Toward Quantum Advantage: Discovering Hidden Structure in Peaked Circuit Distribution

Simulation-based Analysis Using BlueQubit

Presented by: Parth Danve, Soham Bopardikar, Shai Verma, Chirag Dhamange, Henry Lin

BlueQubit

Last Minute

|Y⟩Quantum

Peaked Circuits: Inspired by Random Quantum Circuits (RQC) Layout

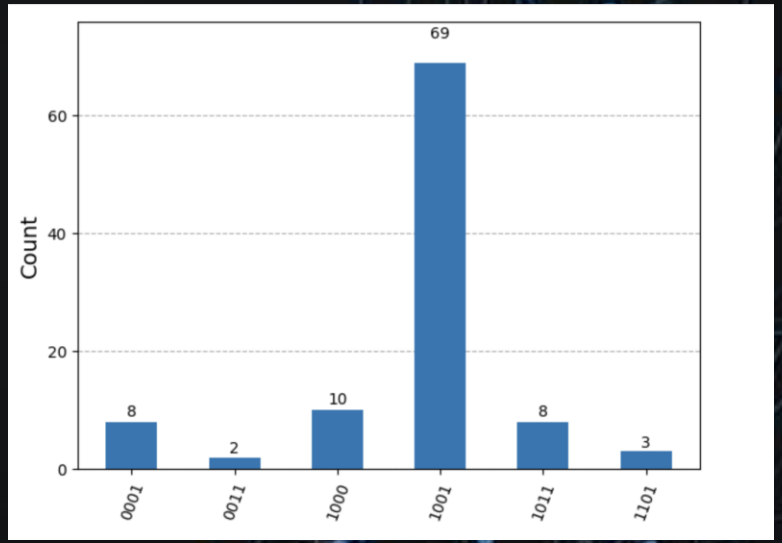
Random Quantum Circuit (RQC)	Peaked Circuit
Use layers of random single-qubit rotations and entangling gates	Maintain the same RQC-like gate structure (depth, layout, randomness)
Follow a brickwall layout and are hard to simulate classically	Are engineered so that one specific bitstring appears with $O(1)$ probability
Output distributions are nearly flat, with all bitstrings having ~equal probability	Enable efficient verification on quantum hardware by checking for the peak

The Peaked Circuit Problem

We were tasked with analysing Peaked Circuits provided in .qasm format.

Each circuit prepares a quantum state where one bitstring appears with unusually high probability.

Our mission is to find those hidden peak bitstring in each circuit utilizing BlueQubit's Quantum Simulations



Problem 1: Little Peak

This problem gives us a 4-qubit quantum circuit in .qasm format. We load it into Qiskit, add measurement gates, and run it on BlueQubit's CPU simulator.

```
qc_1 = QuantumCircuit.from_qasm_file('P1_little_peak.qasm')
qc_1.measure_all()
bq_1 = bluequbit.init("<token>")
shots_1 = 100
result_1 = bq_1.run(qc_1, device='cpu', shots = shots_1)
```

Since it's a small circuit with a strong peak in its output distribution, we only need 100 shots to reliably observe the dominant bitstring.

Problem 2: Swift Rise

This problem gives us a 28-qubit peaked circuit.

To simulate it, we use BlueQubit's CPU backend, since it's efficient enough for circuits of this size.

```
qc_2 = QuantumCircuit.from_qasm_file('P2_swift_rise.qasm')
qc_2.measure_all()
bq_2 = bluequbit.init("tbtC35WEFeL9IXYkJoeUgFIoQzYu8lQQ")
shots_2 = 1024
result_2 = bq_2.run(qc_2, device='cpu', shots = shots_2)
```

We chose 1024 shots, which gives us a high-confidence estimate of the most probable bitstring.

Running this on MPS is possible, but unnecessary here — it would be overkill.

Problem 3: Sharp Peak

- Mps.cpu => >=35 to <=96 qubits
- Bond dimension = 32

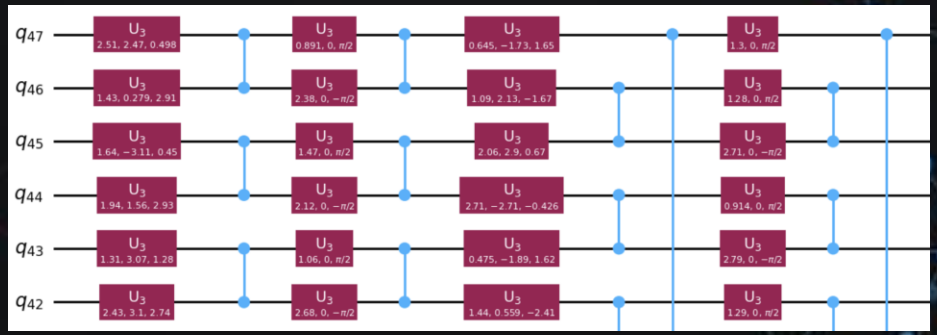
```
options={
    'mps_bond_dimension':32, 'mps_truncation_threshold':1e-16
}
bond_time_start=time()
result_3_b = bq_3.run(qc_3, device='mps.cpu', shots = shots_3, options=options)
```

```
Peaked Bitstring :  10001101010101010000011111001101000100011010
Frequency : 118
Total Shots : 1024
Peaked Amplitude : 0.33946189035000673
Time taken without bond dimension :  126.07059407234192
Time taken with bond dimension :  26.169150829315186
Speedup :  4.817527129352444
```

Problem 4, 5, 6: Vertical Circuit Cutting

Assumption: the amplitude of the peaked quantum state increases faster than non-peaked quantum states as we progress in time.

Divide the circuit into smaller blocks and run the simulation only on the first block as an approximation.



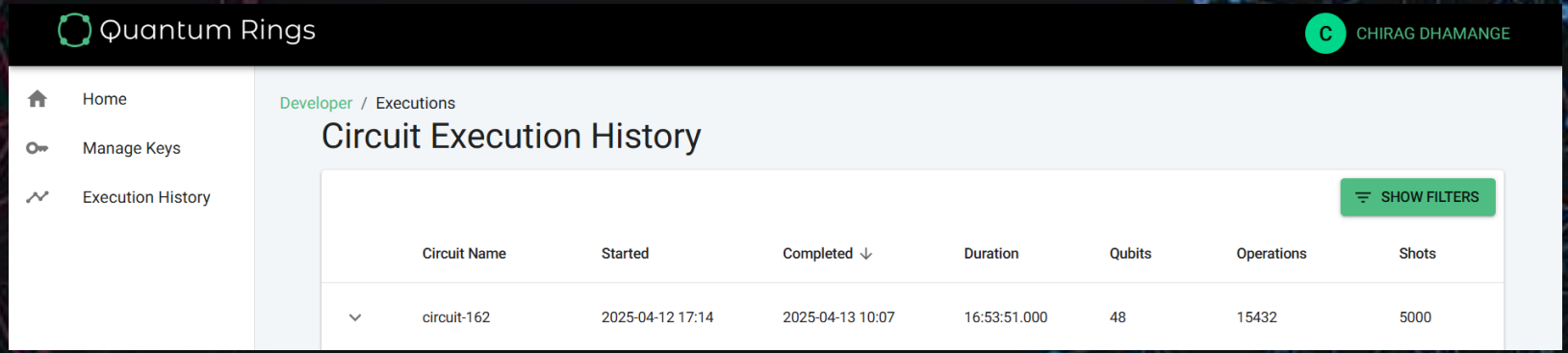
Circuit Cutting: Horizontal

- Check the connectivity of the qubits and partition them into different connected components.
- Run simulations on these different connected components.
- Concatenate the simulation result from each connect component to create the peaked quantum state.

Quantum Rings Simulation – Problem 4

```
root@cdde976ed368:/app# python hackathon.py
Start qc
here
End qc

Circuit time: 36539.146627305 seconds
00000000000000011110101001110011000111011010001 1
```

```python
import QuantumRingsLib
from QuantumRingsLib import QuantumRingsProvider
from quantumrings.toolkit.qiskit import QrBackendV2
from quantumrings.toolkit.qiskit import QrJobV1

provider = QuantumRingsProvider(
               token = "your-token",
               name = "your-name"
             )

mybackend = QrBackendV2(provider, num_qubits = qc.num_qubits)
qc_transpiled = transpile(qc, mybackend,
                          initial_layout=list(range(0, qc.num_qubits))
                         )
print("here")
job = mybackend.run(qc_transpiled, shots = 5000)

result = job.result()
```



IBM Real Backend(ibm_kyiv – 127 qubit) - Problem 4, 5, 6

Ran P5_granite_summit.qasm on a real IBM Quantum device using SamplerV2

Applied hardware-aware transpilation via generate_preset_pass_manager

Automatically selected the least busy backend and executed with 1000 shots

Retrieved and analyzed counts to identify the most likely output bitstring

```
[ ] # runtime imports
    from qiskit_ibm_runtime import QiskitRuntimeService, Session
    from qiskit_ibm_runtime import EstimatorV2 as Estimator

    # To run on hardware, select the backend with the fewest number of jobs in the queue
    service = QiskitRuntimeService(channel="ibm_quantum")
    backend = service.least_busy(operational=True, simulator=False)
```

```
[ ] backend
```

```
<IBMBackend('ibm_kyiv')>
```

```
[ ] from qiskit import transpile
    qc_transpiled = transpile(qc, backend, optimization_level=1)
```

```
from qiskit_ibm_runtime import SamplerV2 as Sampler
sampler = Sampler(backend)
qc_job = sampler.run([qc_transpiled], shots=1000)
```

Quimb – Problem 4, 5, 6

Simulated qasm file using the qiskit-quimb interface.

Converted the circuit into a **tensor network** using quimb_circuit().

Modeled it as a **CircuitMPS** for efficient sampling with limited entanglement.

Generated bitstring samples via .sample(), demonstrating scalable simulation beyond statevector limits.

[Ref: https://github.com/qiskit-community/qiskit-quimb](https://github.com/qiskit-community/qiskit-quimb)

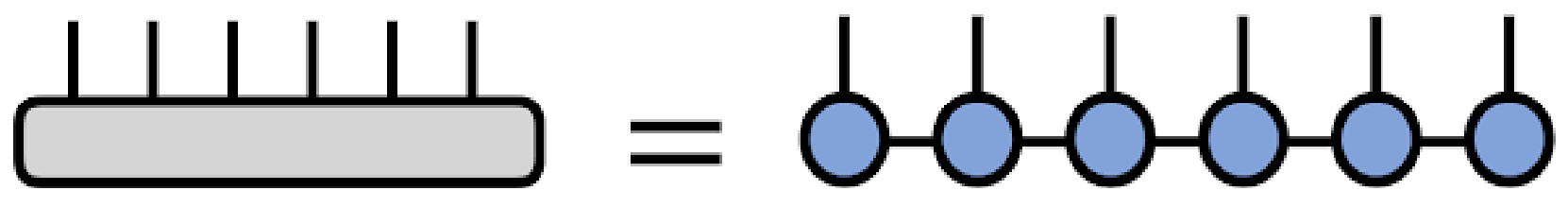
```
# Convert Qiskit circuit to Quimb tensor network format
quimb_circ = quimb_circuit(qc)
```

```
import quimb.tensor

# Use MPS simulation backend with max bond dimension and tweaked this value
quimb_circ = quimb_circuit(
    qc, quimb_circuit_class=quimb.tensor.CircuitMPS, max_bond=256
)
samples = list(quimb_circ.sample(100000, seed=1234))

samples #Random distinct samples for every run
```

R

MPS Simulation – Problem 4

MPS – Matrix Product States



first and last tensors have one fewer auxiliary index

$$\Psi^{n_1 n_2 n_3 \ldots n_l} = \sum_i A^{n_1}_{i_1} A^{n_2}_{i_1 i_2} A^{n_3}_{i_2 i_3} \cdots A^{n_l}_{i_l}$$

"bond" or "auxiliary" dimension
"M" or "D" or "χ"

1D structure of entanglement

$$= \mathbf{A}^{n_1} \mathbf{A}^{n_2} \ldots \mathbf{A}^{n_l}$$

amplitude is obtained as a product of matrices

MPS Simulation – Problem 4

It truncates our search space, helping us solve problems computationally much faster at the cost of some accuracy. But given the nature of our problem we can afford to do this

Bond Dimension gives a measure of how much entanglement is captured in the circuit.

MPS Simulation – Problem 4

Idea is to start with a lower bond dimension (say 4) and we analyze the distribution. Usually for a large search space like this it will be uniform. We keep on doubling the bond dimension so as to see a peak.

MPS Simulation – Problem 4

```
qc = QuantumCircuit.from_qasm_file('P4_golden_mountain_no_cz.qasm')
qc.measure_all()
bq = bluequbit.init("tbtC35WEFeL9IXYkJoeUgFIoQzYu8lQQ")
options={
    'mps_bond_dimension':32, 'mps_truncation_threshold':0.3
    }
result = bq.run(qc, device='mps.cpu', shots=10000, options=options)
```

```
qc1.measure_all()

# Create an Aer simulator configured for MPS simulation
simulator = AerSimulator(method="matrix_product_state",
                         matrix_product_state_max_bond_dimension=64, matrix_product_state_truncation_threshold=10e-16, shots=10000000)


result = simulator.run(qc1).result()
```

Thank you so much:)

Make sure to checkout our code on GitHub

https://github.com/pdd23001/YQuantum2025

Questions?