Resources Estimation for Quantum Computing Algorithms in Multiple Physical Platforms

Cesar Zaragoza Cortes

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Master of Science

University of Washington

2021

Reading Committee:
Jeffrey Wilkes, Chair
Boris Blinov

Program Authorized to Offer Degree:
Physics

University of Washington

Abstract

Resources Estimation for Quantum Computing Algorithms in Multiple Physical Platforms

Cesar Zaragoza Cortes

Chair of the Supervisory Committee:
Professor Emeritus Jeffrey Wilkes
Department of Physics

An important task in the development of quantum computing technologies is to determine the resources needed to execute a particular algorithm in a device with certain characteristics. Resources estimation allows not only to determine whether it is possible to execute an algorithm in a current Noisy Intermediate-Scale Quantum (NISQ) device, but also to experiment with the parameters of quantum hardware to find out how much it would have to scale to achieve quantum advantage. We use the Q# quantum programming language to implement Shor's factorization algorithm, leverage simulators to perform resources estimation for trapped-ion and superconducting quantum hardware platforms, and compare the results.

TABLE OF CONTENTS

Chapter 1
INTRODUCTION

Algorithms designed for quantum computers have the potential to solve some problems that cannot be efficiently solved by algorithms designed for classical computers. However, estimating how much resources are needed to execute a quantum algorithm that outperforms a classical one is a difficult task. There are many quantum programming languages and tools built around them such as Q#[3], Qiskit[4] and Cirq[1] that allow execution of quantum algorithms on simulators but out-of-the-box options to estimate resources are limited to the logical level or not existent.

1.1 The Purpose of This Thesis

This thesis aims to estimate the resources required at the physical level to run Shor's semiprime integer factorization algorithm for trapped-ion and superconducting quantum hardware platforms. To do this, we will extend the simulators infrastructure built around Q# to calculate the maximum number of physical qubits, the total number of physical gates, and the maximum runtime required to execute a particular quantum algorithm.

Furthermore, we will also analyze the effects of errors introduced by the physical gates, analyze the feasibility of obtaining reliable results without implementing fault-tolerance, and estimate the cost of running the algorithm using error-corrected qubits and fault-tolerant gates.

In order to make this thesis more accesible to people from different backgrounds, we dedicate the next few chapters to provide a brief overview of the basics of quantum computing, quantum error correction, and Shor's semiprime integer factorization algorithm.

Chapter 2

BASICS OF QUANTUM COMPUTING

2.1 Dirac Notation

Also known as bra-ket notation, Dirac notation provides a convenient way of expressing the vectors used in quantum mechanics.

Dirac notation defines two elements:

- The *ket* ($|\psi\rangle$), which denotes a column vector in a complex vector space that represents a quantum state.

$$|\psi\rangle = \begin{bmatrix} \psi_1 \\ \psi_2 \\ \cdots \\ \psi_n \end{bmatrix}$$

- The *bra* ($\langle\psi|$), which denotes a row vector that is the conjugate transpose, or adjoint, of a corresponding *ket* ($|v\rangle$).

$$\langle\psi| = \begin{bmatrix} \psi_1^* & \psi_2^* & \cdots & \psi_n^* \end{bmatrix} = \begin{bmatrix} \psi_1 \\ \psi_2 \\ \cdots \\ \psi_n \end{bmatrix}^\dagger$$

An *operator* $\mathbf{A}$, represented by a $n \times n$ matrix, acting on a *ket* $|\psi\rangle$ produces another *ket* $|\psi'\rangle$ such that the produced *ket* can be computed by matrix multiplication:

$$|\psi'\rangle = \mathbf{A}\,|\psi\rangle = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \end{bmatrix} = \begin{bmatrix} A_{11}\psi_1 + A_{12}\psi_2 \\ A_{21}\psi_1 + A_{22}\psi_2 \end{bmatrix}$$

The *inner product* is denoted as a bra-ket pair $\langle\phi|\psi\rangle$ and represents the probability amplitude that a quantum state ψ would be subsequently found in state ϕ:

$$\langle\phi|\psi\rangle = \begin{bmatrix} \phi_1^* & \phi_2^* \end{bmatrix} \begin{bmatrix} \psi_1 \\ \psi_2 \end{bmatrix} = \phi_1^*\psi_1 + \phi_2^*\psi_2$$

This notation also provides a way to describe the state vector of n uncorrelated quantum states, the *tensor product*:

$$|\phi\rangle \otimes |\psi\rangle = |\phi\rangle\,|\psi\rangle = |\phi\psi\rangle = \begin{bmatrix} \phi_1 \\ \phi_2 \end{bmatrix} \otimes \begin{bmatrix} \psi_1 \\ \psi_2 \end{bmatrix} = \begin{bmatrix} \phi_1\psi_1 \\ \phi_1\psi_2 \\ \phi_2\psi_1 \\ \phi_2\psi_2 \end{bmatrix}$$

Note that $|\psi\rangle^{\otimes n}$ represents the tensor product of n $|\psi\rangle$ quantum states:

$$|\psi\rangle^{\otimes n} = |\psi\rangle \otimes \cdots \otimes |\psi\rangle = |\psi\rangle \cdots |\psi\rangle = |\psi \cdots \psi\rangle = \begin{bmatrix} \psi_1 \\ \psi_2 \end{bmatrix} \otimes \cdots \otimes \begin{bmatrix} \psi_1 \\ \psi_2 \end{bmatrix}$$

2.2 Quantum Systems

*ToDo: Explain the Schrodinger equation, describe what a quantum system is and mention basic postulates of quantum mechanics (similar to what Nielsen and Chuang explain).

*ToDo: This is a good place to introduce the concept of coherence.

2.3 Qubits

In classical information theory and classical computing, a bit is the fundamental building block. Analogously, in quantum information theory and quantum computing, a quantum bit or qubit is the fundamental building block.

Physically, a quibit is a two-level quantum-mechanical system. The polarization of a single photon and the spin of the electron are examples of such systems.

Mathematically, a qubit is a linear combination of states $|\psi\rangle = \alpha\,|0\rangle + \beta\,|1\rangle$ where α and β are complex numbers known as amplitudes, and $|0\rangle$ and $|1\rangle$ are the computational basis states.

When a qubit is measured, the result is either $|0\rangle$ with probability $|\alpha|^2$ or $|1\rangle$ with probability $|\beta|^2$. Since the probabilities must sum to one, the qubit's state is normalized:

$$|\alpha|^2 + |\beta|^2 = 1$$

The computational basis states, which are analogous to the two values (0 and 1) that a classical bit may take, form an orthonormal basis represented by the following vectors:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Using the previous definitions, we can see a qubit as a unit vector in two-dimensional complex vector space:

$$|\psi\rangle = \begin{bmatrix} \alpha \\ \beta \end{bmatrix}$$

2.4 Multi-Qubit Systems

*ToDo: Explain how multi-qubit systems are represented. Describe what a state vector is.

2.5 Superposition

The principle of quantum superposition states that the most general state of a quantum-mechanical system is a linear combination of all distinct valid quantum states. A qubit is an example of a quantum superposition of the basis states $|0\rangle$ and $|1\rangle$.

A concrete example of a qubit in superposition that has the same probability of being measured as $|0\rangle$ or $|1\rangle$ is the following:

$$|\psi\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + |1\rangle\right)$$

*ToDo: Expand this section to show multi-qubit superposition.

2.6 Quantum Logic Gates

In classical digital circuits, logic gates are the building blocks. Analogously, in the quantum circuit model of computation, quantum logic gates are the building blocks of quantum algorithms. Quantum gates act on qubits, transform them in different ways, and can be applied sequentially to perform complex quantum computations.

Quantum gates are unitary operators represented as $2^n \times 2^n$ unitary matrices where n is the number of qubits the gate operates on. Unitary matrices are complex square matrices $\mathbf{U}$ which have the property that its conjugate transpose or adjoint $\mathbf{U}^\dagger$ is also its inverse $\mathbf{U^{-1}}$:

$$\mathbf{U}^\dagger\mathbf{U} = \mathbf{U}\mathbf{U}^\dagger = \mathbf{U}\mathbf{U^{-1}} = \mathbf{I}$$

A quantum gate is applied to a qubit system by multiplying the gate's matrix representation by the qubits' state vector. This operation transforms the qubit system:

$$|\psi_1\rangle = \mathbf{U}\,|\psi_\mathbf{0}\rangle$$

Applying a sequence of quantum gates is equivalent to performing a series of these multiplications. For example, applying gate $\mathbf{U_a}$ followed by a gate $\mathbf{U_b}$ to a state vector $|\psi\rangle$ is represented by the follwing expression where the gates closest to the state vector are applied first:

$$\mathbf{U_b}\mathbf{U_a}\,|\psi\rangle$$

Since matrix multiplication is associative, multiplying $\mathbf{U_a}$ by $\mathbf{U_b}$ produces a compund gate $\mathbf{U_b}\mathbf{U_a}$ that is equivalent to applying $\mathbf{U_a}$ followed by $\mathbf{U_b}$:

$$\mathbf{U_b}\mathbf{U_a}\,|\psi\rangle = \mathbf{U_b}(\mathbf{U_a}\,|\psi\rangle) = (\mathbf{U_b}\mathbf{U_a})\,|\psi\rangle$$

Note that all quantum gates are reversible since they are represented by unitary matrices. This means that for any gate, another gate exists that reverts the gate's transformation on a state vector:

$$|\psi\rangle = \mathbf{U}^\dagger(\mathbf{U}\,|\psi\rangle) = \mathbf{U}^\dagger\mathbf{U}\,|\psi\rangle = \mathbf{I}\,|\psi\rangle$$

*ToDo: Describe and show how gates are represented in graphical circuits.

2.7 Single-Qubit Gates

Single-qubit gates are represented by 2×2 matrices.

*ToDo: List the most commonly used single-qubit gates and show how they transform a qubit.

2.8 Multi-Qubit Gates

*ToDo: List the most commonly used multi-qubit gates and show how they transform the qubits they act upon.

2.9 Entanglement

*ToDo: Define what entanglement is, describe why it is important, and show how qubits are entangled.

2.10 Interference

*ToDo: Define what interference is, describe why it is important, and show an example of interference.

2.11 Measurement

*ToDo: Define what it means to measure a qubit or a qubit system and show examples of measurements using different basis.

2.12 Quantum Advantage

*ToDo: Explain how quantum computers can solve a problem that a classical computer can't efficiently by exploiting superposition, entanglement, and interference.

Chapter 3

BASICS OF QUANTUM ERROR CORRECTION

This chapter explains the basic concepts to understand how quantum computations can be performed reliably in the presence of noise. Given the fragility of coherent quantum systems, quantum error correction and fault-tolerant quantum computation are fundamental arbitrarily large quantum algorithms operating on many qubits.

3.1 Noise and Error Correction

Noise can introduce errors in data while it is being processed, sent through a channel and/or stored in a medium. A common error that can be introduced by noise is a bit flip (e.g. $0 \rightarrow 1$, $1 \rightarrow 0$). The task of error correction is to detect when an error has occurred and correct it.

In classical error correction, coding based on data-copying is extensively used. The key idea is to protect data against the effects of noise by adding redundancy. For example, a classical repetition code can encode a logical bit using three physical bits:

$$0 \rightarrow 000$$

$$1 \rightarrow 111$$

The problem is that classical error correction techniques cannot be directly applied to qubits because of the following reasons:

- It is impossible to create an independent and identical copy of a qubit in an arbitrary unknown state. This is known as the no-cloning theorem of quantum mechanics and it has the consequence that qubits cannot be protected from errors by simply making multiple copies. *ToDo: Find reference for the no-cloning theorem.

- Measurement destroys quantum information, implying that we cannot measure a qubit to decide which correcting action to perform.

- In addition to bit flip errors ($|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \rightarrow |\psi\rangle = \alpha|1\rangle + \beta|0\rangle$), qubits are also susceptible to phase flip errors ($|\psi\rangle = \alpha|0\rangle + \beta|1\rangle \rightarrow |\psi\rangle = \alpha|0\rangle - \beta|1\rangle$) that have no classical analogous. Therefore, quantum error correction must be able to simultaneously correct for both.

- Errors in quantum information are continuous. This means that qubits, in the presence of noise, experience angular shifts rather than full bit or phase flips.

3.2 Bit Flip Code

*ToDo: Describe and show how a bit flip code works.

3.3 Phase Flip Code

*ToDo: Describe and show how a phase flip code works.

3.4 Shor Code

*ToDo: Describe and show how the Shor code works and how it can completely correct a qubit from any type of errors.

3.5 Fault-Tolerant Quantum Computation

*ToDo: Describe the concept of fault-tolerant quantum computation and how fault-tolerant quantum gates are constructed.

3.6 Threshold Theorem

*ToDo: Explain the threshold theorem and why it is important.

Chapter 4

SHOR'S SEMIPRIME INTEGER FACTORIZATION ALGORITHM

Shor's algorithm is a polynomial-time quantum algorithm for semiprime integer factorization. In comparison, the time complexity of the most efficient known classical factoring algorithm is superpolynomial.

We chose Shor's algorithm because it is one of the most significant quantum algorithms, and because the number of qubits and the number of quantum operations required are proportional to the input size. The number of operations is particullarly relevant since it makes the errors introduced by the physical gates an important consideration.

4.1 Algorithm

The problem that Shor's algorithm solves is the following: given a semiprime integer N, find its two prime factor p and q.

Shor's algorithm combines both classical and quantum computations, and the procedure to perform it is the following:

1. Pick a random integer $1 < a < N$.

2. Check whether a is a factor by determining whether a and N are comprime. If they are, we can compute the factors. Otherwise continue with the rest of the algorithm.

3. Use a period-finding quantum subroutine to find the period r of the function $f(x) = a^x \ mod \ N$.

4. If r is odd, then go back to step 1. If r is even, go to the next step.

5. If $a^{\frac{r}{2}} \equiv -1 \; mod \; N$, then go back to step 1.

6. Either $a^{\frac{r}{2}} - 1$ or $a^{\frac{r}{2}} + 1$ shares a factor with N.

4.2 Q# Implementation

In order to take advantage of the tools built around Q#, we use a modified implementation of Shor's algorithm found in Microsoft's quantum samples GitHub repository.

*ToDo: Add reference to Microsoft's quantum samples repository.

*ToDo: Breakdown the implementation into different sections and describe each one (similar to what is done in the "Learn Quantum Computing with Python and Q#" book).

The following Q# code presents a top-level implementation of Shor's algorithm.

```
@EntryPoint()
operation FactorSemiprimeInteger(N : Int) : (Int, Int) {

    // Check the most trivial case where N is pair.
    if (N % 2 == 0) {
        return (2, N / 2);
    }

    mutable factors = (1, 1);
    mutable foundFactors = false;
    repeat {

        // Start by guessing a coprime to N.
        let coprimeGuess = DrawRandomInt(1, N - 1);

        // If the guess number is a coprime, use a quantum algorithm for peri
        // Otherwise, the GCD between N and the coprime guess number is one o
```

```
        if (IsCoprimeI(N, coprimeGuess)) {
            let period = EstimatePeriod(N, coprimeGuess);
            set (foundFactors, factors) = CalculateFactorsFromPeriod(N, coprir
        } else {
            let gcd = GreatestCommonDivisorI(N, coprimeGuess);
            set (foundFactors, factors) = (true, (gcd, N / gcd));
        }
    }
    until foundFactors

    return factors;
}
```

4.3 Quantum Subroutine

*ToDo: Show the circuit representation of the quantum subroutine.

The following Q# code implements the period finding quantum subroutine:

```
    @EntryPoint()
    operation EstimatePeriodInstance(N : Int, a : Int) : (Bool, Int) {

        // Prepare eigenstate register.
        let bitSize = BitSizeI(N);
        use eigenstateRegister = Qubit[bitSize];
        let eigenstateRegisterLE = LittleEndian(eigenstateRegister);
        ApplyXorInPlace(1, eigenstateRegisterLE);

        // Prepare phase register.
        let bitsPrecision =  2 * bitSize + 1;
```

```
    use phaseRegister = Qubit[bitsPrecision];
    let phaseRegisterLE = LittleEndian(phaseRegister);

    // Prepare oracle for quantum phase estimation.
    let oracle = DiscreteOracle(ApplyOrderFindingOracle(N, a, _, _));

    // Execute quantum phase estimation.
    QuantumPhaseEstimation(oracle, eigenstateRegisterLE!, LittleEndianAsB

    let phaseEstimate = MeasureInteger(phaseRegisterLE);

    // Reset qubit registers
    ResetAll(eigenstateRegister);

    // Return period calculation based on estimated phase.
    return CalculatePeriodFromPhaseEstimate(N, bitsPrecision, phaseEstima
}
```

Chapter 5

ANALYSIS FRAMEWORK

The strategy we use to analyze this algorithm is very similar to the one proposed by Soeken et al.[2]. The process is the following:

1. Implement a quantum algorithm using a high-level programming language (Q# in this case).

2. Verify the correctness of the implementation by executing the algorithm in a full state simulator.

3. Use the built-in resources estimator to roughly calculate the amount of logical quantum gates used by the algorithm depending on the input size. *ToDo: This part can be done in its own chapter and that chapter can explain in detail how the built-in resources estimator works and how to interpret the data that it produces.

4. For each hardware platform do the following:

   (a) Create a resources estimator that uses hardware specific parameters to calculate the amount of physical qubits, physical gates, and runtime required to run the algorithm.

   (b) Analyze data produced by the resources estimator to determine the maximum input size for the algorithm to run on a real NISQ device.

   (c) Analyze data produced by the resources estimator to determine the characteristics that a device should have to run the algorithm for a specific input size, and determine what would be the runtime.

5. Compare the results for each hardware platform.

*ToDo: Explain the following

- Resources Metrics: Describe the values obtained from the resources estimator (gate count, runtime, accumulated error), and how they are calculated.

- Gate Decomposition: Describe why logical-level gates have to be decomposed into physical-level gates.

- Limitations: Describe the limitations that this resources estimation has in regards to runtime (sum of the runtimes of individual gates rather than the critical path), and types of computations (trouble with mixed states).

5.1 Extending Q# Simulation Infrastructure for Estimation of Physical Resources

Microsoft's Quantum Development Kit (QDK) supports the implementation of custom simulators that can be used to run Q# programs. We leverage this capability and implement a simulator that calculates the resources a quantum algorithm would require to be executed in a hardware platform with specific characteristics.

*ToDo: Explain the following

- QDK Custom Simulators: Describe how custom simulators are implemented using diagrams and code snippets.

- Software Architecture of Physical Resources Estimator Simulator: Describe the software architecture using diagrams and code snippets.

Source code of a working version can be found in in GitHub.

Chapter 6

TRAPPED-IONS HARDWARE PLATFORM

*ToDo: Briefly describe this quantum computing platform.

6.1 Native Gates and Platform Characteristics

*ToDo: Enumerate the native gates that this platform implements, its characteristics (fidelity, gate time), and how logical gates are implemented (using circuits to illustrate them).

6.2 Resources Estimation Analysis

*ToDo: Show (using tables and/or plots) how resources escalate as the size and pattern of the input changes.

6.3 Analysis of Execution in NISQ Devices

*ToDo: Analyze what would be the maximum input size that can be successfully run on a NISQ device based on this platform.

6.4 Analysis of Execution of Algorithm for Input of Specific Size

*ToDo: Use a resource estimator to determine the characteristics that a device should have to run the algorithm for a specific input size, and determine what would be the runtime.

Chapter 7

SUPERCONDUCTING HARDWARE PLATFORM

*ToDo: Briefly describe this quantum computing platform.

7.1 Native Gates and Platform Characteristics

*ToDo: Enumerate the native gates that this platform implements, its characteristics (fidelity, gate time), and how logical gates are implemented (using circuits to illustrate them).

7.2 Resources Estimation Analysis

*ToDo: Show (using tables and/or plots) how resources escalate as the size and pattern of the input changes.

7.3 Analysis of Execution in NISQ Devices

*ToDo: Analyze what would be the maximum input size that can be successfully run on a NISQ device based on this platform.

7.4 Analysis of Execution of Algorithm for Input of Specific Size

*ToDo: Use a resource estimator to determine the characteristics that a device should have to run the algorithm for a specific input size, and determine what would be the runtime.

Chapter 8

COMPARISON BETWEEN TRAPPED-IONS AND SUPERCONDUCTING HARDWARE PLATFORMS

*ToDo: Compare the results obtained from both hardware platforms and comment on the insights obtained.

Chapter 9

FUTURE WORK

*ToDo: Mention how this framework can be used to analyze and compare other hardware platforms.

BIBLIOGRAPHY

[1] Cirq Developers. Cirq, May 2021. See full list of authors on Github: https://github.com/quantumlib/Cirq/graphs/contributors.

[2] Mathias Soeken, Mariia Mykhailova, Vadym Kliuchnikov, Christopher Granade, and Alexander Vaschillo. A resource estimation andverification workflow in q#. *Design, Automation and Test in Europe Conference*, 2021.

[3] Krysta Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level dsl. In *Proceedings of the Real World Domain Specific Languages Workshop 2018*, RWDSL2018, New York, NY, USA, 2018. Association for Computing Machinery.

[4] Matthew Treinish, Jay Gambetta, Paul Nation, Paul Kassebaum, qiskit bot, Diego M. Rodríguez, Salvador de la Puente González, Shaohan Hu, Kevin Krsulich, Laura Zdanski, Jessie Yu, David McKay, Juan Gomez, Lauren Capelluto, Travis-S-IBM, Julien Gacon, Ashish Panigrahi, lerongil, Rafey Iqbal Rahman, Steve Wood, Luciano Bello, Divyanshu Singh, Drew, Joachim Schwarm, MELVIN GEORGE, Manoel Marques, Omar Costa Hamido, RohitMidha23, Sean Dague, and Shelly Garion. Qiskit/qiskit: Qiskit 0.26.2, May 2021.

Appendix A

IMPLEMENTATION OF GENERIC PHYSICAL RESOURCES ESTIMATION FRAMEWORK

*ToDo: Add source code that implements the physical resources estimation framework.

Source code can also be found in GitHub.