CPEN 400Q / EECE 571Q Lecture 09
Overview of quantum compilation, and quantum transforms

Tuesday 8 February 2022

Announcements

- Assignment 2 due Monday 14 Feb 23:59
  - Issue with problem 4: do not worry if last test function doesn't pass; will be checked manually.
- Project details available

Quiz 4 at the end of class today.

Project

Implement the methods of a recent research paper, reproduce the results as closely as possible, share the results with your classmates.

Three equally-weighted parts:

- Software implementation (PennyLane or other framework)
- Presentation with live demo (30+5 minutes; online)
- Companion report

Work in groups of 4.

Project: important dates

- **2022-02-15**: Group and topic selection due.
- **2022-03-08**: Prototype implementation due. (10-15 minute meeting; not graded).
- **2022-03-29**: Presentations start.
- **2022-04-08**: Final report and software implementation due.

Grading rubric, and suggested papers available in repo:

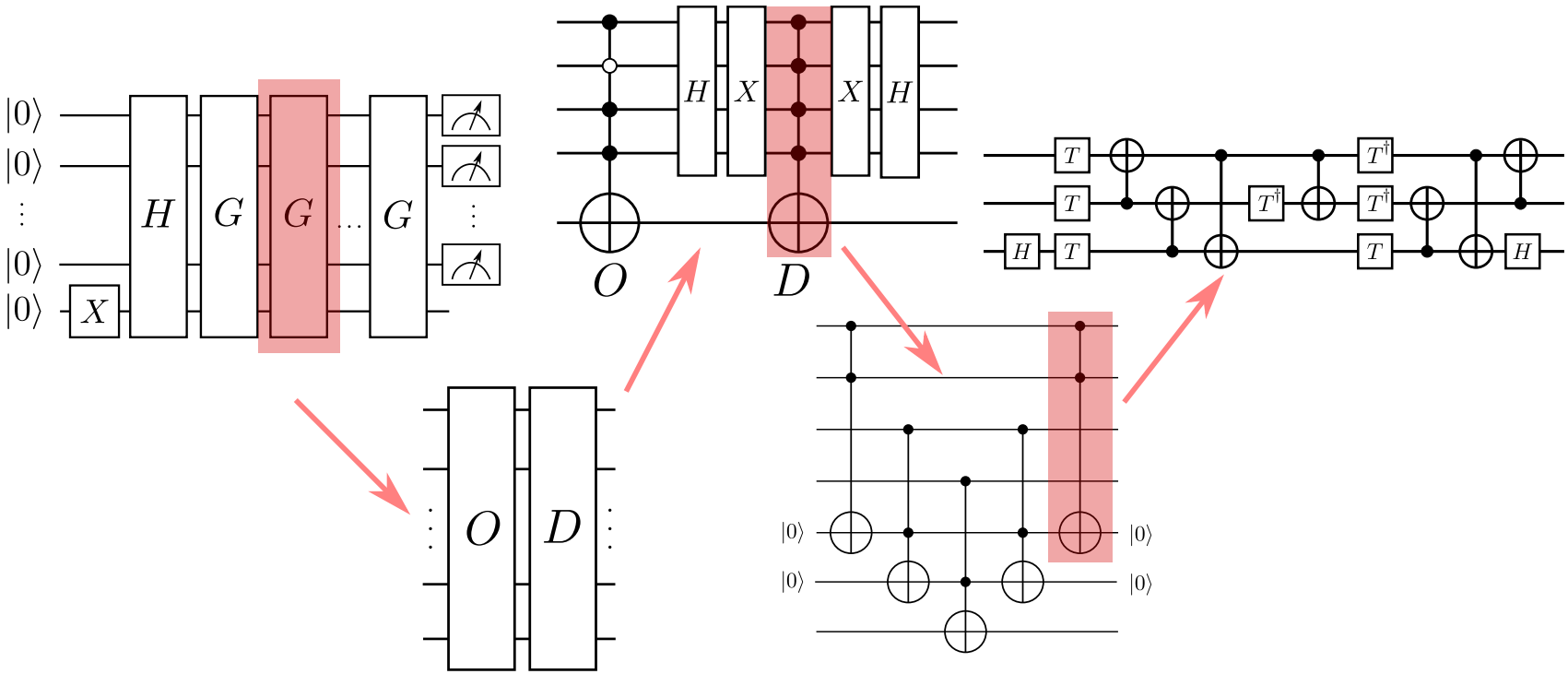
`https://learning.github.ubc.ca/CPEN-400Q-202-2021W2/Final-Project`

Learning outcomes

- List and describe the stages of the quantum compilation process
- Manipulate PennyLane circuits with quantum transforms

Last time

We reviewed Grover's algorithm, and used it to discuss the different ways of estimating quantum resources (queries vs. depth vs. gates, etc.)



I bungled an example with `qml.specs`... let's redo it.

Quantum compilation

Compilation

Compilers turn human-readable code into the executable machine language a computer can understand.

Consider the following 5 lines of C code:

```c
#include<stdio.h>

void main() {
    printf("Hello, world!");
}
```

Compilation

Compiler compiles the code and turns it into *assembly language*.

- Preprocessing
- Lexing
- Parsing
- Semantic analysis
- General optimization
- Hardware-specific optimization

```
        .file   "hello_world.c"
        .text
        .section        .rodata
.LC0:
        .string "Hello, world!"
        .text
        .globl  main
        .type   main, @function
main:
.LFB0:
        .cfi_startproc
        endbr64
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        leaq    .LC0(%rip), %rdi
        movl    $0, %eax
        call    printf@PLT
        nop
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
.LFE0:
        .size   main, .-main
        .ident  "GCC: (Ubuntu 9.2.1-9ubuntu2) 9
```

Still human-readable (sort of). Went from 5 lines to 45 lines.

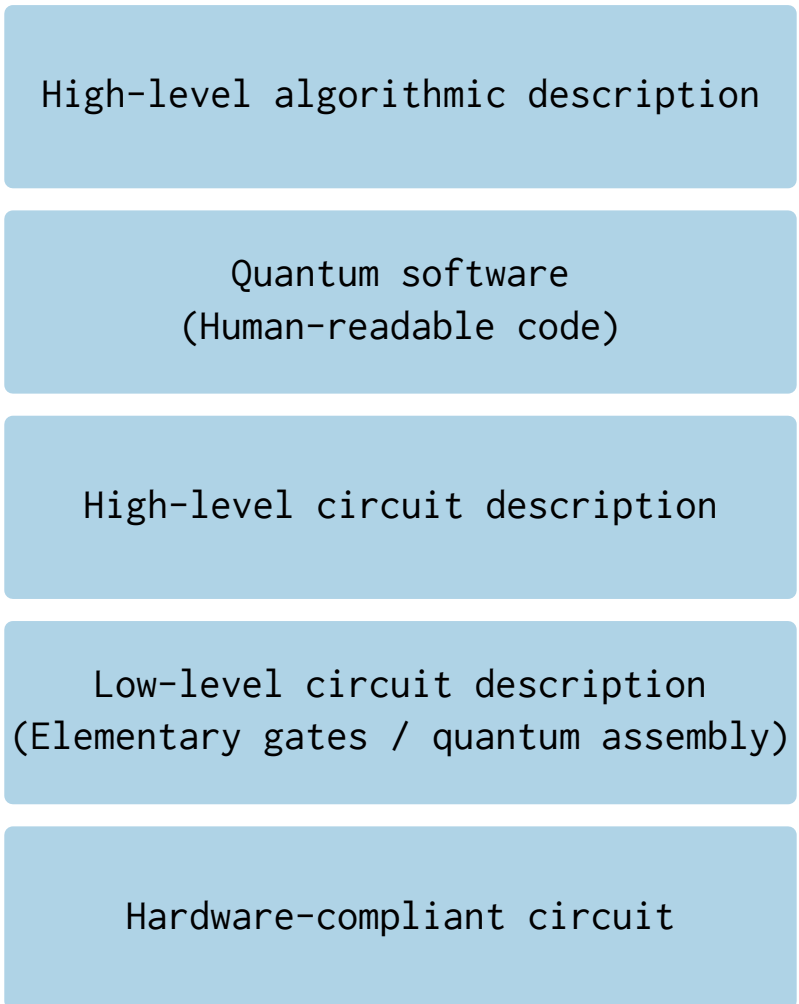
Compilation

Assembler turns this into something machine executable.

176 lines of this... *not something we want to do by hand*.

Quantum compilation

Quantum computers will also need compilation tools to turn human-readable algorithms and code to quantum operations.



Quantum compilation

High-level algorithmic description

Add bits a and b with a half-adder

Quantum software
(Human-readable code)

High-level circuit description

Low-level circuit description
(Elementary gates / quantum assembly)

Hardware-compliant circuit

Quantum compilation

High-level algorithmic description

Quantum software
(Human-readable code)

quantum_half_adder(a, b)

High-level circuit description

Low-level circuit description
(Elementary gates / quantum assembly)

Hardware-compliant circuit

Quantum compilation

High-level algorithmic description

Quantum software
(Human-readable code)

High-level circuit description

Low-level circuit description
(Elementary gates / quantum assembly)

Hardware-compliant circuit

```
quantum_half_adder(a, b)
```

Available quantum computing software frameworks allow users to work mostly at this level.

Quantum compilation

High-level algorithmic description

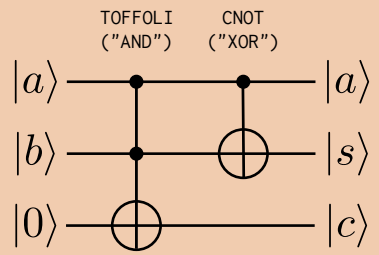
Quantum software
(Human-readable code)

High-level circuit description

Low-level circuit description
(Elementary gates / quantum assembly)

Hardware-compliant circuit

```
def quantum_half_adder(a, b):
    qubit register q[3]
    prepare_bit(a, q[0])
    prepare_bit(b, q[1])
    toffoli(q[0], q[1], q[2])
    cnot(q[0], q[1])
```



Quantum compilation

High-level algorithmic description

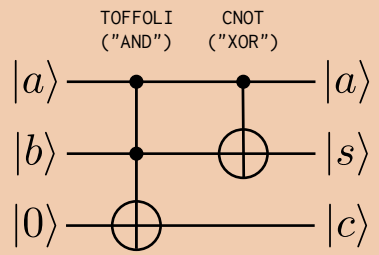
Quantum software
(Human-readable code)

High-level circuit description

Low-level circuit description
(Elementary gates / quantum assembly)

Hardware-compliant circuit

```
def quantum_half_adder(a, b):
    qubit register q[3]
    prepare_bit(a, q[0])
    prepare_bit(b, q[1])
    toffoli(q[0], q[1], q[2])
    cnot(q[0], q[1])
```



If you're developing quantum algorithms, you're probably working at this level.

Quantum compilation

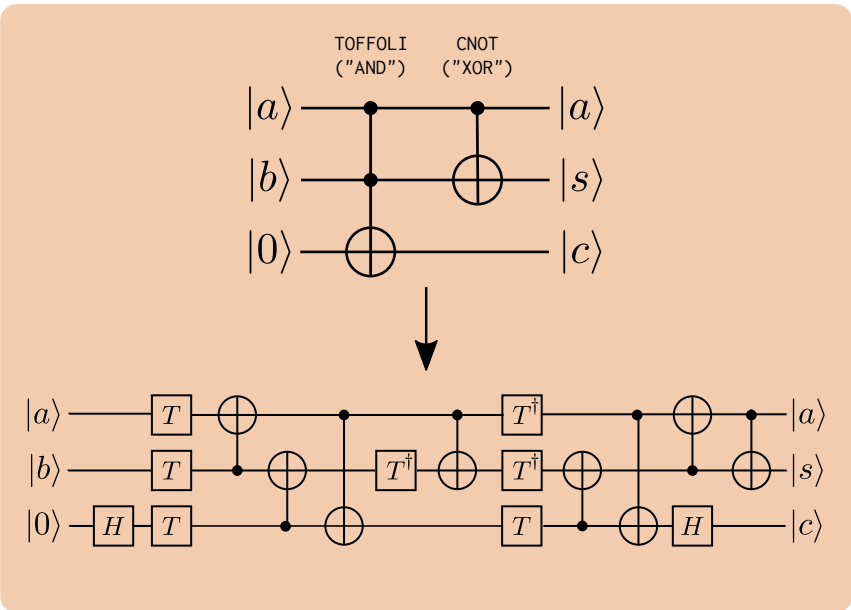
High-level algorithmic description

Quantum software
(Human-readable code)

High-level circuit description

Low-level circuit description
(Elementary gates / quantum assembly)

Hardware-compliant circuit



Quantum compilation

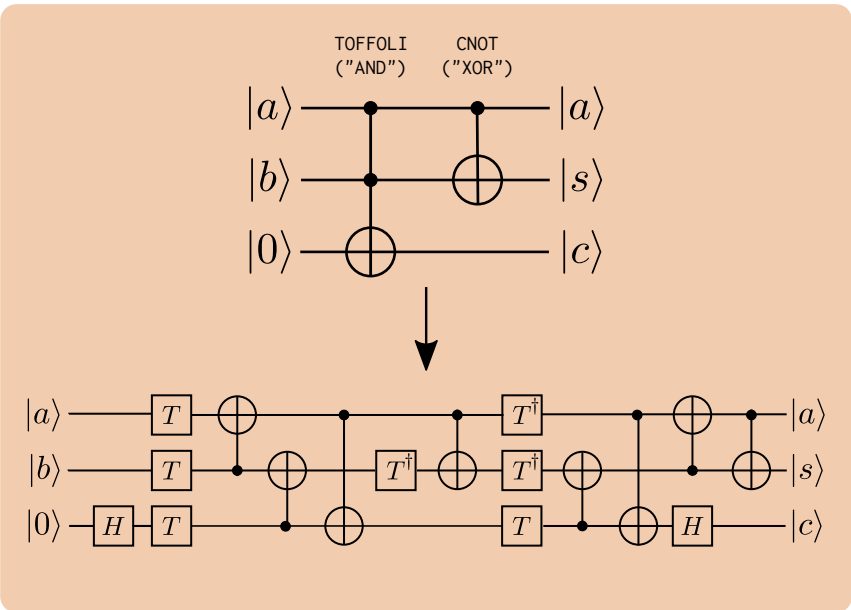
High-level algorithmic description

Quantum software
(Human-readable code)

High-level circuit description

Low-level circuit description
(Elementary gates / quantum assembly)

Hardware-compliant circuit



We should automate this.

Quantum compilation

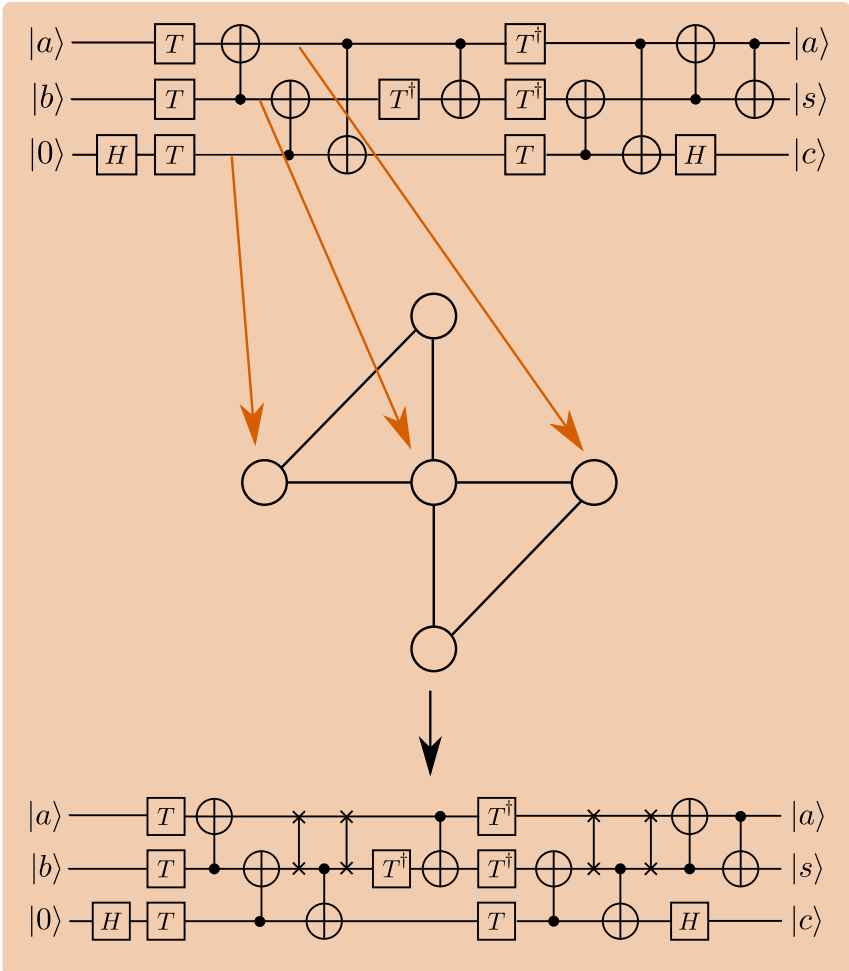
High-level algorithmic description

Quantum software
(Human-readable code)

High-level circuit description

Low-level circuit description
(Elementary gates / quantum assembly)

Hardware-compliant circuit



Quantum compilation

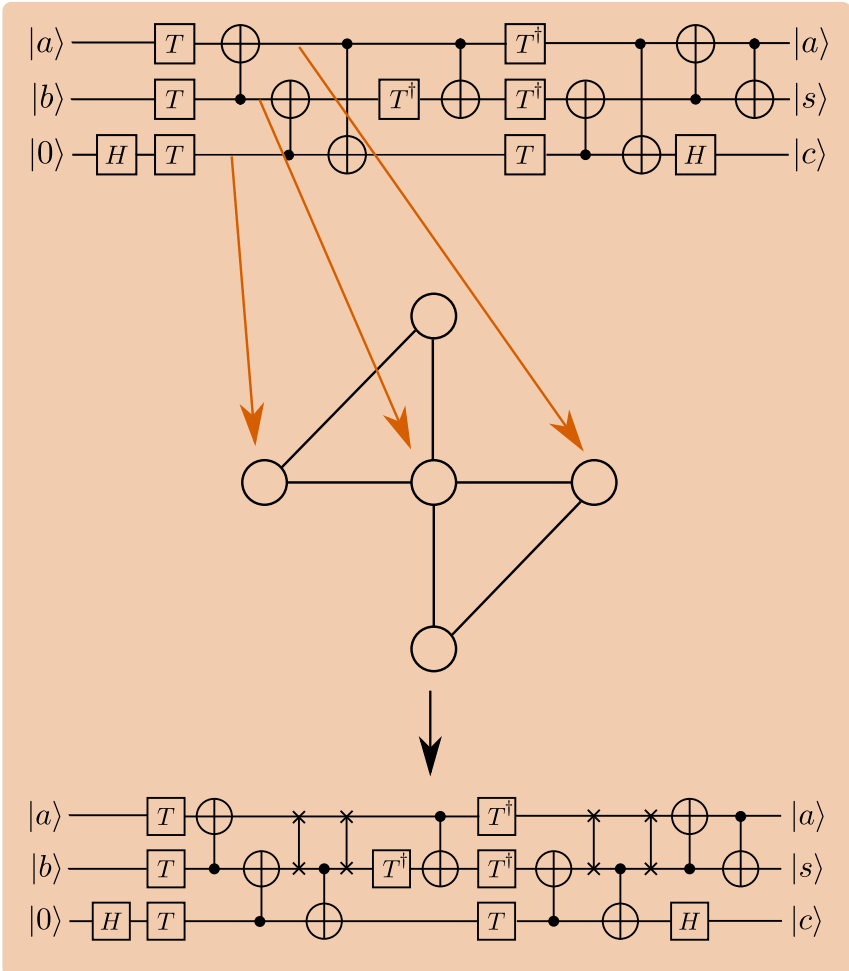
High-level algorithmic description

Quantum software
(Human-readable code)

High-level circuit description

Low-level circuit description
(Elementary gates / quantum assembly)

Hardware-compliant circuit



We should **really** automate this.

Quantum compilation

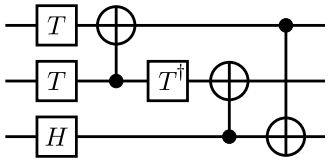
High-level algorithmic description

Quantum software
(Human-readable code)

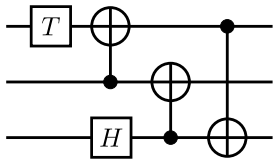
High-level circuit description

Low-level circuit description
(Elementary gates / quantum assembly)

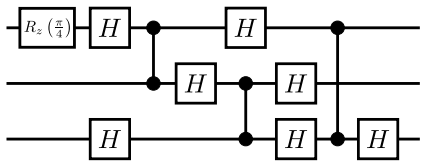
Hardware-compliant circuit



Synthesis

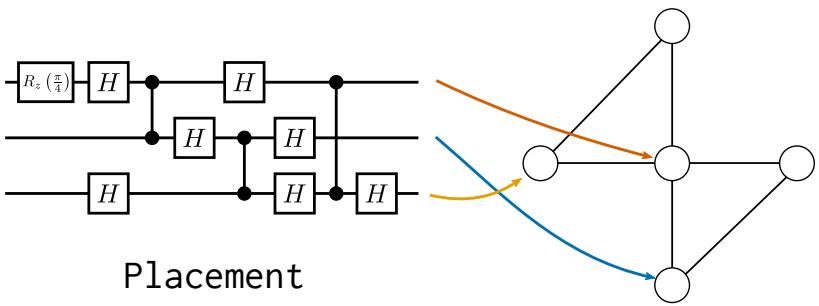


Optimization

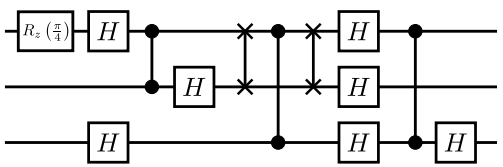


Transpilation

Verification



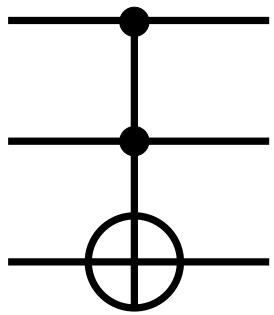
Placement



Routing

Quantum circuit synthesis

Consider a 3-qubit gate such as the Toffoli:



```
qml.Toffoli(wires=[0, 1, 2])
```

$$\text{TOF} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

We understand this and we use it in circuits - what if a quantum computer cannot implement it as-is?

Quantum circuit synthesis

Fact 1: Quantum operations are *unitary* ($UU^\dagger = \mathbb{1}$).
Fact 2: The unitary group is infinite.

Consequence: It is not feasible to build a gate-based qubit quantum computer and specify how to perform every arbitrary n-qubit unitary (and perform it well).

Solution: Figure out how to implement a small, finite set of operations well, and express everything else in terms of them.

Quantum circuit synthesis

Let $\mathcal{G} = \{G_k\}$ be a finite set of gates on one or more qubits.

The task of **quantum circuit synthesis** is, given an arbitrary unitary operation U, find a decomposition

$$\begin{aligned} U &= C_1 C_2 \cdots C_m \qquad \text{(exact synthesis), or} \\ U &\approx C_1 C_2 \cdots C_m \qquad \text{(approximate synthesis),} \end{aligned}$$

where all C_i are constructed solely from elements of $\mathcal{G}$.

If $\mathcal{G}$ is a **universal gate set**, we can do this for *any* U, up to arbitrary precision of approximation.

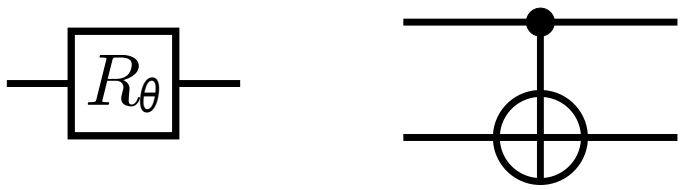
Universal gate sets - 1 qubit

Single-qubit rotations

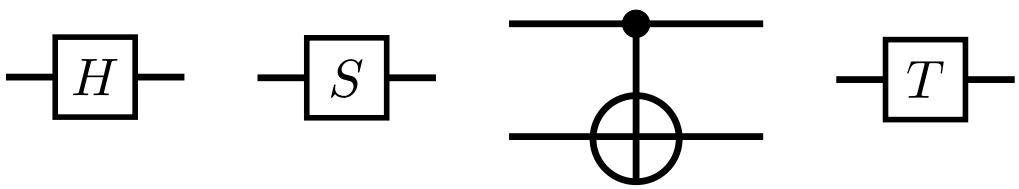
Hadamard and T

Universal gate sets - 2 qubits

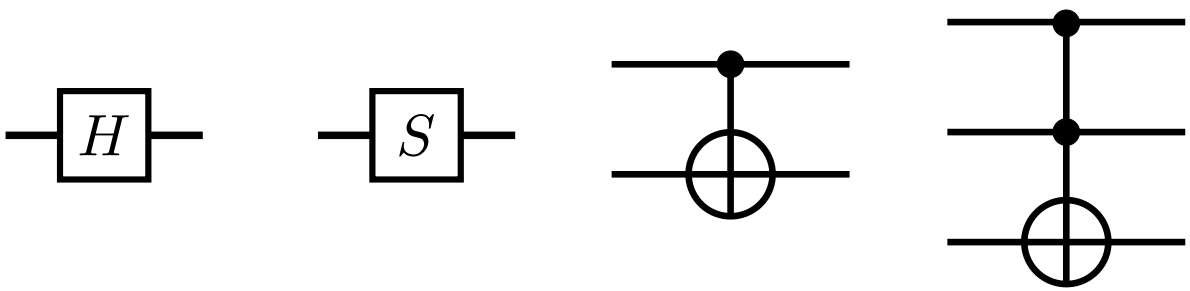
Single-qubit rotations + CNOT



Clifford + T



Clifford + Toffoli



Approximate circuit synthesis

Given gave set $\mathcal{G}$, target U and arbitrary precision ϵ, find

$$U' = C_1 C_2 \cdots C_m \quad \text{s.t.} \quad ||U' - U|| < \epsilon, \quad C_i \in \mathcal{G}$$

Difficulty depends on number of qubits, and your gate set.

Single-qubit case

Gate set general $R(\theta)$: you're all set.

Gate set $\{R_y(\theta), R_z(\theta)\}$: $U = R_z(\alpha)R_y(\beta)R_z(\gamma)$

Gate set $\{H, T\}$: specialized algorithms (Solovay-Kitaev, number-theoretic approaches). Best approaches use $O(\log^c(1/\epsilon))$ gates, where c ranges from 1 to ≈ 4 depending on how many auxiliary qubits are available.

Multi-qubit case

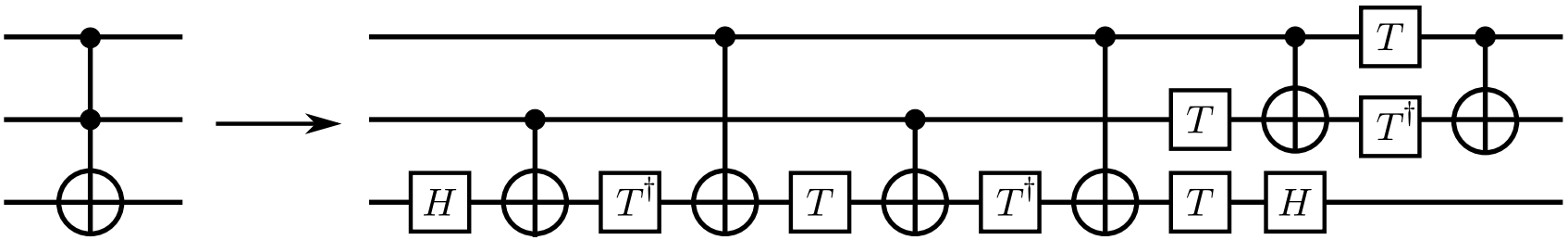
Much harder than single-qubit case...

Gate set $\{R(\theta), CNOT\}$: $O(n^2 4^n)$ 1- and 2-qubit gates. "Unravel" the unitary into $O(4^n)$ 2-level operations, then implement those with $O(n^2)$ gates each.

Gate set Clifford + T: Use number theoretic methods to "round off" to a unitary that can be synthesized exactly, then do so. Requires $O(4^n n \log(1/\epsilon) + n)$ gates.

Exact circuit synthesis

Sometimes structure of an operation allows for *exact* synthesis over Clifford + T.

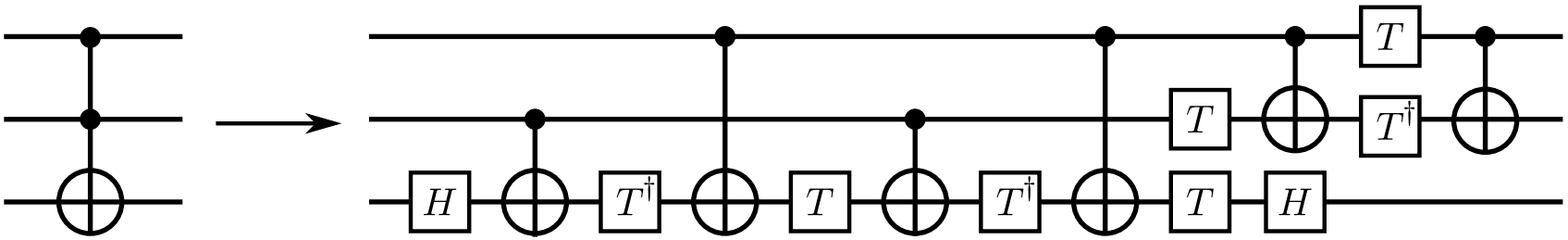


For single qubits, it's known how to do this optimally.

For multiple qubits, often done using search, but the runtimes depend exponentially on the number of qubits and circuit depth.

Making things better: transpilation and optimization

Recall our implementation of the Toffoli:



What if...

- our hardware implements CZs instead of CNOTs?
- we want to optimize for, e.g., depth, or T-count / T-depth?
- we synthesize many Toffolis within a circuit and then put them together?

Making things better: transpilation and optimization

After synthesis, we have the opportunity to transpile (re-express gates in terms of other gates) and optimize a circuit to improve it.

This may include:

- Unrolling multi-qubit gates with known decompositions into 1- and 2-qubit gates
- Remove redundant gates / "gate fusion"
- Transformation rules / canonical forms (shuffle around commuting gates)
- Rewriting rules / template matching
- Parallelization by leveraging auxiliary qubits

The transpilation and optimization process typically includes multiple passes to cover all these points.

Method: T gate parallelization with auxiliary qubits

If you are rich in qubits, some circuit families (including the Toffoli) can be reduced to T-depth 1 with enough auxiliary qubits.

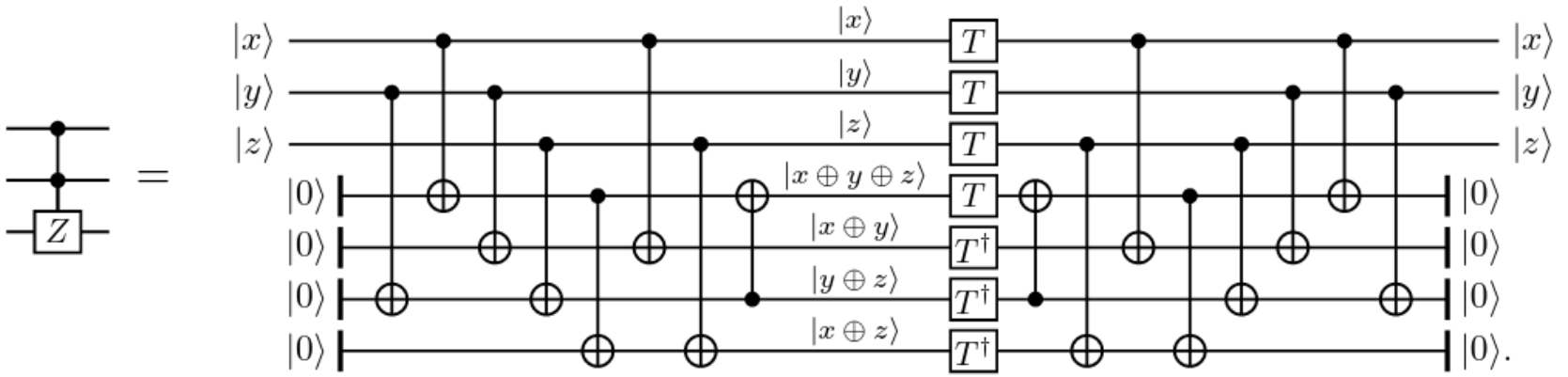


Figure 1: T-depth 1 representation of the Toffoli gate

Many space-time tradeoffs can be made in this way.

P. Selinger. Quantum circuits of T-depth one. Phys. Rev. A 87, 042302 (2013)

Method: template matching

Recognizing patterns in a circuit can lead to savings.

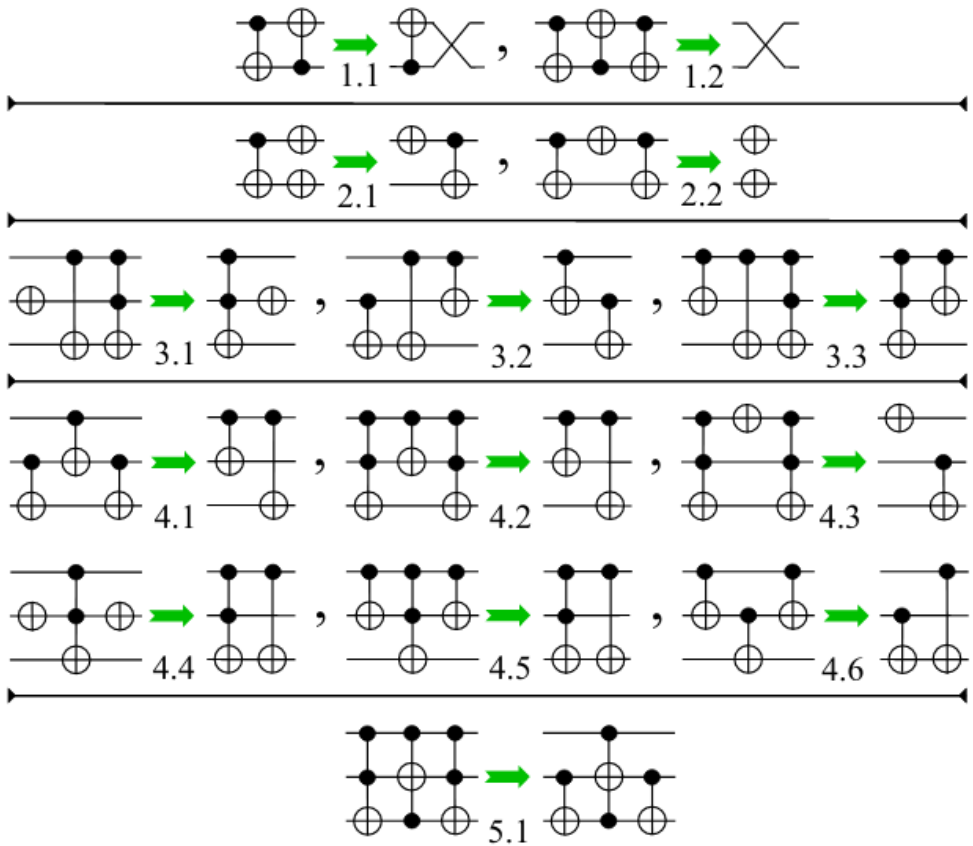


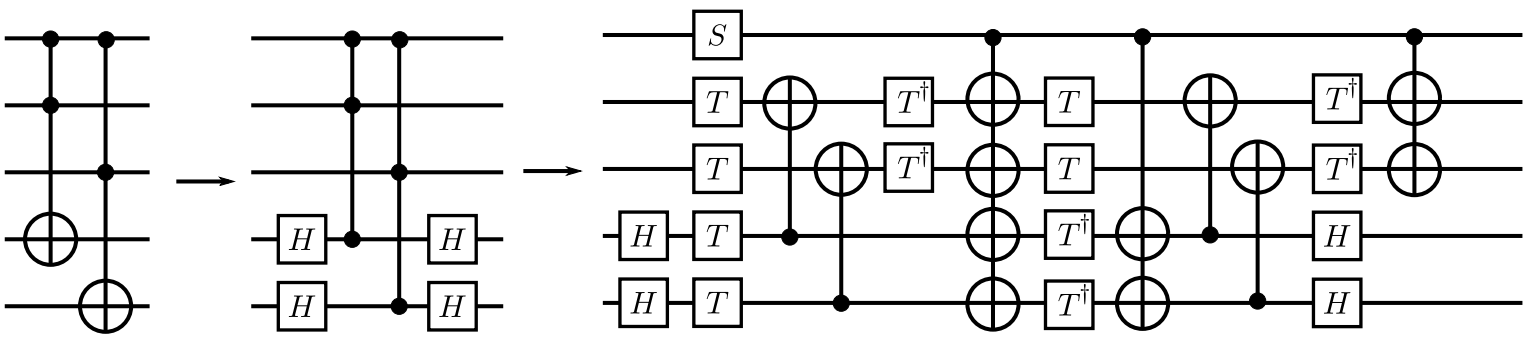
Figure 4: Templates with 2 or 3 inputs.

Image: D. M. Miller, D. Maslov, G. W. Dueck. A Transformation Based Algorithm for Reversible Logic Synthesis. DAC 2003.

Example: optimizing shared-control Toffolis

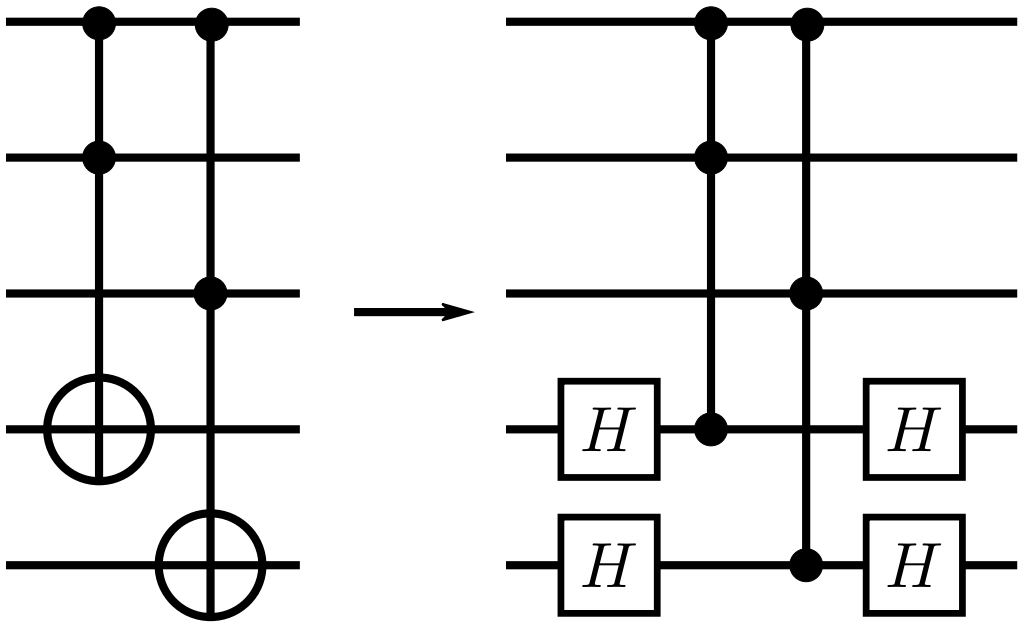
Through transpilation and optimization we can improve the depth, T-depth, and T-count of a pair of Toffolis if they share a control.

```
@qml.qnode(dev)
def shared_controls():
    qml.Toffoli(wires=[0, 1, 3])
    qml.Toffoli(wires=[0, 2, 4])
```



Example: optimizing shared-control Toffolis

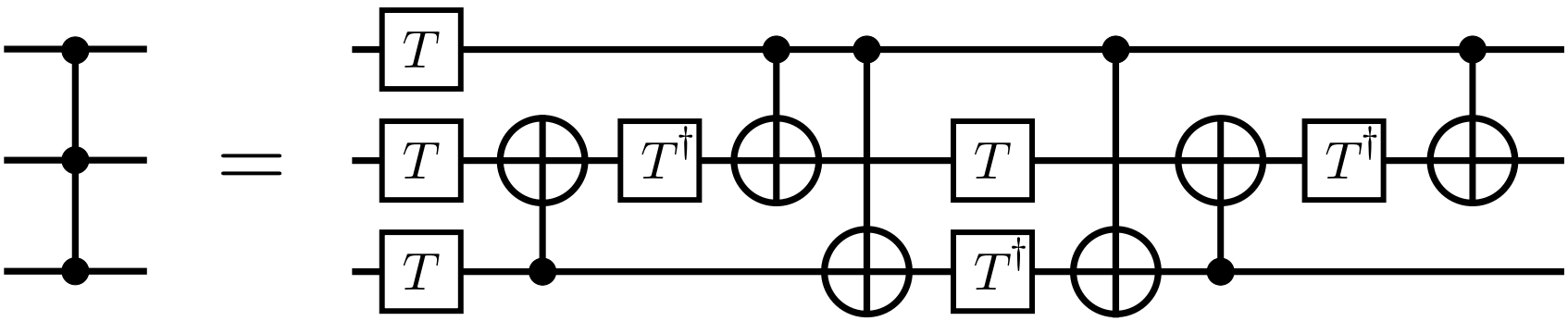
Transpile the circuit by replacing Toffolis with CCZs.



We do this because in a CCZ, the control and target qubits are interchangeable.

Example: optimizing shared-control Toffolis

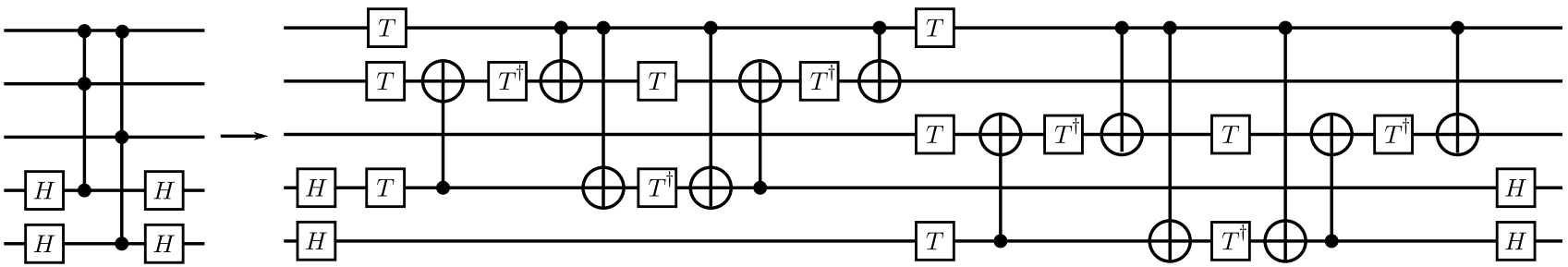
Transpile the CCZs over Clifford + T. Use yet another decomposition of a CCZ/Toffoli.



(This decomposition has T-count 7, T-depth 4, and depth 10.)

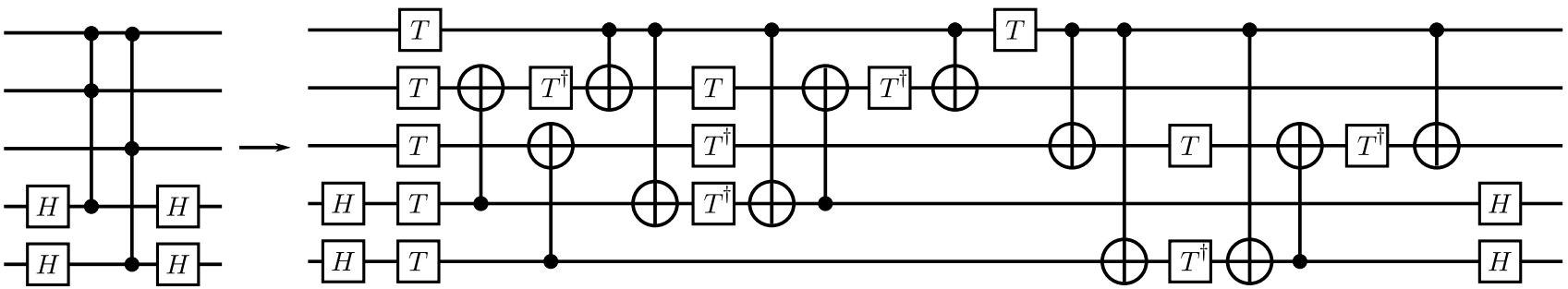
Example: optimizing shared-control Toffolis

Unroll the CCZs:



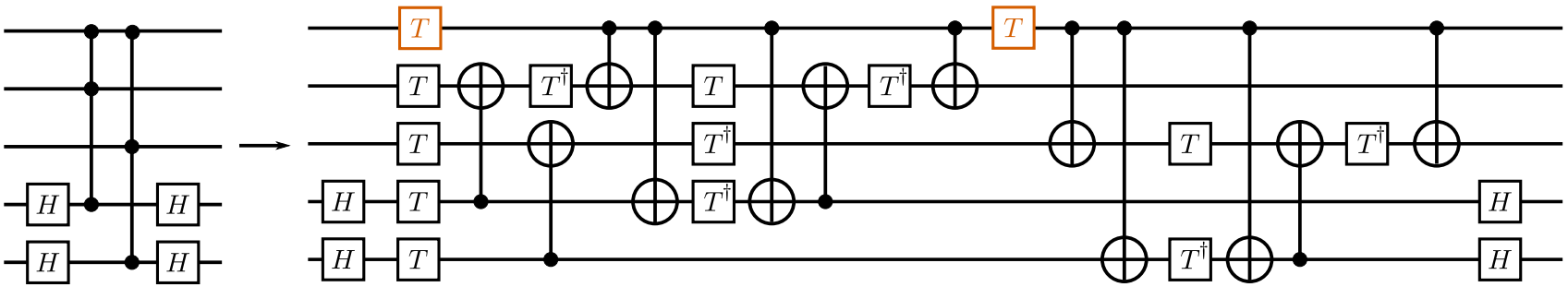
Example: optimizing shared-control Toffolis

Start pushing through obviously commuting gates.



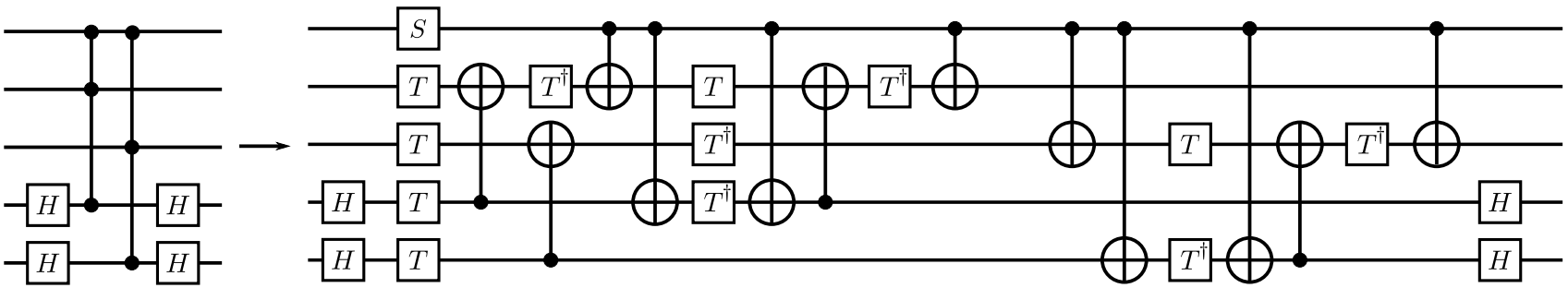
Example: optimizing shared-control Toffolis

Push the T through...



Example: optimizing shared-control Toffolis

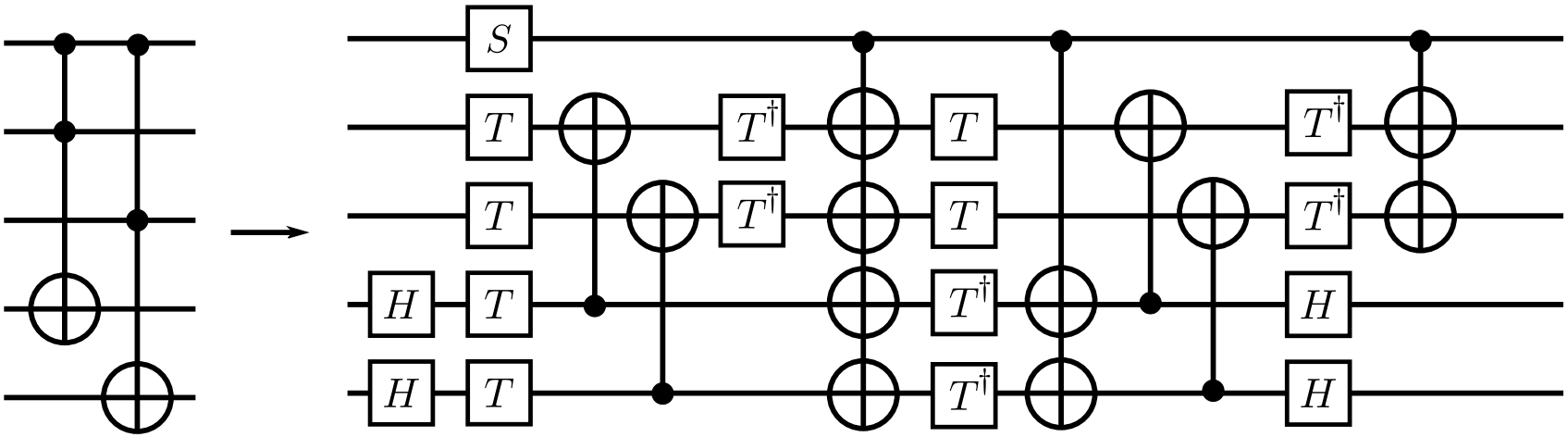
... and simplify!



Let's see this in action.

Example: optimizing shared-control Toffolis

Final version:

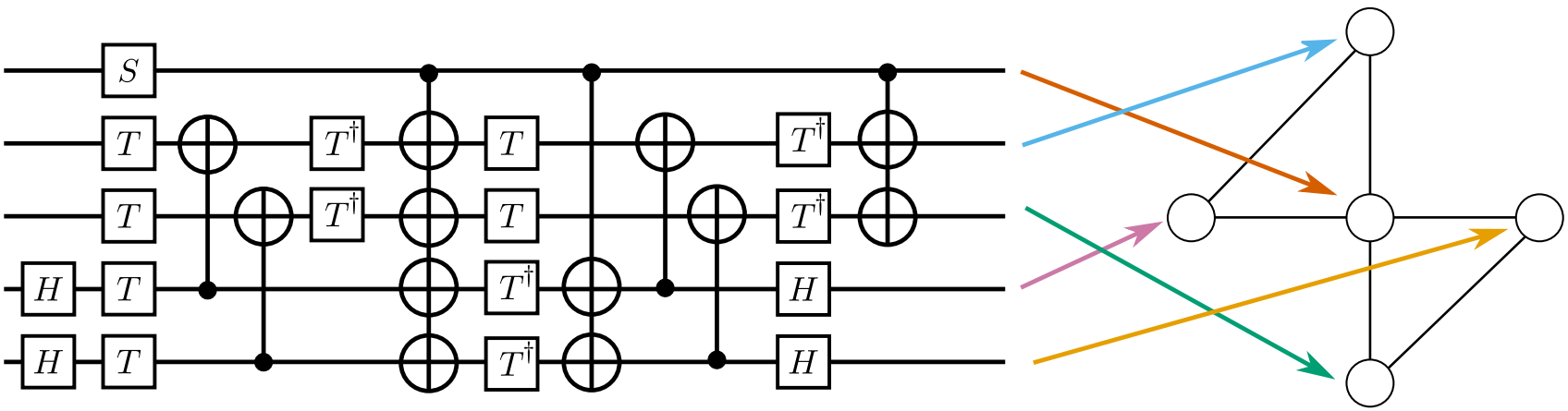


	Original	Optimized
T-count	14	12
T-depth	8	4
Depth	22	15

Obviously we do not want to do this by hand ever again.

Qubit allocation

Initial qubit placement, or **qubit allocation**, is the process of assigning logical qubits in a circuit to physical qubits on hardware.



Qubit allocation is proven to be **NP-complete** [1]

[1] Siraichi, dos Santos, Collange, Pereira. *Qubit Allocation*. Proc. CGO 2018.

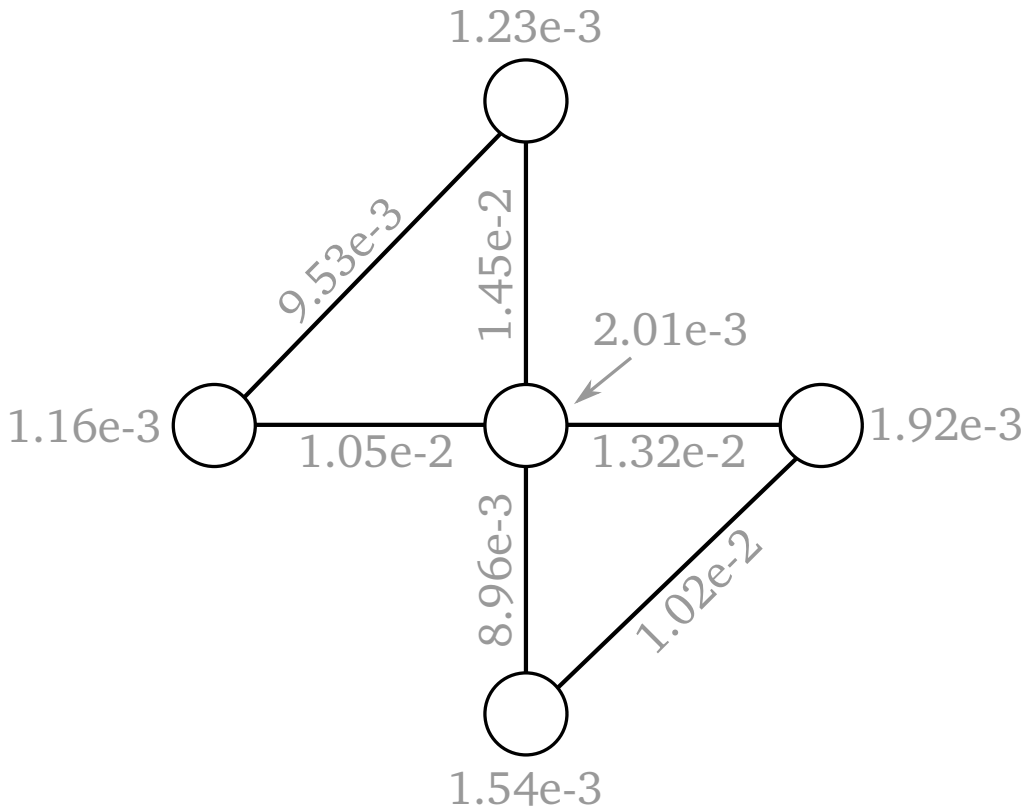
Qubit allocation

Many aspects to consider

- Qubit connectivity
- Gate error rates
- Gate operation times
- Coherence times

Many metrics of solution quality

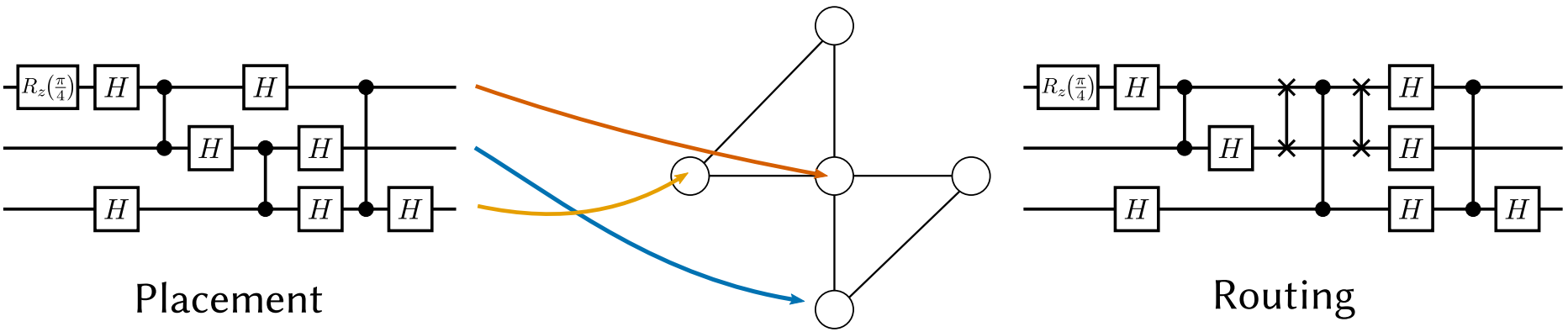
- Number of added SWAPs
- Circuit depth
- Success probability of circuit
- Total execution time



(Random values)

Qubit placement/routing approach

Shuffle qubits around in order to satisfy any connectivity constraints. Very intertwined with placement.



Earlier algorithms considered only the connectivity graph and tried to minimize the number of added SWAPs; more recent techniques incorporate error rates.

Approach through heuristic search

Partition circuit into layers; find optimal placement in each layer; knit them together using SWAPs.

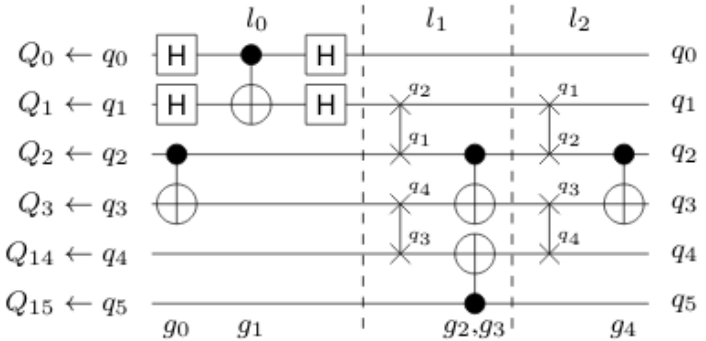


Fig. 6: Circuit resulting from locally optimal mappings

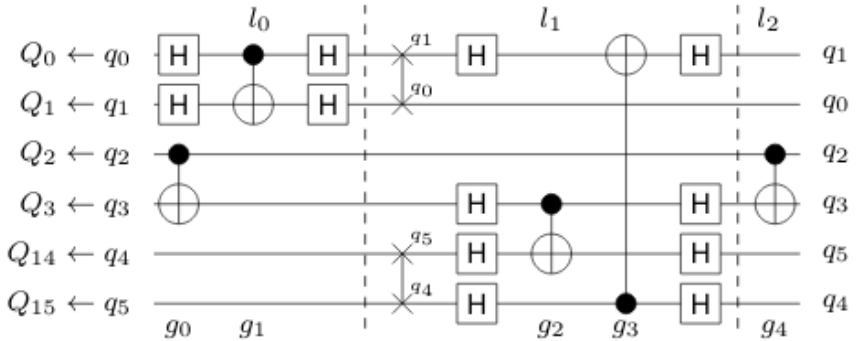


Fig. 7: Circuit generated when using the look-ahead scheme

Using a look-ahead approach improved this significantly.

Image: A. Zulehner, A. Paler, R. Wille. An efficient methodology for mapping quantum circuits to the IBM QX architectures. arXiv 1712.04722.

Combined placement/routing: SABRE

SwAp-based BidiREctional heuristic search algorithm
(Not error-aware)

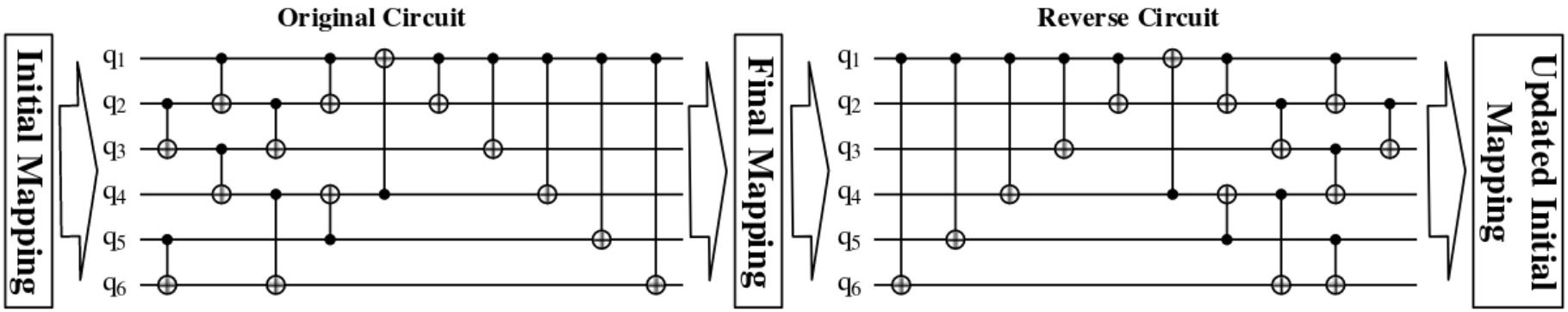


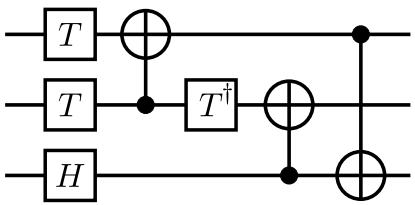
Fig. 5: Initial Mapping Update Using Reverse Traversal Technique

Starts with random initial placement; runs through circuit and uses heuristic techniques to insert SWAPs and permute things along the way. Then run through it backwards to get out a new placement.

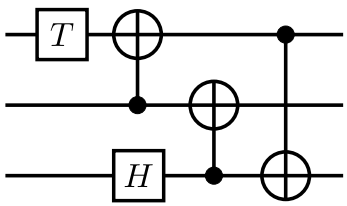
Image: Li, G., Ding, Y., Xie, Y., Tackling the qubit mapping problem for NISQ-era quantum devices, (2019).

Quantum compilation

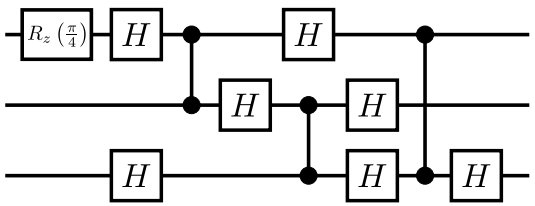
Much of compilation is (computationally) *hard*; therefore we need good-quality heuristics and good-quality automated tools.



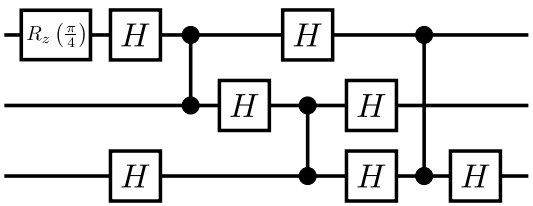
Synthesis



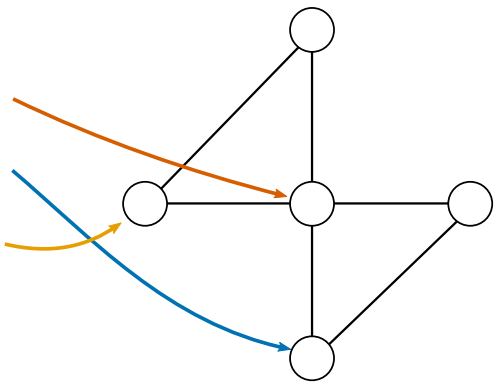
Optimization

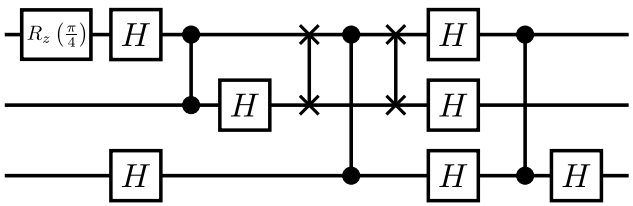


Transpilation



Placement





Routing

Quantum transforms

Quantum transforms

PennyLane's `qml.transforms` module contains a library of transforms. We've already seen some:

- `qml.draw` takes a QNode as input and returns a function that draws it
- `qml.specs` takes a QNode as input and returns a function that computes relevant information about it
- `qml.adjoint` takes a quantum function and returns a function that implements the adjoint
- `qml.ctrl` takes a quantum function and returns a function that implements the controlled version of the function

Quantum transforms

Compilation is implemented using quantum transforms:

Transforms for circuit compilation

This set of transforms accept quantum functions, and perform basic circuit compilation tasks.

`compile`	Compile a circuit by applying a series of transforms to a quantum function.
`cancel_inverses`	Quantum function transform to remove any operations that are applied next to their (self-)inverse.
`commute_controlled`	Quantum function transform to move commuting gates past control and target qubits of controlled operations.
`merge_rotations`	Quantum function transform to combine rotation gates of the same type that act sequentially.
`single_qubit_fusion`	Quantum function transform to fuse together groups of single-qubit operations into a general single-qubit unitary operation (`Rot`).
`unitary_to_rot`	Quantum function transform to decomposes all instances of single-qubit and select instances of two-qubit `QubitUnitary` operations to parametrized single-qubit operations.
`merge_amplitude_embedding`	Quantum function transform to combine amplitude embedding templates that act on different qubits.
`remove_barrier`	Quantum function transform to remove Barrier gates.
`undo_swaps`	Quantum function transform to remove SWAP gates by running from right to left through the circuit changing the position of the qubits accordingly.

https://pennylane.readthedocs.io/en/stable/code/qml_transforms.html

Quantum transforms

Recall this slide from lecture 2...

You probably have some questions...

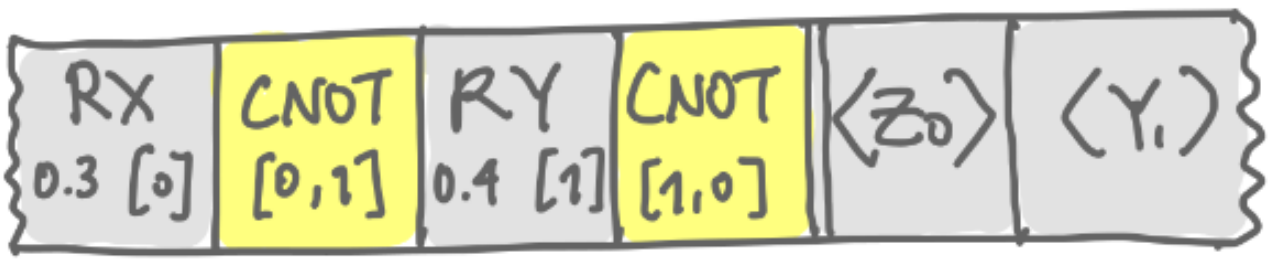
1. Where's the state?
   - Inside the device!
2. What happens to the gates?
   - Operations are recorded onto a "tape"
   - The QNode constructs the tape when it is called
   - The tape is then executed on the device.

19 / 66

Compilation transforms work by manipulating the underlying "quantum tape" that gets constructed in the QNode.

Quantum tapes

Tapes are flat lists of quantum operations and measurements.
They are a pretty low-level data structure in PennyLane.



Let's go inspect one...

Image credit: https://pennylane.ai/blog/2021/08/how-to-write-quantum-function-transforms-in-pennylane/

Single tape transforms

"Tape goes in, tape comes out": the simplest type of transform.

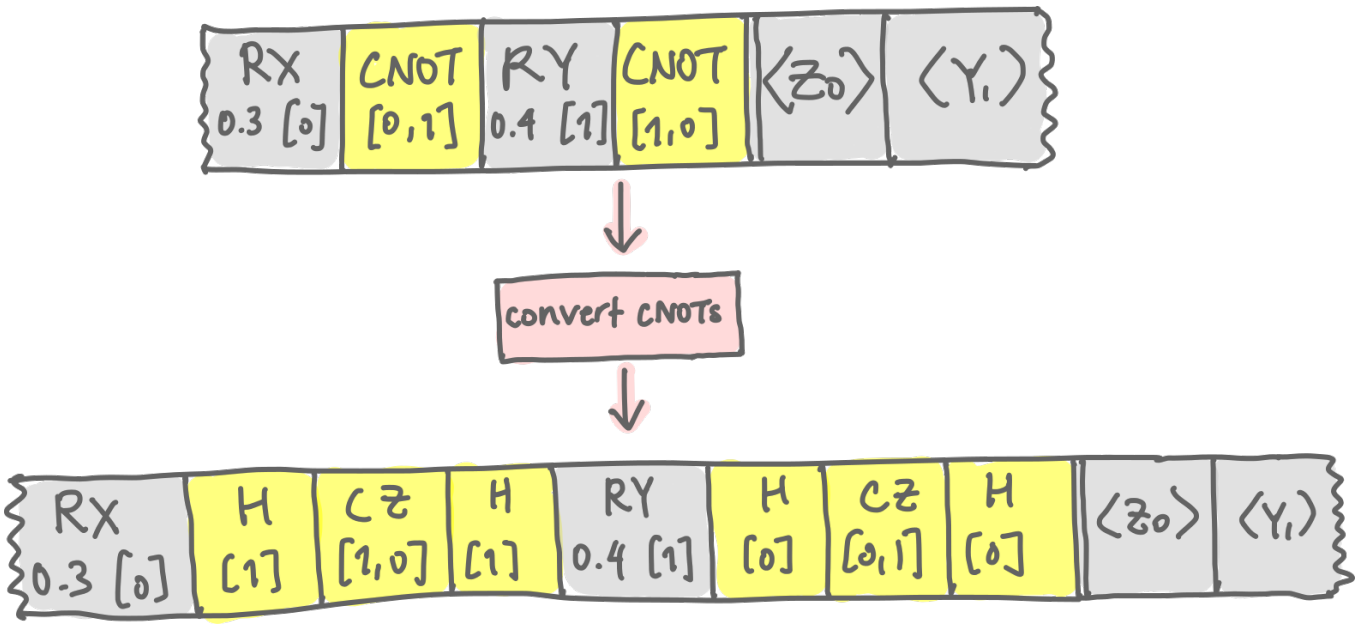


Image credit: https://pennylane.ai/blog/2021/08/how-to-write-quantum-function-transforms-in-pennylane/

Quantum function transforms

These are "elevated" tape transforms - they are constructed in the same way, but can be applied directly to quantum functions.

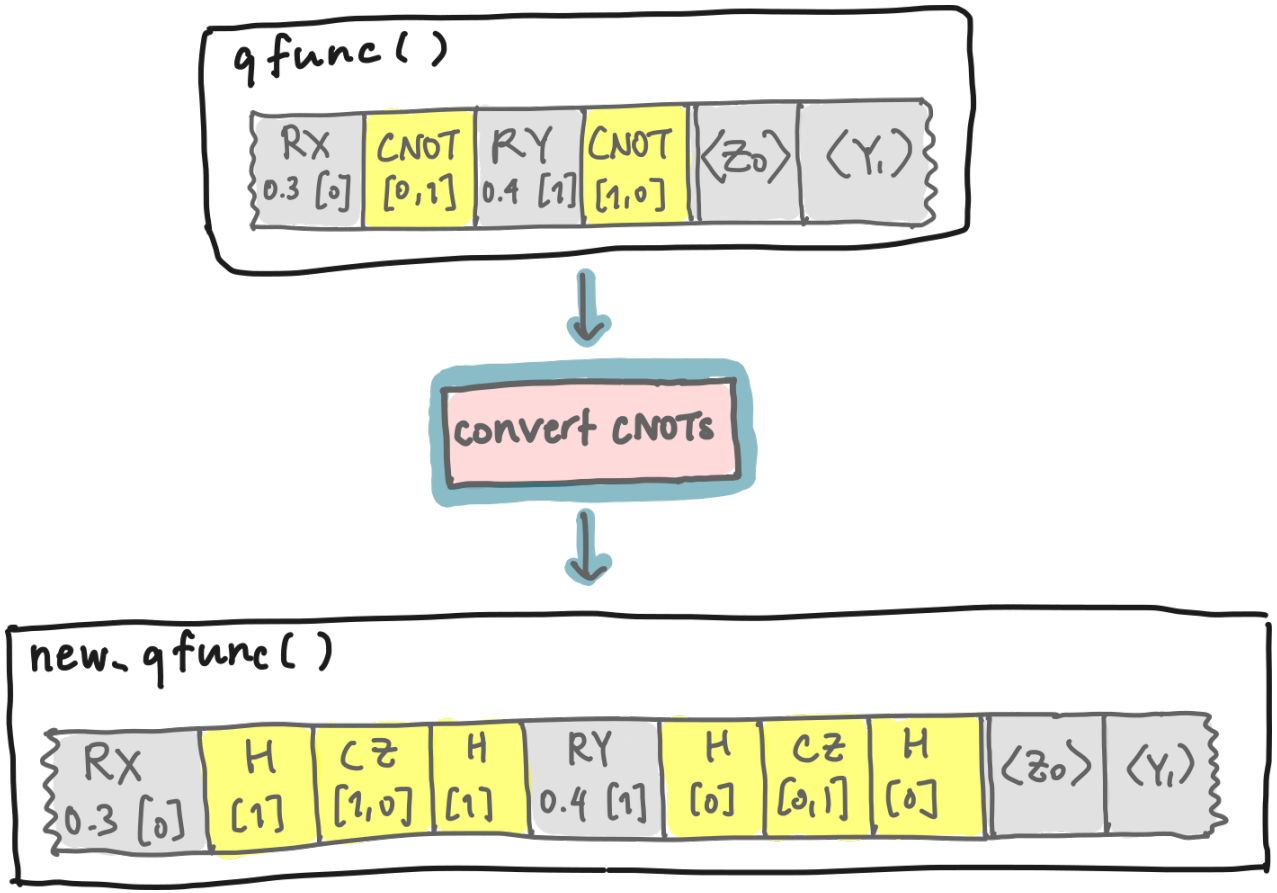


Image credit: https://pennylane.ai/blog/2021/08/how-to-write-quantum-function-transforms-in-pennylane/

Anatomy of a single tape transform

Loop through list of operations on a tape, and change which operations are applied.

```
@qml.single_tape_transform
def my_transform(tape, other_params):
    for op in tape.operations:
        # Change or manipulate operations

    for m im tape.measurements:
        # Change or manipulate measurements

# Apply the transform to a tape
new_tape = my_transform(old_tape)
```

Example: applying a circuit identity

X and Z are related by a Hadamard: $X = HZH$. Let's replace every instance of X with HZH.

```
@qml.single_tape_transform
def x_to_hzh(tape):
    for op in tape.operations:
        if op.name == ``PauliX''
            qml.Hadamard(wires=op.wires)
            qml.PauliZ(wires=op.wires)
            qml.Hadamard(wires=op.wires)
        else:
            qml.apply(op)

    for m im tape.measurements:
        qml.apply(op)
```

Example: applying a circuit identity

We can write a quantum function transform in the same way, using the `@qml.qfunc_transform` decorator:

```
@qml.qfunc_transform
def x_to_hzh(tape):
    for op in tape.operations:
        if op.name == ``PauliX''
            qml.Hadamard(wires=op.wires)
            qml.PauliZ(wires=op.wires)
            qml.Hadamard(wires=op.wires)
        else:
            qml.apply(op)

    for m im tape.measurements:
        qml.apply(op)
```

What is the advantage?

Example: applying a circuit identity

We can apply this directly to a quantum function:

```
def my_qfunc():
    qml.PauliX(wires=0)
    qml.PauliX(wires=1)
    return qml.probs(wires=[0, 1])

my_new_qfunc = x_to_hzh(my_qfunc)
```

Or, we can use it as a *decorator*.

```
@x_to_hzh
def my_qfunc():
    qml.PauliX(wires=0)
    qml.PauliX(wires=1)
    return qml.probs(wires=[0, 1])
```

Let's try it...

`@qml.compile`

PennyLane has a number of special-purpose transforms to perform some more sophisticated optimization and manipulation:

- `commute_controlled`: push gates past targets and controls
- `merge_rotations`: essentially, $RX(x_1)RX(x_2) = RX(x_1 + x_2)$
- `single_qubit_fusion`: merge adjacent single-qubit gates
- `undo_swaps`: removes SWAPs by relabeling qubits

... and more. It would be tedious to apply all of these as decorators to every quantum function we wanted to optimize.

@qml.compile

The top-level `@qml.compile` decorator can be used to apply pipelines of transforms.

```
pipeline = [
    undo_swaps,
    commute_controlled(direction='left'),
    cancel_inverses,
    merge_rotations
]

@qml.qnode(dev)
@qml.compile(pipeline=pipeline)
def my_qfunc():
    # Some gates...
    return qml.probs(wires=[0, 1])
```

Next time

Content:

- The Quantum Fourier transform

Action items:

1. Continue with Assignment 2 (can do all the problems now)
2. Email me your project group and paper selection

Recommended reading:

- Transforms blog post:
  `https://pennylane.ai/blog/2021/08/how-to-write-quantum-function-transforms-in-pennylane/`
- Explore the `qml.transforms` module
  https://pennylane.readthedocs.io/en/stable/code/qml_transforms.html