CS5863: Introuction to Program Analysis and Optimization

Conditioned Quantum Operations

Kartheek Tammana, Kushagra Gupta, Rishit D

Indian Institute of Technology, Hyderabad

Table of contents

Introduction

Introduction

- A flow of a generic quantum program has the following phases:
  1. Fetch quantum circuit & optimize
  2. Load circuit on device
  3. Fetch measurement results & create new state
  4. Update circuit & repeat
- Most research does not consider optimizations on classically-conditioned quantum operations.

Problem Statement

- We introduce the following three optimization passes with the following functionalities:
    1. Branch Merging & Simplification
    2. If-Else Splitting & Recombination via Branch Prediction
    3. Hoare Optimizations across Measurements
- We wish to investigate the viability of these above passes in terms of practicality and gate-counts.

If-Else Splitting & Recombination

The Setting

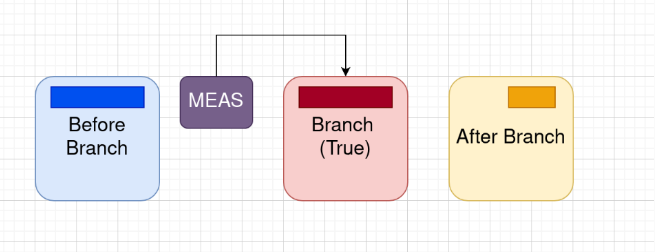


Figure 1: The General Setting

Split

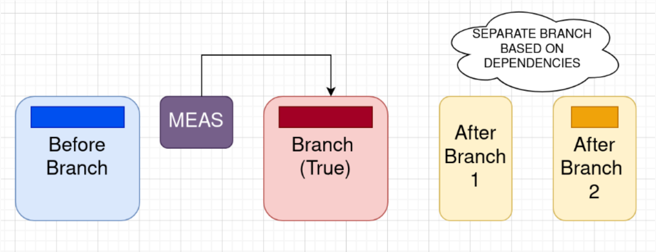


Figure 2: Splitting the Branches

Recombine

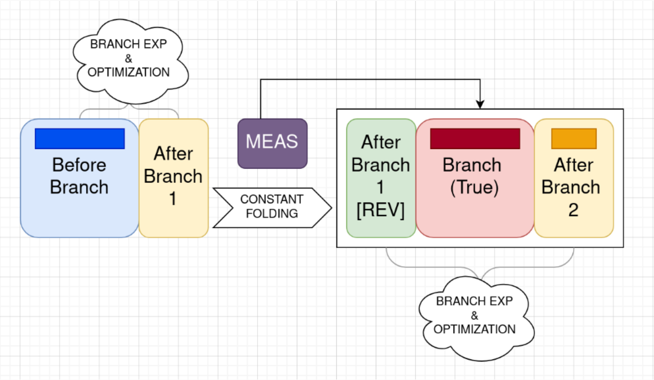


Figure 3: Recombining the Branches

The Issues

- Measurements are the only *non-commutative* operations in quantum circuits. Consequently, one cannot change the order of operations on measured quantum registers.
- Multiple branches are tougher to deal with as we are restricted to the non-dependent (wrt the measured registers) gates and registers. (Although this can be addressed to some extent later)
- Certain cases are easily dealt with in a nested fashion, while some are better flattened out (and some cannot be optimized at all).
- The extent of optimization is also dependent on the circuit area, level of nesting and window-depth.

A Toy Example

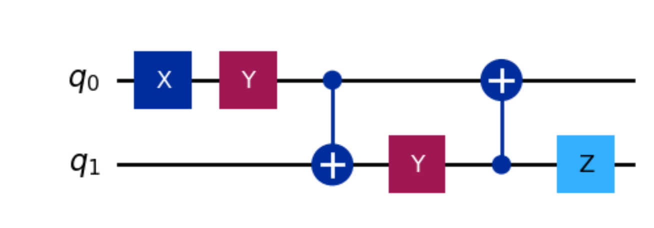


Figure 4: A Toy Condition Subcircuit

The Toy Example

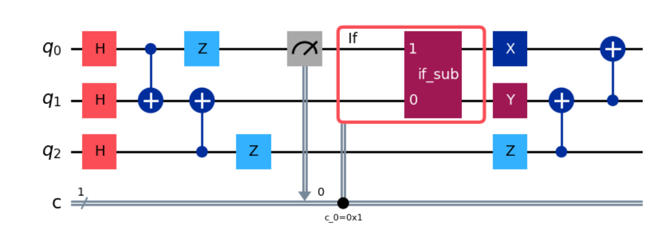


Figure 5: Without Optimization

Analysis

- **Gate Count:** RZ:7, SX:4, CX:3, X:2, IF_ELSE:1
- **Depth:** 10

The Toy Example

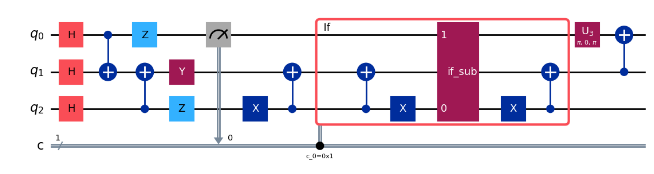


Figure 6: With Optimization

Analysis

- **Gate Count:** RZ:7, SX:3, CX:2, X:1, IF_ELSE:1
- **Depth:** 7

PseudoCode

```
INPUT: [circuit = {before, measure, branch, after}], THRESHOLD

// Circuit Mode
// Temporary Circuits
circuit_duplicate = circuit.insert(barrier, before)
circuit_direct = circuit_duplicate.remove(measure, branch)

// Convert to DAG
circuit -> dag
circuit_duplicate -> dag_duplicate
circuit_direct -> dag_direct

// DAG Mode
[dag = {before, measure, branch, after}]
```

Figure 7: Alternate Circuit Generation

PseudoCode

```
// DAG Mode
[dag = {before, measure, branch, after}]

// Find edges from measure to barrier
dependencies = []
for each edge in dag_duplicate[measure]:
    if edge.target == barrier:
        dependencies.append(edge.register)

// Find nodes in dag_direct which are in dependencies (after barrier)
nodes = []
// Start bfs/dfs from barrier
start = barrier
worklist = [(start, 0)]
while worklist:
    for each edge in out_edges(worklist.top):
        if edge.register in dependencies:
            nodes.append(edge.source, edge.target)
            dag_direct.remove(edge.source, edge.target)
        else if worklist.top[1] > THRESHOLD:
            nodes.append(edge.source, edge.target)
            dag_direct.remove(edge.source, edge.target)
        else:
            worklist.append((edge.target, worklist.top[1] + 1))
// Remove nodes with 0 in-degree
for each node in dag_direct:
    if in_deg(node) == 0:
        dag_direct.remove(node)
        nodes.append(node)
```

PseudoCode

```
// Dag after
dag_after = dag.copy(dag_direct)
dag_after.delete(barrier.ancestors)
dag_after.delete(barrier)

// Dag before
dag_before = dag.copy(dag_direct)
dag_before.delete(barrier.descendants)
dag_before.delete(barrier)

// Dag after-after
dag_after_after = new dag(nodes)
for each edge in dag_after:
    if edge.source and edge.target in nodes:
        dag_after_after.add(edge.source, edge.target)
dag_after_after.delete(barrier)
```

Figure 9: Convert DAGs to Sub-Circuits

PseudoCode

```
// Back to Circuit
dag_after -> after
dag_before -> before
dag_after_after -> after_after
branch = circuit[branch]

// New Circuit
final_circuit = circuit()
final_circuit.add(optimize(before, after))
final_circuit.add(measure)
final_circuit.add(make_branch(optimize(after.inv, branch)))
final_circuit.add(after_after)

return final_circuit
```

Figure 10: Finalize the Circuit

Hoare Optimizations++

Hoare Optimizations++

- Hoare Logic is a formal system using preconditions and postconditions to specify the behavior of a program.
- We can use Hoare Logic to optimize quantum circuits by ensuring that the optimizations do not change the correctness of the program.
- The following Hoare logic used could be state-dependent or state-independent.

Example

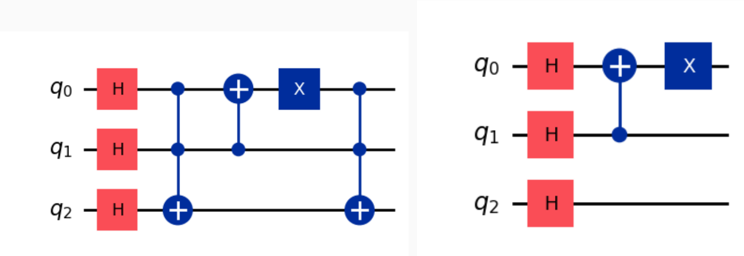


Figure 11: Hoare Logic Example

Works Across Measurements?

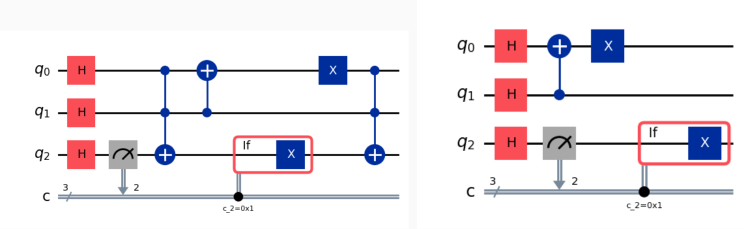
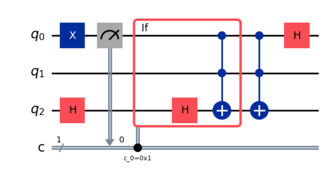


Figure 12: Across Measurements

Analysis
Original Circuit Gate Count: 'rz': 24, 'sx': 11, 'cx': 11, 'measure': 1, 'if_else': 1)
Optimized Circuit Gate Count: 'rz': 6, 'sx': 3, 'cx': 1, 'x': 1, 'measure': 1, 'if_else': 1)

On Measurements



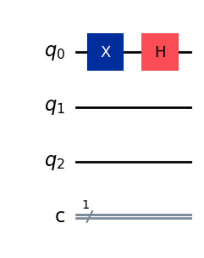
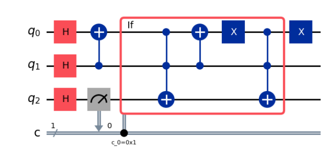


Figure 13: On Measurements

Analysis
Original Circuit Gate Count: 'rz': 13, 'cx': 6, 'sx': 4, 'x': 1, 'measure': 1, 'if_else': 1

Optimized Circuit Gate Count: 'rz': 2, 'sx': 1

Value Independent



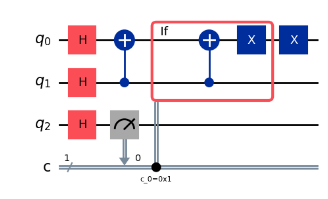
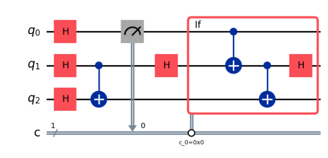


Figure 14: Value Independent

Analysis
Original Circuit Gate Count: 'rz': 6, 'sx': 3, 'cx': 1, 'measure': 1, 'if_else': 1, 'x': 1
Optimized Circuit Gate Count: 'rz': 6, 'sx': 3, 'measure': 1, 'if_else': 1

Value Dependent



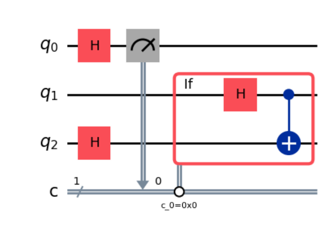


Figure 15: Value Dependent

Analysis
Original Circuit Gate Count: 'rz': 7, 'sx': 4, 'cx': 1, 'measure': 1, 'if_else': 1

Optimized Circuit Gate Count: 'rz': 4, 'sx': 2, 'measure': 1, 'if_else': 1

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.
Quantum computing with Qiskit, 2024.

The CUDA-Q development team.
CUDA-Q.

Yanbin Chen, Innocenzo Fulginiti, and Christian B. Mendl.
Reducing mid-circuit measurements via probabilistic circuits.
In *2024 IEEE International Conference on Quantum Computing and Engineering (QCE)*, page 952–958. IEEE, September 2024.

References ii

Thomas Häner, Torsten Hoefler, and Matthias Troyer.
Assertion-based optimization of quantum programs.
Proceedings of the ACM on Programming Languages, 4(OOPSLA):1–20, November 2020.

Yanbin Chen.
Unleashing optimizations in dynamic circuits through branch expansion, 2025.