Optimizing Ancilla-Based Quantum Circuits with SPARE

RITVIK SHARMA, Stanford University, USA
SARA ACHOUR, Stanford University, USA

Many quantum algorithms instantiate and use ancillas, spare qubits that serve as temporary storage in a quantum circuit. In particular, many recently developed high-level and modular quantum programming languages (QPLs) use ancilla qubits to implement various programming constructs. These are lowered to circuits with nested/cascading compute-uncompute gate sequences that use ancilla qubits to track internal state. We present SPARE, a rewrite-based quantum circuit optimizer that restructures these compute-uncompute gate sequences, leveraging the ancilla qubit state information to optimize the circuit. In this work, we prove the correctness of SPARE's rewrites and link SPARE's gate-level transforms to language-level program rewrites, which may be performed on the input language. We evaluate SPARE on QPL-generated quantum circuits against Unqomp and Spire, two optimizing compilers for QPLs. SPARE achieves a reduction of up to 27.3% in qubit count, 56.7% in 2-qubit gates, 68.2% in 1-qubit gates and 73.9% in depth against Unqomp, and up to 17.8% in qubits, 67.3% in 2-qubit gates, 61.4% in 1-qubit gates and 59.9% in depth against Spire. We also evaluate SPARE against the Quartz, Feynman, and PyZX circuit optimizers: SPARE achieves up to a 70.0% reduction in two-qubit gates, up to a 53.6% reduction in 1-qubit gates, and up to a 56.7% reduction in depth compared to the best result from all the gate-level optimizers.

CCS Concepts: • **Computer systems organization** → **Quantum computing**.

Additional Key Words and Phrases: quantum programming languages, quantum optimizing compilers

ACM Reference Format:
Ritvik Sharma and Sara Achour. 2025. Optimizing Ancilla-Based Quantum Circuits with SPARE. *Proc. ACM Program. Lang.* 9, PLDI, Article 154 (June 2025), 25 pages. https://doi.org/10.1145/3729253

1 Introduction

Quantum computing [37] promises a large computational advantage over classical computing and has applications in cryptography, combinatorial optimization, sensing, and machine learning [5, 7, 16, 45]. Recently, practitioners have developed quantum programming languages (QPLs) [1, 6, 22, 51, 54] that enable ergonomic development of scalable quantum computations. These languages enable modular circuit design through the introduction of functions and composable circuits. Some languages also support the integration of accessible control flow constructs, such as if-else statements and with-do blocks, enabling the implementation of quantum data structures and lowering the barrier to entry to quantum computing. In these usages, a quantum program is translated into a quantum circuit with an optimizing compiler (QOC), which translates language constructs to sequences of quantum gates – the quantum circuit may then be lowered to hardware [38, 50, 55].

1.1 Challenges with Optimizing QPL-Generated Circuits

While high-level QPLs are more accessible to programmers and support modular design and reuse, the circuits generated by their associated compilers have a large depth and require a large number of gates. Even simple quantum programs, when compiled, result in circuits containing hundreds

Authors' Contact Information: Ritvik Sharma, Stanford University, Stanford, USA, rsharma3@stanford.edu; Sara Achour, Stanford University, Stanford, USA, sachour@stanford.edu.

ACM 2475-1421/2025/6-ART154
https://doi.org/10.1145/3729253

to tens of thousands of gates. We observe that these QPL-generated quantum circuits have certain common features that can be leveraged for optimization:

- *Ancilla qubits.* In modular quantum circuits and control-flow blocks, transient internal quantum states are stored in the circuit using *ancilla* qubits. Ancilla qubits are typically instantiated to the $|0\rangle$ state before use, and then reset after the sub-computation is completed.
- *Compute-transform-uncompute gates.* For many of these programs, control-flow logic and compute/uncompute operations when compiled to gate-level circuits have nested and sequential compute-transform-uncompute (CTU) gates. In the CTU gate pattern, information is first stored onto the state of a subset of qubits (usually an ancilla) through compute gates. The transform gates then modify the quantum state of both ancilla and regular qubits. Finally, uncomputation is performed by reversing the compute gate operation.
- *Multicontrolled gates.* Control flow constructs introduce multi-controlled gates into the circuit, where the gate's controls capture information about the predicates enclosing the block.

In practice, circuits with compute/uncompute logic and ancilla qubits are not limited to QPLs and also appear in arithmetic, cryptographic, and error-correction circuits [10, 27, 30, 32, 37].
Challenges with Current QOCs. Compilers that target QPLs and gate-level quantum circuit optimizers often do not fully exploit the CTU gate patterns and their associated ancilla information, missing various optimization opportunities [28, 33, 34, 36, 52, 53]. In addition, some gate-level quantum circuit optimizers lack the scalability to support the frequently large QPL-generated quantum circuits.

1.2 Optimization of Ancilla-Based Quantum Circuits with SPARE

We present SPARE, a gate-level quantum circuit optimizer for circuits that have both compute-transform-uncompute patterns and ancilla qubits, such as those produced by high-level languages. SPARE works with a multi-controlled quantum circuit representation that carries ancilla information and deploys new rewrites to reduce the complexity of circuits with the above features:

- SPARE employs rewrites that optimizes compute/uncompute logic in compute-transform-uncompute (CTU) gate sequences and decompose difficult-to-optimize gates into smaller CTU gate sequences.
- SPARE uses ancilla information to delete "dead" gates whose control conditions will never be satisfied and "trivial" controls which will always be satisfied. These rewrites preserve a relaxed notion of correctness that incorporates ancilla state information. To better utilize ancilla information, SPARE employs variable propagation methods which propagate ancilla information deeper into the circuit.
- SPARE deploys a highly scalable rewrite algorithm that applies rewrites speculatively and utilizes circuit cost heuristics to find sequences of rewrites that effectively optimize the circuit.

SPARE offers front-ends for Unqomp [38] and Spire [55] compilers and enables further optimization. Compared to Unqomp, SPARE reduces qubit counts, 2-qubit gates, 1-qubit gates and depth by an average of 5%, 25.1%, 27.4% and 24.0% and by up to 27.3%, 56.7%, 78.2% and 73.9%, respectively. Compared to Spire,SPARE also qubit counts, 2-qubit gates, 1-qubit gates and depth by an average of 8.3%, 42.2%, 34.3%, 31.6% and by up to 17.7%, 67.3%, 61.4% and 59.9%, respectively.

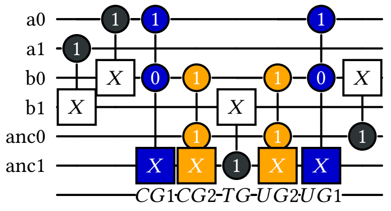
We also re-interpret a subset of SPARE's gate-level rewrites as high-level program transformations amenable to integration into compilers that target high-level QPLs [54, 55]. By lifting SPARE's gate-level rewrites to a program-level representation, we enable the application of gate-level transforms at an earlier stage of compilation, and at a higher level of abstraction.

1.2.1 Contributions. We present the following contributions:

- We define novel correctness-preserving rewrites that target circuits with compute-transform-uncompute (CTU) gate patterns and ancilla qubits and prove their correctness.

```
a,b,anc = QReg(n),QReg(n),AncReg(n)
c = AncCirc() # Declare circuit
for i in range(n-1):
   c.append(carryGate(), anc[i],a[i],b[i],anc[i+1])
   c.append(sumGate(), anc[i],anc[i],b[i])
c.append(sumGate(), anc[n-1],a[n-1],b[n-1])
c.circuitWithUncomputation() # Uncompute
```

(a) Unqomp program for b:=a+b

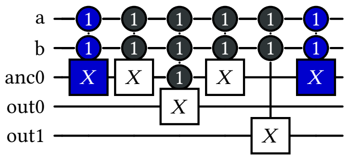


(b) Compiled circuit with CTU pattern.

Fig. 1. Example Adder program and the corresponding Unqomp-generated circuit for $n=2$. The two different compute-uncompute pairs are highlighted as 🟦 and 🟧 respectively.

```
if a
   with { let anc0 <- b;}
   do
      if b {
         let out0 <- not anc0;
         let out1 <- true;}
   Undo with {anc0 -> b;}
```

(a) Example Tower program



(b) Compiled circuit where **with..do..undo** forms CTU gates.

Fig. 2. Tower program and the corresponding circuit with compute-uncompute gates marked 🟦.

- We present the SPARE optimizing compiler, which rewrites circuits with ancilla and CTU gate patterns to reduce depth and gate count. The compiler uses static analysis to infer ancilla information from circuits, and deploys new algorithmic optimizations to improve performance.
- We re-interpret a subset of the gate-level rewrites as high-level program optimizations, identifying a subset of impactful high-level quantum program transformations.
- We evaluate SPARE's ability to optimize both circuits generated from Spire and Unqomp compilers as well as hand-implemented circuits that naturally have CTU structures. Our evaluation compares SPARE against other circuit optimizers, such as Quartz [53], Feynman [2] and PyZX [24].

2 Background: Quantum Programming Languages

Researchers have previously developed quantum programming languages (QPLs) [54, 55] that contain information-rich, high-level programming abstractions to enable the modular development of quantum circuits and offer familiar program constructs such as control flow operations. We consider quantum circuits produced from two different high-level quantum programming frameworks and languages which are:

- Languages and tools that provide explicit support for automated uncomputation with a type-system for qubits like Silq [6], Unqomp [38]. While Silq does not directly compile to quantum circuits, Unqomp can automatically synthesize circuits with efficient uncomputation.
- The Tower language [54] supports writing quantum programs with control flow operations and quantum data structures which can then be lowered to quantum circuits.

These systems offer different programming models – Silq and Unqomp enable safe, automatic uncomputation, while Tower supports control-flow programs. Though the programming models are very different, they all produce circuits with CTU gate patterns that employ ancilla qubits.

2.1 Languages with Automated Uncomputation and Unqomp Compiler

Silq [6] is a modular QPL designed to enable the construction of complex computations from smaller, composable components. Silq is the first language to support automatic uncomputation through a type system that tracks quantum operations and registers, and targets circuits that use ancilla qubits to store intermediate results. This system introduces syntax like `forget` and `dispose` that explicitly mark ancilla or temporary variables for uncomputation. Consequently, the resulting programs exhibit a structured pattern of compute, transform, and uncompute operations. However, Silq does not come with an end-to-end compiler and developing such a compiler is an active research area [38, 39, 50].

Compiling Silq requires automatic synthesis of uncomputation. Unqomp [38] automatically compiles efficient uncompute gates for quantum circuits described in a quantum circuit-based language frontend called Qiskit++ [11]; this frontend extends on Qiskit [21]. It optimizes programs that contain compute and transform operations described with a type system to annotate ancillas, as shown for an Adder in Figure 1a. Unqomp analyzes the compute gates in the circuit and generates the uncompute gates necessary to undo the compute operations, unentangling the ancilla qubits from other qubits in the process. Therefore, compiling programs in Unqomp results in circuits that naturally have compute-transform-uncompute (CTU) gate patterns that may be nested and involve ancilla qubits.

As both Silq and Unqomp have a qubit-type system, certain Silq programs can be ported to equivalent Qiskit++ programs [50]. For instance, the Qiskit++ program for an Adder in Figure 1a is equivalent to a Silq program that performs addition. The corresponding Unqomp-generated circuit is shown in Figure 1b. It has a nested CTU structure: different compute/uncompute gates are highlighted blue and orange, respectively; `TG` is a transform gate, and `anc0`, `anc1` qubits are ancillas. However, Silq also supports features not present in Unqomp, such as recursion and variable-length quantum registers.

2.2 Tower Language and Spire Compiler

Tower [54] is a high-level QPL that supports control flow constructs, such as if-else statements, loops, functions, and mutable data. With Tower, programmers can define quantum data structures, such as stacks and lists, and build modular quantum programs. Figure 2 presents a Tower program and its corresponding quantum circuit. In this example, the conditionals map to controlled gates in the circuit, and the `with` block creates an ancilla qubit for the `anc0` internal variable, forming a CTU gate pattern. In the circuit implementation, the compute/uncompute gates (blue) instantiate and recover the state of the ancilla qubit before and after applying the `do` block. The `do` block is lowered to a sequence of transform gates that operate on the ancilla that maps to the `anc0` temporary variable in the high-level program.

Spire [55] efficiently compiles Tower programs to quantum circuits composed of various quantum sub-circuits corresponding to functions in the Tower program. Such sub-circuits contain input qubits for function arguments, output qubits for results, and ancilla qubits to store variables local to the function. Ancilla qubits are all initialized to zero. Various control flow constructs such as the `with` operation, multi-controlled operations and arithmetic operations are implemented with ancilla qubits. While the ancilla sites corresponding to the `with` block are visible in the high-level code, other ancillae used to implement arithmetic calculations, state swaps, etc., are not. The compiler synthesizes all circuit fragments for each function and assembles them to construct the full quantum circuit.

Spire Program-level Rewrites. Spire also includes two main program-level rewrites, conditional narrowing (`cn`) and conditional flattening (`cf`). These high-level rewrites are applied using a cost model heuristic to reduce circuit complexity in terms of T-gates. The `cn` rewrite moves any conditional statements outside a `with-do-undo` block to inside the `do` block of the program. It simplifies the circuit by dropping from compute/uncompute gates, any controls shared by all compute-transform-uncompute gates. The `cf` rewrite optimizes circuit fragments conditioned on multiple variables by creating new temporary variables that store the results of any multi-variable conditional statements in the program.

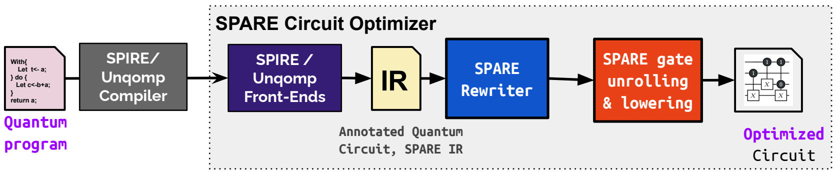


Fig. 3. SPARE compilation flow. **IR** circuits retain ancilla information and are comprised of multi-controlled single-target gates. Initial circuits expressed in SPARE IR and the final optimized circuit have the same number of data qubits (n) but different numbers of ancillas (k and m) while preserving the circuit semantics.

Table 1. Notation reference for rewrite correctness discussions

element	description	element	description
$s \in S \subseteq \{0,1\}$	Set of qubit basis states.	Q	A circuit fragment composed of sequential gates
n	Number of qubits in circuit	$U(Q)$	Unitary of a circuit fragment Q
$C(s) = \|s\rangle\langle s\|$	Control on basis state s.	$Z_G[i]$	2×2 matrix operator applied by gate G on q_i.
$I(2^i)$	$2^i \times 2^i$ identity matrix	$G_i = Gate(Z_G, U, k)$	Gate G with target qubit k and unitary U
U_{2^i}	$2^i \times 2^i$ unitary on i qubits.	$\pi(Z_G[i:j])$	Outer product of 2x2 matrices in Z_G for qubits q_i to q_j inclusive.
$U^\dagger$	Inverse of unitary U.		

3 SPARE Compiler Overview

We present SPARE, a compiler that automatically optimizes quantum circuits with CTU patterns and ancilla qubits. The compilation flow is outlined in Figure 3. SPARE optimizes quantum circuits compiled from high-level programming languages that contain CTU patterns and ancilla qubits (see Section 2). The technical presentation of SPARE is organized as follows:

- We first present the rewrites used by the SPARE compiler to optimize circuits. In Section 4 we present the atomic rewrites and prove they preserve circuit correctness. In Section 4.8 we present the correct-by-construction CTU gate pattern rewrites, which are comprised of these atomic rewrites.
- We then present the SPARE compiler in Section 5. Section 5.1 presents the SPARE IR and describes our custom front-ends that translate Spire/Unqomp-generated circuits to the SPARE IR. Then, the SPARE rewrite engine (Section 5.2) optimizes these circuits using the SPARE IR annotations to identify ancilla. Finally, SPARE lowers circuits to a Clifford+T basis gateset adapting common lowering optimizatons as described in Section 5.4.

4 Correctness-Preserving SPARE Rewrites

We first describe the atomic rewrites used by SPARE, and provide proofs for their correctness. We use the notation described in Table 1 and Section 4.1. We then formally define equivalence in Section 4.2 and discuss a proof sketch for our atomic transformations (Sections 4.3-4.7). The more sophisticated compute-transform-uncompute rewrites (Section 4.8) are composed of these atomic, correctness-preserving rewrites.

4.1 Quantum Circuit Notation and State Information

SPARE rewrites operate on quantum circuits that are comprised of multi-controlled single-target quantum gates with qubit wire annotations. These wire annotations mark the qubit segment if it is restricted to a quantum basis state $|0\rangle, |1\rangle$ and are derived through static analysis of the circuit (see Section 5.1). Any gate given by $G = Gate(Z, U, k)$ can have multiple controls each annotated with a control state $|0\rangle$ *or* $|1\rangle$ and a target qubit q_k on which it applies the unitary matrix U when all the control conditions are satisfied. For the gate G, at each qubit q_i we define a 2×2 matrix operator

$Z[i]$ which is one of the following: ① $I(2)$ if gate G does not operate on qubit q_i, ② $C(s)=|s\rangle\langle s|$ if gate G is controlled on state s on qubit q_i and ③ target unitary U if $i=k$. Since this gate is a multi-controlled single target gate, we use this operator and the tensor-product notation from [18, 26, 42] to describe the unitary of gate G over all n qubits. The gate applies the target unitary if all control conditions are satisfied $(\pi(Z[1:k-1])\otimes U\otimes\pi(Z[k+1:n])])$ and $I(2)$ otherwise $((I(2^n)-\pi(Z[1:k-1])\otimes I(2)\otimes\pi(Z[k+1:n])]))$. Thus, the unitary of gate G is given by $U(G)$ is equal to:

$$U(G)=\pi(Z)+(I(2^n)-\pi(Z[1:k-1])\otimes I(2)\otimes\pi(Z[k+1:n])])=\pi(Z[1:k-1])\otimes(U-I(2))\otimes\pi(Z[k+1:n])+I(2)$$

A circuit fragment Q is defined as a series of sequential gates and is represented as $Q=[G1,G2...Gm]$. The unitary of the circuit fragment is simply the product of these gate unitaries in reverse i.e. $U(Q)=U(Gm)\times...U(Gj)...U(G1)=\prod_{j=1}^{n}U(Gj)$. The unitary of the circuit fragment can be similarly derived as the product of unitaries of individual fragments.

4.2 Correctness Criteria: Preservation of Quantum Circuit Unitary

We define SPARE rewrites as transformations over circuit fragments. Given a rewrite $Q\rightarrow Q'$ that transforms a quantum circuit fragment Q that implements a unitary $U(Q)$ to a circuit fragment Q' that implements a unitary $U(Q')$, a valid rewrite preserves one of two kinds of correctness:

Definition 1. *Strict Correctness (Type-1).* The rewrite exactly preserves the quantum circuit's unitary, that is $U(Q)=U(Q')$. Therefore, the difference between unitaries is:

$$\Delta U(Q)=U(Q)-U(Q')=0 \tag{1}$$

This correctness definition is used in most prior quantum circuit rewrite systems [28, 51–53].

Definition 2. *Ancilla Correctness (Type-2).* The rewrite preserves the quantum circuit's unitary over the subset of states the qubits may be in. In this relaxed definition of correctness, we take advantage of the fact that ancilla qubits may start and end in a basis quantum state ($|0\rangle$ or $|1\rangle$) at the circuit interfaces. Consider a quantum fragment Q that operates on qubits $[q_1,q_2...q_k...q_n]$, where a subset of qubits $q_l\in AncQ0$ are ancilla qubits that are restricted to the quantum state $|0\rangle$, and $q_k\in AncQ1$ are qubits restricted to quantum state $|1\rangle$. A rewritten quantum circuit fragment Q' is Type-2 equal to Q if the unitary difference $\Delta U(Q)$ has the following form:

$$\Delta U(Q)=U(Q')-U(Q)=\sum_{q_l}^{AncQ0}U_{2^{l-1}}\otimes C(1)\otimes U_{2^{n-l}}+\sum_{q_l}^{AncQ1}U_{2^{l-1}}\otimes C(0)\otimes U_{2^{n-l}} \tag{2}$$

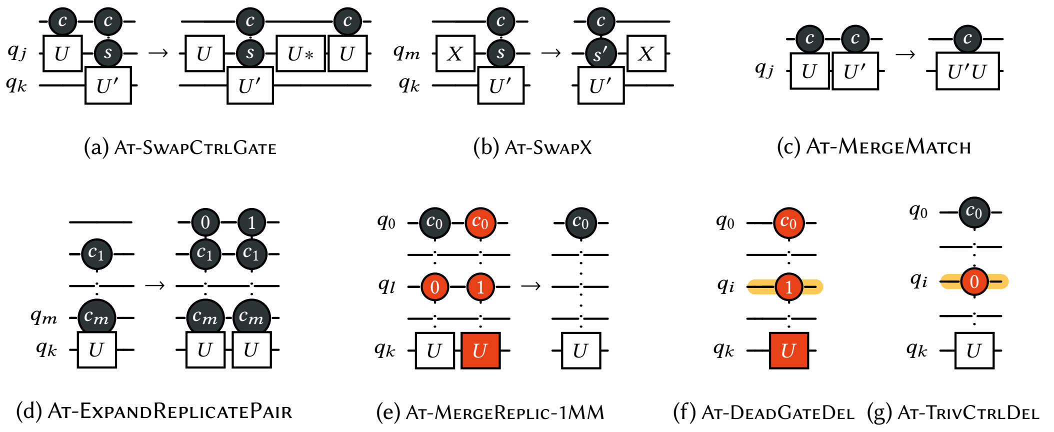
Intuitively, the quantum circuit fragments are equal if, when the difference is taken, all non-zero summands contain a control $C(0)$ or $C(1)$ on a qubit restricted to a basis state $|1\rangle$ or $|0\rangle$ respectively. For ancilla qubits $q_l\in AncQ0$, each summand contains at least one control $C(1)$ for a qubit that is only in the $|0\rangle$ state. Similarly, for qubits $q_l\in AncQ1$, each summand contains at least one control $C(0)$ for a qubit that is only in the $|1\rangle$ state. Because these qubits are never in the quantum state required to make the summand non-zero, the unitary difference $\Delta U(Q)$ evaluates to zero when only the valid quantum states over all the qubits are considered.

Lemma 1. *Preserving correctness over fragments preserves circuit correctness.* If a transformation that modifies a circuit $Q_C=[Q_A,Q,Q_B]$ to $Q_C=[Q_A,Q',Q_B]$ preserves correctness over the circuit fragment ($\Delta U(Q)$ equals Equation 1 or 2) then the transformation preserves correctness on the entire circuit ($\Delta U(Q_C)$ equals Equation 1 or 2).

Proof Sketch. For the given quantum circuit Q_C the circuit unitary before and after transformation is given by $U(Q_C)=U(Q_B)\times U(Q)\times U(Q_A)$ and $U(Q'_C)=U(Q_B)\times U(Q')\times U(Q_A)$. The difference between the unitaries factoring out terms is

$$\Delta U(Q_C)=U(Q_B)\times[U(Q')-(Q))]\times U(Q_A)=\Delta U(Q_C)=U(Q_B)\times\Delta U(Q)\times U(Q_A)$$

If $\Delta U(Q)$ satisfies Type-1 equality, the above equation will trivially satisfy Type-1 equality as well. If $\Delta U(Q)$ satisfies Type-2 equality, any qubit $q_i\in AncQ0$ is restricted to $|0\rangle$ state at the fragment interface.



(a) At-SwapCtrlGate (b) At-SwapX (c) At-MergeMatch

(d) At-ExpandReplicatePair (e) At-MergeReplic-1MM (f) At-DeadGateDel (g) At-TrivCtrlDel

Fig. 4. Atomic rewrites for moving, deleting, merging, and expanding gates. Deleted controls and gates are highlighted in red (🟥), wires with qubits in the basis $|0\rangle$ state are shaded in yellow (🟨).

We replace $U(Q_A)$,$U(Q_B)$ with general unitary matrices represented in the tensor product notation as $U(Q_A) = \sum \pi(Z_A[1:i-1]) \otimes U_A \otimes \pi(Z_A[i+1:n])$,$U(Q_B) = \sum \pi(Z_B[1:i-1]) \otimes U_B \otimes \pi(Z_B[i+1:n])$. We calculate the $\Delta U(Q_C)$ term and use the properties of tensor product notation to get:

$$\Delta U(Q_C) = \sum [pi(Z_B[1:i-1])U_{2^{i-1}}pi(Z_A[1:i-1])] \otimes [U_A C(1) U_B] \otimes [pi(Z_B[i+1:n])U_{2^{i-1}}pi(Z_A[i+1:n])]$$

Since qubit q_i state was restricted, the fragments Q_A,Q_B (and hence U_A,U_B) will only modify qubit q_i between the basis states $|0\rangle$,$|1\rangle$. If the qubit q_i started in the basis $|0\rangle$ state, then qubit q_i would be retained in $|0\rangle$ at the start of the circuit $i \in AncQ_c0$ (and vice versa for $|1\rangle$ state). Thus U_A,U_B either retain the $C(1)$ if qubit $q_i \in AncQ_C0$ or switch both the control to $C(0)$ and the qubit q_i to be restricted to $|1\rangle$ ($q_i \in AncQ_C1$). In either scenario, the above equation will follow Type-2 equality (Definition 2).

Lemma 2. *Any sequence of correctness-preserving transformations will be a correctness-preserving transformation.* If two correctness-preserving (satisfying Definition 1 or 2) circuit transformations modify circuit fragment $Q \rightarrow Q' \rightarrow Q''$, then the transitive relation $Q \rightarrow Q''$ also preserves correctness.

Proof Sketch. The unitary difference for the $Q \rightarrow Q''$ transformation is $\Delta U(Q'') = U(Q'') - U(Q)$. We add and subtract the $U(Q)$ term and get $\Delta U(Q'') = (U(Q'') - U(Q')) + (U(Q') - U(Q))$. Each term here follows Equation 1/2; thus, circuit transformations are transitive.

4.3 Atomic Rewrites: Swap and Merge (Type 1)

The movement rewrites swap adjacent gates, and the merge rewrite combines adjacent gates and, in special cases, deletes them. Quantum gates in the circuit may be shuffled by successively swapping pairs of gates using movement rewrites, which support three cases: (1) swapping commuting gates, (2) swapping gates that share controls, (3) a non-commuting X gate that modifies the control of the following gate. The merge rewrite applies to gates with shared controls operating on the same qubit.

▶ *At-SwapComm.* If two gates G1, G2 are commutative $U(G1)U(G2) = U(G2)U(G1)$, then they may be swapped without affecting the correctness of the circuit.

▶ *At-SwapCtrlGate (Figure 4a).* This rewrite operates on two sequential gates with shared controls (c) where the first gate G_1 applies unitary U to qubit q_j and the second gate G_2 is controlled by qubit q_j on an arbitrary quantum state s. To swap these gates, the first gate's unitary U is applied to qubit q_j before applying G_2, and then inverted ($U*$) before applying G_1. Because gate G_1 is always applied when G_2 is applied and G_2 does not affect G_1, this swap operation preserves correctness.

▶ *At-SwapX (Figure 4b).* This rewrite operates on an X gate that swaps the quantum states $|0\rangle,|1\rangle$ of qubit q_m followed by a controlled gate G with a control on state $s \in S$ on q_m. This rewrite swaps the X gate with the controlled gate and changes the control from s to s' (the complementary basis state of s).
At-MergeMatch (Figure 4c) This rewrite operates on two successive gates that have the same control qubits c and apply unitaries U,U' to the same qubit q_j. These gates are merged into a single gate applying $U'U$ with the original controls. In case where $U' = U^{-1}$, the merged gate applies the identity matrix $U'U = U^{-1}U = I(2)$, therefore deleting both gates. When At-MergeMatch rewrite is applied to a compute gate followed by an uncompute gate, both are deleted since they together apply the identity matrix.
Proof Sketch: These rewrites have been previously proven as preserving Type 1 (exact) equality $\Delta U(Q) = 0$ (Defintion 1) in previous literature [42].

4.4 Atomic Rewrite: Expansion to Replicate Pairs (Type 1)

The expansion rewrite (At-ExpandReplicatePair, Figure 4d) replaces a gate G with no control on qubit q_i with two *replicate* gates $RG1$ and $RG2$ that are now controlled on q_i. Two gates are replicates if they apply the same unitary U to the same qubit q_k and have controls on the same set of qubits, where the control states differ on at least one qubit (q_i for $RG1,RG2$). Because replicates have at least one mismatched control state, they are applied mutually exclusively and may be swapped.
Proof Sketch: We prove that this rewrite preserves Type-1 (Definition 1) equality. Consider the circuit fragment $Q = [G]$, where G is not controlled on qubit q_i. The unitary for the fragment Q is:

$$U(Q) = \pi(Z_G[1:i-1]) \otimes I(2) \otimes \pi(Z_G[i+1:n]) + I(2^n)$$

We then use the fact that identity $I(2) = (C(0)+C(1))$, the sum of the control unitaries $C(0)$, $C(1)$:

$$U(Q) = \pi(Z_G[1:i-1]) \otimes [C(0)+C(1)] \otimes \pi(Z_G[i+1:n]) + I(2^n)$$

We distribute $C(0)+C(1)$ term and complete the square to rewrite $U(Q)$ as a composition of two gates:

$$[\pi(Z_G[1:i-1] \otimes C(0) \otimes \pi(Z_G[i+1:n]) + I(2^n)] \times [\pi(Z_G[1:i-1]) \\ \otimes (C(1)) \otimes \pi(Z_G[i+1:n]) + I(2^n)] = U(RG1) \times U(RG2) = Q'$$

The rewritten unitary describes the two-gate circuit $[RG1,RG2]$. $RG1,RG2$ are replicates with control on state $|0\rangle,|1\rangle$ for qubit q_i respectively. Thus, the unitary of $Q = [G]$ is equivalent to the unitary of $Q' = [RG1,RG2]$ and $\Delta U(Q) = 0$. This matches equation 1, thereby proving Type-1 equality.

4.5 Atomic Rewrite: Single Mismatch Replicate Merge Rewrite (Type 1)

The replicate merge rewrite (At-MergeReplic-1MM, Figure 4e) merges a pair of replicates $RG1,RG2$ that have only one mismatched control on say qubit q_i. It deletes the control on qubit q_i for $RG1$ and deletes $RG2$, reducing both the number of gates and the number of controls on the remaining gate.
Proof Sketch: This rewrite preserves Type-1 equality (Definition 1). Note that this rewrite is the exact reverse of the At-ExpandReplicatePair rewrite that we have previously shown preserves Type-1 equality (section 4.4). Type-1 equality implies that $\Delta U(Q) = U(Q') - U(Q) = 0$. This is a symmetric property and therefore At-MergeReplic-1MM will also preserve Type-1 equality.

4.6 Atomic Rewrite: Dead Gate Deletion: At-DeadGateDel (Type 2)

The dead gate deletion rewrite (At-DeadGateDel, Figure 4f) deletes controlled gates that will never be applied on the circuit, based on the quantum state annotations. Any gate controlled on state $|1\rangle$ for qubit q_i that is in basis state $|0\rangle$ can be deleted since its control condition on qubit q_i is never satisfied. The same principle holds for a gate controlled on $|0\rangle$ of some qubit q_i that is in the basis $|1\rangle$ state.
Proof Sketch: This rewrite preserves Type-2 equality (Definition 2). Consider a quantum circuit fragment Q with a controlled gate G that applies the unitary $\pi(Z_G[1:i-1]) \otimes C(1) \otimes \pi(Z_G[i+1:n]) + I(2^n)$

with the control $C(1)$ over qubit q_i which is in the basis state $|0\rangle$, where $q_i \in AncQ0$. After deleting gate G, the rewritten quantum circuit fragment Q' implements an identity $I(2^n)$. The unitary difference $\Delta Q = Q' - Q$ between the original and rewritten circuits is as follows:

$$\Delta Q = I(2^n) - (\pi(Z_G[1:i-1]) \otimes C(1) \otimes \pi(Z_G[i+1:n]) + I(2^n)) = \pi(Z_G[1:i-1]) \otimes C(1) \otimes \pi(Z_G[i+1:n])$$

The ΔQ expression matches equation 2, therefore proving Type-2 equality. The same strategy may be repeated for the case where $q_i \in AncQ1$ and the control on qubit q_i is over state zero ($C(0)$).

4.7 Atomic Rewrite: Trivial Control Deletion: At-TrivCtrlDel (Type 2)

The trivial control deletion rewrite (At-TrivCtrlDel, Figure 4g) deletes controls from gates that are always satisfied by the qubit they operate on. For a gate G with a control $C(0)$ controlled on state $|0\rangle$ of qubit q_i, where qubit q_i is an ancilla in the basis $|0\rangle$ state, the control may be safely deleted. The same holds for controls on state $|1\rangle$ of a qubit q_i in the basis $|1\rangle$ state.
Proof Sketch: This rewrite preserves Type-2 equality (Definition 2). Consider a quantum circuit fragment Q with a controlled gate G control on state $C(0)$ for qubit $q_i \in AncQ0$. The gate applies the unitary $\pi(Z_G[1:i-1]) \otimes C(0) \otimes \pi(Z_G[i+1:n]) + I(2^n)$. After deleting control on q_i, the rewritten circuit fragment Q' implements the unitary $\pi(Z_G[1:l-1]) \otimes I(2) \otimes \pi(Z_G[l+1:n]) + I(2^n)$. The difference between the rewritten and original circuit fragments $\Delta U(Q) = U(Q') - U(Q)$ is therefore:

$$\Delta U(Q) = \pi(Z_G[1:i-1]) \otimes C(1) \otimes \pi(Z_G[i+1:n])$$

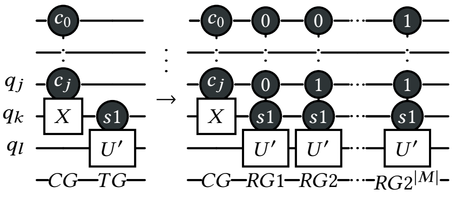
The $\Delta U(Q)$ expression matches equation 2, proving Type-2 equality. The same strategy is repeated for the case where q_i is in basis state $|1\rangle$ ($q_i \in AncQ1$) and the control on qubit q_i is over state zero ($C(0)$).

4.8 Compute-Transform-Uncompute Rewrites

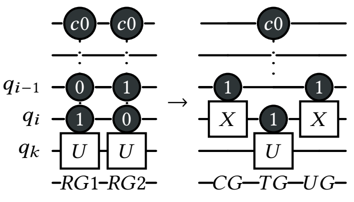
SPARE's optimizer, presented in Section 5.2, works with more sophisticated rewrites shown in Figure 5 that are comprised of the atomic rewrites from Sections 4.3-4.7. These rewrites operate over a compute-transform-uncompute (CTU) circuit structure that contains compute gates $CG_1 ... CG_k$, followed by transform gates $TG_1 ... TG_l$, and uncompute gates $UG_1 ... UG_k$.

4.8.1 Rewrite: Transform Gate Expansion Rewrite (Rw-ExpandTG). Given an CTU structure, the Rw-ExpandTG rewrite operates on transform gate TG preceded by a compute gate CG, where CG is controlled on qubits in set M but TG is not. This rewrite recursively applies the At-ExpandReplicatePair atomic rewrite on TG for each qubit in set M, resulting in a sequence of 2^M replicate gates $[RG_1 ... RG_{2^M}]$ as shown in Figure 5a. These replicate gates share common control qubits with the compute-uncompute gates, enabling SPARE to use movement rewrites (section 4.3) on compute-uncompute gates to find optimization opportunities. For circuit fragments with nested CTU patterns, the Rw-ExpandTG rewrite is performed over the qubits of all compute gates in the fragment.

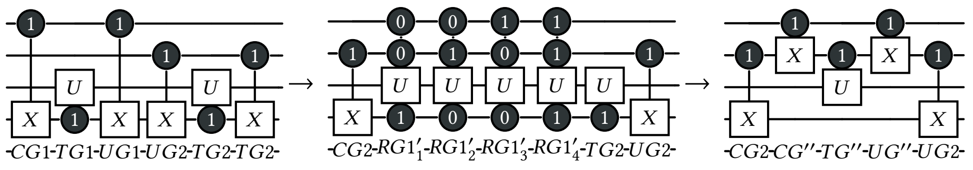
4.8.2 Rewrite: Merge Replicates with Multiple Mismatches (Rw-MergeReplic-MultMM). The Rw-MergeReplic-MultMM rewrite merges two replicate gates $RG1, RG2$ with multiple control mismatches into a CTU circuit fragment. Given $RG1$,$RG2$ with mismatched controls on qubits in set $M = \{q_{m,0}, q_{m,1} ... q_{m,|M|-1}\}$, the rewrite first resolves the mismatch for one control qubit $q_{m,i} \in M$. To resolve the mismatch, two controlled CNOT gates (CX) acting on qubit $q_{m,i}$ with controls on qubit $q_{m,i-1} \in M$ are inserted, transforming Q to $Q \rightarrow Q_1 = [RG1, RG2, CX, CX]$. The movement rewrites from Section 4.3 are then used to move the left CX gate before $RG1$,$RG2$, resulting in $Q_1 \rightarrow Q_2 = [CX, RG_1', RG_2', CX]$. When applying these movement rewrites, either CX commutes (At-SwapComm) because it is mismatched on control $q_{m,i-1}$, or it has the same controls as the swapped replicate in which case theAt-SwapX and At-SwapCtrlGate rewrites are used, flipping the control on qubit $q_{m,i}$. The net effect of these movement rewrites is that the control mismatch on $q_{m,i}$ is resolved. This rewritten circuit fragment has a CTU gate pattern, where the CX gates form the compute-uncompute gates.



(a) Rw-ExpandTG



(b) Rw-MergeReplic-MultMM


1
0
U
X
$-CG1-TG1-UG1-UG2-TG2-TG2-$
$\rightarrow$
$-CG2-RG1'_1-RG1'_2-RG1'_3-RG1'_4-TG2-UG2-$
$-CG2-CG''-TG''-UG''-UG2-$

(c) Rw-MergeCTUSeq

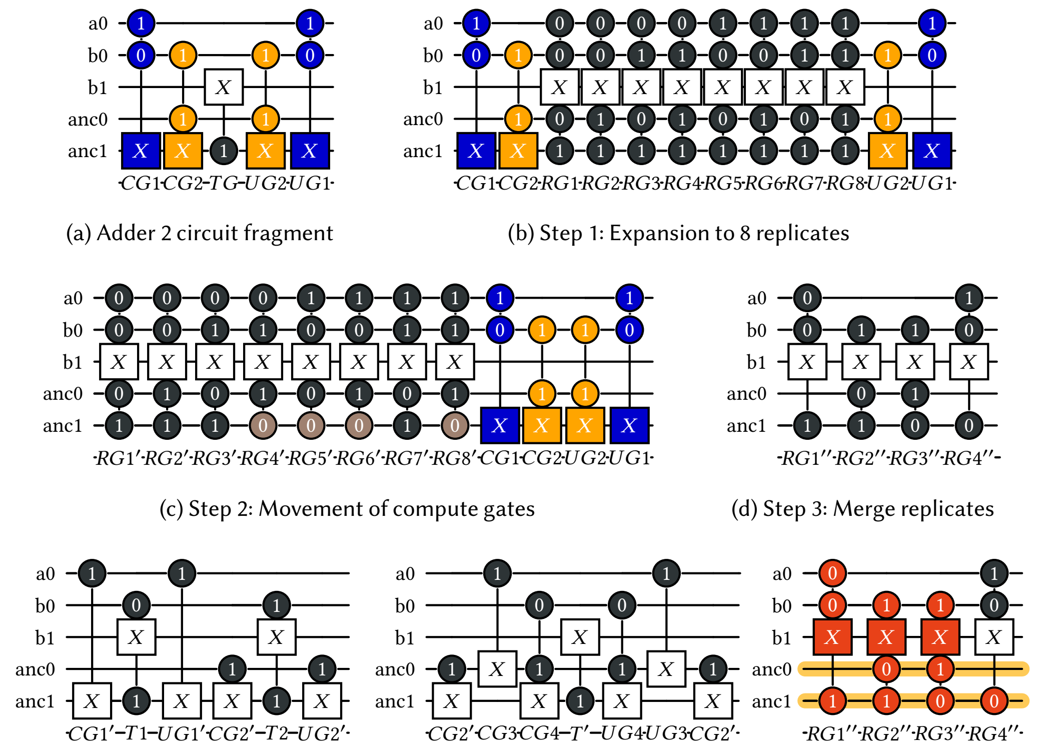
Fig. 5. **Compute-Transform-Uncompute (CTU) rewrites. |M|: # of CG control qubits with no TG control.**

This procedure is repeated, resolving mismatches on qubits $q_{m,i-1}...q_{m,1}$, adding more CX gates in the process until only one mismatch remains. Now the rewrite applies At-MergeReplic-1MM to produce the transform gate in the middle ($RG1'$,$RG2' \rightarrow TG$) of this CTU. An example is shown in Figure 5b.

Algorithm 1 Consecutive CTU Circuit Fragment Fusion (Rw-MergeCTUSeq) Algorithm

```
Input: Circuit fragment $Q = [CG1,TG1,UG1,CG2,TG2,UG2]$ where transform gates $T1,T2$ have the same target qubit and
    unitary and number of controls.
Output: Merged CTU fragment with a new transform gate $TQ$.
1: $Q \rightarrow Q_1 = [CG1,RG1_1...RG1_N,UG1,CG2,TG2,UG2]$ with Rw-ExpandTG
2: $Q_1 \rightarrow Q_2 = [RG1_1...RG1_N,CG1,UG1,CG2,TG2,UG2]$ with At-SwapComm, At-SwapCtrlGate
3: $Q_2 \rightarrow Q_3 = [RG1_1...RG1_N,CG2,TG2,UG2]$ with At-MergeMatch
4: $Q_3 \rightarrow Q_4 = [CG2,RG1'_1..RG1'_N,TG2,UG2]$ with At-SwapComm, At-SwapCtrlGate
5: $Q_4 \rightarrow Q_5 = [CG2,RG1'_1..RG1'_N,RG2_1..RG2_N,UG2]$ with Rw-ExpandTG
6: $Q_5 \rightarrow Q_6 = [CG2,TG'_1,TG'_2,..TG_L,UG2]$ with At-MergeMatch, Rw-MergeReplic-MultMM
7: $CQ,TQ,UQ$ = findCTUPattern($Q_6$) return $CQ,TQ,UQ$
```

4.8.3 Rewrite: Fusion of Consecutive CTU Circuit Fragments (Rw-MergeCTUSeq). Rw-MergeCTUSeq merges consecutive CTUs in a fragment $Q = [CG1,TG1,UG1,CG2,TG2,UG2]$ into a single CTU fragment using Algorithm 1 , provided the transform gates $TG1$, $TG2$ both apply the same unitary U to qubit k and have the same number of controls. The control count condition is optional, but constrains the rewrite to finding good patterns. The algorithm expands $T1$ into N replicates and swaps the first compute gate $CG1$ with each of the replicates, moving $CG1$ next to $UG1$ (lines 1-2). The $CG1$ and $UG1$ gates are merged together and eliminated, then the remaining replicates are moved into the second CTU fragment $[CG2,TG2,UG2]$. The second transform gate $TG2$ is then expanded into N replicates, and then the merge rewrites are used to coalesce the replicates back into a smaller number of transform gates (line 6). Finally, the simplified quantum circuit is broken down into compute-transform-uncompute circuit fragments, where the merged transform gates may become part of the compute/uncompute fragments. An example is shown in Figure 5c.


a0
b0
b1
anc0
anc1
CG1 CG2 TG UG2 UG1
CG1 CG2 RG1 RG2 RG3 RG4 RG5 RG6 RG7 RG8 UG2 UG1
RG1′ RG2′ RG3′ RG4′ RG5′ RG6′ RG7′ RG8′ CG1 CG2 UG2 UG1
RG1″ RG2″ RG3″ RG4″
CG1′ T1 UG1′ CG2′ T2 UG2′
CG2′ CG3 CG4 T′ UG4 UG3 CG2′
RG1″ RG2″ RG3″ RG4″

(a) Adder 2 circuit fragment (b) Step 1: Expansion to 8 replicates

(c) Step 2: Movement of compute gates (d) Step 3: Merge replicates

(e) Step 4: Merge mismatched gates. (f) Step 5: Merge sequential CTU pairs (g) Gate/control deletion

Fig. 6. Applying SPARE rewrites on CTU circuit fragment from Adder 2 circuit computing $b \leftarrow a+b$ with ancilla qubits. Modified controls highlighted 🟫. Qubits in basis state $|0\rangle$ marked 🟨. Deleted gates and controls marked 🟥

4.9 Illustrative Example: Quantum Adder

Many quantum circuits, particularly those produced from high-level languages [6, 54], have compute-transform-uncompute patterns that can be optimized using rewrites. To demonstrate, we optimize a CTU fragment in Figure 6a, pulled from the 2-bit adder circuit generated by Unqomp in Figure 1b. *Steps 1-3.* This circuit contains two compute gates ($CG1$, $CG2$), two uncompute gates ($UG1$, $UG2$), and a transform gate (TG) between them. The initial goal is to cancel out the compute/uncompute gates $CG1$,$UG1$ and $CG2$,$UG2$, as shown in Figure 6b-6d. In Figure 6b, the Rw-ExpandTG rewrite is applied to expand TG into eight replicates [$RG1$,$RG2$...$RG8$] over all compute gate controls. In Figure 6c, the movement rewrites from Section 4.3 are used to move $CG1$ next to $UG1$, and $CG2$ next to $UG2$. In Figure 6d, the matching compute/uncompute gates are merged together with At-MergeMatch, and the replicates are merged back together with At-MergeReplic-1MM. In this merging operation, replicates $RG1'$,$RG2' \rightarrow RG1''$, and $RG3'$,$RG7' \rightarrow RG2''$, and $RG4'$,$RG8' \rightarrow RG3''$, and $RG5'$,$RG6' \rightarrow RG4''$.
Steps 4-5. Now, we are left with gates that have no obvious CTU gate pattern. To re-introduce this pattern, we use the Rw-MergeReplic-MultMM rewrite to merge replicates with mismatched controls, transforming $RG1''$,$RG4''-> (CG1',T1,UG1')$ and $RG2''$,$RG3''-> (CG2',T2,UG2')$ as shown in Figure 6e. It results in a sequence of two CTU gate patterns [$CG1'$, $T1$, $UG1$, $CG2'$,$T2$, $UG2$]. The Rw-MergeReplic-MultMM rewrite is used to merge the two sequential CTU gate patterns, yielding a single CTU gate pattern [$CG2'$, $CG3$, $CG4$, T', $UG4$, $UG3$, $UG2'$], as shown in Figure 6f.

]

Gate Deletion Rewrites. The At-TrivCtrlDel, At-DeadGateDel rewrites may also be used to delete gates and controls from the circuit using ancilla information. For example, three gates and one control from the circuit in Figure 6d can be deleted, yielding the circuit from Figure 6g.
Optimized Circuit Complexity. Comparing lowered circuit complexities, the original circuit fragment (Figure 6b) contains 4 Toffoli compute/uncompute gates and 1 CNOT gate, totaling 25 CNOTs (as 1 Toffoli requires 6 CNOTs). The optimized circuit in Figure 6e contains only 2 Toffoli gates and 5 CNOT gates, totaling 17 CNOTs. Even with optimized lowering of Toffoli gates to Margolus gates (where 1 Margolus requires 3 CNOTs, see Section 5.4). The original circuit requires $4*3+1=13$ CNOTs, and the circuit in Figure 6f requires 2 Margolus gates and 5 CNOTs, totaling $2*3+5=11$ CNOTS. Therefore, SPARE rewrites can deliver gate count benefits for both lowering schemes. The circuit optimized with ancilla state information in Figure 6g only has one Toffoli gate is implementable with 6 CNOTs .

5 SPARE Compiler

We next introduce the SPARE intermediate representation (IR) and rewrite algorithm and describe SPARE's qubit ancilla elimination and lowering pass optimizations. We also discuss the generality of SPARE rewrites and identify a subset of rewrites that can be re-cast as higher program-level rewrites.

5.1 SPARE Intermediate Representation (IR)

SPARE works with a quantum circuit IR comprised of multi-controlled single-target gates. Each gate control is annotated with the quantum state it is controlled on ($|0\rangle$ or $|1\rangle$), and each wire segment is annotated with the quantum states of the qubit at that point in time. Each wire segment in the IR is annotated with **0** if the qubit is in the $|0\rangle$ basis state, **1** if the qubit is in the $|1\rangle$ basis state, or **0/1** if the wire carries a superposition of both states. The control annotations are inferred from a partially annotated quantum circuit generated by SPARE's Spire/Unqomp frontends.
Ancilla Inference from Spire/Unqomp. The SPARE compiler front-end translates Unqomp/Spire-generated circuits to the SPARE IR and annotates ancilla qubits. For Unqomp-generated circuits, SPARE uses the circuit's type information to identify ancilla qubits. For Spire-generated circuits, SPARE tracks local variables through the Spire compiler and annotates them in the generated circuit.
Static Analysis - State Propagation. SPARE annotates the ancilla qubit wire segments at the input interface of the circuit, the **0** state, and then uses a two-pass static analysis to derive the internal wire annotations for each segment. In the first pass, SPARE traverses the circuit left to right, symbolically applying gates to check if a qubit's state changes from a basis **0** or **1** state, to a superposition **0/1** state and annotates wires accordingly. The ancilla are set back to a basis state after all the uncomputation gates are similarly applied. SPARE then propagates ancilla state information from right to left, starting from the circuit's output interface, using the same algorithm. A similar pass is applied to track and store any CTU circuit fragments in the circuit as well. After analysis, all internal wire segments in the circuit are labelled with the quantum state of the qubit at that time. Unlike previous wire annotation-based IRs (like presented in [42]), SPARE also supports "output qubits", these qubits start in a basis state on the input interface but are used to store the circuit result on the output and thus are in a superposition.

5.2 SPARE Rewrite Algorithm

SPARE finds CTU gate patterns in the circuit and iteratively optimizes them. It first searches for all possible CTU patterns and samples a target CTU circuit fragment (*trgtQ*). This sampling selects CTU circuit fragments that have a higher number of gates. SPARE then applies the *Basic Rewrite Algorithm* (Algorithm 2) to optimize this target fragment. The algorithm first estimates the cost of the circuit fragment *trgtQ* using a heuristic cost model of circuit complexity (line 2). Second, it uses Rw-ExpandTG to expand the middle transform gates of the *trgtQ* fragment (line 4) and uses the swap rewrites to move the compute gate *CG* next to the uncompute gate *UG*. Third, SPARE

Algorithm 2 Basic Rewrite Algorithm

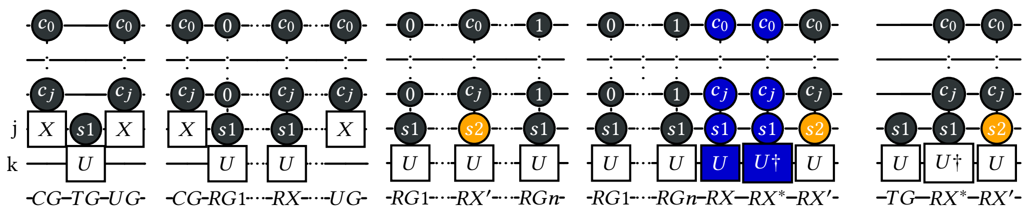
```
 1: function OptimizeCircuit(trgtQ=[CG, TG, UG])      ▷ target CTU circuit fragment from the circuit
 2:     origComplexity, origQ = trgtQ.getCmplxHeuristic(), trgtQ.copy()
 3:     $trgtQ \rightarrow trgtQ_1 = [CG,RG1..RGn,UG]$ with Rw-ExpandTG on TG.
 4:     $trgtQ_1 \rightarrow trgtQ_2 = [RG1'..RGn',CG,UG]$ with At-SwapComm, At-SwapCtrlGate, At-SwapX on $CG$
 5:     $trgtQ_2 \rightarrow trgtQ_3 = [RG1'..RGn']$ with At-MergeMatch on CG, UG.
 6:     $trgtQ_3 \rightarrow currQ$ with At-TrivCtrlDel, At-TrivCtrlDel on $[RG1'..RGn']$.
 7:     while True do
 8:         prevComplexity = GetCmplxHeuristic()
 9:         $currQ \rightarrow currQ_4$ with At-MergeMatch applied exaustively
10:         $currQ_1 \rightarrow currQ_2$ with At-MergeReplic-1MM applied exhaustively
11:         $currQ_2 \rightarrow currQ_3$ with Rw-MergeReplic-MultMM applied exhaustively
12:         currQ, currComplexity = $currQ_4$, getComplexityHeuristic()
13:         if currComplexity >= prevComplexity then break
14:     $currQ \rightarrow finQ$ with Rw-MergeCTUSeq applied speculatively
15:     return if getComplexityHeuristic() < origComplexity then $finQ$ else origQ
```

applies At-MergeMatch and uses the ancilla state information to prune replicate gates (line 5-6). Fourth, it applies At-MergeReplic-1MM, Rw-MergeReplic-MultMM rewrites exclusively, one after another (line 7-12). Finally, after completing these optimizations, SPARE speculatively applies the Rw-MergeCTUSeq rewrite and returns the modified fragment if circuit complexity has been reduced (line 15-16). The optimizer then repeats the process, identifying a new CTU fragment. SPARE applies this entire process 5 times with different random seeds, and returns the circuit with the lowest complexity.

5.2.1 Circuit Complexity-Improving Optimizations. SPARE uses several optimizations to further reduce the complexity of the optimized circuits and better navigate the search space.
Speculative Execution. In practice, the basic flow misses optimization targets because the compute-uncompute gate elimination (lines 4-6) and replicate simplification (lines 7-12) steps sometimes eliminate optimization targets or increase circuit complexity. SPARE therefore applies algorithm 2 *speculatively* (line 15) and rolls back the rewrites if the estimated circuit complexity increases.
CTU target caching: SPARE samples target CTU circuit fragments and speculatively applies rewrites. To avoid redundant exploration, it caches previously examined and rejected CTU targets. Thus, during optimization, SPARE focuses solely on new targets, ensuring efficient and diverse exploration.
Small Expansion Optimization. Because the Rw-ExpandTG rewrite in line 4 of algorithm 2 drastically increases the circuit complexity, we limit it to targets that produce less than 2^6 replicates for each transform gate. This can be easily estimated without applying the rewrite; the number of replicates is 2^k where k is the number of qubits that compute gate CG is controlled on but transform gate TG is not.

5.2.2 Performance Optimization for CTU Fragments with Many Qubits. Applying the Rw-ExpandTG rewrite independently of the replicate optimizations (lines 6-12) results in many intermediate replicate gates, slowing down the rewrite engine. Therefore, to efficiently optimize large CTU circuit fragments, we combine expansion, movement, merge, and elimination rewrites (lines 4-14) into a single optimization step.

Figure 7 shows the improved optimization flow, which directly generates the optimized circuit. First, figure 7a presents the circuit after the Rw-ExpandTG rewrite is applied. We observe that only one replicate produced has identical controls as the compute gate CG. We call this replicate RX (Figure 7b). Moving CG to UG will thus apply At-SwapCtrlGate and At-SwapX on only RX, changing it to RX' and will apply At-SwapComm on all other replicates (Figure 7c). We introduce the modified replicate and its inverse RX,RX^* into the circuit (■ in Figure 7d). With this modification, the replicates $RG1-RGn$ and RX can now be merged back to TG. Therefore, we can directly transform the CTU circuit



(a) CTU Fragment Example (b) Expansion of TG to replicates (c) Moving CG to UG only modifies RX (🟧) (d) Add original and inverted replicate RX',RX^* (e) Merge $RG1-RGn$, RX to TG

Fig. 7. Optimization flow for large CTU targets: directly delete CG,UG gates and for each TG gate add an inverted replicate gate (RX^*) and a replicate with control modified (RX') on CG's target qubit (🟧).

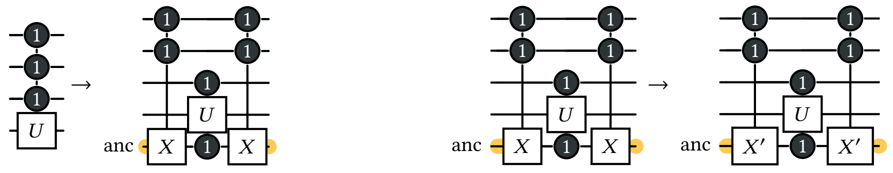
fragment to produce the circuit in Figure 7e, in which the compute-uncompute gate pair is deleted and two replicate gates RX', RX^* gates are introduced after the transform gate TG. The RX' gate is derived by swapping the control on Cg's target for the replicate RX, and the RX^* gate applies the inverse of RX.

5.2.3 Performance Optimization for CTU Fragments with only CX Gates (Rw-DelCX-CG-UG). For CTU circuit fragments that consist only of CNOT compute/uncompute gates that all act on an ancilla qubit q_m and are controlled by the same data qubit q_k, SPARE applies an optimized version of Algorithm 2 which merges lines 3-7 into a single transformation for efficiency. This is achieved by eliminating the compute/uncompute CNOTs and rewriting all transform gates in the fragment, replacing controls on qubit q_m with q_k with the same control state. If a transform gate is already controlled on q_k, SPARE verifies that the original control states of q_k and q_m match; otherwise, it removes the gate. Since CNOTs acting on data and ancilla qubits q_k and q_m only serve to copy the state of q_k onto q_m, any operation on q_m can be directly applied on q_k.

5.3 Qubit Elimination Optimization

SPARE rewrites reduce the number of gates applied to ancilla qubits, often resulting in ancilla qubits that can be removed. These qubits are deleted from the circuit itself after all the SPARE optimizations are complete. SPARE targets CTU circuit fragments that may use ancilla qubits, where Compute/Uncompute gates act on an ancilla qubit while transform gates are controlled on it. SPARE optimizations delete the Compute/Uncompute gates and use the ancilla state information to delete gates and controls from the transform gate replicates (Sections 4.6, 4.7). As a result, SPARE reduces the number of gates operating on any ancilla, and if all CTU circuit fragments that use an ancilla are similarly optimized, the ancilla can be marked for deletion from the circuit.

Verifying Circuit Correctness with Qubit optimizations. We retain the otherwise dead ancilla qubits to ensure easy verification of rewrites. Transformations that optimize away qubits change a fragment Q acting on qubits $q_1..q_{k-1},q_k,q_{k+1}..q_n$ to a fragment Q' acting on qubits $q_1..q_{k-1},q_{k+1}..q_n$ (q_k is deleted). Thus, $U(Q)$ is of the form $U_{2^k-1} \otimes U_2 \otimes U_{2^{n-k}}$ whereas, $U(Q')$ is of the form $U'_{2^k-1} \otimes U'_{2^{n-k}}$. Because the corresponding unitaries have different sizes after qubit elimination, verifying Type-1/Type-2 equality is less straightforward. Therefore, to simplify analysis, we retain the qubit q_k in fragment Q' and apply a no-op $I(2)$ gate on it. With this change, the unitary of the transformed fragment would be of the form $U'_{2^{k-1}} \otimes I(2) \otimes U'_{2^{n-k}}$. We can now calculate $\Delta U(Q) = U(Q) - U(Q') = U_{2^{k-1}} \otimes U_2 \otimes U_{2^{n-k}} - U'_{2^{k-1}} \otimes I(2) \otimes U'_{2^{n-k}}$ and check if it satisfies Type-1 or Type-2 equality (Equation 1/ 2). The qubit q_k that only contains $I(2)$ gate is deleted from the final circuit after all SPARE optimizations are complete.



(a) Implement n-controlled U gates using 2-controlled U, Toffoli gates and ancillas.

(b) Replace compute/uncompute Toffoli gates with Margolus gates (shown as X'), which require 3 CNOT and 6 1-qubit gates.

Fig. 8. Lowering pass optimizations applied by SPARE. Wire with qubit in basis state $|0\rangle$ marked ■.

5.4 SPARE Lowering Pass

SPARE lowers Toffoli gates into a Clifford+T gateset using a popular implementation [8, 21, 42] that requires six CNOT and eleven one-qubit gates and has a depth 11 [12, 43]. SPARE also adapts two common lowering optimizations on the optimized logical-level circuits shown in Figure 8.

For n-controlled gates where $n > 2$, SPARE adapts an ancilla-based lowering from [37, 55] that yields an $O(\log(n))$ depth circuit using $n-2$ extra qubits. These qubits store control results using extra Toffoli gates and can be reused once reset to $|0\rangle$ state. Figure 8a shows an example for $n=3$.

SPARE also adapts a lowering optimization from previous compiler [6, 38] and expert circuit implementation [4] work that replaces Toffoli gates in compute/uncompute gates of the circuit with Margolus (RCCX) gates [20] as shown in Figure 8b. Margolus gates are equivalent to a Toffoli gate up to a phase. They transform the state $|111\rangle \rightarrow -|110\rangle$ instead of $|110\rangle$ but only require three CNOTs and six single-qubit gates, compared to Toffoli gates that require six CNOTs and nine single-qubit gates. Applying this substitution on compute/uncompute gates preserves circuit semantics [38, 50].

5.5 Discussion: Generality of SPARE Rewrites

SPARE optimizations target CTU gate patterns along with ancilla qubits. These patterns frequently appear on mapping classical operations like quantum arithmetic or controlled operations, which require computing intermediate results that must be explicitly uncomputed [37]. CTU gate patterns are seen important quantum algorithms like phase estimation [25], quantum chemistry simulations [23], the quantum Fourier transform [44], implemented quantum memory with QROMs [3], and Shor's algorithm [31], where modular exponentiation relies on CTU structures. They also arise in translating classical computations into reversible quantum circuits with frameworks like Quipper [15]. SPARE optimization approach optimizes CTU circuit fragments with circuit-level rewrites that use ancilla state information and synthesize new efficient CTU circuit fragments. SPARE does not incorporate higher-level rewrites, such as control simplifications [47]. Instead, SPARE delegates such transformations to its frontends. For example, Spire [55] applies conditional narrowing (`cn`), which is similar to control simplification. Additionally, SPARE currently does not support measurement-based computation.

5.6 Discussion: Interpreting SPARE Rewrites at the Language-Level

Some of SPARE's gate-level optimizations can be lifted to the high-level language for control flow-based languages like Tower. We describe how some rewrites that work with CTU gate patterns correspond to transforming the `with..do..undo` pattern seen in the control flow representation of the program. We also describe the limitations of applying such rewrites to only the high-level program.

5.6.1 Rewrites on `With..Do..Undo` *Block and CTU Gatesets.* In languages like Tower, CTU patterns are only visible in the high-level language in special constructs like `with..do..undo`. Figure 9a shows

```
let qk <- 0
with # compute
    if (q0=1 && q1=1 &&
      ... qj=1)
        qk <- $\overline{qk}$
do # transform
    if (qk = s1)
        ql <- U′ state(ql)
Undo with # uncompute
    if (q0=1 && q1=1 &&
      ... qj=1)
        qk -> $\overline{qk}$
```

(a) **with-do-undo** block

```
...
do # transform
    #replicate 1
    if (q0=0 && q1=0 ...qj=0 && qk=1)
        ql <- U′ state(ql)
    #replicate 2
    if (q0=0 && q1=0 ...qj=1 && qk=1)
        ql <- U′ state(ql)
    ...
    # replicate n
    if (q0=1 && q1=1 ...qj=1 && qk=1)
        ql <- U′ state(ql)
...
```

(b) Do-block unrolling (Rw-ExpandTG)

```
let qk <- 0
if (q0=0 && q1=1 && ... qj=0 && qk=1)
    ql <- U′ state(ql)
...
if ((q0 && .. & qj) && qk=0)
    ql <- U′ state(ql)
with #computation
    if (q0=1 && q1=1 && .. && qj=1)
        qk ← $\overline{qk}$
Undo with # uncomputation
    if (q0=1 && q1=1 && .. && qj=1)
        qk → $\overline{qk}$
```

(c) Moving **with** block (At-SwapComm, At-SwapCtrlGate) and dead code elimination (At-DeadGateDel).

Fig. 9. **With-do-undo** block and high-level transformations that are similar to equivalent to SPARE CTU rewrites/transforms (in bracket). ■: changed clauses, ■: deleted code.

```
if (q0 && q1 && .. qm && ..)
    let qk  <- 1
if (q0 && q1 && .. $\overline{qm}$ && ..)
    let qk  <- 1
```

(a) Conditionals with 1 mismatch

```
if (q0 && q1 && .. && ~~qm~~ && ..)
    let qk  <- 1
```

(b) Merging 1 mismatch (At-MergeReplic-1MM)

```
#controlled gate 1
if (q0 && q1 && .. && $\overline{qj}$ && qk)
    ql <- U state(ql)
#controlled gate 2
if (q0 && q1 && .. && qj && $\overline{qk}$)
    ql <- U state(ql)
```

(c) Conditionals with 2+ mismatches.

```
if (qj) # compute
    qk <- $\overline{qk}$
if (q0 && q1 && ..&&
    ..&& qk) # transform
    ql <- U state(ql)
if (qj) # uncompute
    qk <- $\overline{qk}$
```

(d) Merging with 2+ mismatches (Rw-MergeReplic-MultMM)

Fig. 10. Rewrites on unrolled **Do** block statements with equivalent gate level rewrites (in bracket).

an example **with...do...undo** block that we will optimize with SPARE rewrites. In the high-level program, the **with** block performs computation, the **do** block uses the computation to transform data variables, and the **undo** block undoes the computation in **with**.

▶ *Rw-ExpandTG rewrite as Do Block Unrolling.* Rw-ExpandTG rewrite flattens the **do** block into a sequence of conditionals that enumerate all combinations of qubits in the **with**/**undo** predicates. Figure 9b shows the **with..do..undo** block after performing the gate expansion rewrite.

▶ *At-MergeMatch rewrite and With/Undo Deletion.* The movement and merge rewrites from Section 4.3 move and optimize away compute-uncompute gates. This gate level rewrite is similar to optimizing **with..do..undo** blocks as shown in Figure 9b. Here, the **with** block is moved down through the statements in the **do** block until it is next to the **undo** block. Now both the **with** and **undo** blocks can be deleted, leaving just the transformed **do** statements, as shown in Figure 9c.

▶ *Gate Deletion as Dead Code Elimination.* At-DeadGateDel operation eliminates gates with controls that can never be satisfied. This is similar to constant propagation and dead code elimination. For example, in Figure 9c the variable qk is set to state $|0\rangle$. Thus, the control statements in lines 2-4 will never be satisfied and can be deleted. Dead code elimination eliminates gates using constant propagation; SPARE achieves a similar task propagating wire annotations.

▶ *At-MergeReplic-1MM as Merging Conditionals.* At-MergeReplic-1MM merges two control gates with one mismatched control condition. A natural site for a controlled gate to be visible in a high-level programming language is in a conditional statement. Thus, a pair of conditional statements with one mismatched control condition can be merged. Figure 10a presents an example, and Figure 10b presents

the same program after merging conditional statements. This transformation is equivalent to applying At-MergeReplic-1MM when the target qubit (**qk**) is an ancilla (due to the **if** statement body).
▶ *Rw-MergeReplic-MultMM as Merging Conditionals with multiple mismatches.* Rw-MergeReplic-MultMM translates controlled gates with multiple mismatches together into a CTU fragment. In special cases, such an optimization can be performed in the higher-level program as shown in Figure 10c to obtain the program in Figure 10d. The resulting program has a set of conditional statements that form an CTU pattern. But this type of CTU pattern is not natural to the high-level programming language. In Tower [54] for example, a programmer can only express CTU structures using the **with..do..undo** block, which in turn instantiates new variables and ancillas. On the other hand, the Rw-MergeReplic-MultMM rewrite adds the compute-uncompute gates "in place" and does not require extra ancilla qubits.

5.6.2 Limitations of High-Level Rewrites. The described high-level program rewrites are more limited than the gate-level rewrites because not all ancilla qubits or CTU structures are visible in the high-level program. Upon compilation of a program, various new temporary variables are often instantiated that do not exist at higher levels of abstractions. Therefore, the generated circuit will have far more CTU circuit fragments and ancilla qubits the is visible in the high-level program.

SPARE also uses rewrites that are not easily accessible in high-level languages. For example, Rw-ExpandTG can be applied even when the compute/uncompute gates of an CTU circuit fragment do not target ancilla qubits. This is unlike applying Block Unrolling (Figure 9a) in the high-level code, where the only site with a CTU gate pattern is a **with..do..undo** block, where the **with** block instantiates new variables and thus requires the compute target to be an ancilla. Similarly, high-level transformations that correspond to At-MergeReplic-1MM and Rw-MergeReplic-MultMM are only applied in special CTU fragments. There is also no clear high-level equivalent transformation for Rw-MergeCTUSeq. Thus, only a subset of optimization opportunities are available at the high-level abstraction.

6 Results

We evaluate SPARE on 26 compute-uncompute circuits from Spire and Unqomp-generated benchmark sets against three different compilers/circuit optimizers. For all the Spire-generated and Unqomp-generated Arithmetic benchmarks, we lower circuits into a logical circuit implementation using a Clifford+T gate set, which consists of $\{H,X,Z,S,CNOT,T\}$ gates. This gateset is frequently used for fault-tolerant quantum computing [9, 13, 14, 49]. The Unqomp-generated quantum benchmarks were lowered to the Clifford+T+U gates used in Unqomp lowering [38]. Table 2 summarizes the number of qubits (**#qs**), 2-qubit gate count (**2q**), the 1-qubit gate count (**1q**), and the circuit depth (**depth**) for each benchmark, and provides a circuit description.
Spire-generated Benchmarks. Spire [55] compiles quantum programs described in Tower [54], a control-flow based language, to quantum circuits. We apply SPARE and the baseline optimizers to nine benchmark programs with recursion depth of 1, generated with all Spire optimizations enabled.
Unqomp-generated Benchmarks. The Unqomp compiler [38] optimizes compute/uncompute quantum circuits by automatically synthesizing uncompute gates, therefore optimizing CTU circuit patterns. It can be used to optimize lower-level descriptions of certain Silq programs [6] that do not use features like recursion or variable-length quantum registers. We apply SPARE and the baseline optimizers on 17 Unqomp-generated quantum circuits compiled from nine applications across 1-3 problem sizes, where five circuits are non-arithmetic. The baseline circuits are generated using the Unqomp compiler with all the circuit generation and lowering optimizations enabled. The optimized lowering pass replaces Toffoli gates in the compute/uncompute sequences with Margolus gates [20], which are easier to implement.

Table 2. Benchmark summary. Q=Quartz, F=Feynman, and ZX=PyZX. For benchmarks marked † we found and fixed a bug in the Unqomp implementation. ✓ indicates the optimizer successfully optimized the circuit. ✓* indicates that a circuit was produced, but the algorithm did not terminate. *times* indicates the optimizer did not support the circuit. *times** indicates the optimizer timed out with no result after 72 hours.

benchmarks	name	Q	F	ZX	# qs	2q gates	1q gates	depth	description
Spire-generated	len1	×*	✓	✓	54	912	1294	703	list length, n=1
	rm1	×*	✓	✓	65	3224	4712	3728	remove element, n=9
	fpos1	×*	✓	✓	57	924	1262	781	find position, n=1
	sum1	×*	✓	✓	64	1006	1334	796	sums elements in list, n=1
	pushb1	×*	✓	✓	49	2328	3358	3090	appends element, n=1
	popf1	✓*	✓	✓	36	526	726	451	pops element, n=1
	match1	×*	✓	✓	256	4862	6196	4260	count matches, n=1
	prefix1	×*	✓	✓	135	1781	2216	1617	string prefix, n=1
	cmp1	×*	✓	✓	154	2194	2885	2001	compare strings, n=1
Unqomp-generated (Arithmetic)	add6	✓*	✓	✓	18	92	120	129	6 bit adder
	add12	✓*	✓	✓	36	200	264	267	12 bit adder
	incmp6	✓*	✓	✓	11	30	82	76	integer comparison, 6 bits
	incmp12	✓*	✓	✓	23	66	384	383	integer comparison, 12 bits
	wa4	✓*	✓	✓	13	209	384	383	weighted adder, 4 bits
	wa12	×*	✓	✓	27	1086	2004	1885	weighted adder, 12 bits
	mul3	✓*	✓	✓	15	150	216	205	multiplier, 3 bits
	mul6	×*	✓	✓	30	678	972	826	multiplier, 6 bits
	mul12	×*	✓	✓	60	2868	4104	3310	multiplier, 12 bits
	WAsvq4†	✓*	✓	✓	12	239	444	443	efficient weighted adder, 4 bits
	WAsvq8†	×*	✓	✓	18	683	1276	1284	efficient weighted adder, 8 bits
	WAsvq12†	×*	✓	✓	24	1302	2436	2467	efficient weighted adder, 12 bits
Unqomp-generated (Quantum)	plr3	×	✓	✓	7	76	134	137	piecewise linear rotations
	plr12	×	✓	✓	25	262	508	523	piecewise linear rotations
	mcry12	×	✓	✓	24	68	134	161	12 controlled Y rotation
	dj10	✓*	✓	✓	19	54	127	126	deutch-jozsa algorithm
	grov8	×*	✓	✓	15	936	2218	2184	8 bit grover algorithm

Experimental Setup. We compare SPARE against the baseline compiler-generated circuit implementations and against Quartz (**Q**) [53], Feynman (**F**) [2, 35] and PyZX (**ZX**) [24, 40] circuit optimizers. These optimizers were used previously to optimize Spire-generated circuits in [55]. For Quartz, we use the **Nam** gateset target. For Feynman, we use the **-mctExpand -O2** configuration on the input circuits described in the “qc" front-end. These settings were also used in Spire [55]. For PyZX, we use the Python front-end with **opt** setting to apply all optimizations. After these tools or SPARE is applied, we use a Qiskit transpiler pass to ensure the generated circuits are in the target gateset. We evaluate these optimizers on all benchmarks and allocate 72 hours each. In this execution setting, SPARE and the baseline optimizers identify circuit optimization opportunities that Spire and Unqomp compilers missed. In Table 2 we indicate a successful/unsuccessful optimizer run with ✓/×. If the optimizer did not terminate its search in 72 hrs but did produce an intermediate optimized circuit we mark it as ✓*. If the optimizer timed out with no result we report it with ×*. Whereas SPARE successfully optimizes all benchmarks. Along with the absolute counts for **#qs**, **2q**, **1q**, and **depth**, we also report the improvement relative to the original compiler-generated circuit implementations for each benchmark. This relative improvement is calculated as: $(optMetric - inputMetric)/inputMetric \times 100\%)$. A negative percent change indicates an improvement and is annotated in green, and a positive percent change indicates worse performance and is annotated in red.

Table 3. Comparison of SPARE to Quartz, Feynman and PyZX. These optimizers did not reduce *#qs*.

name	Quartz			Feynman			PyZX			SPARE (us)			
	2q gates	1q gates	depth	2q gates	1q gates	depth	2q gates	1q gates	depth	# qs	2q gates	1q gates	depth
len1				874	1007	498	1041	1128	794	49	558	923	408
				-4%	-22%	-29%	+14%	-13%	+13%	-9%	-39%	-29%	-42%
rem1				2560	2901	2097	4307	4159	3129	61	1829	3135	2486
				-21%	-38%	-44%	+34%	-12%	-16%	-6%	-43%	-33%	-33%
fpos1				894	1003	573	1146	1281	812	51	463	744	454
				-3%	-21%	-27%	+24%	+2%	+4%	-11%	-50%	-41%	-42%
sum1				974	1039	586	1359	1279	803	59	686	995	571
				-3%	-22%	-26%	+35%	-4%	+1%	-8%	-32%	-25%	-28%
pushb1				1546	1732	1501	2249	2498	2058	49	1283	2121	2262
				-34%	-48%	-51%	-3%	-26%	-33%	0%	-45%	-37%	-27%
popf1	530	601	461	524	604	329	713	708	544	36	172	280	181
	0%	-17%	+2%	0%	-17%	-27%	+36%	-2%	+21%	0%	-67%	-61%	-60%
match1				4678	4879	3214	16098	12769	7842	234	3286	4527	3679
				-4%	-21%	-25%	+231%	+106%	+84%	-9%	-32%	-27%	-14%
prefix1				1725	1824	1239	3825	3871	2394	111	1169	1650	1335
				-3%	-18%	-23%	+115%	+75%	+48%	-18%	-34%	-26%	-17%
cmp1				2176	1968	1585	3712	3520	2278	131	1379	2035	1577
				-1%	-32%	-21%	+69%	+22%	+14%	-15%	-37%	-29%	-21%
add6	87	89	125	142	120	127	143	150	142	17	76	100	101
	-5%	-26%	-3%	+54%	0%	-2%	+55%	+25%	+10%	-6%	-17%	-17%	-22%
add12	282	264	333	310	264	271	285	335	286	35	160	208	217
	+41%	0%	+25%	+55%	0%	+1%	+43%	+27%	+7%	-3%	-20%	-21%	-19%
incnp6	38	71	81	54	84	71	82	104	125	8	20	50	46
	+27%	-13%	+7%	+80%	+2%	-7%	+173%	+27%	+64%	-27%	-33%	-39%	-39%
incnp12	86	166	179	126	194	160	142	217	205	18	48	122	100
	+30%	-57%	-53%	+91%	-49%	-58%	+115%	-43%	-46%	-22%	-27%	-68%	-74%
wa4	389	474	571	335	383	381	474	530	584	12	154	296	284
	+86%	+23%	+49%	+60%	0%	-1%	+127%	+38%	+52%	-8%	-26%	-23%	-26%
wa12				1770	1944	1933	2769	2776	3388	28	996	1829	1721
				+63%	-3%	+3%	+155%	+39%	+80%	+4%	-8%	-9%	-9%
mul3	196	210	224	246	203	182	214	186	226	13	65	107	100
	+31%	-3%	+9%	+64%	-6%	-11%	+43%	-14%	+10%	-13%	-57%	-50%	-51%
mul6				1104	857	773	949	1027	870	28	305	545	412
				+63%	-12%	-6%	+40%	+6%	+5%	-7%	-55%	-44%	-50%
mul12				4656	3515	3251	4414	4541	3715	58	1326	2450	1654
				+62%	-14%	-2%	+54%	+11%	+12%	-3%	-54%	-40%	-50%
WAssvq4	389	474	571	395	471	452	596	645	709	12	154	296	284
	+63%	+7%	+29%	+65%	+6%	+2%	+149%	+45%	+60%	0%	-36%	-33%	-36%
WAssvq8				1625	1728	1374	2021	1949	2282	18	557	1158	1111
				+138%	+35%	+7%	+196%	+53%	+78%	0%	-18%	-9%	-13%
WAssvq12				3210	3419	2659	3538	3716	4022	24	1078	2238	2183
				+147%	+40%	+8%	+172%	+53%	+63%	0%	-17%	-8%	-12%
plr3				68	34	75	76	105	141	7	40	68	73
				-11%	-75%	-45%	0%	-22%	+3%	0%	-47%	-49%	-47%
plr12				302	168	318	341	467	588	25	197	364	381
				+15%	-67%	-39%	+30%	-8%	+12%	0%	-25%	-28%	-27%
mcry12				134	47	157	191	259	272	23	68	126	85
				+97%	-65%	-2%	+181%	+93%	+69%	-4%	0%	-6%	-47%
dj10	19	70	127	102	79	122	135	172	167	19	54	124	70
	0%	+30%	+0%	+89%	-38%	-3%	+150%	+35%	+33%	0%	0%	-2%	-44%
grov8				1728	2290	2089	1992	2464	3086	15	1068	2229	1622
				+85%	+3%	-4%	+113%	+11%	+41%	0%	+14%	0%	-26%

6.1 Comparison to Quartz, Feynman, and PyZX Circuit Optimizers

Table 3 presents the qubit counts (**# qs**), 2-qubit gate counts (**2q gates**), 1-qubit gate counts (**1q gates**), and circuit depth (**depth**) of SPARE-optimized circuits, compared to Quartz, Feynman and PyZX optimized circuits, along with percentage improvement over original Unqomp/Spire-generated implementations. We find that SPARE produces circuits with up to 24 fewer qubits, 1576 fewer 2-qubit gates, 2362 fewer 1-qubit gates, and a reduction of 1932 in circuit depth compared to the original circuits. Across all benchmarks, SPARE provides a +4% to -27% change in the number of qubits, a +14% to -67% change in 2-qubit gates, a 0% to -68% change in 1-qubit gates, and a +14 to -74% change in circuit depth.

For all but two benchmarks, SPARE delivered strict performance improvements. First, for the **wa12** benchmark, SPARE produces a circuit with one additional qubit but achieves a reduction of 90 two-qubit gates, 175 one-qubit gates, and 164 in circuit depth—corresponding to 8%, 9%, and 9% improvements, respectively. This input circuit, generated by Unqomp, leaves limited opportunity for qubit reduction because it trades off extra qubits for a reduction in gate count and circuit depth [38]. Second, the **grov8** benchmark contains various n-controlled Toffoli gates interleaved with non-commuting gates that can not be swapped around the Toffoli gates using SPARE's movement rewrites. Thus, SPARE can only optimize these n-controlled Toffoli gates in isolation. For n-controlled Toffoli gates when $n = 2^i + 1$ where $i \in \mathbb{N}$, SPARE's lowering implementation prioritizes depth over gate counts, thus the SPARE-generated circuit results in 14% more 2-qubit gates, the same number of 1-qubit gates, but a reduction of 26% (total of 562) in circuit depth.

Comparison to Quartz. Quartz reduces original compiler-generated circuits by up to 5 2-qubit gates, 218 1-qubit gates and reduces circuit depth by up to 204 gates. Quartz does not support Margolus gate-based lowering, leading to worse results for Unqomp-generated circuits. We find that SPARE strictly outperformed Quartz for 8 out of 9 benchmarks. For the **add6** benchmark, SPARE optimized circuit uses 11 more 1-qubit gates but reduces 2-qubit gates by 11 and circuit depth by 14. Overall, we find that SPARE-generated circuits show -11 to -235 change in 2-qubit gates, +11 to -178 change in 1-qubit gates and a -24 to -287 change in circuit depth against Quartz.

Comparison to Feynman. Feynman reduces the original compiler-generated circuits by up to 782 2-qubit gates, 1811 1-qubit gates and reduces circuit depth by up to 1631. It does not implement Margolus gate-based lowering or eliminate qubits from the circuit, limiting its gate and qubit savings. SPARE strictly outperforms Feynman for 17 of 26 benchmarks. We next discuss the benchmarks Feynman outperforms SPARE on. For **mcry12**, SPARE gets 45 more 1-qubit gates but improves on all other metrics. In **match1, prefix1**, SPARE produces circuits with a 96-465 higher depth, but removes up to 24 qubits, 1392 2-qubit gates, and 352 1-qubit gates. In **cmp1, plr3, dj10** SPARE uses 26-67 more 1-qubit gates but removes up to 797 2-qubit gates and reduces circuit depth by up to 72. Finally, in **rem1, pushb1, plr12**, SPARE circuits have 196-389 more 1-qubit gates and 63-761 higher circuit depth, but up to 4 fewer qubits and 731 fewer 2-qubit gates. To summarize, SPARE always beats Feynman in qubit and 2-qubit gate counts, and achieves mixed results on 1-qubit gate counts and depth. We note that 2-qubit gates are far more expensive than 1-qubit gates in quantum circuits [19, 29].

Comparison to PyZX. PyZX reduces the original compiler-generated circuits by up to 79 2-qubit gates, 860 1-qubit gates and reduces circuit depth by up to 1032. PyZX does not implement Margolus gate-based lowering or eliminate any qubits. PyZX generally results in more 2-qubit gates but finds a decent reduction in 1-qubit gates and circuit depth compared to the unoptimized baseline. SPARE strictly outperforms PyZX for 25 out of 26 benchmarks. For the **pusb1** benchmark, SPARE gets 204 higher circuit depth but removes up to 966 2-qubit and 377 1-qubit gates. Overall, SPARE-optimized circuits have up to 12812 fewer 2-qubit gates, 8242 fewer 1-qubit gates, and a reduction of 4163 in circuit depth. We conclude that ZX calculus-based circuit optimizers such as PyZX can lead to smaller depth, but SPARE can optimize CTU circuits far more efficiently, especially in 2-qubit and 1-qubit gate counts.

Table 4. Circuit complexity of SPARE-optimized circuit with different input sizes, compilation passes.

length-simplified		Spire				**SPARE**				**relative change**			
Recurse Depth	Pass	#qs	2q gates	1q gates	depth	# qs	2q gates	1q gates	depth	Δ #qs	Δ2q gates	Δ1q gates	Δ depth
1	all	26	224	318	254	19	72	105	107	-27%	-67%	-67%	-58%
1	cf	26	346	530	453	17	52	83	86	-35%	-85%	-84%	-81%
1	cn	26	224	318	254	19	72	105	107	-27%	-68%	-67%	-58%
1	none	26	346	530	453	17	52	83	86	-35%	-85%	-84%	-81%
5	all	86	1270	1778	1046	49	235	379	304	-43%	-81%	-78%	-71%
5	cf	77	3026	4350	4417	54	846	1349	1307	-30%	-72%	-69%	-70%
5	cn	82	6352	9460	7689	61	1478	2334	1845	-26%	-77%	-75%	-76%
5	none	82	11450	17139	11882	61	1868	2721	1929	-26%	-84%	-84%	-84%
9	all	146	2320	3244	1845	82	330	544	380	-44%	-86%	-83%	-79%
9	cf	146	5706	8158	8389	113	1598	2493	2487	-23%	-72%	-69%	-70%
9	cn	128	44236	66282	40483	129	5056	6720	5003	+1%	-88%	-90%	-87%
9	none	138	60992	91452	52946	138	5449	5680	4019	0%	-91%	-94%	-92%

Table 5. SPARE-optimization of hand implemented circuits. No ancilla reduction was found.

	Hand Implemented [32]			**SPARE**			**relative change**		
benchmark	2q gates	1q gates	depth	2q gates	1q gates	depth	Δ2q gates	Δ1q gates	Δ depth
2of5	93	126	144	80	108	124	-14%	-14%	-14%
sixsim	169	235	167	161	275	160	-5%	+15%	-4%
rd53	107	144	126	94	126	115	-12%	-13%	-9%

Runtime Comparison. SPARE optimizes Unqomp circuits in 0.85 seconds to 10 minutes 29 seconds, and Spire circuits in 22.6 seconds - 2 hours 4 minutes (longest for `match1`). Quartz did not complete the optimization process for any circuit within 72 hours, and took at least 3 hours to find an optimized circuit for each reported benchmark. Quartz is much slower as it is a super-optimizing compiler and operates on a much lower level of abstraction. The Feynman optimizer is much faster, requiring 0–14.34 seconds depending on the circuit. PyZX has runtimes of 2.1 seconds to 1 hour 9 minutes 29 seconds, operating on a similar timescale as SPARE. Both SPARE and PyZX are implemented in Python, whereas Feynman is implemented in Haskell, which may explain some of this performance difference.

6.2 Effect of Spire Compiler Optimizations on SPARE

We next examine how SPARE's performance changes when different Spire optimizations are enabled and how it scales with recursion depth. We use a simplified version of the length benchmark (`len-simplified`) used in the scaling study in Spire [55] and generate circuits with different optimization settings: all optimizations (`all`), only conditional narrowing (`cn`), only conditional flattening (`cf`), and none (`none`) for program depth 1,5,9. For a fair ablation study, we disable the Margolus-gate lowering and lower Toffoli gates exactly, as done in Spire. Table 4 summarizes the results.

We note that in some cases, Spire's high-level rewrites result in suboptimal circuits after SPARE is applied. In fact, after applying SPARE rewrites with recursion depth 1, the circuit where `none` of the Spire optimizations are enabled has lower circuit depth and complexity (in terms of number of gates) compared to the circuit where `all` of optimizations are enabled.

We also observe that SPARE performs better with increasing input size. With all optimizations enabled, SPARE results in an increasing 2-qubit gate reduction of 67%, 81% and 86%. The 1-qubit gates are reduced by 67%, 78% and 83%. The circuit depth is reduced by 58%, 71% and 79%.

6.3 Optimization of Gate-Level CTU Circuits

Table 5 presents SPARE's performance on hand-optimized CTU circuits. These circuits were previously used to assess the Square compiler [10] and sourced from [32]. We disable Margolus-gate-based lowering to ensure fair comparison since the baseline circuits were optimized for lowering to Toffoli basis gates. We find SPARE is still able to achieve marginal improvements, reducing the circuits by up to 13 2-qubit gates, 18 1-qubit gates and 20 depth. This is a -5 to -14% change in 2-qubit gates, +15% to -14% change in 1-qubit gates and a -4 to -14% change in depth.

7 Related Work

Quantum Programming Languages. Various high-level quantum languages that offer different programing constructs like the with-do blocks [54], within-apply blocks [48], quantum conditionals [1, 22], and modular compute/transform constructs [51] have been proposed. These programming constructs often yield circuits with CTU gate patterns and ancilla qubits. Existing QOCs corresponding to these languages, like Spire [55], employ language-level rewrites, instantiating new ancilla qubits and CTU gate patterns in the process. SPARE uses gate-level rewrites specialized to optimising CTU gate patterns present in these circuits. While SPARE's rewrites map to high-level rewrites in certain cases, we find SPARE's gate-level optimizations exploit opportunities not present in the high-level program.
Compilers for Automatic-Uncomputation. Languages like Silq [6] consist of compute and transform operations and use a type-system to annotate ancilla qubits and operations for uncomputation. Although compiling Silq is an active research area, various compilers that can handle some of its features exist [38, 39, 50]. These compilers optimize CTU programs by lowering compute/transform operations and automatically synthesizing uncompute gates. They particularly focus on achieving efficient uncomputation and do not rewrite the program's compute operations. In contrast, SPARE rewrites the entire CTU gate pattern to optimize the circuit, while leveraging ancilla information to delete gates and controls.
Gate-Level Circuit Optimizers. Many previous circuit optimizers such as Quartz [53], Nam [36], Qiskit [21] and more [17, 28, 46, 52] operate on lowered quantum circuits with 1-qubit/2-qubit gates. Thus, the optimization presented in these tools can be used in conjunction with SPARE. In contrast, SPARE rewrites circuits with CTU patterns at multi-controlled gate level.

Other circuit optimizers that work with multi-controlled gates [33] are typically limited to a single CTU pattern comprised of Toffoli-like gates. In contrast, SPARE supports optimizing across multiple CTU patterns, enabling the optimization of a broader set of circuits with CTU patterns. The DARE compiler also targets multi-controlled single-target gates but is focused on qutrit-based circuits only [42]. SPARE instead targets ancilla-based qubit circuits generated from high-level languages. While DARE cannot compile to qubit circuits, its qutrit circuits follow similar, compute-uncompute gate patterns.

8 Conclusion

The explosive growth in the software tooling and support for programming abstractions in quantum computers has led to a large increase in the kind and scale of problems that can be mapped to quantum circuits. These tools utilize ancillas that are used with compute-transform-uncompute pattern operations to support such programming constructs. We present SPARE, a compiler that targets quantum circuits with compute/uncompute patterns using multi-controlled gate-level rewrites. We use ancilla state information to break down and restructure CTU operations while preserving functionality and supporting multiple front-end language compilers. SPARE improves upon the generated circuits from Spire and Unqomp compilers, as well as gate-level rewriting tools like Quartz, Feynman and PyZX. Thus, with SPARE, we bridge the gap between high-level and low-level optimization techniques in quantum computing for circuits with ancillas and CTU gate patterns.

Data Availability Statement

The code/artifact for this work is made available on Zenodo [41].

Acknowledgments

We thank Mark Horowitz for his great insights and advice on the paper. We would also like to thank Charles Yuan for his help in understanding prior work, as well as Dev Iyer and Pu (Luke) Yi for their suggestions and feedback on the paper. Ritvik Sharma was supported by the Stanford Graduate Fellowship. Furthermore, this work was also partially supported by the Stanford Portal Center and the System X Alliance. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

References

[1] T. Altenkirch and J. Grattage. 2005. A functional quantum programming language. In *20th Annual IEEE Symposium on Logic in Computer Science (LICS' 05)*. 249–258. doi:10.1109/LICS.2005.1

[2] Matthew Amy, Dmitri Maslov, and Michele Mosca. 2014. Polynomial-Time T-Depth Optimization of Clifford+T Circuits Via Matroid Partitioning. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33, 10 (2014), 1476–1489. doi:10.1109/TCAD.2014.2341953

[3] Ryan Babbush, Craig Gidney, Dominic W. Berry, Nathan Wiebe, Jarrod McClean, Alexandru Paler, Austin Fowler, and Hartmut Neven. 2018. Encoding Electronic Spectra in Quantum Circuits with Linear T Complexity. *Phys. Rev. X* 8 (Oct 2018), 041015. Issue 4. doi:10.1103/PhysRevX.8.041015

[4] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. 1995. Elementary gates for quantum computation. *Phys. Rev. A* 52 (Nov 1995), 3457–3467. Issue 5. doi:10.1103/PhysRevA.52.3457

[5] Charles H. Bennett, Gilles Brassard, Claude Crépeau, Richard Jozsa, Asher Peres, and William K. Wootters. 1993. Teleporting an unknown quantum state via dual classical and Einstein-Podolsky-Rosen channels. *Phys. Rev. Lett.* 70 (Mar 1993), 1895–1899. Issue 13. doi:10.1103/PhysRevLett.70.1895

[6] Benjamin Bichsel, Maximilian Baader, Timon Gehr, and Martin Vechev. 2020. Silq: a high-level quantum language with safe uncomputation and intuitive semantics. (2020), 286–300. doi:10.1145/3385412.3386007

[7] Kostas Blekos, Dean Brand, Andrea Ceschini, Chiao-Hui Chou, Rui-Hao Li, Komal Pandya, and Alessandro Summer. 2024. A review on Quantum Approximate Optimization Algorithm and its variants. *Physics Reports* 1068 (2024), 1–66. doi:10.1016/j.physrep.2024.03.002 A review on Quantum Approximate Optimization Algorithm and its variants.

[8] Qiskit Community. 2024. Qiskit: Quantum SDK - X Gate Implementation. https://github.com/Qiskit/qiskit/blob/stable/1.4/qiskit/circuit/library/standard_gates/x.py#L483-L495. Accessed: 23 Mar. 2025.

[9] Alexandre A. A. de Almeida, Gerhard W. Dueck, and Alexandre C. R. da Silva. 2019. CNOT Gate Mappings to Clifford+T Circuits in IBM Architectures. In *2019 IEEE 49th International Symposium on Multiple-Valued Logic (ISMVL)*. 7–12. doi:10.1109/ISMVL.2019.00010

[10] 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. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, 570–583. doi:10.1109/ISCA45697.2020.00054

[11] ETH-SRI. 2023. Unqomp: Uncomputation-Aware Quantum Compilation. https://github.com/eth-sri/Unqomp. Accessed: 2025-03-31.

[12] Craig Gidney. 2015. Constructing Large Controlled Nots. https://algassert.com/circuits/2015/06/05/Constructing-Large-Controlled-Nots.html.

[13] Google Quantum AI. 2023. Suppressing quantum errors by scaling a surface code logical qubit. *Nature* 614 (2023), 676–681. doi:10.1038/s41586-022-05434-1

[14] Daniel Gottesman. 1998. Theory of fault-tolerant quantum computation. *Phys. Rev. A* 57 (Jan 1998), 127–137. Issue 1. doi:10.1103/PhysRevA.57.127

[15] Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. 2013. Quipper: a scalable quantum programming language. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 333–342. doi:10.1145/2491956.2462177

[16] Lov K. Grover. 1996. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing* (Philadelphia, Pennsylvania, USA) *(STOC '96)*. Association for Computing Machinery, New York, NY, USA, 212–219. doi:10.1145/237814.237866

[17] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A verified optimizer for Quantum circuits. *Proc. ACM Program. Lang.* 5, POPL, Article 37 (Jan. 2021), 29 pages. doi:10.1145/3434318

[18] Keli Huang and Jens Palsberg. 2024. Compiling Conditional Quantum Gates without Using Helper Qubits. *Proc. ACM Program. Lang.* 8, PLDI, Article 206 (June 2024), 22 pages. doi:10.1145/3656436

[19] IBM. 2018. Quantum devices and simulators. https://www.research.ibm.com/bm-q/technology/devices/.

[20] IBM Quantum Team. [n. d.]. Qiskit Documentation: RCCXGate. https://docs.quantum.ibm.com/api/qiskit/qiskit.circuit.library.RCCXGate. Accessed: 2024-11-14.

[21] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. 2024. Quantum computing with Qiskit. doi:10.48550/arXiv.2405.08810 arXiv:2405.08810 [quant-ph]

[22] Ali JavadiAbhari, Shruti Patil, Daniel Kudrow, Jeff Heckey, Alexey Lvov, Frederic T. Chong, and Margaret Martonosi. 2014. ScaffCC: a framework for compilation and analysis of quantum computing programs. In *Proceedings of the 11th ACM Conference on Computing Frontiers* (Cagliari, Italy) *(CF '14)*. Association for Computing Machinery, New York, NY, USA, Article 1, 10 pages. doi:10.1145/2597917.2597939

[23] Isaac H. Kim, Ye-Hua Liu, Sam Pallister, William Pol, Sam Roberts, and Eunseok Lee. 2022. Fault-tolerant resource estimate for quantum chemical simulations: Case study on Li-ion battery electrolyte molecules. *Phys. Rev. Res.* 4 (Apr 2022), 023019. Issue 2. doi:10.1103/PhysRevResearch.4.023019

[24] Aleks Kissinger and John Wetering. 2020. PyZX: Large Scale Automated Diagrammatic Reasoning. *Electronic Proceedings in Theoretical Computer Science* 318 (04 2020), 230–242. doi:10.4204/EPTCS.318.14

[25] Alexei Y. Kitaev. 1995. Quantum measurements and the Abelian Stabilizer Problem. *Electron. Colloquium Comput. Complex.* TR96 (1995). https://api.semanticscholar.org/CorpusID:17023060

[26] Hochang Lee, Kyung Chul Jeong, Daewan Han, and Panjin Kim. 2024. An Algorithm for Reversible Logic Circuit Synthesis Based on Tensor Decomposition. *ACM Transactions on Quantum Computing* 5, 3, Article 15 (July 2024), 28 pages. doi:10.1145/3673242

[27] Muyuan Li, Daniel Miller, and Kenneth R. Brown. 2018. Direct measurement of Bacon-Shor code stabilizers. *Phys. Rev. A* 98 (Nov 2018), 050301. Issue 5. doi:10.1103/PhysRevA.98.050301

[28] Zikun Li, Jinjun Peng, Yixuan Mei, Sina Lin, Yi Wu, Oded Padon, and Zhihao Jia. 2024. Quarl: A Learning-Based Quantum Circuit Optimizer. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 114 (April 2024), 28 pages. doi:10.1145/3649831

[29] Norbert M. Linke, Dmitri Maslov, Martin Roetteler, Shantanu Debnath, Caroline Figgatt, Kevin A. Landsman, Kenneth Wright, and Christopher Monroe. 2017. Experimental comparison of two quantum computing architectures. *Proceedings of the National Academy of Sciences of the United States of America* (March 2017). doi:10.1073/pnas.1618020114

[30] Daniel Litinski. 2018. A Game of Surface Codes: Large-Scale Quantum Computing with Lattice Surgery. *Quantum* (2018). https://api.semanticscholar.org/CorpusID:53386026

[31] Daniel Litinski. 2023. How to compute a 256-bit elliptic curve private key with only 50 million Toffoli gates. https://api.semanticscholar.org/CorpusID:259164752

[32] David Maslov. 2021. Quantum CIrcuit Benchmarks. https://reversiblebenchmarks.github.io/.

[33] D. Maslov, G.W. Dueck, and D.M. Miller. 2005. Toffoli network synthesis with templates. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24, 6 (2005), 807–817. doi:10.1109/TCAD.2005.847911

[34] D. Maslov, G.W. Dueck, and D.M. Miller. 2005. Toffoli network synthesis with templates. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 24, 6 (2005), 807–817. doi:10.1109/TCAD.2005.847911

[35] Meamy. 2024. Feynman. https://github.com/meamy/feynman. Accessed: 2024-11-14.

[36] Yunseong Nam, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. 2018. Automated optimization of large quantum circuits with continuous parameters. *npj Quantum Information* 5 (May 2018).

[37] Michael A. Nielsen and Isaac L. Chuang. 2010. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press, Cambridge. doi:10.1017/CBO9780511976667

[38] Anouk Paradis, Benjamin Bichsel, Samuel Steffen, and Martin Vechev. 2021. Unqomp: synthesizing uncomputation in Quantum circuits. (2021), 222–236. doi:10.1145/3453483.3454040

[39] Anouk Paradis, Benjamin Bichsel, and Martin Vechev. 2024. Reqomp: Space-constrained Uncomputation for Quantum Circuits. *Quantum* 8 (Feb. 2024), 1258. doi:10.22331/q-2024-02-19-1258

[40] QuiZX Contributors. 2025. QuiZX: A Scalable ZX-calculus Based Quantum Circuit Optimizer. https://github.com/zxcalc/quizx. Accessed: 2025-03-25.

[41] Sara Achour Ritvik Sharma. 2025. Spare Compiler. doi:10.5281/zenodo.15307149

[42] Ritvik Sharma and Sara Achour. 2024. Compilation of Qubit Circuits to Optimized Qutrit Circuits. *Proc. ACM Program. Lang.* 8, PLDI, Article 158 (June 2024), 24 pages. doi:10.1145/3656388

[43] Vivek V. Shende and Igor L. Markov. 2009. On the CNOT-cost of TOFFOLI gates. *Quantum Info. Comput.* 9, 5 (May 2009), 461–486.

[44] P.W. Shor. 1994. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*. 124–134. doi:10.1109/SFCS.1994.365700

[45] Peter W. Shor. 1997. Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer. *SIAM J. Comput.* 26, 5 (1997), 1484–1509. doi:10.1137/S0097539795293172 arXiv:https://doi.org/10.1137/S0097539795293172

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

[47] Damian S. Steiger, Thomas Häner, and Matthias Troyer. 2018. ProjectQ: an open source software framework for quantum computing. *Quantum* 2 (Jan. 2018), 49. doi:10.22331/q-2018-01-31-49

[48] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. 2018. Q#: Enabling Scalable Quantum Computing and Development with a High-level DSL. In *Proceedings of the Real World Domain Specific Languages Workshop 2018* (Vienna, Austria) *(RWDSL2018)*. Association for Computing Machinery, New York, NY, USA, Article 7, 10 pages. doi:10.1145/3183895.3183901

[49] Maika Takita, A. D. Córcoles, Easwar Magesan, Baleegh Abdo, Markus Brink, Andrew Cross, Jerry M. Chow, and Jay M. Gambetta. 2016. Demonstration of Weight-Four Parity Measurements in the Surface Code Architecture. *Phys. Rev. Lett.* 117 (Nov 2016), 210505. Issue 21. doi:10.1103/PhysRevLett.117.210505

[50] Hristo Venev, Timon Gehr, Dimitar Dimitrov, and Martin Vechev. 2024. Modular Synthesis of Efficient Quantum Uncomputation. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 345 (Oct. 2024), 28 pages. doi:10.1145/3689785

[51] Finn Voichick, Liyi Li, Robert Rand, and Michael Hicks. 2023. Qunity: A Unified Language for Quantum and Classical Computing. *Proc. ACM Program. Lang.* 7, POPL, Article 32 (Jan. 2023), 31 pages. doi:10.1145/3571225

[52] Amanda Xu, Abtin Molavi, Lauren Pick, Swamit Tannu, and Aws Albarghouthi. 2023. Synthesizing Quantum-Circuit Optimizers. *Proc. ACM Program. Lang.* 7, PLDI, Article 140 (June 2023), 25 pages. doi:10.1145/3591254

[53] Mingkuan Xu, Zikun Li, Oded Padon, Sina Lin, Jessica Pointing, Auguste Hirth, Henry Ma, Jens Palsberg, Alex Aiken, Umut A. Acar, and Zhihao Jia. 2022. Quartz: Superoptimization of Quantum Circuits. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 625–640. doi:10.1145/3519939.3523433

[54] Charles Yuan and Michael Carbin. 2022. Tower: data structures in Quantum superposition. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 134 (Oct. 2022), 30 pages. doi:10.1145/3563297

[55] Charles Yuan and Michael Carbin. 2024. The T-Complexity Costs of Error Correction for Control Flow in Quantum Computation. *Proc. ACM Program. Lang.* 8, PLDI (June 2024). doi:10.1145/3656397

Received 2024-11-15; accepted 2025-03-06