Qdislib — Circuit Cutting Cheat Sheet

Run circuits bigger than your device: cut, execute the pieces, recombine.

Qdislib is a distributed circuit-cutting library: it splits a quantum circuit into smaller subcircuits that fit your hardware, runs them independently (in parallel, even across backends or QPUs) and recombines the results classically. It is **backend-agnostic** — the same code drives Qibo, Qiskit and CUDA-Q — and scales with PyCOMPSs.

WAYS TO CUT

Gate cutting replaces a two-qubit gate by a sum of single-qubit *measure-and-prepare* operations, splitting the circuit where that gate was. Cuts **any two-qubit gate** (CZ/CX natively; others auto-decomposed to CZ). **Cost: 6^k** terms for *k* cuts.

Wire cutting cuts a qubit's wire (its timeline) into "measure here / re-prepare there". **Cost: 8^k** terms for *k* cuts.

Hadamard gate cutting (`gate_cutting_hadamard`) cuts a gate with one **local ancilla** per fragment instead of mid-circuit measurement — cuts **any** gate with the **optimal** overhead (γ(CZ)=3, $\to$1 for weak gates). Great for superconducting QPUs.

Every cut multiplies the work **exponentially**, so use as few, and as well-placed, as possible. Preview the blow-up with `cut_cost(cut)` before you run.

THE WORKFLOW

Get one **expectation value** <O> in a single call, or use the **three-step** form to control how each piece runs (cluster, QPU):

```
cut = qd.find_cut(circ, gate_cut=True)
val = qd.gate_cutting(circ, cut,
          observables='ZZZZ', shots=8192)

subs = qd.gate_cutting_subcircuits(
         circ, cut, observables='ZZZZ')
vals = [qd.expectation_value(s)
        for s in subs]     # run anywhere
val = subs.reconstruct(vals)
```

`estimate(circ, 'ZZZZ', max_qubits=n)` does find-cut + cut + reconstruct in one call.

QUICKSTART

```
import Qdislib as qd
from qibo import Circuit, gates

circ = Circuit(4)
circ.add(gates.H(0))
circ.add(gates.CZ(0, 1))
circ.add(gates.CZ(1, 2))
circ.add(gates.CZ(2, 3))

cut = qd.find_cut(circ, gate_cut=True,
        wire_cut=False, seed=0)
val = qd.gate_cutting(circ, cut,
        observables='ZZZZ', shots=8192)
```

CORE API

`find_cut(circ, ...)`	Choose where to cut (returns it).
`gate_cutting(circ, cut)`	<O> via gate cutting.
`wire_cutting(circ, cut)`	<O> via wire cutting.
`gate_cutting_hadamard(...)`	<O> via the ancilla cut.
`gate_wire_cutting(...)`	Mixed gate + wire cut.
`..._sampling(circ, cut)`	Full output distribution.
`..._subcircuits(...)`	3-step: pieces + .reconstruct().
`expectation_value(sub)`	Run one subcircuit.
`estimate(circ, ...)`	One-call cut + run + reconstruct.
`cut_cost(cut)`	Preview terms = base**#cuts.
`analytical_solution(...)`	Exact <O> (to check).
`draw_cut(circ, cut, path)`	Visualise the cut.
`from_qasm(qasm, ...)`	Import any OpenQASM 2.

HADAMARD (ANCILLA) GATE CUT

An alternative gate cut (Harrow–Lowe, *arXiv:2403.01018*): each fragment carries **one extra ancilla** running a Hadamard test — **no mid-circuit measurement**, and it cuts **any** two-qubit gate (iSWAP, fSim, …) at the **optimal** overhead.

```
val = qd.gate_cutting_hadamard(
        circ, ['ISWAP_3'], 'ZZ', software='qibo')
# Monte-Carlo (bounded cost for many cuts):
val = qd.gate_cutting_hadamard_sampling(
        circ, cut, 'ZZ', num_samples=5000)
# three-step, runs on any backend / QPU:
subs = qd.gate_cutting_hadamard_subcircuits(
         circ, cut, 'ZZ')
val = subs.reconstruct(
  [qd.expectation_value(s) for s in subs])
```

Fragments are independent (own ancilla, no feed-forward) → they fan out as PyCOMPSs tasks like every other cut.

MIXED WIRE + GATE CUTTING

Cut some **gates** and some **wires** on the **same circuit** in one workflow. The two quasiprobability decompositions tensor, and every fragment across the combined space runs in parallel.

```
val = qd.gate_wire_cutting(circ,
        gates_cut=['CZ_2'],
        wire_cut=[('RY_3', 'CZ_4')],
        observable='ZZZ', software='qibo')
```

SAMPLING THE DISTRIBUTION

Reconstruct the full **output distribution** `{bitstring: prob}` — to read out an image or inspect a state — instead of a single number:

```
dist = qd.gate_cutting_sampling(
         circ, cut, shots=200_000)
# three-step (parallel under PyCOMPSs):
subs = qd.gate_cutting_sampling_subcircuits(
         circ, cut)
dist = subs.reconstruct(
  [qd.sample_subcircuit(s) for s in subs])
```

`wire_cutting_sampling` too. Sampling carries a quasiprobability overhead, so it needs **more shots** than an expectation value.

FIND_CUT OPTIONS

`max_qubits=n`	Max qubits per subcircuit (device size).
`max_cuts=k`	Cap the number of cuts.
`gate_cut / wire_cut`	Which cut types to consider.
`coupling_map=`	Hardware-aware: cut the longest-routing gates (SparseCut, arXiv:2511.05492).
`mixed=True`	Combined {gate, wire} cut for gate_wire_cutting (cost-aware).
`implementation=`	'qdislib' (default) or 'ibm-ckt'.
`seed=n`	Reproducible cut selection.

OBSERVABLES

Per-qubit Pauli string over **I X Y Z** (qubit-0-first): `'ZZZZ'`, `'IZIZ'`, `'XZZY'`. A Hamiltonian is a weighted sum:

```
[(0.5, 'ZZII'), (-1.0, 'IIZZ')]
# or  {'ZZII': 0.5, 'IIZZ': -1.0}
```

`I` ignores a qubit — and works even on a *cut* qubit (CPU & GPU).

BACKENDS

`software=`	'qibo' | 'qiskit' | 'cudaq'.
`gpu=True`	Aer GPU (needs CUDA).
`qpu=True, qpu_dict=`	{'MN_Ona': n} / {'IBM_Quantum': 127}.

Qdislib — Circuit Cutting Cheat Sheet

Run circuits bigger than your device: cut, execute the pieces, recombine.

`available_backends()`	List registered backends.
`build_execution_backend`	How software/gpu/qpu pick one.
`QdBackEnd(pool)`	Load-balance across devices.
`register_backend(...)`	Plug in your own (BaseBackend).

CACHING

```
cache = qd.default_circuit_cache(
          'memory', max_entries=5000)
qd.gate_cutting(circ, cut, cache=cache)
```

The **semantic cache** reuses functionally-equivalent subcircuits (e.g. H·H·H = H). Stores: `'memory'` / `'lmdb'` / `'redis'`. `set_log_level('DEBUG')` for diagnostics.

DISTRIBUTED WITH PYCOMPSS

Qdislib is built for HPC: every subcircuit is a PyCOMPSs `@task`, so the pieces run **in parallel across cores and nodes**. The three-step workflow fans them out automatically — the same code scales from a laptop to a supercomputer.

```
import pycompss.interactive as ipycompss
ipycompss.start()   # or:  runcompss app.py
subs = qd.gate_cutting_subcircuits(circ, cut)
vals = [qd.expectation_value(s) for s in subs]
val = subs.reconstruct(vals)   # auto-parallel
ipycompss.stop()
```

CHOOSING A METHOD

`gate_cutting`	Any 2-qubit gate; distribution too; no ancilla.
`gate_cutting_hadamard`	Any gate at optimal γ; no mid-circuit measure (best on superconducting QPUs); expectation only.
`wire_cutting`	Cut a qubit's wire between two subcircuits.
`gate_wire_cutting`	Both cut types on one circuit at once.
`*_sampling`	Reconstruct the full output distribution.

OVERHEAD AT A GLANCE

`gate cut / cut`	6 terms ($\gamma = 3$ for CZ/CX).
`wire cut / cut`	8 terms (or 3^n+4^n with optimized=True).
`hadamard / cut`	γ(CZ)=3, γ(SWAP)=7, $\to$1 for weak gates; sampling caps cost at num_samples.
`general gate`	$\leq$ 3 CZ $\to$ up to 6^3 terms (one cut with the Hadamard method).

SCALE THE PARALLELISM

Every subcircuit is an independent task. Extra knobs for big fan-outs:

`sync=False`	Return futures; overlap several cuts (wait once).
`reduce_chunk=n`	Tree reduction instead of one fan-in task.
`optimized=True`	Wire cut: 3^n+4^n fragments, not 8^n.
`num_samples=n`	Hadamard sampling: bound the cost, not 6^k.

GOTCHAS

- Cost is exponential in #cuts (6^k / 8^k) — fewer, well-placed cuts win; preview with `cut_cost`.
- Any two-qubit gate is cuttable now; a non-CZ/CX gate becomes **3 CZ cuts**, so prefer `gate_cutting_hadamard` (1 cut, optimal) for exotic gates.
- Sampling needs **more shots** than an expectation value (quasiprobability overhead).
- Identity on a cut qubit: **CPU & GPU yes**, **QPU no**; for a general gate use `software='qibo'` or the Hadamard cut.
- Use `seed=` for a reproducible `find_cut`.

REAL HARDWARE

Route the subcircuits to GPUs or QPUs — the cutting code is unchanged:

```
qd.gate_cutting(circ, cut, gpu=True)      # Aer GPU
qd.gate_cutting(circ, cut, qpu=True,
        qpu_dict={'MN_Ona': 4})           # BSC QPU
qd.gate_cutting(circ, cut, qpu=True,
        qpu_dict={'IBM_Quantum': 127})    # IBM
```

Under PyCOMPSs different fragments can even run on **different devices at once** (`QdBackEnd(pool)`).

BRING YOUR OWN CIRCUIT

Pass a **Qibo, Qiskit, CUDA-Q or PennyLane** circuit — Qdislib converts it through a neutral DAG that speaks every backend (U3/CU3/… included). Or import OpenQASM 2:

```
circ = qd.from_qasm(open('c.qasm').read())
qd.gate_cutting(circ, cut, software='qiskit')
```

LEARN MORE

Full docs, the nine tutorial notebooks (Bell states $\to$ 25-qubit lattices) and the API reference:

`Docs`	qdislib.readthedocs.io
`Source`	gitlab.bsc.es/wdc/quantum/qdislib
`Notebooks`	examples/ — cutting, sampling, scaling, QPUs.
`estimate(...)`	One call: find-cut + cut + reconstruct.

SANITY CHECK

Validate a cut against the exact value on a small instance before you scale it up (cutting reconstructs the same <O>, within shot noise):

```
exact = qd.analytical_solution(circ, 'ZZ')
val   = qd.gate_cutting(circ, cut, 'ZZ',
                        shots=8192)
assert abs(val - exact) < 0.05
qd.draw_cut(circ, cut, 'cut.png')   # see it
```