CMSC 655 Project Report: Simulation Speedup for the HHL Algorithm

Ameya Bhave
Reece Robertson

November 28, 2023

Contents

1 Introduction, Significance, & Background

Quantum computers may be the computers of the future. Compared to the familiar computers that we use every day (hereafter called "classical computers") which operate on inputs in serial, quantum computers are naturally parallel. Although classical computers can use *multiple* cores of a processing unit to handle inputs in parallel, a *single* "core" of a quantum processing unit can act on many different inputs simultaneously. In more precise terms, a single register of quantum bits (qubits) can leverage the quantum phenomena of superposition, entanglement, interference, and measurement to operate on many different potential solutions to a problem simultaneously.

Several quantum algorithms have been discovered that solve certain problems with provable speedups over the corresponding fastest known classical methods for the same problems. One such algorithm is known as the Harrow, Hassidim, and Lloyd (HHL) algorithm, which samples from the solution to a system of linear equations and (under favorable conditions) gains an exponential speedup over the comparable classical algorithm [1]. This algorithm will be explained in more detail in section 2.

Ironically, the current state-of-the-art for small-scale *error-free* quantum computation is to avoid real quantum computers altogether—we have not yet attained perfect control over any real quantum system. To address this issue, researchers have simulated quantum algorithms (including the HHL

algorithm) on classical computers using matrix operations. Performing these simulations, however, requires an exponential increase in resources as the problem scales. Our objective in this project is to improve upon the HHL simulation using optimized numerical methods. We show that this results in a constant run time for any valid linear system involving a 2×2 matrix, while the simulator faces exponential scaling as the ratio between the eigenvalues of the matrix increases (this is described in more detail in section 2).

A subtlety to note is that in this paper we are not concerned with solving a linear system of equations. Instead, we are concerned with replicating the behavior of the HHL quantum algorithm on a classical computer. Quantum computers solve problems by creating and sampling from specific distributions from which meaningful information can be extracted. We implement two methods of simulating the HHL algorithm, one using the standard simulation method and one using an optimized algorithm. We show that both methods ultimately produce and sample from the same distributions that a perfect quantum computer would, and then we show that the optimized method achieves significantly better scaling than the standard simulation method.

We close the introduction with a word of terminology. Throughout the paper we use the term *simulation* to refer to the standard method for simulating quantum algorithms. We use the term *emulation* to describe our method of simulating the algorithm directly as efficiently as possible using an optimized algorithm.

2 Methodology Description

We begin with a discussion of how the HHL algorithm works on a quantum computer. Following this we describe in detail how we implement this on a classical computer via the simulation method, and then via the emulation method.

2.1 The HHL Algorithm on a Quantum Computer

The HHL algorithm begins with a system of linear equations $A|x\rangle = |b\rangle$, where A is an $N \times N$ hermitian matrix, $|b\rangle$ is an $N \times 1$ unit vector, and $|x\rangle$ is the unknown solution vector to this equation. Usually we have that $N = 2^n$ for some $n \in \mathbb{N}$. Let $|u_j\rangle$ be the eigenvectors of A with λ_j as the corresponding eigenvalues. Given the correct coefficients β_j, we can represent $|b\rangle$ in the eigenbasis of A as $|b\rangle = \sum_{j=1}^{N} \beta_j |u_j\rangle$. Now, as stated in [1], we have that

$$\sum_{j=1}^{N} \beta_j \lambda_j^{-1} |u_j\rangle = A^{-1}|b\rangle = |x\rangle. \tag{1}$$

Our quantum computer produces this solution by preparing the state

$$\sum_{j=1}^{N} \beta_j |u_j\rangle \left(\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle + \frac{C}{\lambda_j} |1\rangle \right), \tag{2}$$

which, upon measuring a $|1\rangle$ in the second register, becomes

$$K \sum_{j=1}^{N} \beta_j \frac{C}{\lambda_j} |u_j\rangle, \tag{3}$$

where C and K are constants that we need not discuss here, but are explained in detail in [1]. Observe that (3) is identical to (1) up to the presence of the normalization constant CK, and hence (3) represents a solution to our problem.

The algorithm used to prepare the state (2) is rather complex, and we will not describe it in detail here. Suffice it to say that the crux of the algorithm lies in the decomposition of $|b\rangle$ in the eigenbasis of A [1]. This is done using a subroutine known as quantum phase estimation (QPE), which involves converting the hermitian matrix A into a unitary matrix U and repeatedly applying U to evolve the quantum state $|b\rangle$ into $|x\rangle$ [2]. A high-level picture of the algorithm, presented in the language of quantum circuit diagrams, is given in Figure 1.

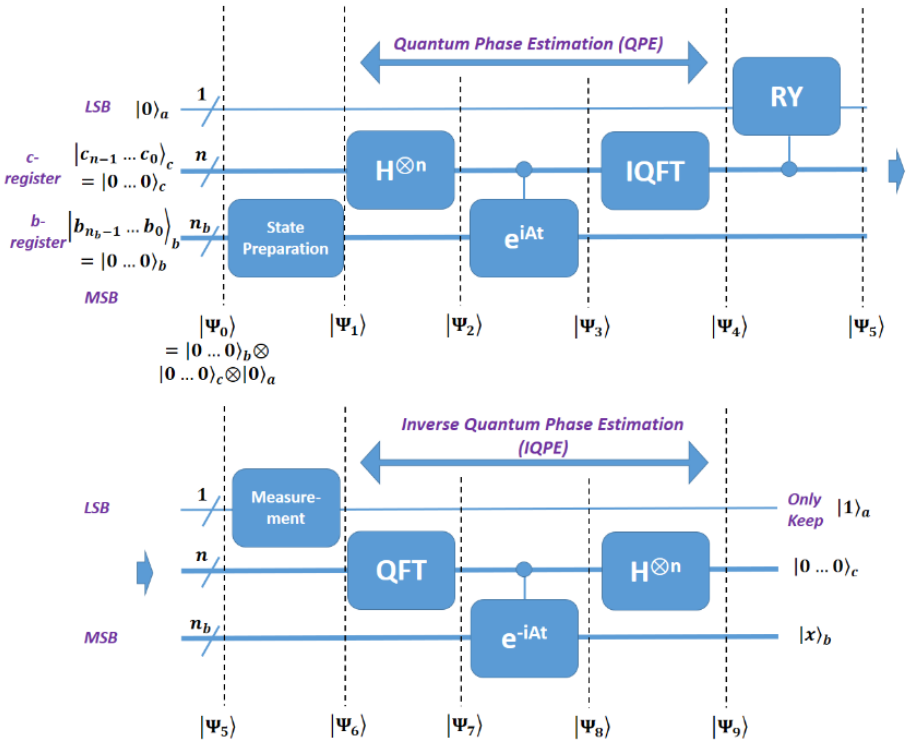


Figure 1: A high-level overview of the HHL algorithm, courtesy of [4].

One last point to note is that the algorithm makes use of three quantum registers, labeled a, c, and b from top to bottom in Figure 1. Each quantum register contains some number of qubits, and each additional qubit doubles the representative capacity of the register. Register a always consists of a single "ancilla" qubit. Register b is initialized to the state $|b\rangle$ and is evolved into the state $|x\rangle$ over the course of the algorithm. Thus, this register needs to contain enough qubits to represent the $N \times 1$ vector $|b\rangle$. In our experiments below we fix the size of the b register at 1 qubit, which allows for $|b\rangle$ vectors of size 2×1. Finally, the c register represents the "counting" qubits. This register requires enough qubits to accurately approximate the ratio between the eigenvalues of A. For example, our first experiment uses an A matrix that has eigenvalues of $2/3$ and $4/3$, hence the ratio between the eigenvalues is $1 : 2$. Therefore, 2 qubits, which gives us $2^2 = 4$ states, is enough to express this ratio. In our second example, however, our A matrix has eigenvalues of 6 and 7, hence the ratio is $6 : 7$. This requires 3 qubits, or $2^3 = 8$ states, to represent exactly. If we were to limit ourselves to 2 qubits the best approximation we could make would be $2 : 3$. In general, increasing the number of counting qubits increases the accuracy of the HHL algorithm until the ratio between the eigenvalues can be expressed exactly [4].

2.2 Simulating the HHL Algorithm

To simulate the HHL algorithm we reproduce every operation required by the algorithm on a classical computer. It is well known that any quantum system of m qubits can be represented by a *state vector* of size 2^m. Moreover, any quantum gate operation can be represented by a $2^m \times 2^m$ unitary matrix, where m is the number of qubits involved in the operation. To apply an operation to a set of qubits we matrix multiply the qubit's state vector by the operation's matrix. To apply multiple operations, we matrix multiply the associated matrices one after another in sequence. Consequently, without getting into the details, the simulation method starts with an initial state vector, creates a matrix to represent each quantum operation, and then multiplies the matrices to the state vector in order. These matrices double in size with each additional qubit added to the algorithm.

Recall that the HHL algorithm involves three registers, a, c, and b. For our purposes, registers a

and b are held constant with 1 qubit each, however for the simulation method the register c will vary in size (as described above). For every additional qubit added to the c register, we expect that the time the simulation method will take to run will (at least) double. This means that the simulation method scales exponentially in time with respect to the number of counting qubits required by the algorithm, even if the size of the matrix A is held constant.

2.3 Emulating the HHL Algorithm

When it comes to emulating the HHL algorithm we do not concern ourselves with exactly replicating every step of the algorithm as it would be performed on a quantum computer. Rather, we are concerned with producing the same output that a perfect quantum computer would produce if it ran the algorithm. For the HHL algorithm, this means that we just need to produce equation (2), and we don't need to worry about how we do so. This focus on the end result allows us to skip over many of the details of the algorithm, such as the quantum phase estimation, that the simulator must replicate.

The emulation proceeds by first computing the eigenvalues λ_j and eigenvectors $|u_j\rangle$ of the matrix A, where $j \in [1, 2]$. The coefficients β_j to represent vector $|b\rangle$ in the eigenbasis of A are then computed. Finally, we set c to be the minimal eigenvalue of A. Once we have that information we can compute (2) directly. Then we can normalize the state and measure the ancilla qubit to get equation (3), and measure the b qubit to get the final output.

Note that equation (2), and hence the above description, does not include the c register at all. While the c register is a critical component to the quantum algorithm (and the simulation), its value has been reset to its initial state by the end of the algorithm. Thus, if we only care about producing the final output of the algorithm we can entirely omit this register from our emulation. Therefore, if we hold the size of the a and b registers constant, the speed of the emulation will likewise run in constant time regardless of the ratio between the eigenvalues of A. This is an exceptional improvement over the simulation method.

3 Methodology Implementation Summary

In this section we describe how we implemented the HHL simulator and emulator using Python. The source code for each method can be found in appendix A.1 and A.2, respectively.

3.1 Python Simulation

Creating a simulator for the HHL algorithm was a complex process. It required a thorough understanding of the exact details of the HHL algorithm, which we have glossed over in this paper. Notably, it requires that one precompute several rotation matrices used in the quantum phase estimation (QPE) and quantum Fourier transform (QFT) subroutines (and their inverses). It also requires that one implement those subroutines explicitly using the precomputed matrices. We chose to use the Intel Quantum Simulator [3] as the framework on which we built implementation for these subroutines, and for our implementation of the HHL algorithm itself.

Our simulation of the HHL algorithm can be found in the `simulateHHL` algorithm beginning on line 211 of appendix A.1. Like the emulator, parameters of this function include the matrix A, the vector $|b\rangle$, the number of shots (times to repeat the experiment), a filename for a plot of the results, and the DPI for the same plot. Unlike the emulator, the simulator requires four extra parameters: a function `applyRYs`, a variable `t`, and a variable `numQubits`, and a variable `extraAncillas`, all of which are necessary to the simulation but specific to the system of equations $A|x\rangle = |b\rangle$ to which the algorithm is applied. Because this function and these variables do not have patterns which easily generalize, we opted to require the user to provide these values for the problem at hand. This has the advantage of simplifying our simulator code, but the trade-off is that it requires the users to do some work before they can run the simulation. One other point to note is that even our "simplified" simulator code is over three times as long as the corresponding emulator code.

Once the parameters are provided, the `simulateHHL` function will precompute all the required matrices, then perform the HHL algorithm making use of the QPE and QFT subroutines. When the algorithm is complete it will measure the resulting state using the rules of quantum measurement. This will produce one of four possible values: $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$. The function performs this experiment

`shots` times, and produces a histogram of the result. It also records the amount of time taken to complete all the shots, and it will return an estimate of the state $|x\rangle$. This estimate is given by the ratio of the probability of a $|11\rangle$ outcome with the probability of a $|01\rangle$ outcome. Note that these states correspond to the state where the ancilla qubit measured to be a $|1\rangle$, hence these values correspond to (3).

3.2 Python Emulation

Appendix A.2 provides our Python code for emulating the HHL algorithm. This code is found in the `emulateHHL` function found on line 5 of that section. This function takes as input the matrix a NumPy matrix `A` and vector `b`. We first calculate the eigenvalues and the eigenvectors of `A`, and we set `c` to be the minimum eigenvalue of `A` as directed by [4]. Next we compute the values of β which allow us to represent `b` in the eigenbasis of `A`. Finally, we compute the final value of the quantum state, given by (2), and normalize.

The emulation continues by simulating quantum measurements. A dictionary, `measure_count`, is initialized to keep track of the count of measurement outcomes corresponding to the states $|00\rangle$, $|01\rangle$, $|10\rangle$, $|11\rangle$. The emulation runs for a specified number of shots, and the outcomes are sampled based on the probabilities defined by the quantum state. The function records the count of measurements for each outcome. The total time taken for the emulation is computed, and a histogram plot of the measurement outcomes is generated and saved as an image file. The function returns an estimation of the solution vector $|x\rangle$, and the total time taken for the emulation.

4 Results

We tested our emulator against our simulator on two different problems. The first was

$$\begin{pmatrix} 1 & -1/3 \\ -1/3 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \tag{4}$$

chosen because it is the example used in [4], which allowed us to verify the accuracy of our methods by reproducing the results from that paper. In this case the eigenvalues of A are $\lambda_1 = 2/3$ and $\lambda_2 = 4/3$, which gives a ratio between the eigenvalues of $1:2$. Two qubits are required to express this relation, hence we performed a simulation involving 4 qubits.

The solution to this equation is $x_1 = 3/8$ and $x_2 = 9/8$, which has a squared ratio of $1:9$. Thus, we expect that the ratio between the probability of measuring the states where the ancilla qubit is $|1\rangle$, that is, the states $|11\rangle$ and $|01\rangle$, to also be $1:9$. Therefore, our expected result is

$$\begin{pmatrix} 0.1 \\ 0.9 \end{pmatrix}. \tag{5}$$

Moreover, we expect the probability of measuring the states where the ancilla qubit is $|0\rangle$, that is, $|10\rangle$ and $|00\rangle$, to be comparatively low.

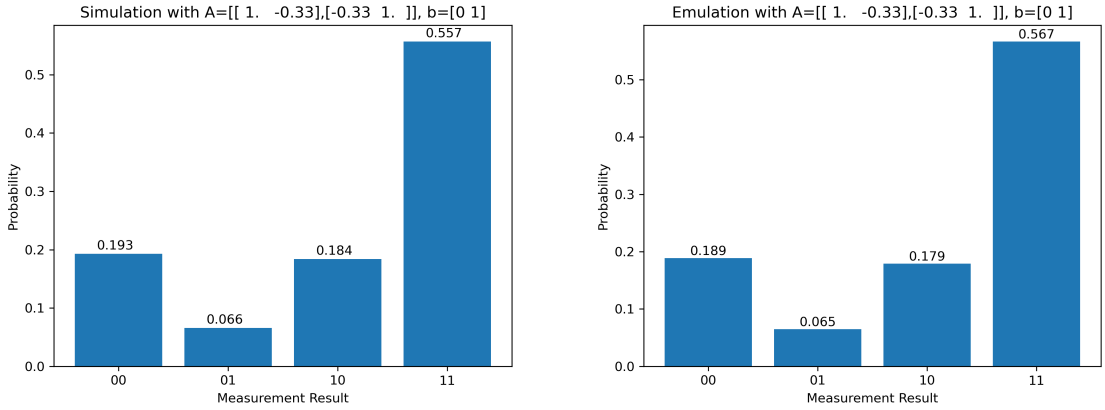
Our expectations are verified in Figure 2, which shows prototypical histogram which results from running 2048 shots of this problem on the simulator (Figure 2a) and on the emulator (Figure 2b). We see that both histograms are close to each other, indicating that both the simulator sample from the same distribution at the conclusion of the quantum algorithm. Moreover, both histograms are very similar to Figure 4 of [4], which further establishes that both our methods are operating correctly.

We ran 2048 shots of this problem on both the simulator and the emulator ten times over, and then we computed the average ratio between the probabilities of $|11\rangle$ and $|01\rangle$ to estimate the value of $|x\rangle$. We found that the simulator produced a result of

$$\begin{pmatrix} 0.0948283 \\ 0.9051717 \end{pmatrix}, \tag{6}$$

and the emulator produced a result of

$$\begin{pmatrix} 0.09411943 \\ 0.90588057 \end{pmatrix}. \tag{7}$$



(a) Simulation measurement result.

(b) Emulation measurement result.

Figure 2: Prototypical results of simulating and emulating the HHL algorithm on (4).

Both results are very close to (5), which gives a final witness that both our simulation method and *especially* our emulation method created and sampled from the distribution that a perfect quantum computer would when running the HHL algorithm on this problem.

Finally, the most important result of this experiment is the run-time. The simulator ran all 2048 shots on average in 2.20873 seconds. This is approximately 0.00108 seconds per shot. The emulator, on the other hand, ran all 2048 shots in 0.07670 seconds on average, which is a time of 0.00003 seconds per shot. All experiments were run on a standard Linux laptop. Here it is already demonstrated that the method of emulation offers a notable time complexity reduction compared to the simulation method.

The second example problem we used to test our simulator and emulator was

$$\begin{pmatrix} 13/2 & -1/2 \\ -1/2 & 13/2 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \tag{8}$$

chosen because the eigenvalues of A are $\lambda_1 = 6$ and $\lambda_2 = 7$, which gives a ratio between the eigenvalues of $6 : 7$. Three qubits are needed to express this relation, and an additional two ancilla qubits are required for one of the steps of the simulation. Consequently, we performed a simulation involving 7 qubits.

The solution to this equation is $x_1 = 1/84$ and $x_2 = 13/84$, which has a squared ratio of $0.00588235 : 0.99411765$ implying that $|x\rangle$ is

$$\begin{pmatrix} 0.00588235 \\ 0.99411765 \end{pmatrix}. \tag{9}$$

Moreover, we ran 2048 shots of this problem on both the simulator and the emulator ten times over, and then estimated the value of $|x\rangle$. We found that the simulator gave a solution of

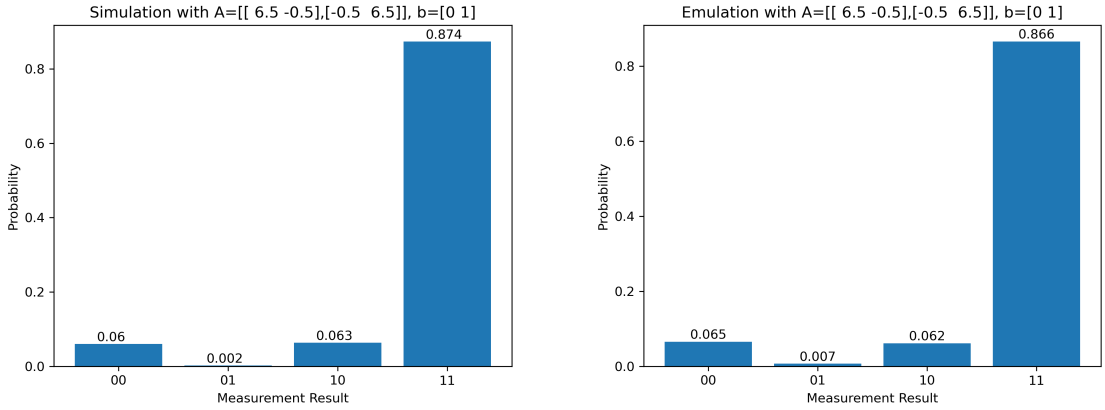
$$\begin{pmatrix} 0.00452281 \\ 0.99547719 \end{pmatrix}, \tag{10}$$

and the emulator gave a solution of

$$\begin{pmatrix} 0.00657847 \\ 0.99342153 \end{pmatrix}. \tag{11}$$

The histogram for the simulator is given in Figure 2a, and the histogram for the emulator is given in Figure 2b.

Once again we find that the emulator greatly outperforms the simulator with respect to run-time. The simulator ran all 2048 shots on average in 30.18495 seconds, which is approximately 0.01474 seconds per shot. This demonstrates the exponential scaling of the simulator with respect to the required number of counting qubits. Recall that the first experiment required 4 total qubits (2 counting, 1 ancilla, 1 b) and ran in 2.20873 seconds. This experiment required 7 qubits total (3 counting, 1 ancilla,



(a) Simulation measurement result.

(b) Emulation measurement result.

Figure 3: Prototypical results of simulating and emulating the HHL algorithm on (8).

1 b, 2 extra ancillas), and ran in 30.18495 seconds—a tenfold increase in time for three additional qubits. The emulator, on the other hand, ran all 2048 shots in 0.07683 seconds on average, which is a time of 0.00003 seconds per shot. This result is practically indistinguishable from the first experiment, demonstrating that the emulator is invulnerable to changes in the required counting qubits. Given a fixed problem size, the emulator runs in constant time while the simulator runs in a variable time that scales exponentially in general.

5 Error Analysis

This work has an unusual relationship with the error inherent in the algorithm. While most algorithms attempt to remove all error from a given computation, our aim is to mimic the behavior of a quantum computer running the HHL algorithm exactly. This means that we attempt to replicate the error that would be observed on a real quantum device, while also eliminating any error that would not be observed on the same device.

There are two sources of error in the HHL algorithm. The first source of error is when the ancilla qubit of the a register is measured to be a $|0\rangle$ instead of a $|1\rangle$. The algorithm minimizes this error through the appropriate number of counting qubits, but it does not eliminate it in general. Therefore, because this error is observed on real devices, both our simulator and our emulator preserve this source of error. This can be seen in Figures 2 and 3, where all plots show some probability mass on the states $|00\rangle$ and $|10\rangle$.

The second source of error arises from the inaccuracy of the approximation of $|x\rangle$. Recall that $|x\rangle$ is approximated by computing the ratio between the probability of measuring a $|11\rangle$ and the probability of measuring a $|01\rangle$. As the number of shots for a given experiment increases we expect that this error will tend toward zero (i.e. we expect that a perfect quantum computer will make an arbitrarily close approximation given enough iterations). Thus, we hope to eliminate this source of error in our simulator and emulator. To show that we have done this, we compute the absolute error between our approximations and the true values of $|x\rangle$ below.

For experiment 1, we have that the simulator solution, equation (6), differs from the true solution, equation (5), by a value of 0.01034. The emulator solution (7) differs from (5) by a value of 0.01709. For experiment 2, the simulator solution (10) differs from the true solution (9), by a value of 0.00316, while the emulator solution (11) differs from the true solution by a value of 0.00436. The error is low on all accounts, and is within the reasonable effects of random chance.

As a final note before we move on from the topic of error correction, let us consider the error between our two approximations. Both the simulator and emulator aim to replicate the behavior of a real quantum device running the HHL algorithm, and hence both should produce the same output. As a check of this, we compute the error between our two approximations. We find that the mean absolute error between the two histograms of experiment 1 is 0.00500 and the mean absolute error

between the two histograms of experiment 2 is 0.00475. Again, both error rates are low enough to be within the realm of random chance. This adds a final witness that our two methods behave identically, which validates our claims about our emulator.

6 Conclusions

In closing, our contribution to the field is an emulator for the HHL algorithm. Our emulator exactly replicates the behavior of a perfect quantum computer when running the HHL algorithm on a specific problem involving a linear system of equations $A|x\rangle = |b\rangle$, where A is a 2×2 matrix. Given that fixed problem size, the emulator runs in constant time regardless of the specifics of the problem. This stands in stark contrast to the traditional simulation method which scales exponentially as the magnitude of the ratio between the eigenvalues of A increases.

We validated the accuracy and speedup of our emulator through comparison with a quantum computing simulator. We performed two experiments using problems of varying complexity, and we verified that the emulator and the simulator agreed in each case. Moreover, our first experiment reproduced the results of [4], which added further validity to the accuracy of both our emulator and our simulator. Accuracy established, our experiments also demonstrated the expected speedups described above.

We hope that our emulator will be used by researchers and students who hope to better understand the behavior of the HHL algorithm on small systems. It is not only faster than the traditional simulation method, but as mentioned in section 3, it is less complex than the traditional simulation method. It is a third the size of the simulator, and it deals with mathematical logic rather than the quantum gate operations that the simulator employs. For students just learning about quantum computing this will provide an accessible place to begin exploring the method in which a quantum computer solves a problem and how that differs from a classical computer, without requiring them to master the language of quantum computing beforehand.

References

[1] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. "Quantum Algorithm for Linear Systems of Equations". In: *Physical Review Letters* 103.15 (Oct. 2009). Publisher: American Physical Society, p. 150502. DOI: 10.1103/PhysRevLett.103.150502. URL: https://link.aps.org/doi/10.1103/PhysRevLett.103.150502 (visited on 09/26/2023).

[2] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. en. ISBN: 9780511976667 Publisher: Cambridge University Press. Dec. 2010. DOI: 10.1017/CBO9780511976667.

[3] Gian Giacomo Guerreschi et al. "Intel Quantum Simulator: a cloud-ready high-performance simulator of quantum circuits". In: *Quantum Science and Technology* 5.3 (May 2020), p. 034007. DOI: 10.1088/2058-9565/ab8505. URL: https://dx.doi.org/10.1088/2058-9565/ab8505.

[4] Hector Jose Morrell Jr au2, Anika Zaman, and Hiu Yung Wong. *Step-by-Step HHL Algorithm Walkthrough to Enhance the Understanding of Critical Quantum Computing Concepts*. 2023. arXiv: 2108.09004 [quant-ph].

A Source Code

All source code for this project can be found at https://github.com/reecejrobertson/hhlEmulator. The source code for the simulator is given in section A.1, the source code for the emulator is given in section A.2, and the code that performs the experiments comparing the two is given in section A.3.

A.1 simulator.py

```
import sys
sys.path.insert(0, '../intel-qs/build/lib') # Change this to match your installation location.
import intelqs_py as simulator
import numpy as np
from matplotlib import pyplot as plt
import time

def computeQPERotations(A, t, numQubits):
    '''
    Compute the single qubit gates needed in the QPE algorithm.
    Parameters:
        A (np.ndarray)  : The A matrix.
        t (float)       : A variable used in the computation.
            For details, see https://arxiv.org/pdf/2108.09004.pdf eqs (17)-(21).
        numQubits (int) : The total number of qubits to use.
    Returns:
        A lsit of the QPE rotation matrices, and a list of the IQPE matrices.
    '''

    # Get the eigenvalues and eigenvectors and sort them by eigenvalue size.
    evals, evecs = np.linalg.eig(A)
    idx = evals.argsort()
    evals = evals[idx]
    evecs = evecs[:, idx]

    # Compute the diagonal matrix hamiltonian.
    Udiag = np.diag(np.exp(1j * evals * t))

    # Create a list to hold the QPE and IQPE rotation matrices.
    qpeRotations = []
    qpeInvRotations = []

    # Populate the matrices with successive powers of the diagonal matrix.
    for i in range(numQubits-2):
        Ui = evecs.T @ np.linalg.matrix_power(Udiag, 2**i) @ evecs
        qpeRotations.append(Ui)
        qpeInvRotations.append(np.linalg.inv(Ui))

    # Return both lists.
    return qpeRotations, qpeInvRotations

def computeQFTRotations(numQubits):
    '''
    A helper function to get the QFT and IQFT rotation matrices.
    Parameters:
        numQubits (int) : The number of qubits involved in the QFT.
    Returns:
        The list of QFT rotation matrices, followed by that of IQFT matrices.
    '''
    return QFTRotationsHelper(numQubits), QFTRotationsHelper(numQubits, True)

def QFTRotationsHelper(numQubits, inverse=False):
    '''
    Computes the QFT (or IQFT) rotation matrices.
    Parameters:
        numQubits (int) : The number of qubits involved in the QFT.
        inverse (bool)  : If true, compute IQFT matrices, else QFT matrices.
    Returns:
```

```
        The list of QFT (or IQFT) rotation matrices.
    '''

    # Create a list to hold the matrices.
    rotations = []

    # Instantiate an inverse oscillating factor if necessary.
    sign = -1 if inverse else 1

    # Compute each matrix using the QFT formula and append it to the list.
    for n in range(2, numQubits-1):
        rotations.append(
            np.array(
                [[1, 0], [0, np.exp(sign*1j*2*np.pi/2**n)]],
                dtype=complex
            )
        )

    # Return the list.
    return rotations

def simulateIQFT(psi, numQubits, rotations):
    '''
    Simulate the Inverse Quantum Fourier Transform.
    Parameters:
        psi (QuantumRegister)   : A quantum state.
        numQubits (int)         : The number of qubits in the circuit.
        rotations (list)        : The list of matrices needed for the algoithm.
    Returns:
        The quantum state after the IQFT has been applied.
    '''

    # Apply a Hadamard gate and then the needed rotations to each qubit.
    for i in range(numQubits-2, 0, -1):
        psi.ApplyHadamard(i)
        for j in range(1, i):
            psi.ApplyControlled1QubitGate(j, i, rotations[i-j-1])

    # Apply swaps as needed.
    for i in range(1, numQubits//2):
        psi.ApplySwap(i, (numQubits-1)-i)

    # Return psi.
    return psi

def simulateQFT(psi, numQubits, invRotations):
    '''
    Simulate the Quantum Fourier Transform.
    Parameters:
        psi (QuantumRegister)   : A quantum state.
        numQubits (int)         : The number of qubits in the circuit.
        invRotations (list)     : The list of matrices needed for the algoithm.
    Returns:
        The quantum state after the IQFT has been applied.
    '''

    # Apply swaps as needed.
    for i in range(numQubits//2-1, 0, -1):
        psi.ApplySwap(i, (numQubits-1)-i)

    # Apply a Hadamard gate and then the needed rotations to each qubit.
```

```
    for i in range(1, numQubits-1):
        psi.ApplyHadamard(i)
        for j in range(numQubits-2, i, -1):
            psi.ApplyControlled1QubitGate(j, i, invRotations[j-i-1])

    # Return psi.
    return psi

def simulateQPE(psi, numQubits, qpeRotations, qftInvRotations):
    '''
    Simulate the Quantum Phase Estimation.
    Parameters:
        psi (QuantumRegister)   : A quantum state.
        numQubits (int)         : The number of qubits in the circuit.
        qpeRotations (list)     : The list of matrices needed for the algoithm.
        invRotations (list)     : The list of matrices needed for the IQFT.
    Returns:
        The quantum state after the QPE has been applied.
    '''

    # Apply a hadamard transformation to psi.
    for i in range(1, numQubits-1):
        psi.ApplyHadamard(i)

    # Apply the controlled-U**i operations to psi.
    for i in range(1, numQubits-1):
        psi.ApplyControlled1QubitGate(i, numQubits-1, qpeRotations[i-1])

    # Apply the IQFT to psi.
    psi = simulateIQFT(psi, numQubits, qftInvRotations)

    # Return psi.
    return psi

def simulateIQPE(psi, numQubits, qpeInvRotations, qftRotations):
    '''
    Simulate the Inverse Quantum Phase Estimation.
    Parameters:
        psi (QuantumRegister)   : A quantum state.
        numQubits (int)         : The number of qubits in the circuit.
        qpeInvRotations (list)  : The list of matrices needed for the algoithm.
        qftRotations (list)     : The list of matrices needed for the QFT.
    Returns:
        The quantum state after the IQPE has been applied.
    '''

    # Apply the QFT to psi.
    psi = simulateQFT(psi, numQubits, qftRotations)

    # Apply the inverse controlled-U**i operations to psi.
    for i in range(numQubits-2, 0, -1):
        psi.ApplyControlled1QubitGate(i, numQubits-1, qpeInvRotations[i-1])

    # Apply a hadamard transformation to psi.
    for i in range(1, numQubits-1):
        psi.ApplyHadamard(i)

    # Return psi.
    return psi

def measure(psi, target):
```

```
    '''
    Measure a qubit of a given quantum state.
    Parameters:
        psi (QuantumRegister)   : A quantum state.
        target (int)            : The index of the qubit to measure.
    Returns:
        The QuantumRegister post measurement, as well as the measurement result.
    '''

    # Compute the probability of qubit 0 being in state |1>.
    prob = psi.GetProbability(target)
    result = None

    # Draw random number in [0,1)
    r = np.random.rand()

    # If the random number is less than the probability, collapse to |1>.
    if r < prob:
        psi.CollapseQubit(target, True)
        result = '1'

    # Otherwise collapse to |0>.
    else:
        psi.CollapseQubit(target, False)
        result = '0'

    # In both cases we renormalize the wavefunction and return.
    psi.Normalize()
    return psi, result

def simulateHHL(A, b, applyRYs, t, numQubits=4, extraAncillas=0, shots=2048,
↪   figfile='simulator.png', dpi=300):
    '''
    A function that simulates the HHL algorithm using the process explained in
    section II of https://arxiv.org/pdf/2108.09004.pdf.
    Parameters:
        A (np.ndarray)      : The matrix A from the system Ax = b.
        b (np.ndarray)      : The vector b from the system Ax = b.
        applyRYs (func)     : A function to apply the controlled y rotations.
        t (float)           : A value needed for the QPE matrices.
        numQubits (int)     : The number of qubits used in the algorithm.
        extraAncillas (int) : The number of extra ancillas (used in applyRYs).
        shots (int)         : The number of shots to run.
        figfile (string)    : The file name to save the histogram under.
        dpi (int)           : The dpi of the histogram.
    Returns:
        An estimation of x as given by the HHL algorith.
        Also returns the time taken to perform the emulation.
        Finally saves a plot of the distribution generated by the algorithm.
    '''

    # Define the initial state of psi and get the QPE and QFT matrices.
    initialSate = 0
    qftGates, iqftGates = computeQFTRotations(numQubits)
    qpeGates, iqpeGates = computeQPERotations(A, t, numQubits)

    # Define a dictionary to hold the experimental results.
    probs = {'00': 0, '01': 0, '10': 0, '11': 0}

    # Compute the matrix to prepare the b state.
    bMatrix = np.array([[b[0], -b[1]], [b[1], b[0]]])
```

```
241
242     # Note the time before starting the simulation proper.
243     startTime = time.time()
244
245     # For each shot:
246     for i in range(shots):
247
248         # Initialize a quantum system.
249         psi = simulator.QubitRegister(numQubits + extraAncillas, 'base', initialSate, 0)
250
251         # Intialize the target (last) qubit to the b state.
252         psi.Apply1QubitGate(numQubits-1, bMatrix)
253
254         # Apply the QPE to the target qubit (involving the c qubits).
255         psi = simulateQPE(psi, numQubits, qpeGates, iqftGates)
256
257         # Apply the controlled Y rotations to the anscilla (top) qubit.
258         psi = applyRYs(psi)
259
260         # Apply the IQPE to the target qubit.
261         psi = simulateIQPE(psi, numQubits, iqpeGates, qftGates)
262
263         # Measure the anscilla and target qubit, and track the result.
264         psi, rightResult = measure(psi, target=0)
265         psi, leftResult = measure(psi, target=numQubits-1)
266         probs[leftResult + rightResult] += 1
267
268     # Compute the total amount of time it took to perform the simulation.
269     elapsedTime = time.time() - startTime
270
271     # Compute the percentage of each measurement result.
272     for key in probs:
273         probs[key] /= shots
274
275     # Plot the histogram with exact y-values on top of the bars.
276     fig = plt.figure()
277     plt.bar(range(len(probs)), list(probs.values()), align='center')
278     plt.xticks(range(len(probs)), list(probs.keys()))
279     xlocs, xlabs = plt.xticks()
280     for i, v in enumerate(probs.values()):
281         plt.text(xlocs[i] - 0.2, v + .007, str(round(v, 3)))
282     plt.xlabel('Measurement Result')
283     plt.ylabel('Probability')
284     plt.title(
285         'Simulation with A=['
286         + np.array_str(A[0], precision=2,) + ','
287         + np.array_str(A[1], precision=2) + '], b='
288         + np.array_str(b, precision=2)
289     )
290     plt.savefig('plots/' + figfile, dpi=dpi)
291
292     # Return the estimation of x.
293     return np.array([
294         probs['01'] / (probs['01'] + probs['11']),
295         probs['11'] / (probs['01'] + probs['11'])
296     ]), elapsedTime
```

A.2 emulator.py

```python
import numpy as np
import matplotlib.pyplot as plt
import time

def emulateHHL(A, b, shots=2048, figfile='emulator.png', dpi=300):
    '''
    A function that emulates the HHL algorithm using the fourth equaiton on
    page 3 of https://arxiv.org/abs/0811.3171.
    Parameters:
        A (np.ndarray)  : The matrix A from the system Ax = b.
        b (np.ndarray)  : The vector b from the system Ax = b.
        shots (int)     : The number of shots to run.
        figfile (string): The file name to save the histogram under.
        dpi (int)       : The dpi of the histogram.
    Returns:
        An estimation of x as given by the HHL algorith.
        Also returns the time taken to perform the emulation.
        Finally saves a plot of the distribution generated by the algorithm.
    '''

    # Note the time before starting the emulation proper.
    startTime = time.time()

    # Create a dict for the measure count of the states |00>, |01>, |10>, |11>.
    measure_count = {'00': 0, '01': 0, '10': 0, '11': 0}

    # Repeat the following experiment 'shots' times.
    for i in range(shots):

        # Compute the eigenvalues and eigenvectors of A.
        evals, evecs = np.linalg.eig(A)

        # Compute the coefficients to represent b in the eigenbasis of A.
        beta = np.linalg.solve(evecs, b)

        # Get c, which we take to be the minimal eigenvalue.
        c = min(evals)

        # Compute the state of the anscilla qubit as given in the fourth equation.
        a = np.array([np.sqrt(1 - c**2/evals**2), c/evals])

        # Generate the complete state according to the forth equation and normalize.
        psi = np.kron(beta * evecs, a)
        psi = psi[:,1] + psi[:,2]
        psi = psi /np.linalg.norm(psi)

        # Sample from the distribution.
        r = np.random.rand()
        if r < psi[0]**2:
            measure_count['00'] += 1
        elif r < (psi[0]**2 + psi[1]**2):
            measure_count['01'] += 1
        elif r < (psi[0]**2 + psi[1]**2 + psi[2]**2):
            measure_count['10'] += 1
        else:
            measure_count['11'] += 1

    # Compute the total amount of time it took to perform the simulation.
    elapsedTime = time.time() - startTime
```

```
60
61      # Extract the keys and values from the dictionary.
62      x_values = list(measure_count.keys())
63      y_values = list(measure_count.values())
64
65      # Calculate the probability of each measurement outcome.
66      total_counts = sum(y_values)
67      probabilities = [count / total_counts for count in y_values]
68
69      # Plot the histogram with exact y-values on top of the bars.
70      fig, ax = plt.subplots()
71      bars = ax.bar(x_values, probabilities)
72      for bar, prob in zip(bars, probabilities):
73          yval = bar.get_height()
74          ax.text(
75              bar.get_x() + bar.get_width() / 2, yval,
76              round(yval, 3), ha='center', va='bottom'
77          )
78      plt.xlabel('Measurement Result')
79      plt.ylabel('Probability')
80      plt.title(
81          'Emulation with A=['
82          + np.array_str(A[0], precision=2,) + ','
83          + np.array_str(A[1], precision=2) + '], b='
84          + np.array_str(b, precision=2)
85      )
86      plt.savefig('plots/' + figfile, dpi=dpi)
87
88      # Return the estimation of x.
89      return np.array([
90          measure_count['01'] / (measure_count['01'] + measure_count['11']),
91          measure_count['11'] / (measure_count['01'] + measure_count['11'])
92      ]), elapsedTime
```

A.3 comparison.py

```
1  import numpy as np
2  import emulator as em
3  import simulator as sim
4
5  def runExperiment(
6          A, b, applyRYs, t, numQubits, extraAncillas, shots=2048,
7          emFig='emulator.png', simFig='simulator.png', dpi=300
8      ):
9      '''
10     A function that simulates the HHL algorithm using the process explained in
11     section II of https://arxiv.org/pdf/2108.09004.pdf.
12     Parameters:
13         A (np.ndarray)      : The matrix A from the system Ax = b.
14         b (np.ndarray)      : The vector b from the system Ax = b.
15         applyRYs (func)     : A function to apply the controlled y rotations.
16         t (float)           : A value needed for the QPE matrices.
17         numQubits (int)     : The number of qubits used in the algorithm.
18         extraAncillas (int) : The number of extra ancillas (used in applyRYs).
19         shots (int)         : The number of shots to run.
20         emFig (string)      : The name to save the emulator histogram under.
21         simFig (string)     : The name to save the simulator histogram under.
22         dpi (int)           : The dpi of the histogram.
23     Returns:
24         An estimation of x as given by the HHL algorith.
```

```
        Also returns the time taken to perform the emulation.
        Finally saves a plot of the distribution generated by the algorithm.
    '''

    print('Begin Experiment')

    # Compute the solution, the simulator estimate, and the emulator estimate.
    x = np.linalg.solve(A, b)
    actual = (x / np.linalg.norm(x))**2
    simEstimate, simTime = sim.simulateHHL(
        A, b, applyRYs, t, numQubits, extraAncillas,
        shots=shots, figfile=simFig, dpi=dpi
    )
    emEstimate, emTime = em.emulateHHL(
        A, b, shots=shots, figfile=emFig, dpi=dpi
    )

    # Print results.
    print('A=['
        + np.array_str(A[0], precision=2,) + ','
        + np.array_str(A[1], precision=2) + '], b='
        + np.array_str(b, precision=2)
    )
    print('Actual x\t:', actual)
    print('Simulated x\t:', simEstimate)
    print('Emulated x\t:', emEstimate)
    print('Sim Time\t:', simTime)
    print('Em Time \t:', emTime)
    print('Sim Avg Time\t:', simTime/shots)
    print('Em Avg Time\t:', emTime/shots)
    print('End Experiment')
    print()

def applyToffoli(psi, a, b, c):
    '''
    Apply a Toffoli gate to a quantum register.
    Parameters:
        psi (QuantumRegister)   : A quantum state.
        a (int)                 : The index of the first control qubit.
        b (int)                 : The index of the second control qubit.
        c (int)                 : The index of the target qubit.
    Returns:
        The QuantumRegister after the Toffoli gate has been applied.
    '''

    # Apply the Toffoli using the decomposition in Nielsen and Chuang.
    psi.ApplyHadamard(c)
    psi.ApplyCPauliX(b, c)
    psi.ApplyRotationZ(c, -np.pi/4.)
    psi.ApplyCPauliX(a, c)
    psi.ApplyRotationZ(c, np.pi/4.)
    psi.ApplyCPauliX(b, c)
    psi.ApplyRotationZ(c, -np.pi/4.)
    psi.ApplyCPauliX(a, c)
    psi.ApplyRotationZ(b, -np.pi/4.)
    psi.ApplyRotationZ(c, np.pi/4.)
    psi.ApplyCPauliX(a, b)
    psi.ApplyHadamard(c)
    psi.ApplyRotationZ(b, -np.pi/4.)
    psi.ApplyCPauliX(a, b)
    psi.ApplyRotationZ(a, np.pi/4.)
```

```
    psi.ApplyRotationZ(b, np.pi/2.)

    # Return psi.
    return psi

def RY1(psi):
    '''
    The controlled y-rotation gates for experiment 1.
    Parameters:
        psi (QuantumRegister)   : A quantum state.
    Returns:
        The QuantumRegister after the y-rotations have been applied.
    '''

    # Apply the needed y rotations and return.
    psi.ApplyCRotationY(1, 0, np.pi)
    psi.ApplyCRotationY(2, 0, np.pi/3.)
    return psi

def RY2(psi):
    '''
    The controlled y-rotation gates for experiment 2.
    Parameters:
        psi (QuantumRegister)   : A quantum state.
    Returns:
        The QuantumRegister after the y-rotations have been applied.
    '''

    # Apply the needed y rotations and return.
    psi = applyToffoli(psi, 2, 3, 5)
    psi.ApplyCRotationY(5, 0, np.pi)
    psi = applyToffoli(psi, 2, 3, 5)
    psi = applyToffoli(psi, 1, 2, 5)
    psi = applyToffoli(psi, 3, 5, 6)
    psi.ApplyCRotationY(6, 0, np.pi/3.)
    psi = applyToffoli(psi, 3, 5, 6)
    psi = applyToffoli(psi, 1, 2, 5)
    return psi

runExperiment(np.array([[1 , -1/3], [-1/3 , 1]]), np.array([0, 1]), RY1, 3 * np.pi / 4,
↪  4, 0, emFig='em1.png', simFig='sim1.png')
runExperiment(np.array([[13/2., -1/2.], [-1/2., 13/2]]), np.array([0, 1]), RY2, np.pi/4,
↪  5, 2, emFig='em2.png', simFig='sim2.png')
```