Optimization of a 12-Qubit Quantum Circuit via Structural Pattern Matching and Iterative Peephole Compilation

Ibraheem AlYousef
Term Project, COE 530

May 17, 2026

1 Introduction

The COE 530 term project requires compiling a fixed 12-qubit benchmark circuit into an OpenQASM 3 program with minimum cost under the weighted score

$$\text{Score} \;=\; 10 \cdot \text{Qubits} \;+\; \text{Depth} \;+\; \text{GateCount},$$

subject to the constraint that the compiled circuit's unitary equals the baseline unitary up to a single global phase, with maximum element-wise residual below 10^{-8} [4]. Barriers, measurements, delays, and resets are excluded from the depth and gate counters; everything else (including `ccx` and `swap`) counts as one gate.

The input `benchmark_circuit.qasm` has 12 qubits, depth 105, and 381 gates, yielding a baseline score of $10 \cdot 12 + 105 + 381 = 606$. The compilation pipeline presented in this report produces `benchmark_iterall.qasm` with 12 qubits, depth 46, and 219 gates, a final score of **385** — a 36% reduction. Unitary equivalence is verified by Qiskit Aer at a residual of 1.9×10^{-16}, well within the 10^{-8} tolerance.

The approach combines four ingredients: (i) syntactic pattern matching that recovers the four atomic Toffoli gates the baseline expresses as their textbook 15-gate decomposition, (ii) the Qiskit transpiler at `optimization_level=3` configured with a basis that preserves those Toffolis, (iii) an algebraic permutation-correction step that rebuilds exact unitary equivalence after Qiskit's elide-permutations pass absorbs terminal swaps into the layout, and (iv) a Clifford-rename pass that re-exposes named 1-qubit gates to a subsequent commutative-cancellation sweep. These passes are iterated to a fixed point on the (depth, gate-count) pair.

2 Techniques

The compilation entry point `iter_all.py` loads the baseline, performs an initial Toffoli-collapse pass, then iterates a body that combines Qiskit's optimization-level-3 transpilation with a peephole and Clifford-renaming loop, choosing the lower-scoring outcome each round and terminating when the score stalls.

2.1 Textbook-Toffoli pattern matching

The baseline expresses every CCX as the standard Nielsen & Chuang decomposition [2, §4.3],

$$\begin{aligned}\text{CCX}(a,b,t) \;= {} & H_t,\ \text{CX}_{b,t},\ T_t^\dagger,\ \text{CX}_{a,t},\ T_t,\ \text{CX}_{b,t},\ T_t^\dagger,\ \text{CX}_{a,t},\\ & T_t,\ T_b,\ H_t,\ \text{CX}_{a,b},\ T_a,\ T_b^\dagger,\ \text{CX}_{a,b}.\end{aligned}$$

The function `collapse_toffolis` in `recognize_toffoli.py` slides a length-15 window through the instruction list and tests two conditions:

1. **Name signature.** The 15 gate names must equal the sequence `[h, cx, tdg, cx, t, cx, tdg, cx, t, t, h, cx, t, tdg, cx]`.
2. **Qubit binding.** Let t be the qubit of the first `h`. Both `h` gates, the four `tdg`/`t` singletons on the target ladder (indices 2, 4, 6, 9), and the second-rail target of the alternating `cx`s must all equal t. Let b and a be the controls of the `cx`s at indices 1 and 3 respectively; then indices 5 and 7 must repeat those controls, index 8 must carry `t` on b, index 12 must carry `t` on a, index 13 must carry `tdg` on b, and indices 11 and 14 must be `cx`(a, b).

A matching window is replaced with a single `CCXGate` on (a, b, t). Symbolically, the predicate is

$$\text{match}(W) = \begin{cases} (a, b, t) & \text{if name signature and qubit constraints both hold,} \\ \bot & \text{otherwise,} \end{cases}$$

and the matched window contributes 14 gates of scoreboard savings (15 original gates collapse to 1 atomic `ccx`). The baseline contains four such windows, for a total of $4 \times 14 = 56$ gates saved on the first pass alone.

2.2 Qiskit `optimization_level=3` with a Toffoli-aware basis

After Toffoli collapse, the circuit is handed to Qiskit's transpiler [1] at `optimization_level=3` with `basis_gates=["cx","ccx","u"]`. The level-3 preset chains the heavy synthesis stack:

- `Collect2qBlocks` groups maximal subsequences of operations on at most two qubits;
- `ConsolidateBlocks` contracts each block into a 4×4 unitary;
- `UnitarySynthesis` re-emits each 4×4 as at most three `CX` gates surrounded by single-qubit rotations (a KAK-style decomposition);
- `Optimize1qGatesDecomposition` merges chains of single-qubit gates into a single `u(theta,phi,lambda)`;
- `CommutativeCancellation` and `InverseCancellation` eliminate adjacent commuting cancellations and adjacent involutive pairs.

Including `ccx` in the basis is essential: it tells the partitioner not to unroll the four Toffolis recovered in the previous step, so the 56-gate structural compression survives the synthesis stack.

2.3 Permutation correction

At `optimization_level=3` the transpiler's elide-permutations pass may absorb terminal `swap` gates into the layout, so that the emitted circuit's unitary U_{opt} differs from the reference unitary U_{ref} by a basis-permutation matrix P induced by a qubit permutation $\sigma : \{0, \ldots, n-1\} \to \{0, \ldots, n-1\}$. Setting `qc._layout = None` discards the latent permutation tracker; an explicit SWAP network is then appended to restore exact unitary equivalence.

The function `fix_against_reference` in `sliced_optimize.py` computes

$$D \;=\; U_{\text{opt}}\, U_{\text{ref}}^{\dagger} \qquad (\text{or } U_{\text{ref}}^{\dagger}\, U_{\text{opt}} \text{ when the permutation is on the right}).$$

A basis-permutation matrix P has the property that for every computational basis vector $\mathbf{e}_r$ with $r = 2^i$, $P\mathbf{e}_r$ is a basis vector $\mathbf{e}_c$ whose index c also has Hamming weight 1, i.e. $c = 2^{\sigma(i)}$. The detector reads off

$$\sigma(i) \;=\; \log_2\!\big(\arg\max_{c} |D_{r,c}|\big), \qquad r = 2^i,$$

after dividing D by the unit phase at its largest entry. It verifies $|D_{r,\sigma(i)}| > 0.99$ and popcount$(c) = 1$ for each row; if either test fails on any row, no clean permutation exists and the circuit is returned unmodified.

To realize σ^{-1} as a SWAP network, the algorithm bubble-sorts the array σ into the identity, emitting one `SWAP` per bubble step. A permutation with c cycles requires $n - c$ transpositions; for the final submitted circuit, σ has five cycles, so seven SWAP gates are appended.

2.4 Clifford renaming

After Qiskit's 1-qubit absorber runs, every 1-qubit gate is a generic `u(theta, phi, lambda)`. Even when (θ, ϕ, λ) corresponds exactly to a named Clifford such as H or S, the gate is opaque to `CommutativeCancellation`, whose commutation library is keyed on gate *name*.

`rewrite_1q` in `clifford_rewrite.py` walks the circuit and, for each 1-qubit operation, computes its 2×2 unitary and looks it up against a table of eleven matrices: $\{I, H, X, Y, Z, S, S^\dagger, \mathrm{SX}, \mathrm{SX}^\dagger, T, T^\dagger\}$. Matches are taken modulo global phase: $A \sim B$ iff there exists $\varphi \in [0, 2\pi)$ with $\max_{ij} |A_{ij} - e^{i\varphi} B_{ij}| < 10^{-6}$. A matched gate is replaced by the corresponding named gate (identities are dropped). This step does not directly remove gates, but it makes the next `CommutativeCancellation` sweep substantially more effective.

2.5 Cancellation PassManager

The custom PassManager `safe_pm` runs four cycles of

`InverseCancellation` → `CommutativeCancellation` → `Optimize1qGatesDecomposition`,

where the inverse-pair list covers $\{H, X, Y, Z, \mathrm{SWAP}, \mathrm{CX}, \mathrm{CZ}, \mathrm{CCX}\}$ as self-inverses and $\{(S, S^\dagger), (T, T^\dagger), (\mathrm{SX}, \mathrm{SX}^\dagger)\}$ as paired inverses. All three passes are unitary-preserving by construction (no resynthesis, no layout change), so no permutation-correction step is required afterwards.

2.6 Iteration to a fixed point

After the initial Toffoli collapse, the body runs both the O3-with-permutation-correction strategy and the `safe_pm` + `rewrite_1q` strategy on the current circuit and takes the lower-scoring outcome as the next iterate. Each round ends with a final `collapse_toffolis` pass that captures any Toffoli patterns newly exposed by the peephole. Iteration continues until the score stops decreasing; on this benchmark, the score converges in a single round and remains stable thereafter.

3 Results

Table 1 reports the score after each major stage of the pipeline.

Stage	Qubits	Depth	Gates	**Score**
Baseline (`benchmark_circuit.qasm`)	12	105	381	606
After `collapse_toffolis`	12	70	325	515
After O3 transpile + Clifford / peephole loop	12	46	219	**385**
Net reduction		−56%	−43%	−36%

Table 1: Score progression. The Toffoli collapse step alone removes 56 gates (four matches, fourteen scoreboard gates each), and the O3 + peephole loop removes another 106 gates and roughly halves the remaining depth.

The gate-type histogram, broken down by name, confirms the structural change between the two circuits. The baseline contains zero `ccx` gates (Toffolis are pre-decomposed), 110 `cx` gates, 23 `swap` gates, and a large population of named 1-qubit gates from the Toffoli ladders. The optimized circuit retains the four recovered `ccx` gates, 70 `cx` gates, and 17 `swap` gates (10 in the body plus 7 appended by the permutation-correction step). Named 1-qubit gates are largely absorbed into 32 generic single-qubit rotations (`u3`/`u2`/`u1`).

Gate	Baseline	Optimized
`cx`	110	70
`cz`	19	19
`ccx`	0	4
`swap`	23	17
`h`	37	5
`t`	28	6
`tdg`	24	5
`sdg`	16	8
`s`	11	3
`sx`	9	7
`x`	18	6
`y`	20	10
`z`	13	6
`rx`	16	5
`ry`	10	7
`rz`	27	9
`u3`	0	18
`u2`	0	13
`u1`	0	1
Total	**381**	**219**

Table 2: Gate-type histogram before and after optimization. The 1-qubit Clifford block contracts because runs of named gates collapse into a single generic `u3` or `u2`, while the four `ccx` gates re-emerge from the textbook ladder.

The final QASM file is `benchmark_iterall.qasm`, produced by `python3 iter_all.py`. Verification is via `python3 scoreboard.py submit benchmark_iterall.qasm`, which reports `accepted=True`, `score=385`, and a unitary residual of 1.91×10^{-16}.

4 Analysis

Why Toffoli matching is the single largest contribution. The scoreboard treats `ccx` as a single gate even though its physical implementation is roughly 15× larger. Each Toffoli match therefore delivers a one-time savings of 14 gates that no amount of subsequent peephole can replicate; the four matches in the baseline yield 56 gates of structural compression, more than a quarter of the total reduction observed.

Why `ccx` must remain in the transpiler basis. If `ccx` were omitted from `basis_gates`, Qiskit's `UnitarySynthesis` pass would unroll each `ccx` back into a sequence of CX and 1-qubit gates as part of the 3-qubit synthesis routine, immediately erasing the 56-gate gain from the previous step. Including

`ccx` signals the partitioner to leave these gates atomic, while still allowing all 2-qubit blocks (which are the structural majority of the circuit) to be re-synthesized at the KAK-optimal ≤ 3 `CX` cost.

Why permutation correction is essentially free. Allowing the elide-permutations pass to absorb terminal swaps lets the layout-aware portion of the transpiler treat swap connections as virtual, which opens up additional gate cancellation opportunities throughout the body. The cost of this trade-off is bounded by the structure of the absorbed permutation σ: a permutation on n qubits with c cycles requires $n - c$ transpositions to invert. For the final circuit the appended SWAP network has size 7, which is small compared to the body savings the elision enables.

Why Clifford renaming helps without removing gates. Qiskit's `CommutativeCancellation` pass matches against a static commutation table keyed on gate name; a generic `u(theta,phi,lambda)` whose matrix happens to equal H is opaque to the table, while a gate named `h` is not. Renaming therefore acts as a multiplier on the next peephole pass: the absorber consolidates 1-qubit chains, the renamer re-identifies named gates inside the consolidation, and the next commutative-cancellation sweep can then move those gates through 2-qubit gates whose commutation rules are defined per name.

Why atomic `ccx` interacts well with 2-qubit resynthesis. Keeping `ccx` atomic prevents the synthesis stack from re-touching what is already inside the Toffoli decomposition. The textbook construction is already near-optimal in T-count for an unconstrained 3-qubit Toffoli, and any local re-synthesis would risk producing a longer 1-qubit-and-CX sequence than the atomic representation costs on the scoreboard. The pipeline therefore makes a clean separation: 3-qubit structure is handled by pattern matching, while 2-qubit and 1-qubit structure is handled by the transpiler.

Termination. The pipeline is monotone in the (depth, gate-count) lexicographic order because every body step is unitary-preserving and the best-of-strategies selector never accepts a worse iterate. Because the state space (set of (depth, gates) pairs reachable from the initial circuit) is finite, the iteration must terminate. In practice, convergence occurs after one body iteration on this benchmark.

5 Reproduction

The submission package contains the following Python modules, the baseline `benchmark_circuit.qasm`, and the compiled `benchmark_iterall.qasm`:

- `iter_all.py` — entry point.
- `recognize_toffoli.py` — the textbook-Toffoli matcher.
- `clifford_rewrite.py` — Clifford rename pass and `safe_pm` PassManager.
- `sliced_optimize.py` — permutation detector and SWAP-network synthesizer (`fix_against_reference`).
- `scoreboard.py` — course-provided scorer and unitary verifier.

The Python dependencies are `qiskit==2.4.1`, `qiskit-aer`, and `numpy`. To reproduce the result:

```
python3 -m venv .venv
source .venv/bin/activate
pip install -r requirements.txt
python3 iter_all.py # writes benchmark_iterall.qasm
python3 scoreboard.py submit benchmark_iterall.qasm
```

The pipeline runs in well under a minute on a modern laptop; verification through Aer's 4096×4096 unitary contraction takes a few additional seconds.

References

[1] Qiskit Contributors. *Qiskit: An Open-Source Framework for Quantum Computing*, version 2.4.1, 2024. https://github.com/Qiskit/qiskit.

[2] M. A. Nielsen and I. L. Chuang. *Quantum Computation and Quantum Information*, 10th anniversary edition. Cambridge University Press, 2010. Toffoli decomposition: §4.3, Figure 4.9.

[3] D. Maslov. "Advantages of using relative-phase Toffoli gates with an application to multiple control Toffoli optimization," *Phys. Rev. A*, vol. 93, no. 2, p. 022311, Feb. 2016.

[4] M. Felemban. *QCompiler-Scoreboard*, KFUPM, 2026. https://github.com/mfelemban/QCompiler-scoreboard.