How Qiskit 2.6 compiles quantum programs for IBM hardware

Overview

1. **Why compilation**
   Ideal circuits vs real hardware
2. **Hardware model**
   IBM devices, ISA circuits, IBM-focused gate implementation
3. **The six transpiler stages**
   Init, layout, routing, translation, optimization, scheduling + the dependency graph
4. **Levels' comparison**
5. **Fault tolerance (very briefly)**
6. **Wrap-up**

Programs vs hardware

What you write

- Any gate: H, Toffoli, QFT, arbitrary rotations
- Any pair of qubits can interact
- Gates are instantaneous and perfect



compile

What the chip offers

- A few native gates (e.g. CZ, RZ, SX, X)
- Interactions only along edges of a coupling graph
- Each gate takes time and has its own error rate

What “compiling” means

The contract

Input: a circuit C on n qubits, and a device description T: the gates the device can run, which qubit pairs can interact, the error and duration of each operation, and the timing rules.

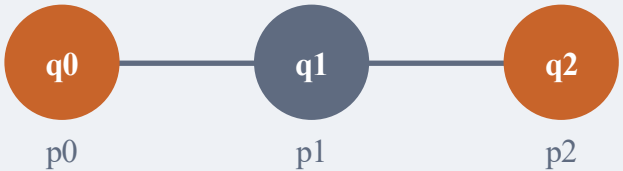
Output: a circuit C′ such that

- C′ uses only gates that T lists
- every two-qubit gate of C′ acts on a qubit pair that T allows
- C′ respects the timing rules of T
- $U(C') = \exp(i\varphi)\, U(C)$, up to the tracked qubit permutation

U(C) is the $2^n \times 2^n$ unitary matrix that C multiplies out to, and φ is a real number (a global phase).

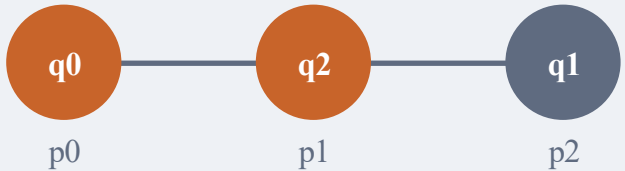
Toy example: one distant CX

Before: CX(q0, q2)



q0 and q2 sit on p0 and p2: no edge between them

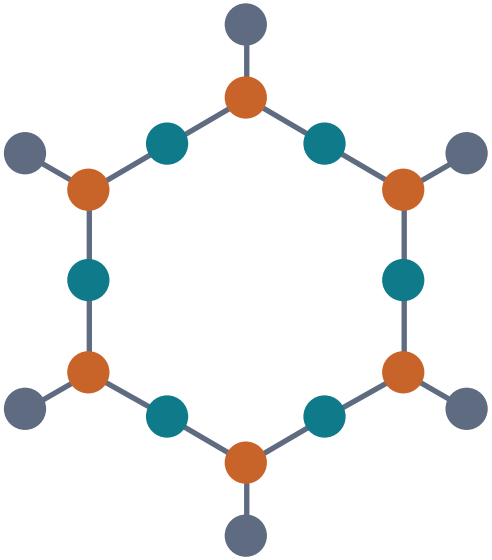
After SWAP on (p1, p2)



q0 and q2 now sit on p0 and p1: adjacent

Cost: SWAP = 3 CX gates, so every SWAP adds depth and error

Default IBM hardware



Heavy-hex cell (Falcon, Eagle, Heron)

Family	Layout	2-qubit gate	Neighbors
Falcon	heavy-hex	CX	2 to 3
Eagle	heavy-hex	ECR	2 to 3
Heron	heavy-hex	CZ	2 to 3
Nighthawk	square lattice	CZ	up to 4

Single-qubit gates on all four: RZ, SX, X

The compiler reads whatever device description it is given. On all four families a SWAP costs 3 two-qubit gates.

ISA circuits: the compiler's output



Makes the circuit fit the device

Uses only the device's instructions

Takes ISA circuits as input

Compilation is what turns the circuit you wrote into an ISA circuit

IBM-specific requirements in Qiskit

IBM requirement	How Qiskit encodes it
Native gates: CX, ECR or CZ by family; RZ, SX, X	Device gate list; translation stage
Fractional gates: RX(θ), RZZ(θ)	use_fractional_gates=True; RZZ needs $0 < \theta \leq \pi/2$ (FoldRzzAngle)
Uneven noise across qubits and edges	Per-qubit and per-edge errors; error-aware layout scoring
Timing rules	dt, granularity, alignments; scheduling stage
Mid-circuit measurement, feed-forward	if_else, for_loop, switch_case, reset as device operations
Topology: heavy-hex, or square lattice (Nighthawk)	Coupling map changes; the passes stay the same

The six-stage pipeline*

1	2	3	4	5	6
init	**layout**	**routing**	**translation**	**optimization**	**scheduling**
Reduce to 1- and 2-qubit gates	Which physical qubits?	Insert SWAPs	Rewrite in native gates	Remove redundancy	When does each gate run?

```
from qiskit.transpiler import generate_preset_pass_manager
pm = generate_preset_pass_manager(optimization_level=2, backend=backend)
isa_circuit = pm.run(circuit)
```

The six-stage pipeline*

1	2	3	4	5	6
init	**layout**	**routing**	**translation**	**optimization**	**scheduling**
Reduce to 1- and 2-qubit gates	Which physical qubits?	Insert SWAPs	Rewrite in native gates	Remove redundancy	When does each gate run?

*The lowest optimization “effort” doesn’t call stage 5!

Stage 1: init

6

CX gates for one Toffoli (CCX)

- Layout and routing only handle 1- and 2-qubit gates
- So any gate on 3 or more qubits is decomposed first
- Init is also the hook for custom early optimizations (plugins)

Counted with Qiskit 2.6: one CCX → 6 CX plus single-qubit gates

Stage 2: layout, the problem

Layout

A layout is an injective map

$$\ell : \{0, \ldots, n-1\} \to V$$

from virtual qubits of the circuit to physical qubits V of the device. Injective: no two virtual qubits share a physical qubit.

$$\approx 1.2 \times 10^{21}$$

placements of the demo's 10 qubits on FakeTorino's 133 qubits (133!/123!)

A good layout puts

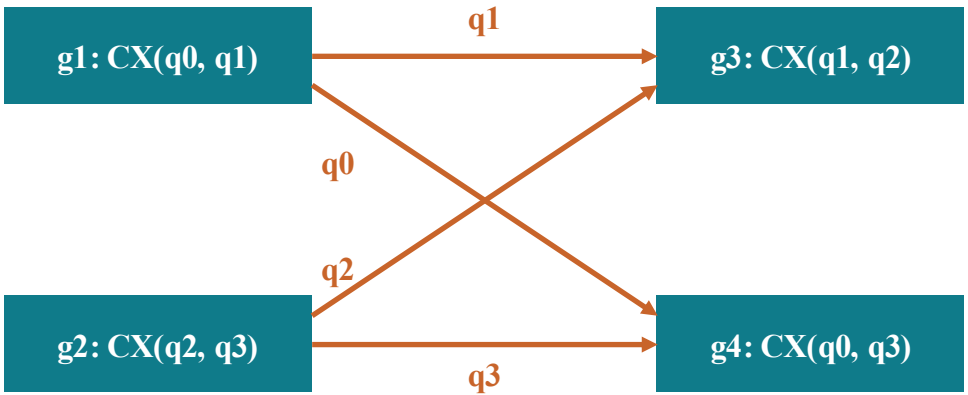
- interacting qubits on adjacent nodes
- the circuit on low-error qubits and edges

Layout primitives

Option	What it does	Where the presets use it
TrivialLayout	Virtual qubit i goes to physical qubit i	level 0; first try at level 1
DenseLayout	Most connected subgraph of the right size; on a real backend it also weighs error rates	not in 2.6.2 presets; layout_method="dense"
VF2Layout	Every interacting pair on a device edge, so no SWAPs; prefers low-error placements	levels 1 to 3
SabreLayout	Random starts refined by SabreSwap runs	fallback when no perfect fit

Also available, never used by the presets: SabrePreLayout (seeds Sabre via an augmented coupling map), CSPLayout (backtracking). SetLayout applies your initial_layout. External plugins by name.

The dependency graph



Layers: {g1, g2}, then {g3, g4}. Depth 2.

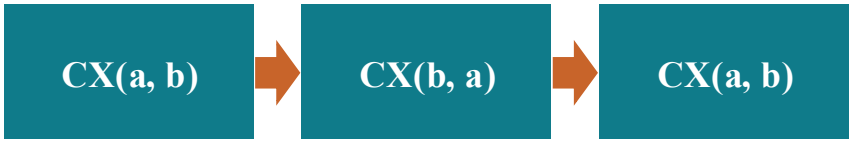
- Qiskit's DAGCircuit
- Nodes are operations; an edge joins an operation to the next one on the same qubit wire
- Acyclic: nothing depends on its own future
- Front layer: operations with no unfinished predecessor
- Depth: the longest path, equal to the number of layers

Stage 3: routing, inserting SWAPs

The problem

- Given a layout, insert SWAP gates so every 2-qubit gate acts on a hardware edge
- Finding the optimal SWAP set is hard, so Qiskit uses heuristics (SabreSwap by default)
- Goal: few SWAPs, low depth

SWAP(a, b) =



Each SWAP costs three two-qubit gates for any IBM architecture, the noisiest ops on the chip

Routing: the options

Option	What it does
sabre (default)	SabreSwap; heuristic basic, lookahead or decay; many trials
basic	BasicSwap; swaps along a path for each blocked gate
lookahead	LookaheadSwap; searches over sequences of SWAPs
none	No routing; errors if the circuit is not already mapped

User-designed routing conglomerates also exist: a user can chain several routing-related passes in a custom pass manager instead of choosing one routing_method. Also: external plugins; AIRouting in qiskit-ibm-transpiler (add-on).

SabreSwap: a greedy heuristic

1. **Front layer:** Gates whose predecessors are done

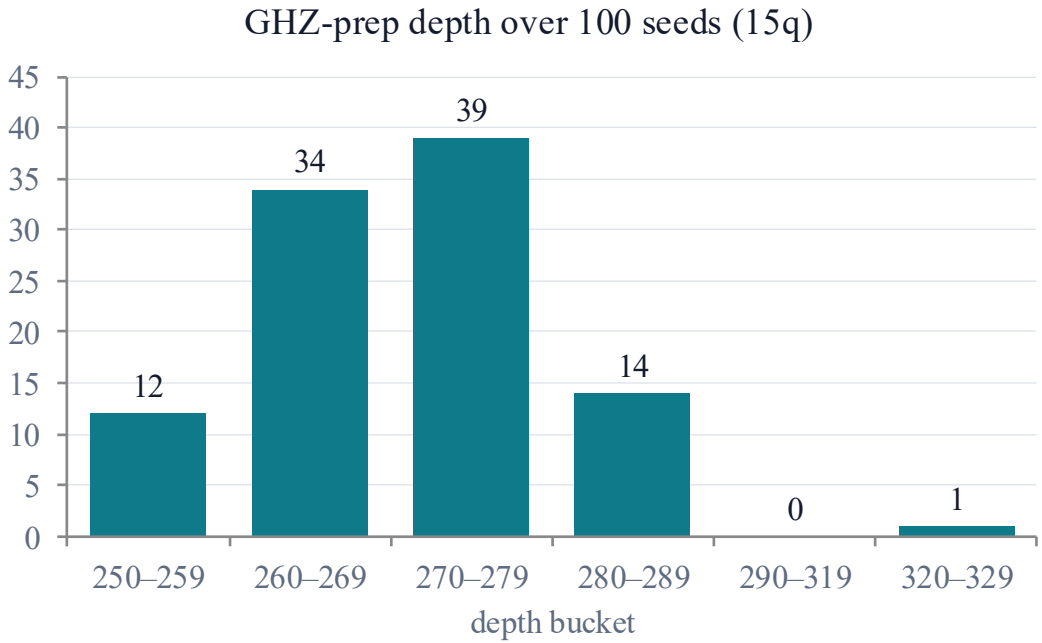
2. **Execute:** Run every front-layer gate whose qubits are adjacent

3. **Score SWAPs:** Try each SWAP touching a front-layer qubit; prefer ones that shrink distances, with lookahead and a penalty for repeats

4. **Apply best, repeat:** Ties are broken randomly, so results vary by seed

Li, Ding and Xie, arXiv:1809.02573 (the algorithm behind SabreSwap and SabreLayout)

Heuristic means results vary



253–321

depth range, same input

Run several seeds and keep the best result

Stage 4: translation

On the Eagle backend



This example is on the Eagle (CZ-based) architecture. ECR-based backends (Heron) translate CXs differently.

Stage 5: optimization



Why two-qubit gates get the attention: RZ costs almost nothing on IBM hardware, while two-qubit gates are slower and noisier, so the loop spends its effort removing CX and CZ. SWAPs are three such gates, so they are often merged away.

Stage 6: scheduling

ASAP: start as early as possible



Measured on FakeTorino qubits 0 to 2, Qiskit 2.6.2. Measurement bars not to scale.

ALAP: stop as late as possible (up to circuit execution depth!)



- Delays make idle time explicit
- ALAP anchors the end: idle time moves toward the start of each qubit, where it is spent in the ground state

Where every pass lives

Stage	Passes, in the order they run (preset pass managers, Qiskit 2.6.2)
1 init	UnitarySynthesis, HighLevelSynthesis, BasisTranslator (reduce to 1- and 2-qubit gates). Level 1 and up: InverseCancellation. Levels 2 and 3 also: ElidePermutations, RemoveDiagonalGatesBeforeMeasure, RemoveIdentityEquivalent, CommutativeCancellation, ConsolidateBlocks, Split2QUnitaries.
2 layout	SetLayout (your initial_layout), TrivialLayout, VF2Layout, SabreLayout; DenseLayout only on request. After the first three, FullAncillaAllocation, EnlargeWithAncilla and ApplyLayout pad the circuit to the full device and relabel it onto physical qubits.
3 routing	CheckMap (is routing needed?), BarrierBeforeFinalMeasurements, SabreSwap, FilterOpNodes (removes that barrier). SabreSwap is skipped when SabreLayout has already routed. Levels 1 to 3 add VF2PostLayout, unless VF2Layout already found a perfect layout.
4 translation	UnitarySynthesis, HighLevelSynthesis, BasisTranslator again, now targeting the backend's native gates; this is where each SWAP becomes three two-qubit gates.
5 optimization	Level 1: Size, Depth, FixedPoint (loop bookkeeping), Optimize1qGatesDecomposition, InverseCancellation, GatesInBasis. Level 2: TwoQubitPeepholeOptimization once before the loop; loop body RemoveIdentityEquivalent, Optimize1qGatesDecomposition, CommutativeCancellation. Level 3: the peephole pass moves into the loop, MinimumPoint replaces FixedPoint, and VF2PostLayout runs after the loop.
6 scheduling	With scheduling_method set: TimeUnitConversion, ALAPScheduleAnalysis or ASAPScheduleAnalysis, InstructionDurationCheck, PadDelay. Without it: only ContainsInstruction, TimeUnitConversion (if delays exist) and InstructionDurationCheck.

Stages are not strictly sequential

Where the order bends	What happens	Stages
Loop inside stage 5	Optimization passes repeat in a do-while loop until size and depth stop changing (FixedPoint at levels 1 and 2, MinimumPoint at level 3).	5 → 5
Stage 5 re-runs stage 4	After each loop pass, GatesInBasis checks that every gate is still native; if not, the translation stage is run again inside the loop. It never fired in my traces.	5 → 4
Stage 5 revisits stage 2	At level 3, VF2PostLayout runs after the loop and may move the finished circuit to lower-error qubits (same gates, new qubit labels). Levels 1 to 3 also run it inside stage 3.	5 → 2
Stages 2 and 3 merge	SabreLayout chooses the layout and routes in one pass, so stage 3 finds nothing left to do; a perfect layout from VF2Layout skips routing entirely.	2 ↔ 3
Stage 1 reaches ahead	ElidePermutations deletes SWAPs and permutations before layout and records them as a qubit permutation on the output circuit.	1 → output

Short answer: the six stages run once, in a fixed order, and no stage jumps backwards. Every “go back” is a loop or a condition inside a stage, or a pass that appears in two stages.

Optimization levels: search budgets

Level	Layout search	VF2Layout budget	Sabre (layout, routing)
0	TrivialLayout only	not run	routing: basic heuristic, 5 trials
1	Trivial, then VF2, then Sabre	50,000 calls, 1,000 trials	layout 2 iterations × 5 trials; routing: decay, 5
2	VF2, then Sabre	5 million calls, 10,000 trials	layout 2 iterations × 20 trials; routing: decay, 20
3	VF2, then Sabre	30 million calls, 100,000 trials	layout 4 iterations × 20 trials; routing: decay, 20

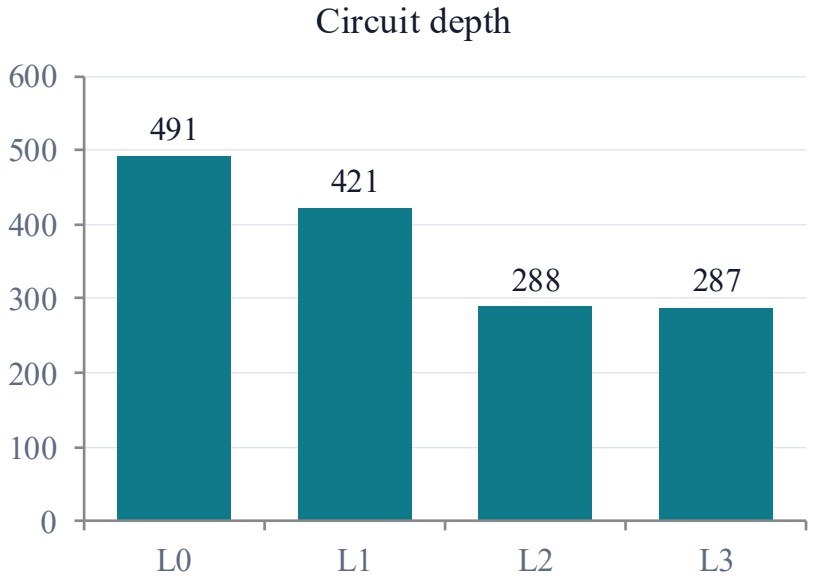
Read from the Qiskit 2.6.2 source, not the docs; budgets may change between releases. A trial is one Sabre run from a new random seed.

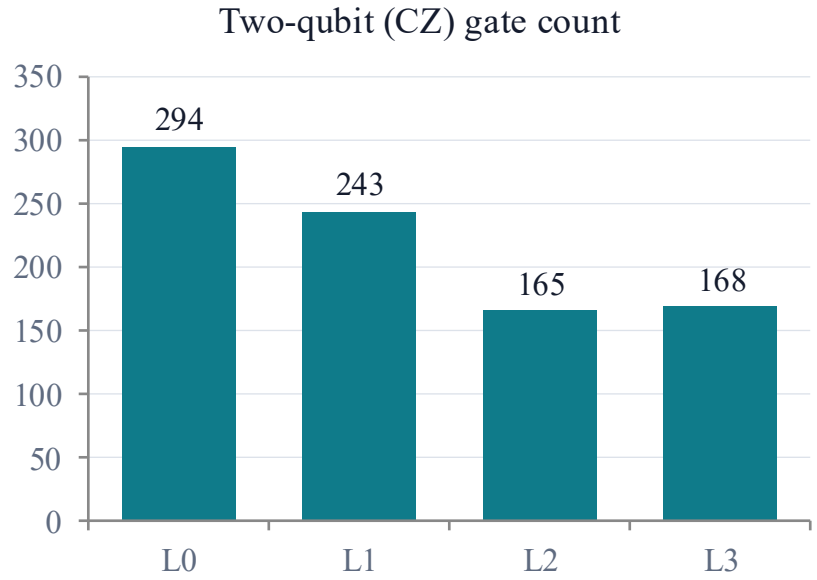
Optimization levels: the passes

Level	Init stage adds	Optimization stage
0	only reduces to 1- and 2-qubit gates	none
1	inverse cancellation	loop: 1-qubit merge + inverse cancellation, to a fixed point
2	+ commutative cancellation, block consolidation, permutation elision	2-qubit peephole once, then loop; fixed point
3	same as level 2	2-qubit peephole inside the loop; stop at minimum of depth and size; then VF2PostLayout

SWAP example: a random 2-qubit unitary plus a SWAP costs 6 ECR at levels 0 and 1 but 3 at levels 2 and 3, because the SWAP is merged into the block.

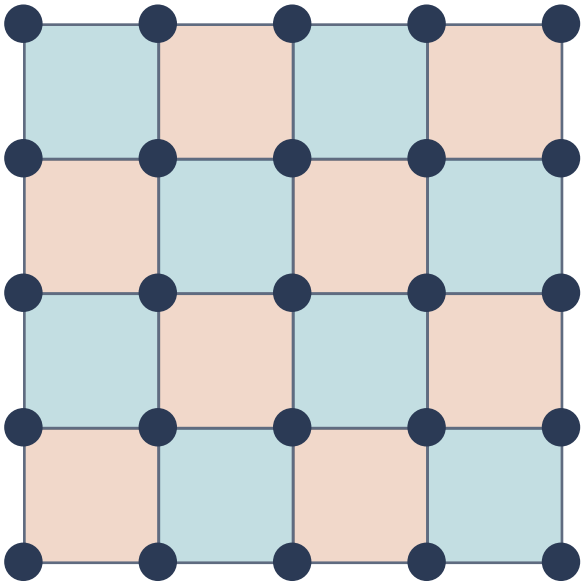
Demo: levels on a 10-qubit QFT





FakeTorino is a backend, not a circuit: an offline snapshot of a 133-qubit Heron device (CZ native). Circuit: 10-qubit QFT, seed 7. Level 3 is not always best (168 versus 165 CZ gates).

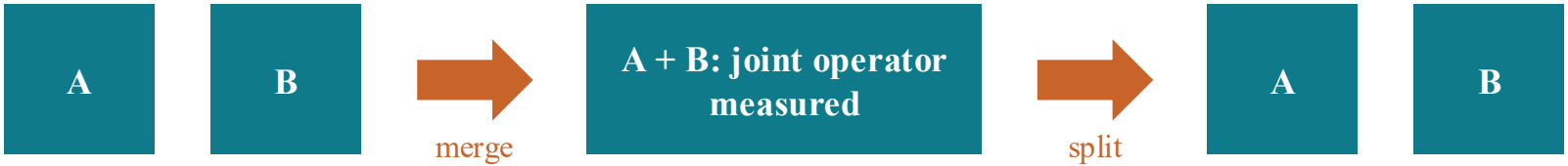
Logical qubits and the surface code



Schematic, d = 5: dots are data qubits, shaded cells are parity checks (interior only)

- Logical qubit: one qubit encoded across many physical qubits
- Distance d: fewest physical errors that can silently corrupt it
- Rotated surface code: $2d^2 - 1$ physical qubits per logical qubit (49 for d = 5)
- Needs a 2D square lattice of nearest-neighbor couplings
- Cost becomes space (patches) and time (error-correction rounds)

Lattice surgery and Pauli-based form

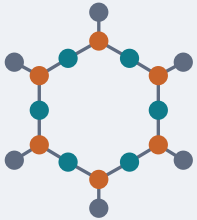


Clifford and T
Cliffords (H, S, CX) map Pauli strings to Pauli strings and are cheap here: patch merges and splits. T gates are not Clifford; they consume a magic state, a prepared resource state, by teleportation. That is the costly part.

Pauli-based form
Cliffords are commuted to the end, leaving Pauli product rotations $\exp(-i\theta P/2)$ and measurements (Litinski 2019). Qiskit: LitinskiTransformation, PauliProductRotationGate, PauliProductMeasurement; CommutativeOptimization works on this form (v2.3).

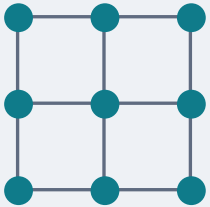
IBM connectivity versus code requirements

Heron



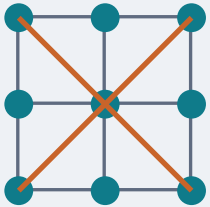
Heavy-hex, CZ, up to 3 neighbors.

Nighthawk



120 qubits, square lattice, up to 4 neighbors, 218 tunable couplers. Surface-code memory run on Nighthawk r2 (arXiv:2610.23840).

Loon



Experimental chip with long-range C-couplers, a building block for qLDPC codes (press reports).

Surface code: yes. A rotated surface-code memory has been run on a Nighthawk r2 processor (arXiv:2610.23840).

BB code: no. It needs every qubit coupled to 6 others, in a graph made of two edge-disjoint planar subgraphs (Bravyi et al.). Nighthawk has at most 4 neighbors in one planar lattice.

Takeaways

1. **Compilation bridges ideal circuits and real chips**
   native gates, connectivity, time

2. **One device description feeds every stage**
   IBM specifics are data in it

3. **Layout and routing are heuristic searches**
   seeds matter; level 3 is not always best

4. **Fault tolerance changes what the compiler optimizes**
   Pauli-based circuits, lattice surgery; IBM’s roadmap uses BB qLDPC codes

Questions?

Sources and further reading

Qiskit docs: Transpiler stages

Qiskit docs: Transpile with pass managers

Qiskit API: describing a backend to the transpiler

Qiskit docs: Fractional gates

IBM blog: What are ISA circuits?

Qiskit SDK 2.3 release summary

Qiskit API: LitinskiTransformation

Qiskit Transpiler Service release notes

Li, Ding, Xie: SABRE (arXiv:1809.02573)

Litinski: A Game of Surface Codes (Quantum 3, 128)

Quantum Computing Report: IBM roadmap and BB codes

Tom's Hardware: Nighthawk and Loon

Layout, routing, scheduling and optimization-level details: Qiskit 2.6.2 source (preset_passmanagers/builtin_plugins.py, common.py) and pass docstrings