C2Q: A Compiler Framework for Translating Classical C Programs to Quantum Circuits

Pietro Pizzoccheri

Department of Electronics, Information and Bioengineering
Politecnico di Milano, Milan, Italy
pietro.pizzoccheri@mail.polimi.it

Abstract

***Abstract*— This paper presents **C2Q**, a complete compilation framework that translates classical C programs into executable quantum circuits using Draper's QFT-based quantum arithmetic. The compiler implements a multi-stage pipeline consisting of frontend parsing, MLIR-based intermediate representation using the xDSL framework, an integrated optimization pipeline, and backend circuit generation with Qiskit. The system supports basic arithmetic operations (addition, subtraction, multiplication) with mixed-precision operands and features a comprehensive optimization framework that achieves 15–36% gate count reduction and 7–51% depth reduction across a benchmark suite of 9 test cases. All optimized circuits pass validation against expected results using Matrix Product State simulation. The project demonstrates the feasibility of automated quantum circuit synthesis from high-level classical code, bridging the gap between classical programming paradigms and quantum computing architectures.**

1 Introduction

Quantum computing promises exponential speedups for specific computational problems, yet developing quantum algorithms remains challenging due to the significant conceptual gap between classical programming paradigms and quantum circuit design. While classical programmers work with familiar abstractions like variables, arithmetic operations, and control flow, quantum programming requires reasoning about qubits, superposition states, and gate sequences—a barrier that limits accessibility to quantum computing.

The **C2Q** (C-to-Quantum) compiler addresses this challenge by automatically translating a subset of Standard C into executable quantum circuits. The system accepts classical C programs with integer arithmetic and generates functionally equivalent quantum circuits that can be simulated or executed on quantum hardware. This approach enables developers to express algorithms using familiar high-level constructs while the compiler handles the complex details of quantum gate synthesis.

1.1 Motivation and Scope

The primary motivation for C2Q is to explore automatic quantum circuit synthesis from classical code, particularly for arithmetic-intensive algorithms. Rather than manual quantum circuit design—which is error-prone and requires deep quantum expertise—the compiler automates the translation process, ensuring correctness while applying optimizations that would be tedious to implement manually.

The current implementation focuses on **integer arithmetic** using Draper's Quantum Fourier Transform (QFT) based algorithms [1], supporting:

- Addition and subtraction of arbitrary-width integers
- Multiplication through repeated addition
- Mixed-precision arithmetic with automatic width handling
- Multi-stage optimization to reduce gate count and circuit depth

1.2 Key Contributions

This project makes the following contributions:

1. A complete compilation infrastructure from C source to validated quantum circuits, including lexer, parser, IR generator, optimizer, and backend.
2. Implementation of Draper QFT arithmetic with support for mixed-width operands and dynamic register allocation.
3. An integrated optimization framework achieving 22.5% average gate reduction through iterative phase precision filtering and supporting passes.
4. SSA-aware backend circuit generation that correctly handles aggressive optimization and register renaming.
5. Comprehensive validation framework using Matrix Product State simulation for circuits with 24+ qubits.

The remainder of this paper is organized as follows: Section II describes the compiler architecture and pipeline stages. Section III explains the quantum arithmetic algorithms. Section IV details the optimization framework. Section V presents evaluation results on a benchmark suite. Section VI concludes with future directions.

2 Architecture

The C2Q compiler follows a classical multi-stage compilation pipeline, transforming C source code through increasingly specialized representations until producing executable quantum circuits. Figure **??** illustrates the complete dataflow through four major stages: Frontend, Intermediate Representation, Optimization, and Backend.

2.1 Frontend: Parsing and AST Construction

The frontend consists of three components implemented in the `frontend/` module:

Lexer (`lexer.py`): Implements tokenization of C source code, extending xDSL's generic lexer with C-specific token types including keywords (`int`, `return`), operators (`+`, `-`, `*`), and punctuation. The lexer produces a stream of tokens with associated source locations for error reporting.

Parser (`parser.py`): A recursive-descent parser that constructs an Abstract Syntax Tree (AST) from the token stream. The parser implements a symbol table with nested scoping to track variable declarations, enforce initialization-before-use semantics, and detect common errors like undefined variables or type mismatches. The implementation supports:

- Function definitions with `main` entry point
- Variable declarations with initializers
- Assignment statements
- Arithmetic expressions with operator precedence
- Return statements

AST Representation (`c_ast.py`): Defines node types for all supported C constructs, including `FunctionAST`, `VarDeclExprAST`, `BinaryExprAST`, and `ReturnExprAST`. Each node contains source location information for diagnostic messages.

2.2 Intermediate Representation: Quantum MLIR

After parsing, the IR generator (`ir_gen.py`) transforms the AST into quantum operations using the Multi-Level Intermediate Representation (MLIR) infrastructure via the xDSL framework.

Quantum Dialect (`dialects/quantum_dialect.py`): Defines a custom MLIR dialect using xDSL's IRDL (IR Definition Language) framework. The dialect includes over 20 operation types:

- **Initialization:** `InitOp` creates quantum registers
- **Single-qubit gates:** `OnQubitHadamardOp`, `OnQubitPhaseOp`, `OnQubitNotOp`
- **Multi-qubit gates:** `OnQubitCNotOp`, `OnQubitCCnotOp`, `OnQubitControlledPhaseOp`, `OnQubitSwapOp`
- **Measurement:** `MeasureOp` extracts classical results
- **Metadata:** `CommentOp` for IR annotation

The quantum dialect represents quantum registers as `VectorType` with element type `IntegerType(1)`, allowing the IR to model multi-qubit registers as composite SSA values. This representation naturally supports mixed-width arithmetic where operands may have different bit widths.

Register Naming Convention: Quantum registers follow the `qX_Y` naming scheme where `X` is the register number (unique per variable) and `Y` is the version number (incremented after each operation). This SSA-style naming enables precise value tracking through optimization passes.

IR Generation (`ir_gen.py`): Maps C operations to quantum gate sequences:

- Variable declarations → `InitOp` with classical value encoding
- Arithmetic operations → Draper QFT-based circuits
- Return statements → `MeasureOp` on result register

The IR generator maintains a symbol table mapping variable names to SSA values representing quantum registers. When generating arithmetic operations, it delegates to the `QuantumArithmetic` class (Section III).

2.3 Middle-End: Optimization Pipeline

The `middle_end/optimizations/` module implements pattern-based and analysis-based optimization passes coordinated by the `IntegratedQuantumOptimizer` class. The optimizer applies multiple passes iteratively until convergence, typically achieving stability in 2–3 iterations. Details of the optimization framework are presented in Section IV.

2.4 Backend: Circuit Generation and Validation

The backend (`backend/`) converts optimized MLIR to executable Qiskit circuits:

Circuit Generation (`run_qasm.py`): Traverses the MLIR module and maps quantum operations to Qiskit gates. A critical challenge is handling SSA value chains after aggressive optimization—the backend implements SSA-aware tracking to maintain correct qubit mappings even when register names have been renamed multiple times.

The generator collects metrics during traversal:

- Total gate count (all operations)
- Circuit depth (critical path length)
- Qubit count (maximum register index)
- Individual gate counts (CNOT, Hadamard, SWAP, phase gates)

Validation Framework (`validate.py`): Simulates generated circuits using Qiskit Aer's Matrix Product State (MPS) method. MPS simulation is efficient for Draper arithmetic circuits, which exhibit localized entanglement patterns, allowing validation of circuits with 24+ qubits without exponential memory overhead. The validator compares measured outcomes against expected values, confirming functional correctness.

3 Quantum Arithmetic

The core of C2Q's circuit synthesis is the implementation of Draper's QFT-based quantum arithmetic algorithms [1]. These algorithms perform arithmetic in the frequency domain using the Quantum Fourier Transform, achieving polynomial gate complexity for addition, subtraction, and multiplication.

3.1 Quantum Fourier Transform

The Quantum Fourier Transform maps a quantum state from the computational basis to the Fourier basis, where arithmetic operations become simpler. For an n-qubit register $|x\rangle = |x_0 x_1 \dots x_{n-1}\rangle$, the QFT produces:

$$QFT|x\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{2\pi i x k/2^n} |k\rangle \tag{1}$$

The QFT is implemented using Hadamard gates and controlled phase rotations:

```
def qft_on_vector(builder, vec):
    n = vector_width(vec)
    for i in range(n):
        # Hadamard on qubit i
        vec = OnQubitHadamardOp(vec, i)
        # Controlled phase rotations
        for j in range(i+1, n):
            angle = 2*pi / 2**(j-i+1)
            vec = OnQubitControlledPhaseOp(
                vec, j, i, angle)
    return swap_reverse_qubits(vec)
```

Listing 1: QFT Implementation (simplified)

The circuit depth is $O(n^2)$ due to the nested controlled phase rotations. The inverse QFT (IQFT) reverses this transformation using negated phase angles.

3.2 Addition

Draper addition transforms the second operand to the Fourier basis, applies controlled phase rotations proportional to the first operand, then transforms back:

$$|a\rangle|b\rangle \xrightarrow{QFT} |a\rangle QFT|b\rangle \xrightarrow{\text{phase}} |a\rangle QFT|a+b\rangle \xrightarrow{IQFT} |a\rangle|a+b\rangle \tag{2}$$

The key insight is that in the Fourier basis, addition becomes phase multiplication. For each bit a_i of the first operand, we apply controlled phase rotations to all qubits of the second operand:

```
def quantum_add(builder, a, b):
    # Transform b to Fourier basis
    b = qft_on_vector(builder, b)

    # Apply controlled phases
    n_a = vector_width(a)
    n_b = vector_width(b)
    for i in range(n_a):
        for j in range(n_b):
            angle = 2*pi / 2**(j+1)
            b = OnQubitControlledPhaseOp(
                b, i, j, angle, control=a)

    # Transform back
    return iqft_on_vector(builder, b)
```

Listing 2: Addition Implementation

The total gate count is $O(n^2)$ where n is the register width. For 8-bit addition, the unoptimized circuit contains approximately 120 gates (16 Hadamards for QFT/IQFT, 8 CNOTs for controlled operations, and 92 phase gates).

3.3 Subtraction

Subtraction uses the same structure as addition but with negated phase angles, effectively computing $|a\rangle|b-a\rangle$:

```
def quantum_subtract(builder, a, b):
    b = qft_on_vector(builder, b)
    for i in range(n_a):
        for j in range(n_b):
            angle = -2*pi / 2**(j+1)  # Negative!
            b = OnQubitControlledPhaseOp(
                b, i, j, angle, control=a)
    return iqft_on_vector(builder, b)
```

Listing 3: Subtraction via Negated Phases

3.4 Multiplication

Multiplication is implemented through repeated addition, leveraging the quantum adder as a subroutine. For $a \times b$, we perform b additions of a to an accumulator register:

```python
def quantum_multiply(builder, a, b):
    width = vector_width(a) + vector_width(b)
    acc = InitOp(VectorType(IntegerType(1), [
        width]))

    # Classical iteration over bits of b
    for i in range(vector_width(b)):
        # If bit i of b is 1, add (a << i) to acc
        shifted_a = shift_left(a, i)
        acc = quantum_add_conditional(
            builder, shifted_a, acc,
            control_bit=b[i])
    return acc
```

Listing 4: Multiplication via Repeated Addition

The conditional addition uses controlled arithmetic: if the i-th bit of b is 1, add $(a \ll i)$ to the accumulator. This requires extending the adder with an additional control qubit.

For small operands (e.g., 2×3), the base circuit contains approximately 3,155 gates and depth 2,051. Multiplication has $O(n_a \cdot n_b \cdot (n_a + n_b)^2)$ complexity due to repeated applications of the $O(n^2)$ adder.

3.5 Mixed-Width Arithmetic

A key feature is support for mixed-precision operands. When adding an 8-bit and 16-bit number, the compiler:

1. Detects operand widths from register types
2. Promotes the narrower operand by zero-extension
3. Performs arithmetic at the wider precision
4. Allocates result register with appropriate width

This is implemented transparently in the IR generator's width handling logic, allowing expressions like `int8 + int16` to work correctly.

4 Optimizations

Raw Draper arithmetic circuits contain significant redundancy and precision overhead. The C2Q optimization framework applies multiple passes to reduce gate count and circuit depth while preserving functional correctness. The `IntegratedQuantumOptimizer` class coordinates all optimization passes, applying them iteratively until convergence.

4.1 Optimization Framework Architecture

The optimizer implements a fixed-point iteration strategy:

```python
def optimize_circuit(module):
    iteration = 0
    while iteration < max_iterations:
        prev_metrics = collect_metrics(module)
        apply_all_passes(module)
        new_metrics = collect_metrics(module)
        if no_improvement(prev_metrics,
            new_metrics):
            break
        iteration += 1
    return module
```

Listing 5: Optimization Loop

Typical circuits converge in 2–3 iterations. The first iteration often eliminates outer-layer redundancy (e.g., adjacent Hadamards at QFT/IQFT boundaries), exposing opportunities for subsequent passes to consolidate inner gates.

4.2 Primary Optimization: Phase Precision Filtering

The dominant optimization is **phase precision filtering**, implemented in `draper_optimizer.py`. This pass eliminates phase rotations below a configurable threshold (default: 10^{-4} radians in benchmarks).

Rationale: Draper QFT arithmetic generates many fine-grained phase rotations. For example, an 8-bit addition produces phases like $2\pi/2^8 \approx 0.0245$ radians. For practical integer operands, high-precision rotations (< 0.1 rad ≈ 5.7) have negligible effect on measurement outcomes and can be safely eliminated.

Implementation:

```python
def optimize_phase_precision(module, threshold):
    for op in module.walk():
        if isinstance(op, OnQubitPhaseOp):
            angle = abs(get_phase_angle(op))
            if angle < threshold:
                erase_operation(op)
                stats.gates_eliminated += 1
```

Listing 6: Phase Precision Filter

Impact: Across the benchmark suite, phase precision filtering achieves:

- Simple arithmetic (8-bit add/sub): ∼20% phase gate reduction
- Multiplication circuits: ∼55% phase gate reduction
- Mixed-width operations: ∼55% phase gate reduction

The effectiveness varies with circuit structure—multiplication generates more high-precision phases due to repeated additions, providing more optimization opportunities.

4.3 Supporting Optimization Passes

The framework includes several additional passes, though their impact is circuit-dependent:

Dead Code Elimination (`remove_unused_op.py`): Quantum-safe removal of operations whose results are never used. Preserves measurement operations and their dependencies to maintain observable behavior. Uses SSA def-use chains to identify truly dead operations.

CCNOT Decomposition (`ccnot_decomposition.py`): Decomposes Toffoli gates into 1- and 2-qubit gates using the Barenco construction. The current benchmark circuits do not heavily utilize CCNOT gates, so this pass shows minimal impact but is included for completeness.

Adjacent Phase Consolidation (`draper_optimizer.py`): Merges consecutive phase gates acting on the same qubit:

$$P(\theta_1)P(\theta_2) = P(\theta_1 + \theta_2) \tag{3}$$

After phase precision filtering removes some gates, previously non-adjacent phases may become adjacent, allowing consolidation.

Hadamard Cancellation (`draper_optimizer.py`): Cancels adjacent Hadamard gates at QFT/IQFT boundaries:

$$HH = I \tag{4}$$

When a QFT immediately follows an IQFT (common in compound expressions), the final Hadamards of IQFT cancel with the initial Hadamards of QFT.

QFT Depth Reduction (`draper_optimizer.py`): Optimizes QFT depth when operands use fewer bits than the register width. If only the lower k bits are non-zero, the QFT need not process all n qubits.

Redundant SWAP Elimination (`draper_optimizer.py`): Removes unnecessary SWAP operations in QFT circuits, particularly when SWAPs are used for qubit reordering but the ordering is irrelevant to subsequent operations.

4.4 Optimization Statistics

Table 1 summarizes the per-pass contribution across the benchmark suite. Phase precision filtering is responsible for the majority of improvements, with supporting passes providing incremental benefits depending on circuit structure.

Table 1: Optimization Pass Impact

Pass	Avg Gates Eliminated	Primary Benefit
Phase Precision	19–56%	Gate count
Hadamard Cancel	0–2%	Depth
Phase Consolidation	1–3%	Gate count
Dead Code Elim.	0–1%	Cleanup

The iterative strategy is crucial: the first iteration eliminates outer redundancy, enabling subsequent iterations to optimize inner circuit structure. Without iteration, many consolidation opportunities would be missed.

5 Evaluation

We evaluate C2Q on a benchmark suite of 9 test cases covering basic arithmetic, compound expressions, and edge cases. All tests execute on Python 3.12 with Qiskit 1.3.1 and xDSL 0.25.0. Validation uses Qiskit Aer's Matrix Product State simulator with 1,024 shots per circuit.

5.1 Benchmark Suite

The test suite includes:

- **Add (8-bit):** Addition of 3 + 5 = 8
- **Sub (8-bit):** Subtraction of 8 - 3 = 5
- **Mult (2×3):** Small multiplication 2 × 3 = 6
- **Assignment:** Simple variable assignment without arithmetic
- **Optimization Showcase:** Complex expression with multiple operations
- **Stress Test:** Chain of multiple arithmetic operations
- **Complex Math:** Mixed-width arithmetic (8-bit + 16-bit operands)
- **Overflow (8-bit):** Addition causing wrap-around (255 + 1)
- **Mixed Add/Mult:** Combined addition and multiplication

Each test is compiled twice: once without optimizations (baseline) and once with the integrated optimization pipeline (optimized). We measure:

- MLIR operation count (IR-level operations)
- Total gate count (circuit-level quantum gates)
- Circuit depth (critical path length)
- Individual gate counts (CNOT, Hadamard, SWAP, Phase)

5.2 Performance Results

Table 2 presents detailed benchmark results. The optimization pipeline achieves substantial improvements across all metrics:

Key Observations:

1. Optimization Effectiveness: The pipeline achieves 22.5% average gate reduction and 27.7% average depth reduction. Best-case improvements reach 36.4% gate reduction (Complex Math, Mixed Add/Mult) and 51.4% depth reduction (multiplication-heavy circuits).

2. Circuit-Dependent Performance: Simple addition/subtraction shows modest improvements (15%), while multiplication and compound expressions benefit significantly (35–36%). This reflects the distribution of optimization opportunities—multiplication generates many high-precision phase gates that are aggressively filtered.

Table 2: Benchmark Results: Baseline vs. Optimized Circuits

Test Case	Base Gates	Opt Gates	Reduction	Base Depth	Opt Depth	Depth Δ
Add (8-bit)	120	102	-15.0%	39	36	-7.7%
Sub (8-bit)	119	101	-15.1%	39	36	-7.7%
Mult (2×3)	3,155	2,051	-35.0%	2,051	1,002	-51.1%
Assignment	2	2	0.0%	1	1	0.0%
Opt Showcase	3,274	2,150	-34.3%	2,051	1,002	-51.1%
Stress Test	699	591	-15.5%	142	112	-21.1%
Complex Math	3,573	2,271	-36.4%	2,083	1,012	-51.4%
Overflow	125	107	-14.4%	39	36	-7.7%
Mixed Add/Mult	3,573	2,271	-36.4%	2,083	1,012	-51.4%
Average			**-22.5%**			**-27.7%**

3. Assignment Baseline: The assignment test (no arithmetic) shows 0% improvement, confirming optimizations do not affect non-arithmetic operations. This serves as a sanity check.

4. Depth Reduction: Circuit depth improves more dramatically than gate count for multiplication (51% vs. 35%). Phase precision filtering disproportionately affects the critical path by removing gates in series.

5.3 MLIR-Level Optimization

Table 3 shows the correlation between MLIR operation count and final gate count:

Table 3: MLIR Operations vs. Quantum Gates

Test	Base MLIR	Opt MLIR	MLIR Δ
Add (8-bit)	125	107	-14.4%
Mult (2×3)	3,160	2,056	-34.9%
Complex Math	3,580	2,278	-36.4%

MLIR operation reduction closely tracks gate count reduction, indicating optimizations operate effectively at the IR level before circuit generation. This validates the choice of MLIR as the optimization substrate.

5.4 Phase Gate Analysis

Phase gates dominate Draper arithmetic circuits. Table 4 shows their distribution:

Table 4: Phase Gate Optimization

Test	Base Phase	Opt Phase	Reduction
Add (8-bit)	92	74	-19.6%
Mult (2×3)	1,968	864	-56.1%
Complex Math	2,344	1,042	-55.6%

Phase precision filtering eliminates 20–56% of phase gates depending on circuit complexity. Multiplication circuits benefit most due to cumulative high-precision rotations from repeated additions.

5.5 Validation and Correctness

All 9 test cases pass validation with 100% success rate:

- Unoptimized circuits produce correct results
- Optimized circuits produce identical results
- Measurement distributions are deterministic (single outcome with >99% probability)

Matrix Product State simulation completes in <5 seconds per circuit on a MacBook Pro (M2), demonstrating practical validation for circuits up to 72 qubits (Stress Test).

5.6 Limitations and Edge Cases

Overflow Handling: The compiler correctly implements wrap-around arithmetic for overflow cases (e.g., 255 + 1 = 0 for 8-bit registers), matching C semantics. This is inherent to fixed-width quantum arithmetic.

Assignment-Only Code: Circuits without arithmetic operations show no optimization benefit, as expected. The framework correctly identifies such cases and skips unnecessary passes.

Scalability: Compilation time grows quadratically with register width due to $O(n^2)$ QFT complexity. For 16-bit operands, compilation takes ∼2 seconds; 32-bit operands would require ∼8 seconds. This is acceptable for a research prototype.

6 Conclusion

This paper presented C2Q, a complete compiler infrastructure for translating classical C programs into optimized quantum circuits. The system successfully bridges the gap between high-level programming abstractions and low-level quantum gate sequences, demonstrating that automated quantum circuit synthesis is both feasible and practical for arithmetic-intensive algorithms.

6.1 Achievements

The project delivers several concrete contributions:

Complete Compilation Pipeline: A working end-to-end system including lexer, parser, IR generator, optimizer, and backend. The implementation spans over 4,000 lines of Python code across 20+ modules, with comprehensive error handling and diagnostic messages.

Draper QFT Arithmetic: Full implementation of quantum addition, subtraction, and multiplication with support for mixed-precision operands. The quantum arithmetic module handles arbitrary register widths and automatically manages width promotion.

MLIR-Based IR: A custom quantum dialect using xDSL's IRDL framework, providing a clean separation between frontend parsing and backend circuit generation. The SSA-based representation enables powerful pattern-based optimizations.

Effective Optimization: An integrated optimization framework achieving 22.5% average gate reduction and 27.7% depth reduction across 9 test cases. The iterative optimization strategy with phase precision filtering as the primary pass demonstrates clear practical value.

Validated Correctness: All test cases pass validation with 100% accuracy, confirming that optimizations preserve functional correctness. The MPS-based validation framework efficiently handles circuits with 24–72 qubits.

6.2 Limitations and Challenges

Several limitations remain in the current implementation:

Limited C Support: The compiler only handles a small subset of C (integer arithmetic, basic control flow). Missing features include loops, conditionals, arrays, pointers, and function calls beyond `main()`.

Arithmetic-Only: The focus on Draper arithmetic means other quantum algorithmic patterns (e.g., oracles, amplitude amplification) are not supported. Extending to non-arithmetic operations would require additional algorithm libraries.

Scalability: The $O(n^2)$ gate complexity of QFT arithmetic becomes prohibitive for large bit widths. 64-bit arithmetic would generate circuits with tens of thousands of gates, challenging for current NISQ hardware.

Hardware Constraints: Generated circuits exceed the qubit capacity and coherence times of current quantum hardware. Near-term execution requires either simulation or significant additional work on error correction and resource management.

6.3 Future Directions

Several promising directions could extend this work:

1. **Control Flow:** Implementing quantum conditionals and loops using amplitude amplification or measurement-based control. This would enable compilation of more complex algorithms like GCD or factorization.
2. **Alternative Arithmetic:** Exploring carry-save adders or other arithmetic encodings that may offer better gate complexity or depth characteristics than Draper QFT.
3. **Advanced Optimizations:** Incorporating template-based peephole optimizations, global phase kickback analysis, or machine learning-guided optimization policies.
4. **Target-Specific Backends:** Generating circuits optimized for specific quantum hardware architectures (e.g., ion traps, superconducting qubits) with native gate sets and connectivity constraints.
5. **Reversible Computing:** Extending the compiler to generate reversible circuits with explicit uncomputation, reducing ancilla requirements and enabling garbage collection.
6. **Quantum-Classical Hybrid:** Integrating with classical control systems to support variational algorithms and quantum-classical hybrid workflows.

6.4 Broader Impact

C2Q demonstrates that quantum circuit synthesis from classical code is technically viable, providing a proof of concept for higher-level quantum programming abstractions. While significant work remains before such tools can be used for production quantum applications, this project establishes foundational techniques and validates the overall approach.

The open-source nature of the implementation (available at `github.com/pitesse/C2Q`) enables other researchers to build upon this work, extending the compiler with new optimizations, arithmetic algorithms, or language features. We hope this contributes to the broader effort to make quantum computing more accessible to software developers without quantum expertise.

As quantum hardware matures and error correction becomes practical, compilers like C2Q will play a crucial role in bridging the abstraction gap between classical programming paradigms and quantum execution models. This work represents one step toward that future.

References

[1] T. G. Draper, "Addition on a quantum computer," *arXiv preprint quant-ph/0008033*, 2000.