# === Latency & Computational Cost Profiling (QCAA vs QCAA-Optimized) — ibm\_brisbane ===

# Output dir: log16/

# Artifacts:

# - log16/phase\_timings.csv

# - log16/summary.json

# - log16/mask\_qcaa\_opt.csv

# - log16/\*counts\_pub0\_\*.json

# - log16/run.log, log16/run.jsonl

import os, sys, json, uuid, logging

from time import perf\_counter

from pathlib import Path

from datetime import datetime

import numpy as np

import pandas as pd

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import roc\_curve

from qiskit\_ibm\_runtime import QiskitRuntimeService, SamplerV2 as Sampler

from qiskit import QuantumCircuit, transpile

from neal import SimulatedAnnealingSampler

from dimod import BinaryQuadraticModel

# ---------------- Logging ----------------

RUN\_TS = datetime.now().strftime("%Y%m%d\_%H%M%S")

RUN\_ID = f"profile\_hw\_{RUN\_TS}\_{uuid.uuid4().hex[:6]}"

LOG\_DIR = Path("log16"); LOG\_DIR.mkdir(parents=True, exist\_ok=True)

logger = logging.getLogger(RUN\_ID); logger.setLevel(logging.INFO)

fmt = logging.Formatter("[%(asctime)s] %(levelname)s - %(message)s", "%Y-%m-%d %H:%M:%S")

fh = logging.FileHandler(LOG\_DIR / "run.log", encoding="utf-8"); fh.setFormatter(fmt); fh.setLevel(logging.INFO); logger.addHandler(fh)

sh = logging.StreamHandler(sys.stdout); sh.setFormatter(fmt); sh.setLevel(logging.INFO); logger.addHandler(sh)

def log\_json(event, \*\*kw):

rec = {"ts": datetime.now().isoformat(), "run\_id": RUN\_ID, "event": event, \*\*kw}

with open(LOG\_DIR / "run.jsonl", "a", encoding="utf-8") as f:

f.write(json.dumps(rec, ensure\_ascii=False) + "\n")

def log\_kv(msg, \*\*kw):

logger.info(f"{msg} | " + " ".join(f"{k}={v}" for k,v in kw.items()))

log\_json(msg, \*\*kw)

# ---------------- Credentials (your latest) ----------------

IBM\_TOKEN = "ynRx1PCW4ipBRI-cZeL5LQp4VOh4YxAXjrmJDQHAr0XX"

IBM\_INSTANCE = "crn:v1:bluemix:public:quantum-computing:us-east:a/0a0a05e6d1a44757bc62c3e6b305f821:ac760e90-3c3e-41ed-8f33-a53cd7c2bf39::"

# ---------------- Config ----------------

DATA\_PATH = r"C:\Users\Sandip Dutta\OneDrive\Desktop\biometric\_detailed\_dataset.csv"

LABEL\_CANDS = ["label","y","target","Label","Target"]

SHOTS = 1024

OPT\_LEVEL = 1

DRIFT\_S2 = (0.03, 1.03) # (mean\_shift, scale)

DRIFT\_S3 = (0.06, 1.06)

GLOBAL\_SA\_SEED = 7

SA\_READS = 50

MIN\_ON = 2

np.random.seed(42)

log\_kv("config\_set", data\_path=DATA\_PATH, shots=SHOTS, opt\_level=OPT\_LEVEL)

# ---------------- Backend ----------------

service = QiskitRuntimeService(channel="ibm\_cloud", token=IBM\_TOKEN, instance=IBM\_INSTANCE)

backend = service.backend("ibm\_brisbane")

print(f"✅ Connected to backend: {backend.name}")

log\_kv("backend\_connected", backend=backend.name)

# ---------------- Data ----------------

df = pd.read\_csv(DATA\_PATH)

label\_col = next((c for c in LABEL\_CANDS if c in df.columns), None)

if label\_col is None:

raise ValueError("No label column found (expected one of label/y/target/Label/Target).")

feature\_cols = [c for c in df.columns if c != label\_col and np.issubdtype(df[c].dtype, np.number)]

if not feature\_cols:

raise ValueError("No numeric feature columns besides label.")

X\_raw = df[feature\_cols].to\_numpy(dtype=float)

y = df[label\_col].astype(int).to\_numpy()

session\_col = next((c for c in ["session","timestamp","time","date","Session","Timestamp"] if c in df.columns), None)

order = np.argsort(df[session\_col].values) if session\_col else np.arange(len(df))

X\_raw, y = X\_raw[order], y[order]

scaler = MinMaxScaler()

X = scaler.fit\_transform(X\_raw)

log\_kv("dataset\_loaded", n\_samples=len(X), n\_features=X.shape[1],

n\_genuine=int(y.sum()), n\_impostor=int(len(y)-y.sum()),

feature\_cols=",".join(feature\_cols))

# ---------------- Sessions ----------------

n = len(X)

s1\_end = int(0.6\*n); s2\_end = int(0.8\*n)

idx\_s1, idx\_s2, idx\_s3 = np.arange(0, s1\_end), np.arange(s1\_end, s2\_end), np.arange(s2\_end, n)

X\_s1, y\_s1 = X[idx\_s1], y[idx\_s1]

def driftify(Xpart, mean\_shift, scale):

return np.clip(Xpart \* scale + mean\_shift, 0.0, 1.0)

X\_s2, y\_s2 = driftify(X[idx\_s2], \*DRIFT\_S2), y[idx\_s2]

X\_s3, y\_s3 = driftify(X[idx\_s3], \*DRIFT\_S3), y[idx\_s3]

# ---------------- Circuit builders ----------------

def build\_circuit\_qcaa(x\_vec):

d = len(x\_vec); qc = QuantumCircuit(d)

for i in range(d):

xi = float(np.clip(x\_vec[i], 1e-9, 1-1e-9))

qc.ry(2.0 \* np.arcsin(np.sqrt(xi)), i)

for i in range(d-1): qc.cz(i, i+1)

qc.measure\_all()

return qc

def build\_circuit\_qcaa\_opt(x\_vec, mask):

d = len(x\_vec); qc = QuantumCircuit(d)

active = [i for i,m in enumerate(mask) if m==1]

for i in active:

xi = float(np.clip(x\_vec[i], 1e-9, 1-1e-9))

qc.ry(2.0 \* np.arcsin(np.sqrt(xi)), i)

for a,b in zip(active[:-1], active[1:]): qc.cz(a,b)

qc.measure\_all()

return qc

# --- JSON-safe SA mask (PATCHED) ---

def sa\_mask(d, seed=None, reads=50, min\_on=2):

rng = np.random.default\_rng(seed)

linear = {i: float(rng.uniform(-1,1)) for i in range(d)}

quad = {(i,j): float(rng.uniform(-1,1)) for i in range(d) for j in range(i+1,d)}

bqm = BinaryQuadraticModel(linear, quad, 0.0, vartype='BINARY')

resp = SimulatedAnnealingSampler().sample(bqm, num\_reads=reads, seed=seed)

mask = list(resp.first.sample.values())

# ensure minimum 'on'

if sum(int(v) for v in mask) < min\_on:

for i in sorted(range(d), key=lambda k: abs(linear[k]), reverse=True)[:min\_on]:

mask[i] = 1

# cast to plain Python ints

return [int(v) for v in mask]

# ---------------- Helpers ----------------

def to\_bits\_any(k, width):

if isinstance(k, int): return format(k, f"0{width}b")

if isinstance(k, str):

s = k.replace(" ", "")

if set(s) <= {"0","1"}: return s.zfill(width)[-width:]

try: return format(int(s,2), f"0{width}b")

except Exception: return s.zfill(width)[-width:]

if isinstance(k, tuple):

try: return "".join("1" if bool(v) else "0" for v in k).zfill(width)[-width:]

except Exception: return "".join(str(v) for v in k).zfill(width)[-width:]

return "0"\*width

def extract\_P1\_and\_snapshot(res, num\_qubits, tag, shots):

if hasattr(res, "quasi\_dists") and res.quasi\_dists:

qlist = res.quasi\_dists

N = len(qlist)

P1 = np.zeros((N, num\_qubits), dtype=float)

for i, qdist in enumerate(qlist):

row = np.zeros(num\_qubits)

for k, prob in dict(qdist).items():

bits = to\_bits\_any(k, num\_qubits)[::-1]

for q in range(num\_qubits):

if bits[q] == "1": row[q] += float(prob)

P1[i] = row

try:

snap = {to\_bits\_any(k, num\_qubits): int(float(v)\*shots) for k,v in dict(qlist[0]).items()}

with open(LOG\_DIR / f"counts\_pub0\_{tag}.json","w",encoding="utf-8") as f:

json.dump(snap,f,ensure\_ascii=False,indent=2)

log\_kv("counts\_pub0\_saved", tag=tag, path=str((LOG\_DIR / f"counts\_pub0\_{tag}.json").resolve()))

except Exception:

pass

return P1

try:

results\_iter = list(res)

except TypeError:

results\_iter = getattr(res, "results", None)

if results\_iter is None:

log\_kv("error", note="Unknown result shape", tag=tag)

return np.zeros((0, num\_qubits))

def try\_counts(r):

for getter in (

lambda r: r.data.meas.get\_counts(),

lambda r: r.get\_counts(),

lambda r: r.data.counts,

lambda r: r.metadata.get("counts", None),

):

try:

c = getter(r)

if c: return dict(c)

except Exception:

pass

return None

N = len(results\_iter)

P1 = np.zeros((N, num\_qubits), dtype=float)

for i, r in enumerate(results\_iter):

counts = try\_counts(r)

if not counts:

log\_kv("counts\_unavailable", tag=tag, circuit\_index=i); continue

total = max(int(sum(counts.values())), 1)

row = np.zeros(num\_qubits)

for bitstring, cnt in counts.items():

bits = to\_bits\_any(bitstring, num\_qubits)[::-1]

for q in range(num\_qubits):

if bits[q] == "1": row[q] += int(cnt)

P1[i] = row / float(total)

try:

snap = try\_counts(results\_iter[0])

if snap:

snap\_bits = {to\_bits\_any(k, num\_qubits): int(v) for k,v in snap.items()}

with open(LOG\_DIR / f"counts\_pub0\_{tag}.json","w",encoding="utf-8") as f:

json.dump(snap\_bits,f,ensure\_ascii=False,indent=2)

log\_kv("counts\_pub0\_saved", tag=tag, path=str((LOG\_DIR / f"counts\_pub0\_{tag}.json").resolve()))

except Exception:

pass

return P1

def recon\_error\_from\_P1(P1, Xref):

x\_hat = np.arcsin(np.sqrt(np.clip(P1, 1e-12, 1-1e-12)))

return ((Xref - x\_hat) \*\* 2).mean(axis=1)

def circuit\_metrics(transpiled\_list):

depths = [c.depth() or 0 for c in transpiled\_list]

ops = [c.count\_ops() for c in transpiled\_list]

total\_2q = sum(int(op.get("cx",0) + op.get("cz",0) + op.get("iswap",0)) for op in ops)

total\_1q = sum(int(sum(v for k,v in op.items() if k not in ("measure","barrier","reset","cx","cz","iswap"))) for op in ops)

total\_meas = sum(int(op.get("measure",0)) for op in ops)

return {

"n\_circuits": len(transpiled\_list),

"avg\_depth": float(np.mean(depths)) if depths else 0.0,

"total\_2q": int(total\_2q),

"total\_1q": int(total\_1q),

"total\_measure": int(total\_meas),

}

def j\_opt\_threshold(scores, labels):

fpr, tpr, thr = roc\_curve(labels, -scores)

j = tpr - fpr

return float(thr[int(np.argmax(j))])

# ---------------- Build-batch helpers ----------------

def make\_batched\_circuits\_qcaa(X1, X2, X3):

circuits = []; idx\_map = {}; start = 0

for tag, Xref in [("s1",X1),("s2",X2),("s3",X3)]:

cs = [build\_circuit\_qcaa(x) for x in Xref]

circuits.extend(cs); end = start + len(cs)

idx\_map[tag] = (start, end, Xref); start = end

return circuits, idx\_map

def make\_batched\_circuits\_opt(X1, X2, X3, mask\_vec):

circuits = []; idx\_map = {}; start = 0

for tag, Xref in [("s1",X1),("s2",X2),("s3",X3)]:

cs = [build\_circuit\_qcaa\_opt(x, mask\_vec) for x in Xref]

circuits.extend(cs); end = start + len(cs)

idx\_map[tag] = (start, end, Xref); start = end

return circuits, idx\_map

def slice\_errors(P1\_all, idx\_map):

out = {}

for tag, (s, e, Xref) in idx\_map.items():

out[tag] = recon\_error\_from\_P1(P1\_all[s:e], Xref)

return out

# ---------------- Mask for optimized ----------------

d = X\_s1.shape[1]

mask\_vec = sa\_mask(d, seed=GLOBAL\_SA\_SEED, reads=SA\_READS, min\_on=MIN\_ON)

pd.DataFrame([mask\_vec], columns=[f"f{j}" for j in range(d)]).to\_csv(LOG\_DIR / "mask\_qcaa\_opt.csv", index=False)

# ---------------- Sampler without session (Open plan compatible) ----------------

sampler = Sampler(mode=backend)

records = []

# ===== QCAA =====

t0 = perf\_counter()

qcaa\_circs, qcaa\_map = make\_batched\_circuits\_qcaa(X\_s1, X\_s2, X\_s3)

qcaa\_circs\_t = [transpile(c, backend=backend, optimization\_level=OPT\_LEVEL) for c in qcaa\_circs]

t1 = perf\_counter()

transpile\_sec\_qcaa = t1 - t0

log\_kv("transpile\_done", model="QCAA", seconds=f"{transpile\_sec\_qcaa:.3f}", n=len(qcaa\_circs\_t))

t2 = perf\_counter()

job = sampler.run(qcaa\_circs\_t, shots=SHOTS)

job\_id = job.job\_id(); log\_kv("job\_submitted", model="QCAA", job\_id=job\_id, n\_circuits=len(qcaa\_circs\_t), shots=SHOTS)

res = job.result()

t3 = perf\_counter()

exec\_sec\_qcaa = t3 - t2

log\_kv("job\_completed", model="QCAA", job\_id=job\_id, seconds=f"{exec\_sec\_qcaa:.3f}")

t4 = perf\_counter()

num\_qubits = d

P1\_qcaa\_all = extract\_P1\_and\_snapshot(res, num\_qubits, "qcaa\_batch", SHOTS)

RE\_qcaa = slice\_errors(P1\_qcaa\_all, qcaa\_map)

t5 = perf\_counter()

post\_sec\_qcaa = t5 - t4

log\_kv("postprocess\_done", model="QCAA", seconds=f"{post\_sec\_qcaa:.3f}")

m\_qcaa = circuit\_metrics(qcaa\_circs\_t)

records.append({

"model": "QCAA",

"transpile\_sec": transpile\_sec\_qcaa,

"execute\_sec": exec\_sec\_qcaa,

"postprocess\_sec": post\_sec\_qcaa,

\*\*m\_qcaa

})

# ===== QCAA-Optimized =====

t0 = perf\_counter()

opt\_circs, opt\_map = make\_batched\_circuits\_opt(X\_s1, X\_s2, X\_s3, mask\_vec)

opt\_circs\_t = [transpile(c, backend=backend, optimization\_level=OPT\_LEVEL) for c in opt\_circs]

t1 = perf\_counter()

transpile\_sec\_opt = t1 - t0

log\_kv("transpile\_done", model="QCAA-Optimized", seconds=f"{transpile\_sec\_opt:.3f}", n=len(opt\_circs\_t))

t2 = perf\_counter()

job = sampler.run(opt\_circs\_t, shots=SHOTS)

job\_id = job.job\_id(); log\_kv("job\_submitted", model="QCAA-Optimized", job\_id=job\_id, n\_circuits=len(opt\_circs\_t), shots=SHOTS)

res = job.result()

t3 = perf\_counter()

exec\_sec\_opt = t3 - t2

log\_kv("job\_completed", model="QCAA-Optimized", job\_id=job\_id, seconds=f"{exec\_sec\_opt:.3f}")

t4 = perf\_counter()

P1\_opt\_all = extract\_P1\_and\_snapshot(res, num\_qubits, "qcaa\_opt\_batch", SHOTS)

RE\_opt = slice\_errors(P1\_opt\_all, opt\_map)

t5 = perf\_counter()

post\_sec\_opt = t5 - t4

log\_kv("postprocess\_done", model="QCAA-Optimized", seconds=f"{post\_sec\_opt:.3f}")

m\_opt = circuit\_metrics(opt\_circs\_t)

records.append({

"model": "QCAA-Optimized",

"transpile\_sec": transpile\_sec\_opt,

"execute\_sec": exec\_sec\_opt,

"postprocess\_sec": post\_sec\_opt,

\*\*m\_opt

})

# ---------------- Save phase timings & summary ----------------

timings\_df = pd.DataFrame(records)

timings\_df.to\_csv(LOG\_DIR / "phase\_timings.csv", index=False)

# Baseline thresholds on Session 1 (optional completeness)

RE\_qcaa\_s1 = recon\_error\_from\_P1(P1\_qcaa\_all[qcaa\_map['s1'][0]:qcaa\_map['s1'][1]], qcaa\_map['s1'][2])

RE\_opt\_s1 = recon\_error\_from\_P1(P1\_opt\_all [opt\_map ['s1'][0]:opt\_map ['s1'][1]], opt\_map ['s1'][2])

thr\_qcaa = j\_opt\_threshold(RE\_qcaa\_s1, y[idx\_s1]) if len(RE\_qcaa\_s1) else None

thr\_opt = j\_opt\_threshold(RE\_opt\_s1, y[idx\_s1]) if len(RE\_opt\_s1) else None

# --- JSON sanitizer (PATCHED) ---

def \_to\_builtin(x):

if isinstance(x, dict):

return {k: \_to\_builtin(v) for k, v in x.items()}

if isinstance(x, list):

return [\_to\_builtin(v) for v in x]

if isinstance(x, tuple):

return tuple(\_to\_builtin(v) for v in x)

if isinstance(x, np.ndarray):

return x.tolist()

if isinstance(x, (np.integer,)):

return int(x)

if isinstance(x, (np.floating,)):

return float(x)

return x

summary = {

"RUN\_ID": RUN\_ID,

"backend": backend.name,

"shots": SHOTS,

"opt\_level": OPT\_LEVEL,

"dataset": DATA\_PATH,

"n\_session1": int(len(X\_s1)),

"n\_session2": int(len(X\_s2)),

"n\_session3": int(len(X\_s3)),

"features": feature\_cols,

"mask\_qcaa\_opt": mask\_vec, # now plain ints

"qcaa": records[0],

"qcaa\_optimized": records[1],

"baseline\_thr\_qcaa": float(thr\_qcaa) if thr\_qcaa is not None else None,

"baseline\_thr\_opt": float(thr\_opt) if thr\_opt is not None else None

}

with open(LOG\_DIR / "summary.json","w",encoding="utf-8") as f:

json.dump(\_to\_builtin(summary), f, indent=2)

log\_kv("profiling\_done", timings=str((LOG\_DIR / "phase\_timings.csv").resolve()))

print("\n✅ Profiling completed. Results saved to:", str(LOG\_DIR.resolve()))

print(timings\_df)