# Optimization and Performance Analysis of Shor's Algorithm in Qiskit

Dewang Sun, Naifeng Zhang, Franz Franchetti
*Carnegie Mellon University*
Pittsburgh, USA
{dewangs, naifengz, franzf}@andrew.cmu.edu

***Abstract*—Shor's algorithm is widely renowned in the field of quantum computing due to its potential to efficiently break RSA encryption in polynomial time. In this paper, we optimized an end-to-end library-based implementation of Shor's algorithm using the IBM Qiskit quantum library and derived a speed-of-light (i.e., theoretical peak) performance model that calculates the minimum runtime required for executing Shor's algorithm with input size $N$ on a certain machine by counting the total operations as a function of different numbers of gates. We evaluated our model by running Shor's algorithm on CPUs and GPUs, and simulated the factorization of a number up to 4,757. Comparing the speed-of-light runtime to our real-world measurement, we are able to quantify the margin for future quantum library improvements.**

***Index Terms*—Quantum computing, Shor's algorithm, quantum Fourier transform, performance analysis**

## I. Introduction

In recent years, significant advancements have been achieved in the field of quantum computing. In 2022, IBM launched the quantum computer Osprey which can hold 433 quantum bits (qubits), and plans to launch Condor in 2023, the first universal quantum computer with more than 1,000 qubits [1]. With more powerful quantum computers, quantum computing algorithms can be better studied and tested. Shor's algorithm [2] has been one of the most famous quantum computing algorithms. It can factorize a semi-prime number $N$ (i.e., a natural number that is the product of two primes) with $L$ binary bits, where $L = \lceil \log_2 N \rceil$, in polynomial time. This ability makes Shor's algorithm potentially possible to break the classical public-key cryptography scheme such as the RSA scheme and elliptic curve cryptography (ECC) [3].

The most popular approaches to implementing Shor's algorithm use an extensive amount of quantum Fourier transforms (QFTs), each requiring $O(L^2)$ gates. When the phase shift of a rotation gate gets extremely small in the QFT circuit, the effect of the rotation gate is negligible. Barenco et al. [4] therefore proposed the approximate QFT algorithm that uses few gates but still has comparable performance as the exact QFT algorithm for better runtime performance. In particular for Shor's algorithm, Beauregard [5] introduced approximate QFT as an optimization to reduce the circuit depth, where we ignore the controlled-phase gates if the rotation angle is smaller than a specified threshold. This optimization reduced the gates required for QFT from $O(L^2)$ to $O(L \log L)$. Coppersmith [6] discussed the theoretical error introduced regarding the specified threshold. Currently, to the best of our knowledge, very few published work implements and studies the performance and correctness influence of approximate QFT on Shor's algorithm.

Numerous platforms and software are now available to implement quantum algorithms. Qiskit [7] is an open-source development kit for working with quantum computers that provides a user-friendly Python interface to build quantum circuits, run experiments on local or remote simulators, and connect to IBM's quantum computers. Numerous publications discussed the simulation of Shor's algorithm using Qiskit but few of them analyze the runtime performance of simulating Shor's algorithm on actual hardware. In this work, we first implement Shor's algorithm using approximate QFT in Qiskit and then benchmark and analyze the performance of the implementation of Shor's algorithm on CPU and GPU, aiming to quantify the headroom for future improvements.

**Contributions.** Our key contributions are:

- Firstly implementing in Qiskit and open-sourcing the optimization to reduce the depth of the quantum circuit using approximate QFT proposed by Beauregard.
- Empirically verifying the accuracy of the approximate QFT-based implementation against the exact QFT-based implementation.
- Deriving a speed-of-light model of the minimum runtime required for Shor's algorithm simulation in Qiskit. The model is generalizable to different libraries and platforms.
- Benchmarking the runtime performance of Shor's algorithm simulation using Qiskit on both CPU and GPU machines and comparing it with our speed-of-light model to quantify the margin for future improvements.

## II. Background

### A. Shor's algorithm: Theory

Shor's algorithm [2] is one of the most well-known quantum computing algorithms for factorizing a semi-prime number. Classical algorithms have exponential complexity, while Shor's algorithm reduces the complexity to polynomial.

In Shor's algorithm, we first pick a random base $a$ between 2 and $N-1$. If $a$ and $N$ are not co-prime, then we can find our factor using Euclid's algorithm. If $a$ and $N$ are co-prime, we know that there exists a period $r$, such that

$$a^r \equiv 1 \mod N$$

That is, $a^r - 1$ is a multiple of $N$. If the period $r$ is even, we have

$$a^r - 1 = (a^{\frac{r}{2}} - 1)(a^{\frac{r}{2}} + 1)$$

as a multiple of $N$. Then, computing $gcd(N, a^{\frac{r}{2}}+1)$ may lead to a non-trivial factor of $N$ (i.e., not 1 or $N$). If not, we rerun the algorithm from the start. In this way, we turn the factoring problem into an order-finding problem.

The quantum part of Shor's algorithm solves the order-finding problem by applying quantum phase estimation (QPE), which we lay out in the mathematical details below. The quantum circuit of Shor's algorithm is shown in Fig. 1. We define the unitary operator $U$ as

$$U|y\rangle = |ay \ mod \ N\rangle$$

In QPE, we use two quantum registers and initialize them to $|0\rangle$ and $|\psi\rangle$. Then we apply the Hadamard gate to all qubits in the upper register, as shown in stage 1 in Fig. 1, the state then becomes

$$\begin{aligned} |\psi_1\rangle &= \frac{1}{\sqrt{2^t}} \sum_{j=0}^{2^t-1} |j\rangle |\psi\rangle \\ &= \frac{1}{\sqrt{2^t}} (|0\rangle + |1\rangle)^{\otimes t} |\psi\rangle . \end{aligned}$$

Suppose we have a unitary operator $U$ such that

$$U|\psi\rangle = e^{2\pi i} |\psi\rangle ,$$

applying controlled unitary operation as stage 2 in Fig. 1, the state becomes

$$\begin{aligned} |\psi_2\rangle &= \frac{1}{\sqrt{2^t}} \bigotimes_{i=0}^{t-1} (|0\rangle + e^{2\pi i\theta 2^i} |1\rangle) |\psi\rangle \\ &= \frac{1}{\sqrt{2^t}} \sum_{k=0}^{2^t-1} e^{2\pi i\theta k} |k\rangle |\psi\rangle . \end{aligned}$$

Then by applying inverse QFT to the upper register, we have

$$|\psi_3\rangle = \frac{1}{2^t} \sum_{x=0}^{2^t-1} \sum_{k=0}^{2^t-1} e^{-\frac{2\pi i k}{2^t}(x - 2^t\theta)} |x\rangle |\psi\rangle .$$

If we measure the upper register, there will be peaks at $x = 2^t\theta$. It is highly possible that we can obtain the phase from the measured state.

For Shor's algorithm, we further define the state $|u_s\rangle$ as

$$|u_s\rangle = \frac{1}{\sqrt{r}} \sum_{k=0}^{r-1} e^{-\frac{2\pi i s k}{r}} |a^k \ mod \ N\rangle ,$$

where $r$ is the period that we are trying to find, and $0 \leq s \leq r-1$. It can be shown that

$$U|u_s\rangle = e^{\frac{2\pi i s}{r}} |u_s\rangle$$

and

$$\frac{1}{\sqrt{r}} \sum_{s=0}^{r-1} |u_s\rangle = |1\rangle .$$

Therefore, we can initialize the lower register to state $|1\rangle$ and apply QPE. The measured state will have peaks at

$$x = \frac{2^t s}{r}.$$

We then use the continued fraction algorithm to find the value of $r$ [8].

### B. *Shor's algorithm: Implementation*

One main difficulty in implementing Shor's algorithm is the implementation of modular exponentiation, which is the unitary operator $U$ we defined in the section above. Vedral et al. [10] provided the quantum circuit construction for elementary arithmetic operations like addition, modulo, and modular exponentiation. Using Vederal's quantum circuit, it is possible to implement Shor's algorithm using $7L$ qubits, where $L$ is the number of bits needed to represent the to-be-factored number in binary form. This approach is known as the *VBE* approach. Beauregard provided optimizations for Vedral's algorithm and reduced the number of qubits needed to $2L + 3$. We studied Shor's algorithm based on Beauregard's approach, which adapts 3 main optimizations based on the VBE approach:

1) Using the QFT adder proposed by Draper [11] to eliminate the carry bits used in the adder gate of the VBE approach.
2) Using semi-classical QFT, where the controlling qubit is reused and the bits of the answer are measured sequentially. This reduces the upper register to 1 qubit.
3) Using approximate QFT, where some controlled-phase gates in QFT are ignored. This reduces the gates needed.

### C. *Qiskit*

Qiskit is an open-source software development kit for working with quantum computers at the level of pulses, circuits, and application modules. Qiskit provides Python API for users to construct quantum circuits and specify parameters for quantum simulators. Qiskit Aer, a C++ library, is an element of Qiskit that provides high-performance quantum computing simulators with realistic noise models. The entire pipeline shown in Fig. 2 demonstrates how a quantum circuit is converted from Python API representation to a C quantum object (QObject) that can be simulated in Qiskit Aer.

## III. Related Work

Numerous work has been developed using Qiskit and run on quantum computers. DeCusatis et al. [12] implemented Shor's algorithm using Qiskit to factorize the number 15 with 4 qubits. They ran the circuit on an IBM Q System One quantum computer and found that due to the effect of noise, it was difficult to correctly interpret the result with 4 qubits. Skosana et al. [13] provided a demonstration of Shor's algorithm for factorizing 21 with 5 qubits on multiple IBM quantum processors. Rossi et al. [14] implemented a general quantum circuit for Shor's algorithm with approximations to reduce the circuit depth and factorized up to 57 on an IBM quantum

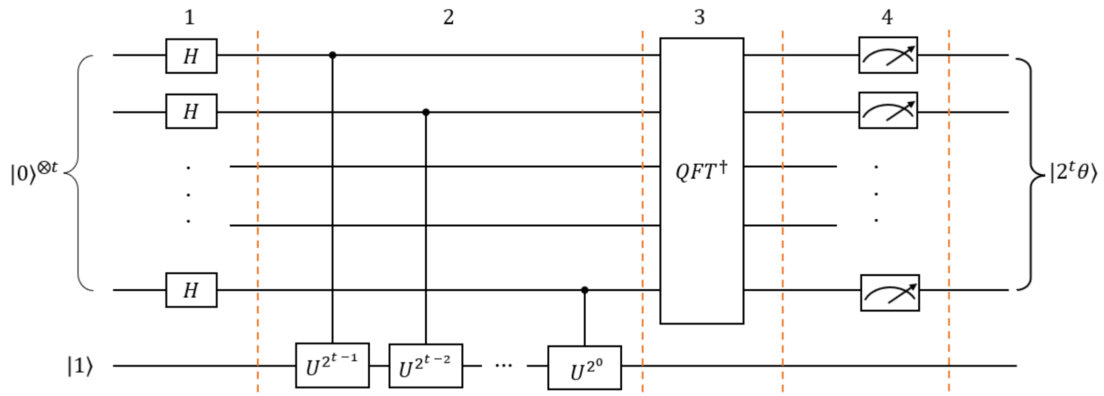



Fig. 1: Quantum circuit of Shor's algorithm, redrawn from [9]. $QFT^{\dagger}$ stands for inverse QFT.

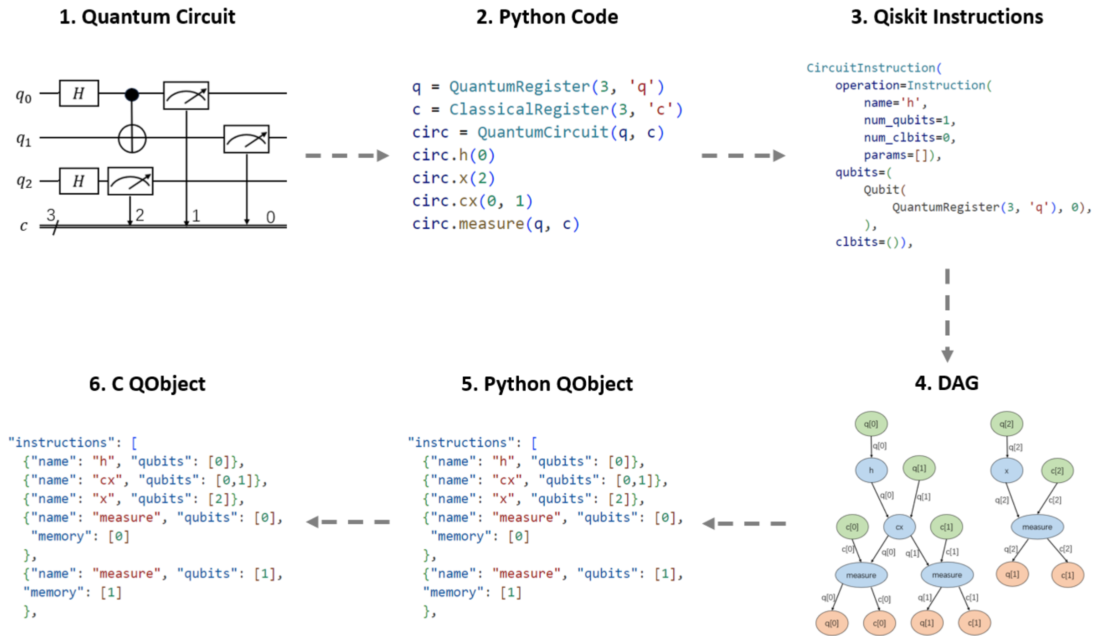



Fig. 2: Data pipeline of Qiskit. First, the user needs to build the quantum circuit (Stage 1) using Qiskit's Python API (Stage 2). Internally, the circuit expressed via the API is stored as Qiskit instructions (Stage 3) that will be translated to a directed acyclic graph (DAG) and eventually to an optimized quantum object (QObject) during the transpilation. The returned QObject is initially compatible with Python (Stage 5) and will be changed to C-compatible to interact with Qiskit Aer (Stage 6), the C++ simulator.

processor. They claimed that current quantum hardware has limited connectivity and fidelity issues.

There are also various studies focused on simulating Shor's algorithm on a classical computer via different approaches. Nam [15] studied Shor's algorithm based on the VBE approach. The author implemented a non-unitary version to reduce the qubits needed and studied the probability spectrum. Larasati et al. [16] simulated the VBE approach using Qiskit, they found Shor's algorithm incorporating this approach can be constructed but not simulated due to a significant amount of lines of code in QASM, a quantum programming language. Leao et al. [17] provided an implementation of Beauregard's paper using Qiskit, with the first two optimizations that we listed in Section II. In this work, we extended their work by further implementing the approximate QFT optimization. Yamaguchi et al. [18] implemented and simulated Shor's algorithm on a distributed GPU cluster for $N$ up to 511. They used 3 different ADD gate implementations with $5L+1$ and $4L+2$ qubits needed, respectively. The authors also estimated the quantum resources needed for different implementations to factorize a 2048-bit integer. Our work further evaluated the runtime performance on different hardware and calculated the headroom of improvements.

## IV. Approximate QFT Implemetation

Beauregard [5] derived the optimized QFT implementation based on Coppersmith's theoretic work [6], which proposed a threshold $k_{\text{max}}$ that is defined as

$$k_{\text{max}} = O\left(\log \frac{L}{\epsilon}\right),$$

where $\epsilon$ is the error.

As an initial attempt, we empirically chose $k_{\text{max}} = \log_2 n$ and enforced this threshold in our implementation. The core of our implementation is shown in Listing 1. To implement approximate QFT, we apply the threshold $k_{\text{max}}$ to the angle of the controlled-phase gate (line 4) and prune the gates with the parameter of an angle larger than the threshold. We open-sourced our implementation on GitHub[1].

```
while i>=0:
    circuit.h(up_reg[i])
    j=i-1
    while j>=0 and (i-j)<=kmax:
        if (np.pi)/(pow(2,(i-j))) > 0:
            circuit.cp(
                (np.pi)/(pow(2,(i-j))),
                up_reg[i],
                up_reg[j]
            )
            j=j-1
    i=i-1
```

Listing 1: Approximate QFT implementation in Qiskit.

## V. Performance Modeling

We conducted the performance modeling and analysis on the exact QFT implementation of Shor's algorithm as it is more widely adopted, while our performance model can be easily generalized to different implementations, libraries, and machines.

### A. Simulation Limit

Using exact QFT, our simulated quantum circuit uses $2L+3$ qubits. Thus, $2^{2L+3}$ states need to be stored in the memory while each state takes $M_s$ bytes memory. As the C++ simulation library, Qiskit Aer uses `std::complex<double>` to store a single state, each state takes 16 bytes in memory. For a number with $L$ binary bits, the memory $M$ needed is

$$\frac{2^{2L+3}M_s}{1024^3} \quad GB.$$

### B. Speed-of-light Analysis for Runtime Performance

The theoretical runtime performance of each gate in simulation is dominated by the number of floating point operations of the computation. To determine the speed-of-light (i.e., theoretical peak) performance of Shor's algorithm in Flop/s, we analyzed the number of gates needed in the quantum circuit and multiplied the number of each type of gates accordingly with the floating point operations required by each gate. We chose the CPhase Gate, CNOT Gate, H Gate, Phase Gate, X Gate, and CSwap Gate as the base gates. The number of gates in the exact QFT quantum circuit is shown in Table I.

TABLE I: Gate count for the quantum circuit using exact QFT.

| Gate Type | Number of Gates |
|---|---|
| CPhase | $8L^4 + 52L^3 + 44L^2$ |
| CNOT | $24L^3 + 32L^2$ |
| H | $16L^3 + 24L^2 + 10L$ |
| P | $4L^3 + 4L^2 + 2L$ |
| X | $2L^2 + 2L$ |
| CSwap | $2L^2$ |
| total | $8L^4 + 96L^3 + 114L^2 + 14L$ |

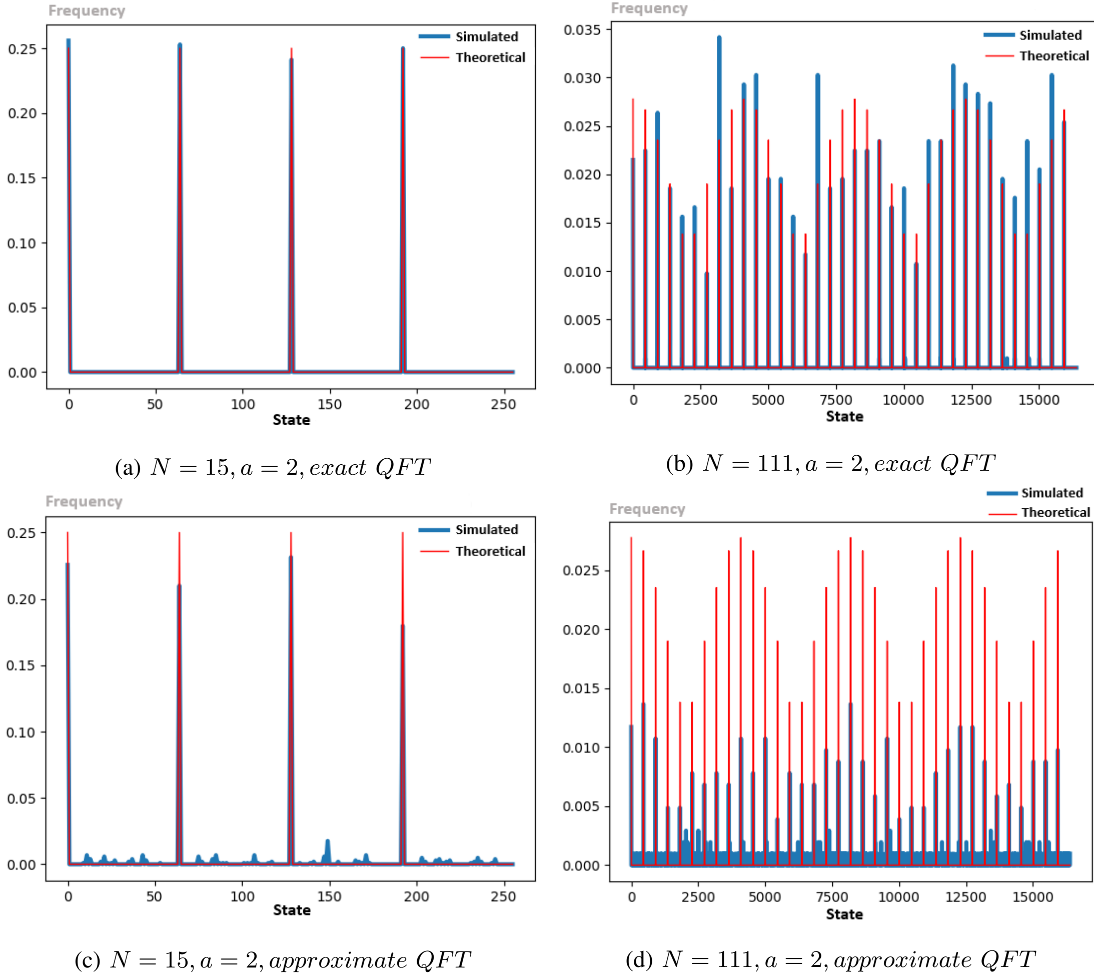
Our circuit has $m = 2L + 3$ qubits, and the length of the state vector is $2^m$. For CNOT Gate, X Gate, and CSwap Gate that only consist of permutation operations rather than float computations, we ignore the overhead for these gates in our speed-of-light estimation. For Hadamard Gate, the transformation matrix size is $2 \times 2$, each transformation needs 4 floating point operations and there are $2^{m-1}$ transformations. So the number of floating point operations per Hadamard Gate is $2^{m+1}$. For the controlled-phase Gate, the transformation matrix size is $4 \times 4$, each transformation needs 6 floating point operations and there are $2^{m-2}$ transformations. Therefore the number of floating point operations per Hadamard Gate is $3 \times 2^{m-1}$. For Phase Gate, the transformation matrix size is $2 \times 2$, each transformation needs 6 floating point operations and there are $2^{m-1}$ transformations. The number of floating point operations per Hadamard Gate is $3 \times 2^m$.

We then define the normalized Flop/s of the implementation as the total number of floating point operations in the circuit divided by the actual runtime. This value can be compared to the maximum Flop/s provided by the hardware as an indicator of how effectively computational resources are being utilized.

## VI. Experimental Setup

We ran the experiments on Pittsburgh Supercomputing Center's Bridges-2 Extreme Memory (EM) CPU and GPU machines for performance evaluation. The EM CPU machine uses 4 Intel Xeon Platinum 8260M with 24 Cores each, with a total of 160 Gflop/s. The GPU machine uses NVIDIA Tesla V100-32GB SXM2 GPU, with 7.8 Tflop/s [19]. We used Qiskit 0.23.1 and Qiskit Aer 0.12.0 with GPU support, running in Python 3.10.9 environment. We chose the noise-free simulators provided by Qiskit Aer.

To evaluate the correctness of the simulation, we ran 1,024 shots (i.e., runs) for each parameter set to get enough data points to generate the probability spectrum. To evaluate the runtime performance of Shor's algorithm, we ran both 1,024-shot and 1-shot experiments to explore the performance under different settings.




(a) $N = 15, a = 2, exact\ QFT$

(b) $N = 111, a = 2, exact\ QFT$

(c) $N = 15, a = 2, approximate\ QFT$

(d) $N = 111, a = 2, approximate\ QFT$

Fig. 3: Possibility spectrum of Shor's algorithm with different parameter settings.

## VII. Evaluation

### A. Correctness Analysis

We evaluated the effectiveness of implementing Shor's algorithm using both exact and approximate QFT compared to the ideal distribution. We ran each circuit with 1,024 shots, the probability spectrum is shown in Fig. 3.

We define correctness as the percentage of shots that can successfully find a factor. With $N$ as the number we are trying to factorize, $a$ as the base, $C_E$ as the correctness for exact QFT, and $C_A$ as the correctness for approximate correctness, we calculate the relative difference of correctness as

$$\frac{|C_E - C_A|}{C_E}.$$

We define the speedup as the runtime of the approximate QFT-based implementation over the runtime of the exact QFT-based implementation. Table I shows the correctness and speedup comparing the exact QFT implementation and the approximate QFT implementation of various parameter combinations.

[1]https://github.com/sundewang233/shor_algorithm_simulation

Effective algorithms should have well-aligned peaks in the probability spectrum. Fig. 3 shows that both approaches work well in the simulation of factorizing 15 and 111. Table I indicates that the approximate QFT approach provides the desired speedup while the correctness of factorizing is not significantly dropped.

### B. Simulation Limit

In our experiments, the largest $L$ we can reach is 13, which requires a 29 qubits quantum circuit, and the memory needed for the states is $8GB$. As shown in Fig. 2, in Qiskit's transpile stage, the list of gate instructions is first translated into a directed acyclic graph (DAG), and then the DAG is converted to a Python QObject [20]. The reason that prevents us from reaching a larger circuit is the edge number overflow in the Rust petgraph library that is used to construct the DAG in Qiskit.

### C. Runtime Performance

We evaluated the runtime performance of Shor's algorithm using the exact QFT implementation. The number of gates is

TABLE II: Correctness and runtime comparison of Shor's algorithm implementations using exact QFT and approximate QFT.

| N | a | $C_E$ (%) | $C_A$ (%) | Relative Difference (%) | Speedup |
|---|---|---|---|---|---|
| 15 | 2 | 74.41 | 75.20 | 1.06 | 0.45 |
| 21 | 2 | 83.50 | 85.44 | 2.32 | 0.45 |
| 35 | 2 | 83.39 | 71.00 | 15.37 | 1.11 |
| 111 | 2 | 97.85 | 90.92 | 7.08 | 1.22 |
| 111 | 5 | 97.16 | 87.21 | 10.24 | 1.22 |
| 143 | 2 | 64.84 | 53.52 | 17.46 | 1.42 |
| 323 | 2 | 97.17 | 97.95 | 0.80 | 1.03 |
| 519 | 2 | 99.94 | 99.80 | 0.14 | 1.24 |
| 1147 | 2 | 68.26 | 61.23 | 10.30 | 1.30 |

dominated by the number of controlled-phase gates since we use many QFT operators for modular exponentiation.

There are two main stages in Qiskit to run a quantum circuit. The instructions are first transpiled into a QObject (as described in Fig. 2), the QObject is then passed to the pre-compiled Qiskit Aer C++ library that executes the operations in the QObject. The transpilation is needed once for multiple circuit runs. That is, if there are multiple shots for factorization, this transpilation overhead will be amortized.

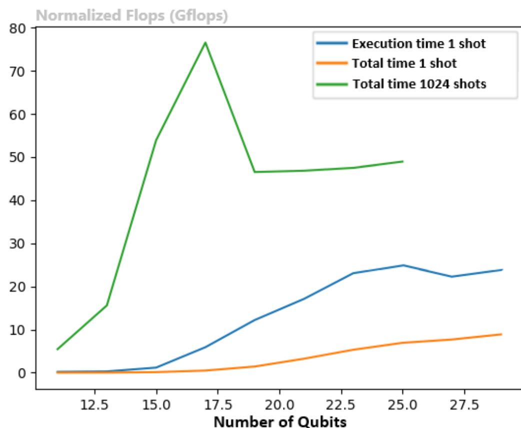



Fig. 4: Normalized Flop/s of single/multiple-shot Shor's algorithm implementation on CPU.

Fig. 4 and Fig. 5 show the normalized Flop/s with respect to the size of the quantum circuit, we measured 3 types of runtime performance, (i) the entire runtime for 1-shot, (ii) the entire runtime for 1,024-shots, and (iii) runtime for 1-shot execution in Qiskit Aer (i.e., after transpilation). For both CPU and GPU, normalized Flop/s for 1,024 shots with the transpilation time are higher than the normalized Flop/s for 1 shot, with or without the transpilation time. The reason is that while Qiskit will perform finer-grained parallelization for 1-shot implementation, Qiskit has a much better parallelization strategy that maps multiple shots to different cores/threads on CPU and GPU. Furthermore, for multiple shots, each CPU core gets its own copy of the state vector, thereby reducing data transfer during the computation. The fact that the margin is larger on CPU than on GPU indicates on GPU the parallelization across threads and blocks is not as optimized

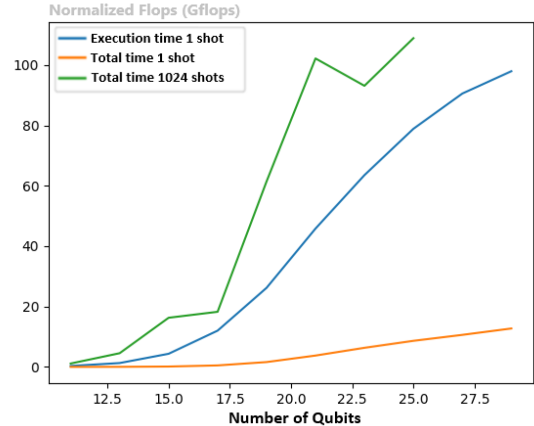



Fig. 5: Normalized Flop/s single/multiple-shot Shor's algorithm implementation on GPU.

as the CPU implementation by Qiskit.

Moreover, the Flop/s on a single CPU machine can reach up to about 1/3 of the maximum hardware Flop/s while the Flop/s on a GPU machine is only about 1/70 of the GPU Flop/s, which also indicates that there is more headroom on GPU than CPU for the community to optimize based on the current Qiskit implementation.

## VIII. Conclusion

In this paper, we implemented and simulated Shor's algorithm in Qikit using both the exact QFT-based and the approximate QFT-based approach. The experiments revealed that the bottleneck of the current simulation limit comes from the DAG building stage in Qiskit transpilation. Moreover, we defined a speed-of-light runtime performance model that calculates the minimum runtime required for executing Shor's algorithm with input size $N$ on a given platform. We evaluated the runtime performance of Shor's algorithm implementations on CPU and GPU using normalized Flop/s and compared them with the speed-of-light estimation to quantify headroom for further improvements. Our study suggests that future work can focus on a new method to construct the QObject to reach a higher simulation limit or optimizing the runtime performance on GPU devices.

# References


[1] C. Q. Choi, "Ibm's quantum leap: The company will take quantum tech past the 1,000-qubit mark in 2023," *IEEE Spectrum*, vol. 60, no. 1, pp. 46–47, 2023.

[2] P. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134, 1994.

[3] M. Roetteler, M. Naehrig, K. M. Svore, and K. Lauter, "Quantum resource estimates for computing elliptic curve discrete logarithms," 2017.

[4] A. Barenco, A. Ekert, K.-A. Suominen, and P. Törmä, "Approximate quantum fourier transform and decoherence," *Phys. Rev. A*, vol. 54, pp. 139–146, Jul 1996.

[5] S. Beauregard, "Circuit for shor's algorithm using 2n+ 3 qubits," *arXiv preprint quant-ph/0205095*, 2002.

[6] D. Coppersmith, "An approximate fourier transform useful in quantum factoring," *arXiv preprint quant-ph/0201067*, 2002.

[7] Qiskit contributors, "Qiskit: An open-source framework for quantum computing," 2023.

[8] M. A. Nielsen and I. Chuang, "Quantum computation and quantum information," 2002.

[9] J. R. Wootton, F. Harkins, N. T. Bronn, A. C. Vazquez, A. Phan, and A. T. Asfaw, "Teaching quantum computing with an interactive textbook," in *2021 IEEE International Conference on Quantum Computing and Engineering (QCE)*, pp. 385–391, IEEE, 2021.

[10] V. Vedral, A. Barenco, and A. Ekert, "Quantum networks for elementary arithmetic operations," *Physical Review A*, vol. 54, no. 1, p. 147, 1996.

[11] T. G. Draper, "Addition on a quantum computer," *arXiv preprint quant-ph/0008033*, 2000.

[12] C. DeCusatis and E. Mcgettrick, "Near term implementation of shor's algorithm using qiskit," in *2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC)*, pp. 1564–1568, IEEE, 2021.

[13] U. Skosana and M. Tame, "Demonstration of shor's factoring algorithm for n= 21 on ibm quantum processors," *Scientific reports*, vol. 11, no. 1, p. 16599, 2021.

[14] M. Rossi, L. Asproni, D. Caputo, S. Rossi, A. Cusinato, R. Marini, A. Agosti, and M. Magagnini, "Using shor's algorithm on near term quantum computers: a reduced version," *Quantum Machine Intelligence*, vol. 4, no. 2, p. 18, 2022.

[15] Y. Nam, "Running shor's algorithm on a complete, gate-by-gate implementation of a virtual, universal quantum computer," 2012.

[16] H. T. Larasati and H. Kim, "Simulation of modular exponentiation circuit for shor's algorithm in qiskit," in *2020 14th International Conference on Telecommunication Systems, Services, and Applications (TSSA*, pp. 1–7, IEEE, 2020.

[17] T. Leao and R. Maia, "Shoralgqiskit," 2019.

[18] J. Yamaguchi, M. Yamazaki, A. Tabuchi, T. Honda, T. Izu, and N. Kunihiro, "Estimation of shor's circuit for 2048-bit integers based on quantum simulator," *Cryptology ePrint Archive*, 2023.

[19] S. T. Brown, P. Buitrago, E. Hanna, S. Sanielevici, R. Scibek, and N. A. Nystrom, "Bridges-2: A platform for rapidly-evolving and data intensive research," in *Practice and Experience in Advanced Research Computing*, PEARC '21, (New York, NY, USA), Association for Computing Machinery, 2021.

[20] M. Treinish, I. Carvalho, G. Tsilimigkounakis, and N. Sá , "rustworkx: A high-performance graph library for python," *Journal of Open Source Software*, vol. 7, p. 3968, nov 2022.