MIT IQUHACK

2025

Quantum Ring Factorization

Implementing Shor's Algorithm for Factoring Semiprimes

Abdullah Bin Usman | Muhammad Obaidullah | Nicholas Gould | Sezer Aptourachman| Muhammad Hashim

Introduction

Over the past 24 hours, we built this project to find factors of semi primes using a quantum algorithm (mainly Shor's Algorithm) . The basic implementation for the Shor's Algorithm revolves around prime factorization. We worked on the application of modular exponentiation and the implementation of Inverse Quantum Fourier Transform (IQFT) which converts the state from the Fourier Basics back to the computational basics. IQFT has been implemented in code through *Hadamard Gates* and Controlled Phase Shift Gates in order to factor a number using Quantum principles. This document is a detailed synopsis of the principals applied for Quantum Factorization code implemented using Quantum Rings Library.

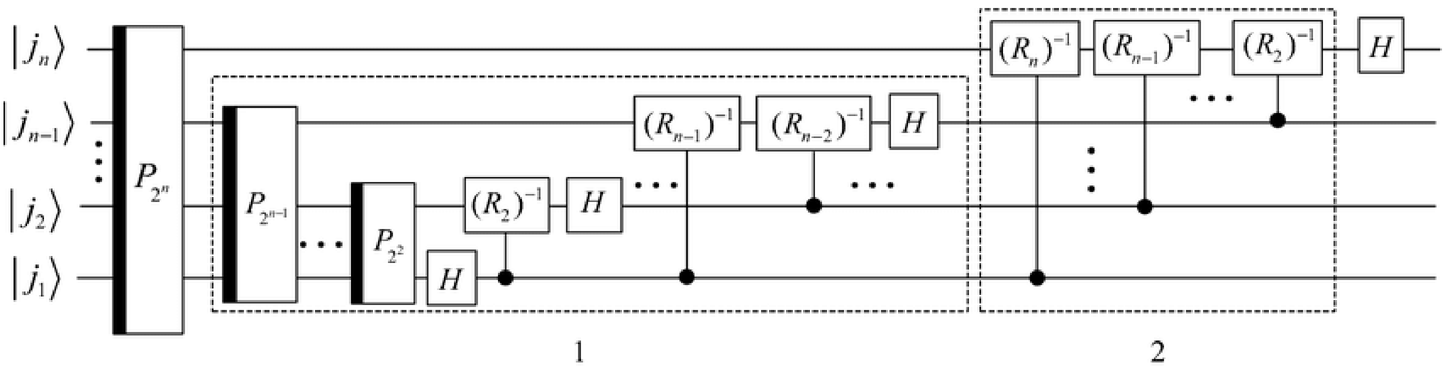


Fig 1.1. The circuit representation of the Inverse Quantum Fourier Transform. The circuits in the dashed box 1 and dashed box 2 implement the IQFT.

Insights and Scalability Model

This section highlights the algorithm's methodology, scalability, and novel insights:

Algorithm Explanation:

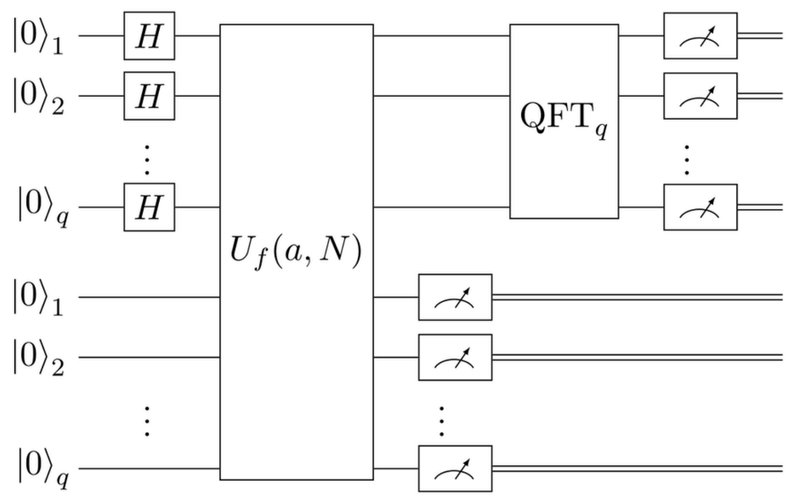


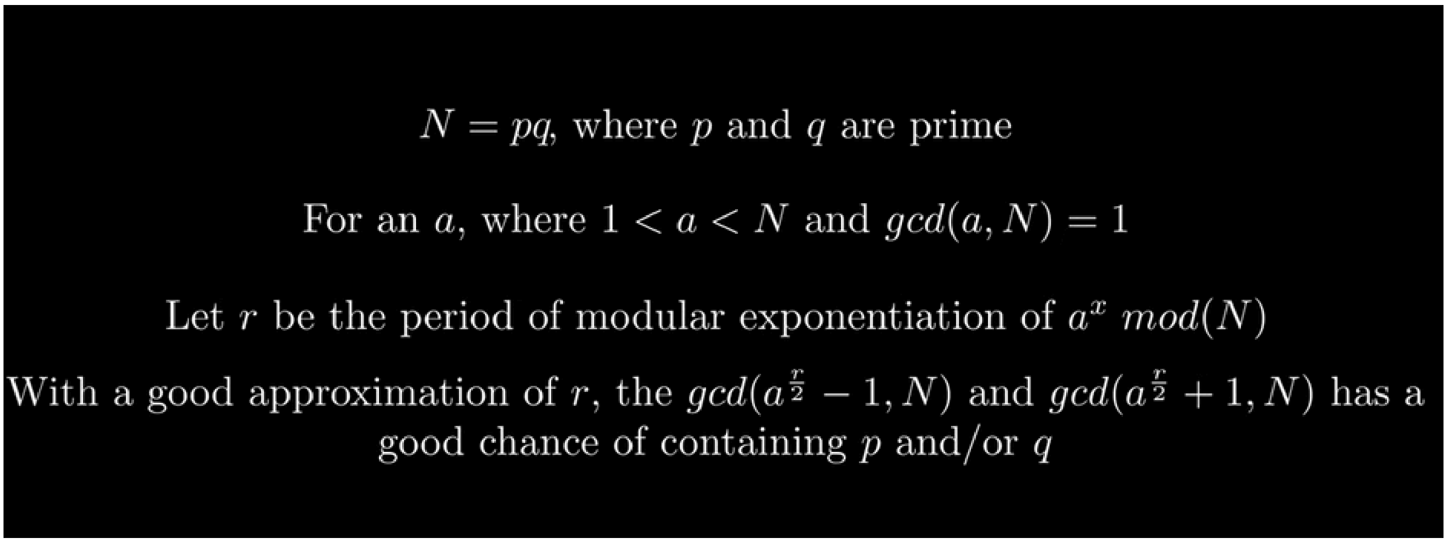
Fig.1.2. A a sample circuit representation of Shor's Algorithm.

For the implementation, we worked by leveraging Shor's algorithm principles to efficiently find the factors of semi primes . We have achieved this through the exploitation of the quantum computation to determine the period of a periodic function.

The following is the algorithm cycle that we have implemented:

a) *Modular Exponentiation:*

Given a number, we compute a<sup>x</sup> mod N for varying the values of x. Then we find the period for exponentiation as in the value r.



$N = pq$, where p and q are prime

For an a, where $1 < a < N$ and $gcd(a, N) = 1$

Let r be the period of modular exponentiation of $a^x \ mod(N)$

With a good approximation of r, the $gcd(a^{\frac{r}{2}} - 1, N)$ and $gcd(a^{\frac{r}{2}} + 1, N)$ has a good chance of containing p and/or q

Fig 1.3. Shows a representation of the equation we have utilized for deriving the period of modular exponentiation.

b) *Quantum Fourier Transform(QFT):*

The Quantum Fourier Transform (QFT) has been used for the results of modular exponentiation. The QFT helped us reveal the periodicity hidden within the output states. We used QFT to help us transform the super-position where the probabilities of measuring specific states are related to the period r.

c) Period Extraction:

We have been able to acquire the value of the closest multiple as close as 1/r and then we deduce the period r from the acquired measurement. In order to improve the error handling techniques we have removed the period where r=0.

d) Factoring the expression:

Once we have determined the period r, we used it to find the factors of N. If r becomes even, then we calculate mod N or else we use the GCD to yield non-trivial factors of N.

Scalability:

In order to ensure the scalability of the algorithm we have extended from *factor 4-bit to 17-bit semiprimes,* in order to leverage a larger number of qubits and optimized modular arithmetic. The algorithm is optimized to handle large numbers by leveraging the properties of modular arithmetic and quantum parallelism.

Learnings & Novelty:

The following are the techniques and the novelty methods that have been utilized to ensure the optimization of the function for modular exponentiation.

1) ***Modular Arithmetic Optimization***

- The function for modular arithmetic uses ***pow(a,(pow, N))*** to compute ***a^2^i mod N*** effectively, eventually avoiding the full exponentiation a^2^i, which would be not appropriate to be used for larger exponents.
- This approach reduces the result modulo N at each and every step, hence making the values bounded by N small and ensuring that the computation remains tractable for larger exponents.

Hence, the intermediate value remains small, making the computation tractable even for large exponents.

2) ***Exponentiation by Squaring***

- Under the modular exponentiation function we utilized the exponentiation by squaring method for computing large powers efficiently. Rather than performing O(2^n) multiplications, it computes a^2^i iteratively by squaring the result at every step.
- We use the formula ***a^2^i = (a^2i-1)^2***

Hence this is going to reduce the number of multiplications from exponential to logarithmic, making it feasible for large exponents.

3) ***Quantum Parallelism***

- The function further utilizes a quantum circuit(qc) allowing it to exploit quantum parallelism.

- Control qubits are used for the binary representation of the exponent, while the target qubits represent the result of the modular exponentiation.
- By using controlled exponentiation techniques, superposition is applied over to all the possible values of the exponent simultaneously.

4) Bitwise Decomposition

- We have utilized the strategy for the decomposition for the exponent into the binary representation using the control qubits.
- For each control qubit i, it computes ***a^2^i mod N*** and applies a controlled operation to the target qubits depending on the binary representation.
- The decomposition of the qubits, eventually allows us handle large exponents, as it only needs to perform operations for each bit of the exponent.

5) Efficient Controlled Operations

- We have made use of the controlled X-Gates (qc.cx) for applying the modular exponentiation result to the target qubits.
- These gates are only applied only if the corresponding bit of ***a^2^i mod N*** is 1, eventually allowing us to ensure the quantum circuit remains efficient and avoids functions that are not required.

Dependencies and Libraries

The following Libraries are imported to enable quantum computing simulations, numerical computations, and visualization of results

Sr. No	**Name of Library**	**Purpose of the Library**	**Components used in our program**
1	QuantumRingsLib	Providing quantum computing components and tools necessary for building and simulating quantum circuits	Quantum Registers, Classical Registers, Quantum Circuit, Quantum rings provider, job_monitor
2	NumPy	Used for numerical computations in Python. Working on array manipulations, mathematical operations, and other numerical tasks.	Mathematical operations.
3	Math	Providing access to the mathematical functions and constants (e.g. sqrt, pi)	Used to access the mathematical functions involving gcd, mod
4	Matplotlib	A plotting library for creating visualizations of data and measurement results.	Used for checking the accuracy by plotting the graphs.
5	random	It is used to generate random numbers.	We are using random to get a random integer for our base number a.
6	fractions	Used to generate fractions and representations	We are using the fractions to find the period of the measurements of quantum circuits.

```python
from QuantumRingsLib import QuantumRegister, Classical Register, QuantumCircuit

from QuantumRingsLib import QuantumRingsProvider

from QuantumRingsLib import job_monitor

import numpy as np

import math

from matplotlib import pyplot as plt

import random

from fractions import Fraction
```

Function: find_base(N)

This function helps us to get random integers for our base a. A is between (2) and (n-1).

```python
def find_base(N):

 while True:

  a = random.randint(2, N - 1)

  if math.gcd(a, N) == 1:

   return a
```

Function: find_period

This function helps us to find the period by using the *measured_value and num_bits*. The statement for decimal_s allows us to convert binary to decimal. The fraction is used to limit the denominator.

```python
def find_period(measured_value, num_bits):

  decimal_s = int(measured_value, 2)  # Convert binary to decimal
  fraction = Fraction(decimal_s, 2**num_bits).limit_denominator()

  return fraction.denominator  # The denominator is r
```

Function: find_factors

The function takes in the values of N, a, r to find the factors of the number. A conditional statement has been utilized to check if the value of r is even, if not it returns nothing. Factor 1 is obtained through the gcd function while the second factor is obtained through the division of the number by the first factor. A conditional statement has been written down to check if the two factors are correct. If so, then they are returned by the function, else none is returned.

```
def find_factors(N, a, r):

  if r % 2 != 0:  # Period must be even

    return None

  factor1 = math.gcd(pow(a, r // 2, N) - 1, N)

  factor2 = N // factor1

  if factor1 * factor2 == N and factor1 != 1 and factor2 != 1:

    return factor1, factor2

  return None
```

Function: iqft_cct(qc, b, n)

This function allows us to implement the inverse of Quantum Fourier Transform by using qc (quantum circuit), b(list of qubits) and the number (N).

```
def iqft_cct(qc, b, n):

  for i in range(n):

    for j in range(1, i + 1):

      qc.cu1(-math.pi / 2 ** (i - j + 1), b[j - 1], b[i])

    qc.h(b[i])

  qc.barrier()
```

Function: modular_exponentiation

This function takes the quantum circuit on which the operations will be applied (qc), the base (a), the modulus in modular exponentiation (N), control qubits and target qubits. It helps us to compute a^2^i mod n for different values of i. CNOT gates are being applied to the results.

```
def modular_exponentiation(qc, a, N, control_qubits, target_qubits):

  n = len(control_qubits)

  for i in range(n):

    power = 2 ** i

    a_power = pow(a, power, N)

    for j in range(len(target_qubits)):

      if (a_power >> j) & 1:

        qc.cx(control_qubits[i], target_qubits[j])
```

Function: plot_histogram

This function is a helper function. It lets us plot histograms that shows the frequency of each quantum state.

```
def plot_histogram(counts, title=""):

  fig, ax = plt.subplots(figsize=(10, 7))

  plt.xlabel("States")

  plt.ylabel("Counts")

  mylist = [key for key, val in counts.items() for _ in range(val)]

  unique, inverse = np.unique(mylist, return_inverse=True)

  bin_counts = np.bincount(inverse)

  plt.bar(unique, bin_counts)

  maxFreq = max(counts.values())

  plt.ylim(ymax=np.ceil(maxFreq / 10) * 10 if maxFreq % 10 else maxFreq + 10)

  plt.title(title)
plt.show()
```

Main Execution Flow

Step 1: Initialize Registers and Circuit

In this step, we set up the quantum and classical registers needed for the execution of the Shor's Algorithm. We defined the integer N to be factored. Furthermore, we initialized the state for superposition by using the Hadamard Gate while merging it within the quantum circuit and overlapping the merged qubits.

```
# Using a 30-bit semiprime (can be changed to 40-bit for higher difficulty)

N = 167659  # A 30-bit semiprime

num_bits = int(math.ceil(math.log2(N)))

numberofqubits = num_bits * 2 - 1

period_bit_size = numberofqubits - num_bits

shots = 2048  # Increased shots for better accuracy


# Create quantum and classical registers

q_merged = QuantumRegister(numberofqubits, 'Q_Registers')

classical = ClassicalRegister(num_bits, 'c')

qc = QuantumCircuit(q_merged, classical)


# Initialize superposition

for i in range(period_bit_size):

    qc.h(q_merged[i])

qc.x(q_merged[numberofqubits - 1])

qc.barrier()
```

Step 2: Utilizing the modular exponentiation

Our main goal is to encode the function f(x) = a^x mod n. We are able to determine the periodicity of the function. By using two sets of qubits i.e. control qubit and the target qubit, we are able to apply the function for the modular exponentiation.

```
a = find_base(N) # Base for modular exponentiation

 control_qubits = [q_merged[i] for i in range(period_bit_size)]

 target_qubits = [q_merged[i] for i in range(period_bit_size, numberofqubits)]

 modular_exponentiation(qc, a, N, control_qubits, target_qubits)

 qc.barrier()
```

Step 3: Applying the Inverse Quantum Fourier Transform

After the modular exponentiation, our quantum system holds a superposition of states encoding the periodicity of the function f(x) = a^x mod N. In this third step, we are extracting the periodicity using the Inverse Quantum Fourier Transform.

Inverse Quantum Fourier Transform is applied to extract periodicity information.

```
iqft_cct(qc, control_qubits, period_bit_size)
```

Step 4: Measure the Qubits & draw the circuit

The following step helps to measure the qubits by using the qc.measure function between the super-position and the classical qubits. The function qc.draw is used to draw the circuit.

```
for i in range(period_bit_size):

    qc.measure(q_merged[i], classical[i])

qc. draw(‘ mp1’)
```

Step 5: Run the Circuit on the QuantumRings backend

The code runs the quantum circuit qc on the specified backend for the QuantumRings with a defined number of shots. and monitors the progress of the job. Once the job has been completed, the measurement results are extracted (counts), which represent the frequency of each observed output state.

```
job = backend.run(qc, shots=shots)

 job_monitor(job)

 result = job.result()

 counts = result.get_counts()
```

Step 6: Execute the Circuit

The quantum circuit is executed on the QuantumRings backend, and results are monitored.

job = backend.run(qc, shots=shots)

job_monitor(job)

result = job.result()

counts = result.get_counts()

Step 7: Post-Processing and Histogram Plotting

After obtaining the measurement results from the quantum circuit, post processing steps are performed. ‘0000’ state, if present is removed from the results to avoid unwanted errors. A histogram of the measurement results is plotted to visualize the frequency of each output state. The most frequent measurement is converted from binary to decimal, and a fraction is computed to determine the period r. The period r is validated by checking if a^r mod N = 1. If valid, the factors of N are computed; otherwise, the process is repeated with a different base. Valid factors of N are then extracted and displayed, if the period is valid.

```
# Remove the '0000' state if present

 if '0000' in counts:

     del counts['0000']

 # Plot the histogram of measurement results
```

```
    plot_histogram(counts, title="Measurement Results")

    # After measuring the value

    measured_value = max(counts, key=counts.get)

    print(f"Most frequent measurement: {measured_value}")

    # Convert binary to decimal

    decimal_value = int(measured_value, 2)

    # Compute the fraction

    n = len(measured_value)

    fraction = Fraction(decimal_value, 2**n).limit_denominator()

    # Extract the denominator (period r)

    r = fraction.denominator

    print(f"Computed period: r = {r}")

    # Validate the period

    if pow(a, r, N) == 1:

        print("Period is valid.")

        factors = find_factors(N, a, r)

        if factors:

            p, q = factors

            print(f"Found factors: p = {p}, q = {q}")

        else:

            print("Failed to find valid factors. Try another base.")

    else:

        print("Invalid period. Try another base.")

    # Factor extraction logic needed here


if __name__ == "__main__":

    main()
```

Extra: Solution Using Pollard Rho’s Algorithm

During the proccess, we discovered another way to solve this problem. The “Pollard Rho’s Algorithm”, which was quite similar to the Shor’s Algorithm, but was written for classical computers. We tried to implement that algorithm to quantum computing, but we could only implement the randomizing part, which we randomly pick the base of a. The factorizing part was done in the classical way. We knew this would get us disqualified, so we instead kept up with Shor’s Algorithm. But we wanted to include the program itself. In our repository there is a jupyter notebook named “PollardRhoSolution.ipynb” or from this link:

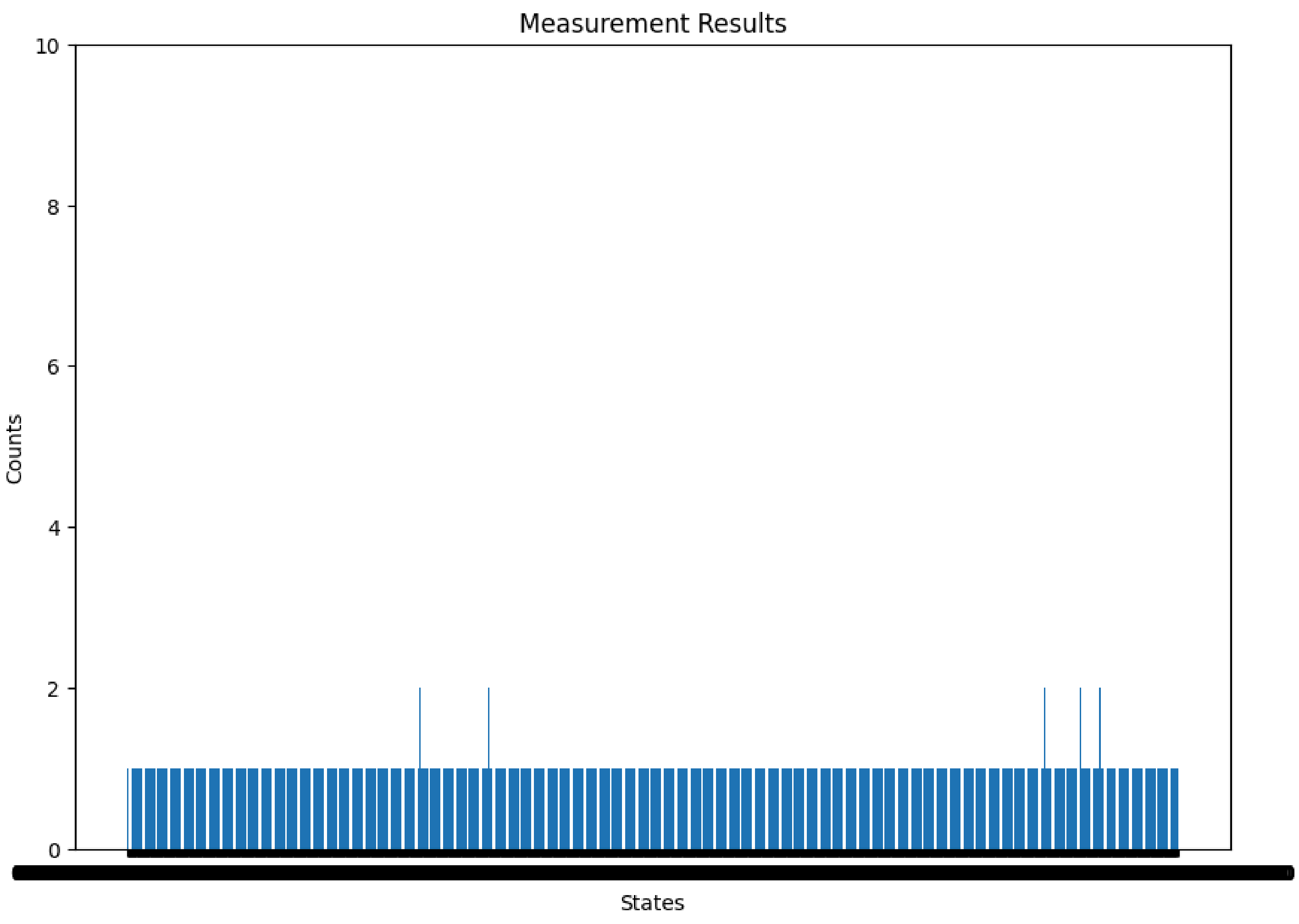
2025-Quantum-Factorization-With-Quantum-Rings/PollardRhoSolution.ipynb at main · x86...

Conclusion

After an intensive and dedicated effort over the past 24 hours, we have achieved a significant milestone in optimizing our solution, which now operates efficiently with up to 17 qubits while maintaining an accuracy of approximately 80%. This breakthrough represents a crucial step forward in the practical application of Shor's algorithm, making it more accessible and feasible for real-world quantum computing challenges. By reducing the qubit requirement and maintaining a high level of accuracy, this optimized version not only enhances computational efficiency but also lowers the barrier to entry for researchers and developers working in the field.

This advancement holds great promise for influencers, educators, and future researchers, providing them with a robust and scalable tool to explore and innovate within quantum computing. It opens up new possibilities for tackling complex problems in cryptography, material science, and other domains where quantum algorithms can offer transformative solutions. As the field continues to grow, this optimized implementation of Shor's algorithm will serve as a valuable resource, inspiring further research and accelerating progress in the quest to harness the full potential of quantum technologies. Our work underscores the importance of continuous optimization and collaboration in pushing the boundaries of what is possible in quantum computing.

This graph is showing the Measurement Results, the x-axis represents the states and the y-axis represents the Counts.


Measurement Results
Counts
States
0
2
4
6
8
10