier [c]	13			11	−15%	✓	✓
PiecewiseLinearR	43			29	−33%	✓	✓
PolynomialPauliR	120			118	−2%	✓	✓
WeightedAdder	70			42	−40%	✓	✓

7.1 Evaluated Algorithms

To address these questions, we evaluated Unqomp on 10 quantum algorithms shown in Tab. 1. We used the implementations from Qiskit 0.22.0 [20] and Cirq v0.9.1 [7], where we re-implemented Cirq examples in Qiskit.

Algorithms. Each quantum algorithm is represented by a Python function which, given some parameters such as the input size, constructs a circuit implementing the algorithm. For example, the MCX (multi-controlled NOT) function accepts a parameter n, and constructs a circuit which takes inputs $(c_1, \ldots, c_n, t)$ to compute $(c_1, \ldots, c_n,\ t \oplus (c_1 \& \ldots \& c_n))$.

Deutsch-Jozsa [16, §1.4.4] and Grover [16, §6] are well-known quantum algorithms. Given an integer v, IntegerComparator flips a target qubit if the number encoded in a list of control qubits is greater than v. Given n and an angle θ, MCRY (multi-controlled Y rotation) rotates a target qubit by θ in the Y basis if all of n control qubits are 1. For parameter n, Multiplier computes the binary representation of $x \cdot y$ for two numbers x and y encoded in n qubits each. Similarly, Adder computes $x + y$. Given n and a piecewise linear function f, PiecewiseLinearR rotates a target qubit by the angle $f(c)$ in basis Y, where c is encoded using n control qubits. In PolynomialPauliR, the function f is polynomial. Finally, given n and a list $v_1, \ldots, v_n$ of classical values, WeightedAdder computes $\sum c_i v_i$ for control qubits c_i.

For our Qiskit++ implementations of these algorithms, we simply removed the parts performing uncomputation or manually passing ancillae (see §7.3) and instead annotated ancilla qubits to enable automatic uncomputation.

Uncomputation Synthesis Time. We ran the Qiskit++ compilation pipeline on a commodity laptop with 8 GB of RAM and 8 CPU cores at 2.40 GHz. For all algorithms presented in Tab. 1, our implementation of Unqomp required less than 1 second to introduce uncomputation.

```
c1 = QuantumRegister(n)
c2 = QuantumRegister(n - 1)
g1 = MCXGate(len(c1))
g2 = MCXGate(len(c2))
a = QuantumRegister(max(
   g1.num_ancillae,
   g2.num_ancillae))
c.append(g1, c1, t, a)
c.append(g2, c2, t, a)
```

(a) Ancilla reuse in Qiskit.

```
c1 = QuantumRegister(n)
c2 = QuantumRegister(n - 1)
a = QuantumRegister(n - 2)
c.mcx(c1, t, a)
c.mcx(c2, t, a)
```

(b) Hardcoded qubits.

```
c1 = QuantumRegister(n - 1)
c2 = QuantumRegister(n)
c.mcx(c1, t)
c.mcx(c2, t)
```

(c) Modularity in Qiskit++.

Figure 8. Exposing ancillae to the caller in Qiskit. We shorten `c.append(MCXGate(len(c)), c, t, a)` to `c.mcx(c, t, a)`.

7.2 Q1: Code Length

Tab. 1 compares the code lengths of the Qiskit (original) and Qiskit++ (with Unqomp) implementations for each algorithm. We observe that Unqomp consistently reduces the number of code lines, by up to 43%.

Most Qiskit algorithm implementations include explicit uncomputation code, which reverts all gates applied to the ancillae (see non-ticked in column U). This is not required in Qiskit++, leading to a significant code reduction. For example, in WeightedAdder, 39% of the source code lines deal with explicit uncomputation, which can be omitted using Unqomp. Explicitly inserting uncomputation not only increases code length but may also require rewriting the actual computation. For example, the original implementation of Adder is convoluted by re-ordered gates and interleaved uncomputation (cp. Fig. 3e). In contrast, uncomputation in Unqomp is implicit and ancillae are uncomputed automatically.

The Deutsch-Jozsa and Grover implementations do not include any uncomputation as they leverage phase kickback for the oracle evaluation. We note that the oracle circuit (which may perform uncomputation internally) is provided as a parameter to these algorithms and hence not part of the code considered here. For this reason, Unqomp cannot save any lines for Deutsch-Jozsa. The significant reduction for Grover originates from a modularity issue discussed next.

7.3 Q2: Modularity

Next, we compare the modularity of ancilla allocation. Overall, we find that Qiskit functions often break modularity, while Qiskit++ allows for modular code.

Exposing Ancillae in Qiskit. Qiskit functions expose the number of required ancillae to the caller using a field `num_ancillae` and rely on the caller to allocate them. This allows developers to manually reuse ancillae across functions, relying on correct uncomputation within the functions. For

Table 2. Percentage of gates and qubits (qbs) saved by Unqomp, where higher numbers are better. Implementations marked with * were improved in Qiskit due to our bug reports (see Footnote 4). Entries in parenthesis require manual intervention, and ✗ indicates that Quipper's `classical_to_reversible_optim` is not applicable.

	Qiskit			Quipper		
	gates		qbs	gates		qbs
Algorithm	all	*CX*		all	*CX*	
Adder	34	35	0	56	62	17
Deutsch-Jozsa	0	0	0	(38)	(50)	(5)
Grover	0	0	0	(40)	(50)	(5)
IntegerComparator	31	48	0	41	51	0
MCRY	99.5	99.5	−4	✗	✗	✗
MCRY *	48	48	−4	✗	✗	✗
MCX	0	0	0	41	51	5
Multiplier	36	38	2	−25	−25	34
PiecewiseLinearR	41	42	29	✗	✗	✗
PolynomialPauliR	81	86	11	✗	✗	✗
PolynomialPauliR *	44	45	11	✗	✗	✗
WeightedAdder	43	55	−12	52	53	78
WeightedAdder *	30	33	−12	52	53	78
WeightedAdder alt. impl.	31	46	0	48	50	82
WeightedAdder alt. impl. *	16	20	0	48	50	82

example, Fig. 8a shows a code snippet where the developer allocates the required ancillae `a` for two `MCX` gates.

While this construction respects modularity, it requires the developer to manually reuse qubits and tediously combine the ancilla requirements of multiple functions to determine the number of ancillae to allocate. This creates significant overhead: for instance, a third of the lines in the original Grover implementation deal with ancilla management. Indeed, this construction is only used for few examples in our benchmark (see ticked ✓ in column M of Tab. 1).

Breaking Modularity. To reduce developer overhead, Qiskit functions are often implemented in a non-modular manner (see column M), where ancilla requirements are hard-coded using hand-crafted formulas relying on explicit knowledge about the implementation of called functions. As a typical example, Fig. 8b allocates exactly $n-2$ ancillae for MCX using knowledge of its internal implementation and of the length of both `c1` and `c2`.

Clearly, this increases coupling and leads to issues if library implementations are changed. Indeed, we found multiple inefficient usages of MCX and MCRY in the Qiskit library, allocating more ancillae than necessary for those gates due to a change in their default implementation. We reported three such issues to the developers, who have fixed them recently. [4] Fixing these issues further ensured that using the

[4] https://github.com/Qiskit/qiskit-terra/issues/4786 for MCRY
https://github.com/Qiskit/qiskit-terra/issues/5320 for WeightedAdder
https://github.com/Qiskit/qiskit-terra/issues/5321 for PolynomialPauliR

```
c.ccx(c0, c1, a0)
c.ccx(c2, a0, r)
c.h(c0)
```

(a) Separable qfree section.

```
c.ccx(c0, c1, a0)
c.ccx(c2, a0, a1)
c.cry(a1, r, 2)
```

(b) Non-qfree gate on ancilla.

Figure 9. Limitation of Classical Uncomputation. Variables `c0, c1, c2, r` are qubits and `a0, a1` are ancillae. The call `c.ccx(c0, c1, a0)` applies a CCX gate with controls `c0, c1` and target `a0`, while `c.cry(a0, r, 2)` applied a controlled rotation with control `a0`, target `r`, and angle 2.

"v-chain" variant of MCX requires only a low number of basic gates by using all allocated ancillae, as we discuss in §7.4.

Modularity in Qiskit++. In contrast to Qiskit, Qiskit++ allows allocating ancillae in a modular manner, as indicated by ✓ in column M of Tab. 1. A library function can locally allocate ancilla qubits for internal use, without exposing them to the caller of the function. Both caller and library can rely on Unqomp to automatically uncompute ancillae, and efficiently allocate and reuse qubits. For example, in Fig. 8c, the caller does not need to know about the ancillae in `MCX`.

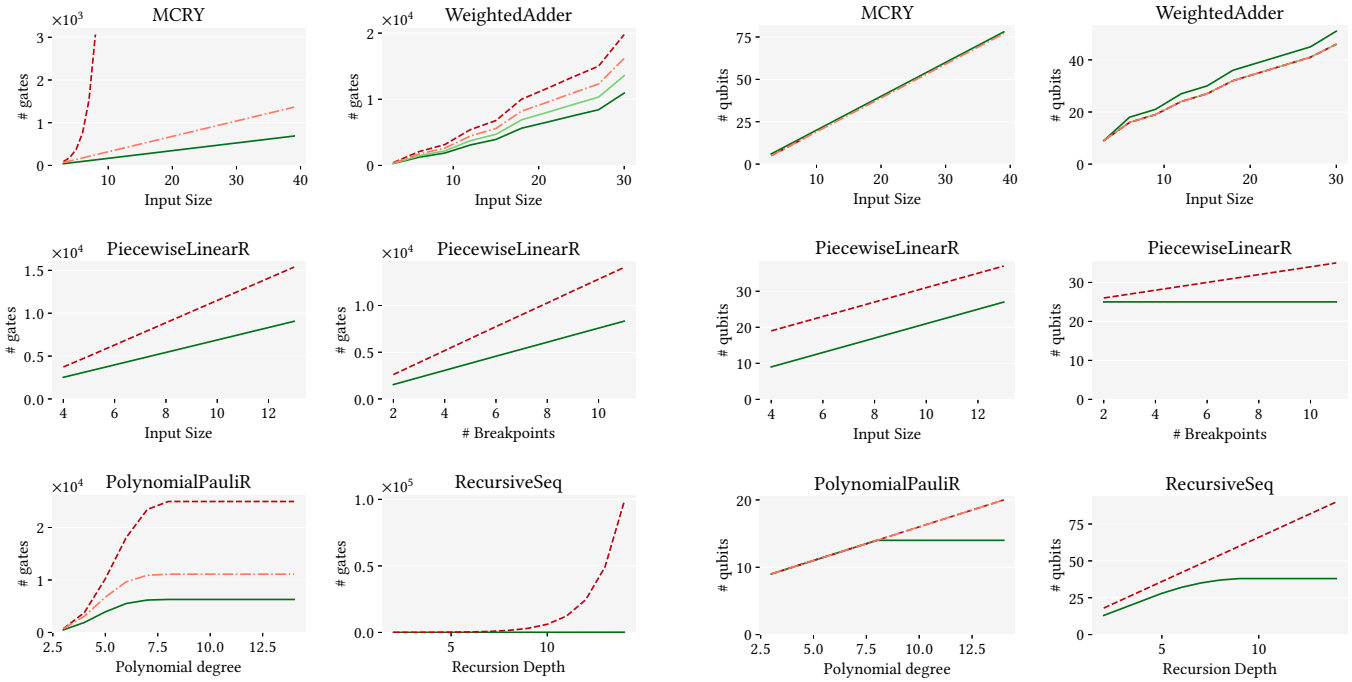
7.4 Q3: Efficiency

We now compare the efficiency of circuits generated by Qiskit++ to circuits generated by Qiskit and Quipper, where Quipper is only applicable for classical programs, i.e., programs only consisting of qfree operations. Overall, we find that Qiskit++ circuits are often significantly more efficient.

Approach. For all algorithms, we ran the full compilation pipeline as shown in Fig. 3, followed by Qiskit's decomposition into the two basic gates *CX* and $U3$ as post-processing, using the "v-chain" variant of MCX. Further, we instantiated the oracle circuit in Grover and Deutsch-Jozsa with an MCX gate. For the comparison to Quipper, we manually translated the algorithms to Quipper using the `classical_to_reversible_optim` construct to insert uncomputation. We then applied the Qiskit decomposition discussed above to the resulting circuits.

We show the resulting reduction from Qiskit to Qiskit++ and Quipper to Qiskit++ for gates and qubits in Tab. 2. For completeness, Tab. 2 also shows the reduction in *CX* gates only (which are typically more expensive than $U3$ gates), with analogous results.

Limitations of Classical Uncomputation. As shown in Tab. 2, the uncomputation offered by Quipper is severely limited: as `classical_to_reversible_optim` only supports classical programs, the presence of non-qfree gates prevents directly applying it on Deutsch-Jozsa, Grover, MCRY, PiecewiseLinearR, and PolynomialPauliR. However, when ancillae are only used in qfree parts of the circuit, it is possible to isolate those qfree parts and apply `classical_to_reversible_optim`



(a) Gate Counts.

(b) Qubit counts.

Figure 10. Gates and qubits for different algorithm parameters using Qiskit++ (——; this work) and Qiskit (- - -). Some implementations were improved as a result of our bug reports (see Footnote 4), shown as -·-·-. For WeightedAdder, we show an alternative Qiskit++ implementation (——) trading gates for qubits. Lower values are better.

to these, before combining them with the non-qfree parts of the algorithm. For example, Fig. 9a shows an extract of the Grover implementation for input size 3. The ancilla `a0` is only used in the first two lines. Thus, we can apply Quipper `classical_to_reversible_optim` to the circuit generated by those two lines, and append to its result the gate corresponding to the third line. This strategy allows applying Quipper uncomputation to Grover and Deutsch-Jozsa. However, this approach not only requires manual intervention but also results in less efficient circuits (see Tab. 2).

When ancillae are used in parts of the circuit that are not purely qfree—as shown for instance in Fig. 9b on the code for MCRY with input size 3—the above separation is not possible. This is the case for MCRY, PiecewiseLinearR and PolynomialPauliR. Thus, Quipper cannot synthesize uncomputation for these examples (see ✗ in Tab. 2).

7.4.1 Reduction in Gate Count. For all but three algorithms, Qiskit++ significantly reduces the number of gates compared to Qiskit, by up to 99.5%. These extremely high savings are partially due to a regression bug in Qiskit (see Footnote 4), which Unqomp helps to avoid. Even for algorithms not affected by this regression, or after fixing this regression (* in Tab. 2), Qiskit++ allows for significant savings of up to 48%. On the three algorithms where Qiskit++ does not outperform Qiskit, both yield circuits with identical size. Being well-studied quantum algorithms, the implementations of Grover and Deutsch-Jozsa have been manually optimized by experts to reduce gate and qubit counts. Similarly, MCX has been heavily studied and optimized [4].

Compared to Quipper, Qiskit++ almost always produces fewer gates, with savings of up to 66%. Quipper outperforms Qiskit++ only on Multiplier, due to an optimization pass performed by Quipper but not Qiskit++. In particular, Quipper applies constant propagation and can for example remove CCX gates if one of their controls is known to be 0. We note that this optimization is orthogonal to our work and could in principle be integrated in Qiskit++ as well. When disabling the optimization in Quipper, Qiskit++ consistently produces fewer gates than Quipper.

Origin of Reductions. Overall, the gate savings of Qiskit++ can be explained by (i) redundant uncomputation in the original implementation (see Fig. 1) and (ii) *CCX* gate optimizations performed during our compilation (see §5.4).

Redundant uncomputation concerns MCRY, PiecewiseLinearR, PolynomialPauliR, and WeightedAdder, as these examples rely on libraries for integrating sub-components in a modular manner.

CCX gate optimizations lead to significant gains for all algorithms, except for those where no savings were observed. Note that this optimization is already partially applied in the

baseline Qiskit implementations (manually to MCX, and indirectly to all examples that use MCX). Consistently applying this optimization to the Qiskit baseline would be virtually impossible without Unqomp: it would require knowing which CCX gates are later uncomputed, which is impossible to determine for library functions computing values which may or may not be used as ancillae.

For Quipper, an additional source of gate savings lies in the more fine-grained control allowed by Qiskit++. Algorithms must be implemented as Haskell functions in Quipper. Hence, many optimizations, such as temporarily changing the value of an input argument in-place, cannot be used.

Asymptotic Gains. Fig. 10a shows the effect of varying some circuit parameters for selected algorithms: efficiency gains often increase with increasing complexity. Even after the fixes following our bug reports (Footnote 4), Unqomp allows for a significant reduction in all shown algorithms.

Gate Explosion for Recursive Calls. As visualized in Fig. 1, using library functions in Qiskit leads to redundant uncomputation. We now demonstrate that this effect is arbitrarily amplified by nested library calls. As a consequence, we should expect Unqomp to yield even larger efficiency gains when quantum algorithms become increasingly complex.

We implemented a toy algorithm RecursiveSeq that computes the sequence $x_{n+1} = 2x_n + 1$ according to its recursive definition: the function computing x_{n+1} uses a recursive function call to compute x_n on an ancilla a and returns $2a + 1$. We implemented this algorithm in Qiskit by explicitly performing uncomputation in each recursive call. In Qiskit++, uncomputation is automatic. As shown in Fig. 10a, this leads to an exponential increase of gate counts for Qiskit, while the count only increases linearly for Qiskit++. This suggests that more generally, Unqomp allows significant efficiency gains for complex algorithms with deep call trees.

7.4.2 Reduction in Qubit Count.

Tab. 2 also compares the number of qubits for the implementations, and Fig. 10b shows the impact of varying algorithm parameters on the number of qubits.

Overall, Quipper yields significantly higher qubit counts. Compared to Qiskit, Qiskit++ results in an identical number of qubits for many algorithms, showing that Unqomp's ancilla allocation (§5.4) can often compete with manual allocation. Furthermore, for some examples, Unqomp yields a significant reduction in qubits, indicating that a completely manual approach fosters errors and missed optimizations.

For RecursiveSeq, Unqomp finds a non-trivial qubit allocation that is hard to detect manually, and thus only requires a constant number of qubits.

For PolynomialPauliR, the number of qubits in Qiskit++ remains constant for polynomial degrees d above 8, while it increases linearly for Qiskit. This is due to the Qiskit implementation allocating d qubits in a hard-coded fashion (see §7.3). However, a detailed analysis of the code shows that the number of required ancillae is actually only $\min(d, n)$, where n is the input size. In contrast to Qiskit, Unqomp automatically finds this improved, non-trivial qubit allocation. Similarly, for PiecewiseLinearR, Unqomp finds a more efficient qubit allocation, using n ancillae for an input size n, instead of $n + b$, where b is the number of breakpoints.

Table 3. Comparing Unqomp to related approaches.

Approach/Language	Autom. Uncomp.	Circuit
Quipper [13]	(qfree)	✓
Revs [17]	(qfree)	✓
ReVerC [3]	(qfree)	✓
ReQWire [21]	(qfree)	✓
Silq [6]	✓	✗
Unqomp (this work)	✓	✓

Trading off Qubits and Gates. Unfortunately, the gate count reduction for WeightedAdder comes at the cost of more qubits. This is a result of Unqomp performing uncomputation later in the circuit than the Qiskit implementation, which prevents some qubit reuse. Still, using a slightly modified alternative Qiskit++ implementation, we can trade Unqomp's gate savings for fewer qubits, resulting in identical qubit counts (see Tab. 2). Similarly, for MCRY, Qiskit++ requires exactly one more qubit than the original Qiskit implementation, as it uses slightly different code—instead of two MCX gates it uses only one MCX with its uncomputation and an extra ancilla qubit. This trade-off is only interesting when using automatic uncomputation: as MCX uses a lot of internal auxiliary values, modular manual uncomputation is quite expensive, while automatic uncomputation is cheap, as illustrated in Fig. 1. This cheap uncomputation allows Qiskit++ to produce half as many gates as Qiskit, at the cost of only one extra qubit.

8 Related Work

We now discuss existing works related to Unqomp both in terms of (i) our goal of synthesizing uncomputation, and (ii) key aspects of our approach to address this goal.

Automatic Uncomputation. Tab. 3 summarizes existing approaches to handle automatic uncomputation. As discussed next, these approaches differ from Unqomp in that they either do not provide a compilation to circuits, or only apply to classical computation. We note that ReQWire [21] can additionally prove that a manually provided uncomputation is safe. However, it cannot automatically synthesize uncomputation, except for purely classical circuits. Tab. 3 omits SQUARE [9] as it does not synthesize uncomputation: SQUARE expects the programmer to manually provide and mark the uncomputation code blocks for each ancilla, and then saves qubits or operations by interleaving those blocks.

```
operation ErroneousUncomputation(x : Qubit) : Unit
{
    use ancilla = Qubit();
    ApplyWith(CopyX, ModifyX, (x, ancilla));
    // Error: Ancilla is not in zero state
}

operation CopyX(x : Qubit, ancilla : Qubit) : Unit is Adj
{
    CNOT(x, ancilla);
}

operation ModifyX(x : Qubit, ancilla : Qubit): Unit is Adj
{
    CNOT(ancilla, x);
    // Error: modifies x which is needed for uncomputation
}
```

Figure 11. `ApplyWith` producing erroneous uncomputation.

Silq: Enable Safe Uncomputation. Silq is the closest work to Unqomp in that it also promises automatic uncomputation and relies on analogous high-level insights for ensuring correctness, namely the notion of *qfree* gates and *controls* (cp. §6.3). However, while Silq's type system ensures that safe uncomputation is possible for all temporary values, Silq does not provide a compiler that synthesizes this uncomputation. In contrast, Unqomp synthesizes uncomputation, which allows extending arbitrary circuit-based languages (such as Qiskit [2]) to support automatic uncomputation. We can view Unqomp as a key step towards compiling Silq, which in particular requires automatically generating safe uncomputation.

Qfree Programs. Quipper [13], Revs [17], ReverC [3], and ReQWire [21] support automatic uncomputation only for classical programs, i.e., programs that only use qfree functions. Further, ReverC only uncomputes boolean expressions, meaning it is not applicable to any of our examples in Tab. 1. In contrast, only Silq and Unqomp support uncomputation in quantum programs that interleave qfree with non-qfree operations. Only supporting qfree computations is a severe restriction—half of the programs we evaluated (see §7) are not fully qfree. Further, the proposed workflow for these approaches is to compile the classical part of a quantum program and then insert the result into the resulting quantum circuit. However, this workflow cannot always be applied, as shown in Fig. 9, and generally results in inefficient circuits, analogously to Fig. 1.

Convenience Functions. Various quantum languages offer convenience functions that simplify manual uncomputation, such as `ApplyWith` in Q# [22] or `with_computed` in Quipper [13]. However, relying on these features cannot guarantee the resulting uncomputation is safe, as incorrectly using them does not result in an error.

For example, Fig. 11 shows Q# code using `ApplyWith` to uncompute a in Fig. 6b. As uncomputing a is physically impossible (see §6.2) and `ApplyWith` performs no static checks, this program results in an incorrect circuit whose error is only detected at runtime (i.e., during simulation).

In addition to being unsafe, convenience functions are often tedious to use. For instance, `ApplyWith` cannot be used in combination with for-loops, forcing even expert Q# developers to resort to manual uncomputation in some cases. [5] Finally, convenience functions often generate inefficient circuits, as explained in Fig. 1.

Circuit Graphs. Various existing works have represented circuits in terms of circuit graphs. In classical computation, dependency graphs have long been used to represent computations without enforcing irrelevant ordering constraints (see e.g., [12], [1, §5.2]). Naturally, works in this domain do not discuss quantum computations or quantum circuits.

Multiple works in quantum computation operate on graph-based circuit representations. However, as none of them are geared towards uncomputation, their graphs (i) do not distinguish between target, control, and anti-dependency edges [10, 11, 15], (ii) are often limited to only a few types of gates [10, 11], and (iii) are not suitable for inserting (uncomputation) gates because they do not contain enough information to reconstruct the circuit they represent [15].

Ancilla Allocation. Our approach to ancilla allocation (§5.4) is an instantiation of linear scan register allocation [18], with one key simplification: instead of a fixed number of registers (and the option of spilling to the heap), we have an unlimited number of potential ancillae. The cost of a specific allocation is hence simply the number of ancillae used, instead of the cost of the operations on spilled registers, allowing for an optimal allocation given a graph linearization.

9 Conclusion

We presented Unqomp, a procedure synthesizing automatic uncomputation for quantum circuits, using an internal representation in terms of circuit graphs. Unqomp can be readily integrated into existing quantum languages, which we demonstrated by extending Qiskit to Qiskit++.

Our evaluation showed that Unqomp reduces the amount of code, improves code modularity, and yields substantially more efficient circuits in terms of number of gates and qubits.

[5] For an example, see https://github.com/microsoft/QuantumKatas/blob/7ba83e55703fda4ff945fc6e89050f4ee179e5bc/RippleCarryAdder/ReferenceImplementation.qs#L73

References

[1] Alfred Aho, Monica Lam, Ravi Sethi, and Jeffrey Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd edition ed.). Addison Wesley, Boston.

[2] 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

[3] Matthew Amy, Martin Roetteler, and Krysta M. Svore. 2017. Verified Compilation of Space-Efficient Reversible Circuits. In *CAV'17*. Vol. 10427. Cham, 3–21. https://doi.org/10.1007/978-3-319-63390-9_1

[4] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. Smolin, and H. Weinfurter. 1995. Elementary gates for quantum computation. *Physical Review A* 52, 5 (Nov. 1995), 3457–3467. https://doi.org/10.1103/PhysRevA.52.3457 arXiv: quant-ph/9503016.

[5] Charles H Bennett. 1973. Logical Reversibility of Computation. *IBM Journal of Research and Development* 17, 6 (Nov. 1973), 525–532. https://doi.org/10.1147/rd.176.0525

[6] 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. https://doi.org/10.1145/3385412.3386007

[7] Cirq Development Team. 2020. Cirq Circuit Examples. https://github.com/quantumlib/Cirq Accessed: 2020-10-29.

[8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to algorithms*. MIT press.

[9] Yongshan Ding, Xin-Chuan Wu, Adam Holmes, Ash Wiseth, Diana Franklin, Margaret Martonosi, and Frederic T. Chong. 2020. SQUARE: Strategic Quantum Ancilla Reuse for Modular Quantum Programs via Cost-Effective Uncomputation. *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)* (May 2020), 570–583. https://doi.org/10.1109/ISCA45697.2020.00054 arXiv: 2004.08539.

[10] Lucas Dixon and Ross Duncan. 2009. Graphical Reasoning in Compact Closed Categories for Quantum Computation. *arXiv:0902.0514 [cs]* (Feb. 2009). http://arxiv.org/abs/0902.0514 arXiv: 0902.0514.

[11] Ross Duncan and Simon Perdrix. 2010. Rewriting Measurement-Based Quantum Computations with Generalised Flow. In *Automata, Languages and Programming (Lecture Notes in Computer Science)*, Samson Abramsky, Cyril Gavoille, Claude Kirchner, Friedhelm Meyer auf der Heide, and Paul G. Spirakis (Eds.). Springer, Berlin, Heidelberg, 285–296. https://doi.org/10.1007/978-3-642-14162-1_24

[12] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (July 1987), 319–349. https://doi.org/10.1145/24039.24041

[13] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *PLDI'13*. ACM Press, Seattle, Washington, USA. https://doi.org/10.1145/2491956.2462177

[14] Arthur B Kahn. 1962. Topological sorting of large networks. *Commun. ACM* 5, 11 (1962), 558–562. https://doi.org/10.1145/368996.369025

[15] Atsushi Matsuo and Shigeru Yamashita. 2012. Changing the Gate Order for Optimal LNN Conversion. In *Reversible Computation (Lecture Notes in Computer Science)*, Alexis De Vos and Robert Wille (Eds.). Springer, Berlin, Heidelberg, 89–101. https://doi.org/10.1007/978-3-642-29517-1_8

[16] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum computation and quantum information* (10th anniversary ed ed.). Cambridge University Press, Cambridge ; New York.

[17] Alex Parent, Martin Roetteler, and Krysta M. Svore. 2015. Reversible Circuit Compilation with Space Constraints. *arXiv:1510.00377 [quant-ph]* (Oct. 2015). http://arxiv.org/abs/1510.00377 arXiv: 1510.00377.

[18] Massimiliano Poletto and Vivek Sarkar. 1999. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21, 5 (1999), 895–913. https://doi.org/10.1145/330249.330250

[19] John Preskill. 2018. Quantum Computing in the NISQ era and beyond. *Quantum* 2 (2018), 79.

[20] Qiskit Development Team. 2020. Qiskit Circuit Library. https://qiskit.org/documentation/apidoc/circuit_library.html Accessed: 2020-10-27.

[21] Robert Rand, Jennifer Paykin, Dong-Ho Lee, and Steve Zdancewic. 2019. ReQWIRE: Reasoning about Reversible Quantum Circuits. *Electronic Proceedings in Theoretical Computer Science* 287 (Jan. 2019), 299–312. https://doi.org/10.4204/EPTCS.287.17 arXiv: 1901.10118.

[22] Krysta Svore, Martin Roetteler, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, and Andres Paz. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*. ACM Press, Vienna, Austria. https://doi.org/10.1145/3183895.3183901

[23] Google AI Quantum Team. 2017. Cirq. (2017). https://github.com/quantumlib/Cirq

[24] Umesh Vazirani. 2013. Quantum Mechanics and Quantum Computation (CS191x). Online Lecture (Lecture 7). https://www.youtube.com/watch?v=XPkKRBk71TY&list=PLDAjb_zu5aoFazE31_8yT0OfzsTcmvAVg&index=30