Implementing Quantum Pattern Matching

Aritra Sarkar
Student ID: 4597982
Quantum & Computer Engineering

Naveen Kumar
Student ID: 4622642
Electrical Sustainable Engineering

Delft University of Technology
AP3421-PR Quantum Information Project
January 2018

Abstract

This paper presents the implementation of the Quantum Pattern Matching [1] algorithm in OpenQL compiler and QX simulator platform. The gate level quantum circuit details are worked out and the final results are analyzed.

1 Introduction

The field of quantum algorithm development came into focus in the mid-1980s with the works of David Deutsch [2] and others. A decade later, Peter Shor [3] showed advantage of quantum computing over classical computation in practical disciplines like cryptography leading to widespread research boost in this domain.

Shor's algorithm for factorization is often partnered with Grover's search algorithm as the two most popular quantum algorithms for demonstrating computational advantage. Most quantum algorithms are categorized as Shor-like [4] and Grover-like. While this stereotyping does not give the respective algorithm developers of the ingenuity, the trend reflect some structure of a quantum algorithm's anatomy. The Shor and Grover algorithm has as their integral part, fundamental quantum algorithm primitives, namely Quantum Fourier Transform and Amplitude Amplification respectively.

While Shor's algorithm gives an exponential speedup with respect to its classical factorization counterpart, Grover's algorithm gives only a quadratic speedup. Grover search is however provably optimal [5], thus no other algorithm, classical or quantum can give a better runtime with the same initial conditions. However, it makes up for this lower improvement benefit in 2 ways:

1. Grover assumes an unstructured database search, which is rarely the case. We often have some idea of the data which can be exploited.
2. Searching is a very general problem in computer science and thus the impact factor of the time reduction is of great interest to researchers.

In this respect, the paper [1] we discuss here can be grouped as a Grover search derivative. However we shall see that it essentially inherits only the state marking concept and amplitude amplification construct from the original idea.

1.1 Grover Search

Lov Grover in his paper [6] describes essentially a *quadratic reduction of query complexity.* It clearly *assumes the existence of an oracle function.* While it is a path breaking discovery in itself, the lack of description to construct an Oracle has led to some works [7] on the practicality of quantum search algorithms.

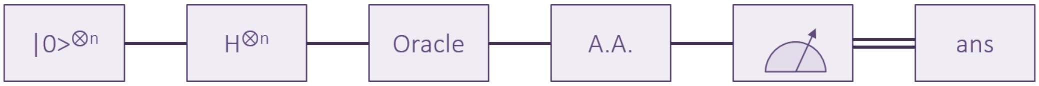


Figure 1: Grover search anatomy

1.2 Oracle Machine

We have progressed to a point in the development of quantum computing hardware that, we can now envision the quantum algorithms for small scale practical application be available in a few years. Thus comes the need to discuss an application based algorithm in its totality.

Oracle functions are integral to an entire class of computer science algorithms, called oracular or black box algorithms. These algorithms assume the existence of an oracle, which simply is a know-it-all answering machine (like the historic bridge between man and god). Oracles interact with a yes or no answer (or a richer state alphabet). As an example, say if it is asked 'what the weather will be like tomorrow?', it cannot process it, however, if it is asked, 'will it rain tomorrow?', it gives the right reply. The easiest way to imagine this system is that the oracles possesses a ledger with the replies of all the potential questions that it can answer. Once an index of a question is supplied, it replies with the value of the answer at the given question index.

A very crucial yet an impractical assumption on oracles is that, it need not be a computable functions, and thus cannot necessarily be constructed with a Turing machine abstraction (as a matter of fact, oracle machines were described by Alan Turing himself for solving unsolvable problems using hyper-computing). However, for every practical algorithm, we need to consider both the construction of the oracle, as well as the oracle query processing.

1.3 Pattern Matching

In this research, we aim to use a quantum search algorithm for pattern matching. Pattern matching is a very useful problem in big data analysis and can be used for a machine learning algorithm's classification stage. Infact, both data mining and quantum search aim to solve the *needle in a haystack* problem.

A 1-dimensional (string) pattern matching problem is formally defined as follows. We assume that the reference pattern is an array $w[0..N-1]$ of length N and that the search pattern is an array $p[0..M-1]$ of length $M \leq N$. We further assume that the elements of w and p are characters drawn from a finite alphabet Σ. The character arrays, w and p are often called strings of characters. We say that pattern p occurs with shift i in text w (or, equivalently, that pattern p occurs beginning at position i in text w) if $0 \leq i \leq N-M-1$ and $w[i+j] = p[j]$, for $0 \leq j \leq M-1$. If p occurs with shift i in w , then we call i a valid shift; otherwise, we call i an invalid shift. The string-matching problem is the problem of finding all valid shifts with which a given pattern p occurs in a given text w.

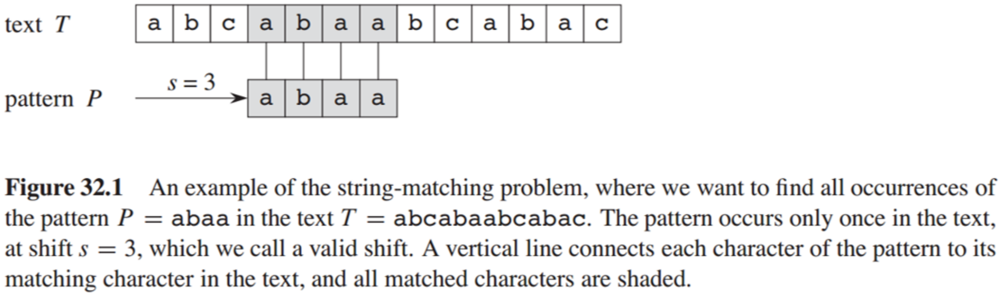


Figure 32.1 An example of the string-matching problem, where we want to find all occurrences of the pattern P = abaa in the text T = abcabaabcabac. The pattern occurs only once in the text, at shift $s = 3$, which we call a valid shift. A vertical line connects each character of the pattern to its matching character in the text, and all matched characters are shaded.

Figure 2: String Matching example (from [8])

2 Development Tools

In the implementation of our algorithm, we used QX simulator, OpenQL, QuInE and MATLAB. A short introduction to these tools are presented in this section.

2.1 QX

Unlike Boolean logic for classical computers, quantum computing can be implemented in different ways like adiabatic quantum computation, cluster state quantum computation or topological quantum computation.

The quantum circuit model is the most common quantum computation model due to its analogy with classical circuits, and thus we prefer an algorithmic description using quantum logic gates operating on qubits.

In order to experiment with algorithms in the absence of a large physical quantum computer, quantum computer simulators are used to verify their feasibility, correctness, scaling and predict their behaviour on a real quantum system.

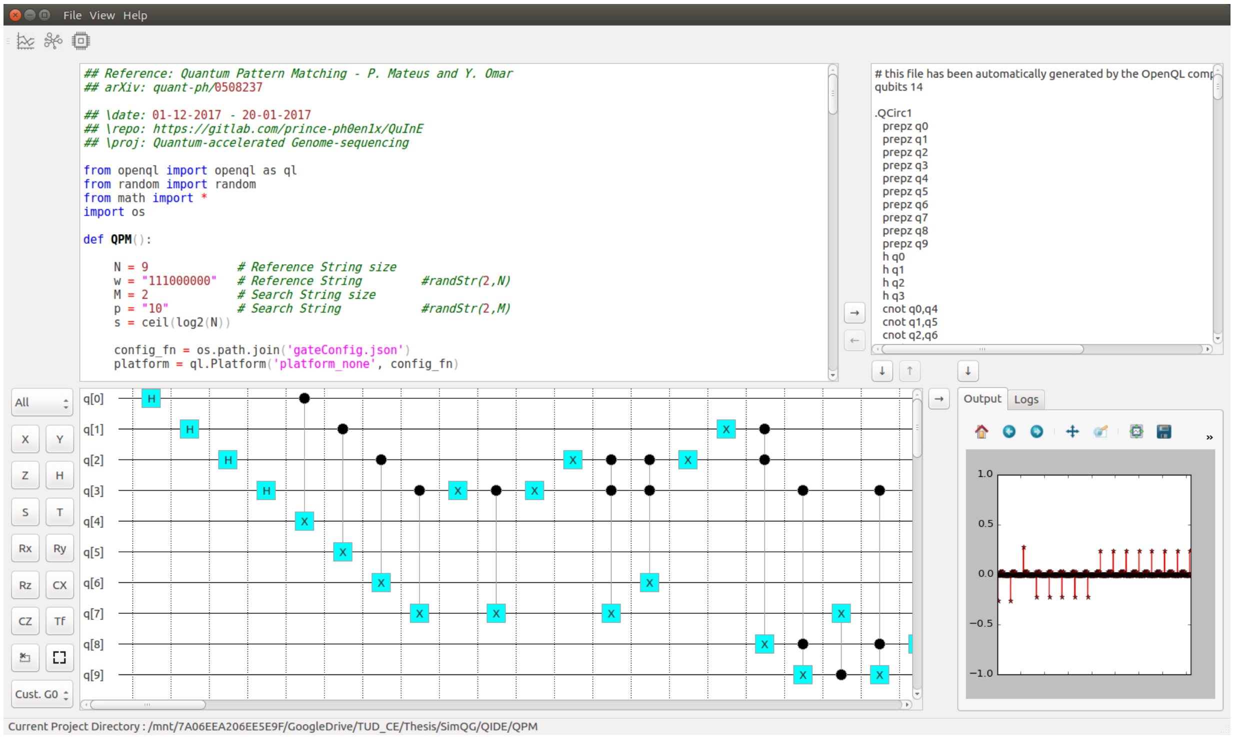
QX [9] is a universal quantum computer simulator that takes as input a specially designed quantum assembly language (QASM) and provides through aggressive optimization, high simulation speeds of up to 34 fully entangled qubits.

2.2 OpenQL

OpenQL is a C++/Python framework for high-level quantum programming. The framework provides a compiler for compiling and optimizing quantum code to produce the intermediate QASM and the compiled micro-code for various target platforms. While the microcode is platform-specific, the QASM is hardware-agnostic and can be simulated on the QX simulator.

2.3 QuInE

Quantum Innovation Environment (QuInE) [10] is an integrated development environment for large-scale quantum algorithm development using OpenQL and QX simulator. It allows parallel development using OpenQL Python, QASM or Circuit building, and integrates features like result display and custom gates as shown in Fig: 3. It is still in pre-alpha release and this project has been instrumental in the understanding of the requirements for a quantum programming tool for high-level algorithms.



```
## Reference: Quantum Pattern Matching - P. Mateus and Y. Omar
## arXiv: quant-ph/0508237

## \date: 01-12-2017 - 20-01-2017
## \repo: https://gitlab.com/prince-ph0en1x/QuInE
## \proj: Quantum-accelerated Genome-sequencing

from openql import openql as ql
from random import random
from math import *
import os

def QPM():

    N = 9               # Reference String size
    w = "111000000"     # Reference String          #randStr(2,N)
    M = 2               # Search String size
    p = "10"            # Search String             #randStr(2,M)
    s = ceil(log2(N))

    config_fn = os.path.join('gateConfig.json')
    platform = ql.Platform('platform_none', config_fn)
```

```
# this file has been automatically generated by the OpenQL comp
qubits 14

.QCirc1
  prepz q0
  prepz q1
  prepz q2
  prepz q3
  prepz q4
  prepz q5
  prepz q6
  prepz q7
  prepz q8
  prepz q9
  h q0
  h q1
  h q2
  h q3
  cnot q0,q4
  cnot q1,q5
  cnot q2,q6
```

Current Project Directory : /mnt/7A06EEA206EE5E9F/GoogleDrive/TUD_CE/Thesis/SimQG/QIDE/QPM

Figure 3: Quantum Innovation Environment

2.4 MATLAB

The proof-of-concept of the algorithm was tested initially in MATLAB using unitary transforms as matrix operations and plotting are easy to develop and test with the pen-and-paper calculations. It is capable of generating the QASM code for the QX, and thus was not only used to check the final algorithm output graphs, but also the OpenQL compilation's QASM output. We will not be discussing much of the MATLAB development as the project's final aim was to use QX.

3 Algorithm Implementation

The Quantum Pattern Matching algorithm [1] is one of the many quantum algorithms curated in the Quantum Algorithm Zoo [11]. In this section we present a detailed description of the algorithm focusing more on the intuition of the circuit and the modifications we carried out. A full worked out and simulated example is presented alongside.

3.1 Jargons

For easy reading of the following sections, the terms associated are described here:

- Σ: The set of characters that composes the pattern matching data, called the alphabet. E.g. the binary digits $\{0, 1\}$
- $A = |\Sigma|$: Size of the alphabet set. E.g. 2 for binary
- N: The length of the reference pattern sequence
- $w \in \Sigma^{\otimes N}$: The reference pattern
- M: The length of the search pattern sequence. $M \leq N$
- $p \in \Sigma^{\otimes M}$: The search pattern
- $i \in N - M + 1$: The match positions for comparison
- $s = \lceil \log_2 i \rceil = \lceil \log_2 (N - M + 1) \rceil$: The number of binary digits required to encode the match positions (refer 1.3)
- $s' = \lceil \log_2 N \rceil$: The number of binary digits required to encode an index of the reference pattern
- Q_σ where, $\sigma \in \Sigma$: Oracle for the character σ

3.2 Initial State Preparation

Grover's algorithm cannot be directly used for pattern matching. In Grover's search, the initial input state is an equal superposition of all the possible strings of the size of the search pattern. Thus, the number of states in the superposition is A^M. The oracle then marks the answer state so that the output is the search pattern. However, if we want to find the index of the pattern matching, we need to initialize the state to a superposition of indices in $N - M + 1$. The oracle marks the index where the pattern matches. So in this implementation, the entire pattern matching comparison is off-loaded to the oracle, and the algorithm is not useful without a description of the oracle construction. The oracle however is an unitary matrix with the diagonal elements being -1 for the answer index, and 1 otherwise. The answer thus needs to be known for the oracle construction, the exact problem we try to avoid.

For matching a sub-string, only a sequential set needs to be considered. The paper describes the input state as:

$$|\psi_0\rangle = \sum_{i=1}^{N-M+1} \frac{1}{\sqrt{N-M+1}} |i, i+1, ..., i+M-1\rangle$$

Henceforth, we would consider an example case of an input alphabet of $\{0, 1\}$ and $w =$ '111000000'. We would like to find the index of the search pattern $p =$ '10'. The above formula for this sample case thus becomes

$$|\psi_0\rangle = \frac{1}{3}\{|0,1\rangle + |1,2\rangle + ... + |7,8\rangle\}$$

However, the circuit described in the paper always generates a unique superposition of N states. How are the last states where the starting index i is, $(N - M + 1) > i > (N - 1)$ described? We see from the state of the input circuit, the digits of the input state saturates at N. Thus, the possible match index states of a pattern length of 2 are

$$\{|0,1\rangle, |1,2\rangle, |2,3\rangle, |3,4\rangle, |4,5\rangle, |5,6\rangle, |6,7\rangle, |7,8\rangle, |8,8\rangle\}$$

Note that the last state has repeats which cannot be described with the recurrence relation. These translate to the search patterns of

$$\{11, 11, 10, 00, 00, 00, 00, 00, 00\}$$

Now, next we need to encode $\{0, 1, .., 8\}$ in binary strings and would thus require $s' = \lceil \log_2 N \rceil = 4$ qubits. The original paper argues that we need to encode only $\{0, 1, .., 7\}$ as the starting index, so only $s = \lceil \log_2(N - M + 1) \rceil = 3$ will suffice. However, since the circuit encodes the remaining pattern index as well for comparison, s qubits are not sufficient. Now, running the initialization circuit with $s' = 4$ qubits gives an equal superposition of states with $i \in \{0, 1, .., 15\}$ Our encoding gives the initial state of

$$|\psi_0\rangle = \frac{1}{4}\{|0000, 0001\rangle, |0001, 0010\rangle, ..., |1110, 1111\rangle, |1111, 1111\rangle\}$$

which translates to,

$$|\psi_0\rangle = \frac{1}{4}\{|0, 1\rangle, |1, 2\rangle, ..., |14, 15\rangle, |15, 15\rangle\}$$

Note that this is different from the initial state till $\frac{1}{3}|7, 8\rangle$ that we wanted to encode. We shall see how this is handled while evaluating the oracle.

The circuit given in the paper shows a gate pattern as an example, but gives no intuition of why it is the way it is. We analyzed this to figure out that, the first set of Hadamard on the first digit place creates a superposition i.e.

$$\frac{1}{4}(|0000\rangle + |0001\rangle + ... + |1111\rangle)$$

on the first s' set of qubits. The set of CNOTs then copies this to the next s' qubits. Thus it creates the state

$$\frac{1}{4}(|0000, 0000\rangle + |0001, 0001\rangle + ... + |1111, 1111\rangle)$$

The Toffolis then implements an increment-by-1 circuit creating the state

$$\frac{1}{4}(|0000, 0001\rangle + |0001, 0010\rangle + ... + |1110, 1111\rangle + |1111, 1111\rangle)$$

Note that the increment saturates at all 1. This idea is not captured in the recurrence equation given in the paper and thus caused some confusion during our design phase.

This part of the circuit is coded as the Python function *Circ1* that initializes the qubits based on the values of s' and M.

3.3 Oracle Construction and Invocation

In Grover's search, the oracle function basically stores the relationship between the database and the search string. This relationship thus needs to change for each search string, making it impractical for implementation.

The core idea in quantum pattern matching is to define a *compile once, run many* approach for the oracle. The algorithm defines multiple oracles, one for each character of the alphabet. The Boolean function that the oracle encodes is -1 for the indices where the reference genome matches the oracle's defining character.

$$f : \{0, 1\}^{2^{s'}} \rightarrow \{-1, 1\}$$

The QPM paper uses a function that maps to $\{0, 1\}$, but we used $\{-1, 1\}$ for convenience with implementing the gate level circuit for the oracle function. Note the oracle construction is independent of the search string, giving this algorithm its usefulness. The oracle circuit assembly depends on the search pattern as shown in Fig: 4. Thus, it can be imagined that at every iteration step, all the A oracles exist in the circuit but only one of them is control activated by the step's corresponding digit of the search pattern. Since the model of quantum computer is based on in-memory computation, the exact circuit need not be pre-compiled if the underlying micro-architecture and classical control is fast enough to allow real time circuit interpretation.

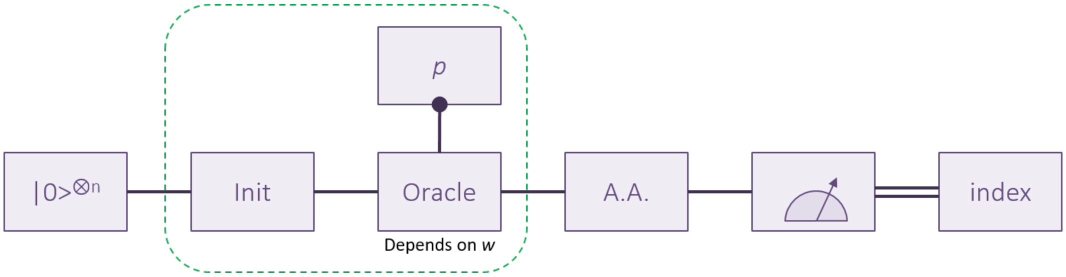


Figure 4: Algorithm anatomy

However, again the circuit for constructing an arbitrary Boolean function is not provided in the paper. We devised our own circuit that can be generated automatically in an OpenQL kernel. In our implementation, we run sequentially through the Boolean function. If the state of a particular index needs to be marked, the Boolean value of the index is taken. Then a CPhase is applied on all the qubits to the oracle, with inverted control on qubits where the Boolean index is 0.

In our example, the Boolean function for σ_1 acting on $s' = 4$ qubits is

$$f_1 = [1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]$$

Thus, we need to mark the positions of $0, 1, 2$, or the qubit states of $|0000\rangle, |0001\rangle, |0010\rangle$ using 3 CPhase Gates over 4 qubits. The first CPhase will have all the controls inverted, for the second CPhase $qb0, qb1, qb2$ are inverted and for the third CPhase $qb0, qb1, qb3$ are inverted. The inversion is carried out by wrapping Pauli-X gate on those qubits before and after the CPhase. The multi-qubit CPhase is converted to a multi-qubit CNOT by wrapping a Hadamard on one of the qubits and then decomposing it as described in Section 3.4.

This part of the circuit is coded as the Python function *Circ2* that adds the required gates to the OpenQL kernel based on the provided Boolean function.

3.4 Amplitude Amplification

The third part of the circuit is the Grover amplification process over the entire non-ancilla qubit set. Both in the initial and the oracle circuits as well as for the Grover gate, the general algorithm of circuit construction has n-qubit control-X gates. These needs to be decomposed to Toffolis using ancillas. This multi-qubit CNOT decomposition as shown in Fig: 5 is coded as the Python function nCX in OpenQL. Further Toffoli decomposition to 1 or 2 qubit gates is not required as Toffoli is natively supported by the QX simulator.

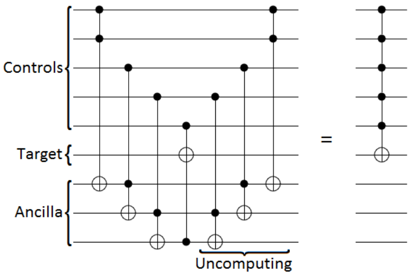


Figure 5: multi-qubit control NOT decomposition

This part of the circuit is coded as the Python function *Circ3* that performs the Grover gate on $s' * M$ qubits.

3.5 Sequential Matching

The randomized run of the algorithm is replaced with a sequential approach as it can gives us an opportunity to break out of the circuit once we get a good dominant solution for a consecutive sub-string match from the entire match pattern. This also simplifies the analysis of the results.

3.6 Qubit Encoding Scheme

The circuits in the paper does not mark the qubit naming and is thus left to us to interpret the LSQ (least significant qubit) and the MSQ (most significant qubit). The ancillas are taken in the MSQ, and the work qubit interpretation is as shown in Fig: 6.

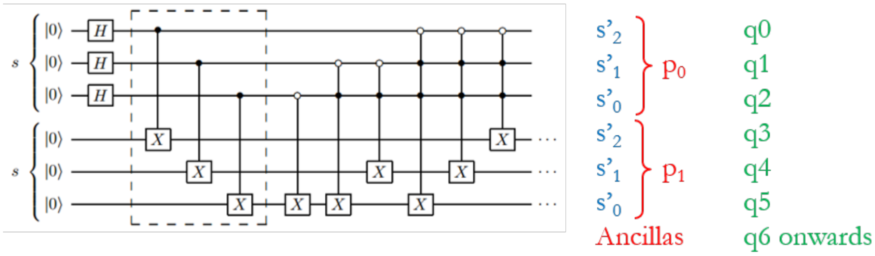
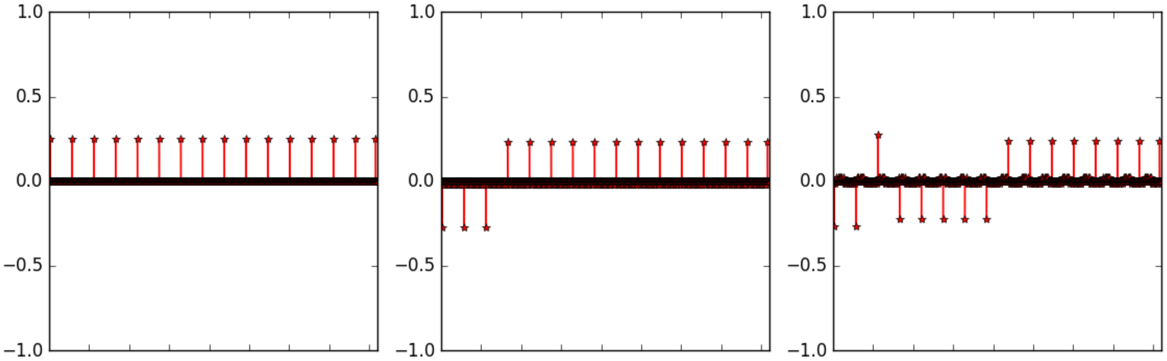


Figure 6: Qubit encoding of the implementation

4 Results

Here the different stages of the run for the search string p = '10' in reference string w = '111000000' is presented.



(a) after QCirc1 in QASM (b) after QCirc2_1 & QCirc3 (c) after QCirc2_0 & QCirc3

Figure 7: Qubit state vector at difference phases of the algorithm's example run

For the example described above, we require $M * s' = 2 * 4 = 8$ data qubits and 6 ancillas. The initial circuit implemented in Kernel 1 picks out only the consecutive states as described in Section 3.2, giving us the peaks of all possible answers as shown in Fig: 7a. Next, the first bit of the pattern is taken, and the oracle σ_1 is called as $p_0 = 1$. This marks the states where the 1st bit of the pattern matches is 1, namely, the 1st 3 positions, reducing the solution set to 3. Next, the second bit of the pattern is taken, and the oracle σ_0 is called as $p_1 = 0$. This marks and amplifies the amplitudes of the states where the 2nd bit is 0. Thus, only the answer state: $i = 2$ gets amplified by both the steps to end in the highest amplitude state.

Note in the figure we show plot the real coefficients of the states for clarity of explaining the marking process. The imaginary coefficients are 0. In the simulator however, we have access to the internal state vector of the defined qubits allowing us to use the directive *display* in QX to probe this state. On measurement in a real quantum device, we will obtain only the eigenvalues of the qubit Hamiltonian, assumed to be 1 for

$|0\rangle$ and 0 for $|1\rangle$. Repeating this process multiple times will give us the amplitude distribution similar to the one obtained in the simulator. How many times we need to repeat the experiment would depend on the confidence with which we want to distinguish the solution state.

For the given example, the solution state is

$$(+0.275390, +0.000000)\,|00000011000100\rangle$$

, while the next two largest amplitudes are of the 1st two states,

$$\{(-0.263672, +0.000000)\,|00000001001000\rangle\,, (-0.263672, +0.000000)\,|00000010000000\rangle\}$$

, up to the precision supported by QX. Thus, we need to distinguish between the probability amplitudes of 0.0758396521 and 0.0.069522923584, requiring in the order of 100 tries to obtain the 3rd decimal place confidence. In this report, we will not take into account the effect of noise as this probability margin itself is already quite small. The solution state is then interpreted based on our encoding scheme described in Section 3.6. It 00000011000100 is first stripped of the already uncomputed ancilla states to get 11000100. It is then grouped in sets of s' bits, to get $\{1100, 0100\}$. The strings in each set is reversed and converted to decimal to get $\{0011, 0010\} = \{3, 2\}$. This ordered set is finally reversed to obtain $\{2, 3\}$, the position where the search string '10' occurs in '111000000'.

4.1 Qubit Complexity

The number of data qubits required for the algorithm is $(M * s')$. The Grover gate requires the largest number of ancilla qubits equaling $(s' - 2)$. Thus the total qubit complexity for this implementation is $2 * M * \lceil \log_2 N \rceil - 2$. There are other ways [12] of expanding a multi-qubit controls to 2-qubit control decomposition which require more gates, but less ancilla qubits. This might be more practical as in a QEC protected quantum processor, qubits are a more scarce resource than the number of gates.

4.2 Gate Complexity

Gate complexity depends on the reference pattern as well as the search pattern as the oracle construction and invoking depends on them respectively. For the example case, Table 1 shows the number of gates required. A worst case upper bound can essentially be calculated but it will not be an interesting one, as it will be searching an all-zero/all-one pattern in a longer all-zero/all-one reference string.

Table 1: Number of gates for searching '10' in '111000000'

Step	**H**	**X**	**CNOT**	**Toffoli**
Initialize	4	8	12	38
1st oracle call	6	20	3	12
Grover gate	18	16	1	12
2nd oracle call	12	26	6	24
Grover gate	18	16	1	12
TOTAL	**58**	**86**	**23**	**98**

Note that this does not include circuit optimization where subsequent same gates are cancelled out to Identity for example. Circuit optimization for such high level algorithm designs like this is tedious to carry out manually and is taken care by the compiler. Currently the OpenQL is not configured to do aggressive optimization. However that allows us to easily identify the designed blocks in the QASM code as well.

In terms of time complexity, it inherits the polynomial speedup from Grover search. The constant factors for gate execution depend of the physical quantum computing hardware. Since simulators are not real quantum devices, we cannot infer the speedup by measuring the run-time on QX simulator.

4.3 Limitations

For binary string matching, there will be two oracle functions, σ_0 for $p_i = 0$, and σ_1 for $p_i = 1$. These oracles will be invoked based on the binary pattern, p. Over all cases in general (i.e. for strings without specific header structure), the first digit of p, $p(0)$ has a 0.5 probability of being 0. For the reference binary string of length N, $eqA = N!/\lceil N/2 \rceil!^2$ is the number of combinations with equal 0 and 1. The rest of the cases, i.e. $2^N - eqA$ cases either has a dominance of 0 or 1 among the first digits of the pattern options. Again for all possible reference string, there will be equal number of such cases. Grover Amplitude Amplification is only effective if the number of marked states are less than unmarked states. So for this case, for $p(0)$, it has a failure chance of $(eqA + (2^N - eqA)/2)/(2^N) = eqA/2^{N+1} + 0.5$, which is more than half! In these failure cases, we end up amplifying the non-marked states as the inversion about mean now depends on a mean more tilted towards the marked states.

We could not come up with an adhoc solution for this, however this probability becomes less when the alphabet size increases, as the chance to dominate 50% of a distribution by a single character diminishes. Thus, the failure probability for an arbitary alphabet set of size A can be calculated as:

$$eq_A = \frac{N!}{\lceil N/A \rceil!^A}$$

$$fp_A = \frac{eq_A + \dfrac{A^N - eq_A}{A}}{A^N} = \frac{(A-1) * eq_A}{A^{N-1}} + \frac{1}{A} \approx \frac{1}{A}$$

This is plotted for varying A in Fig: 8. Thus it is evident that for higher sized alphabet, this problem is not significant. Note that this failure probability is independent of N.

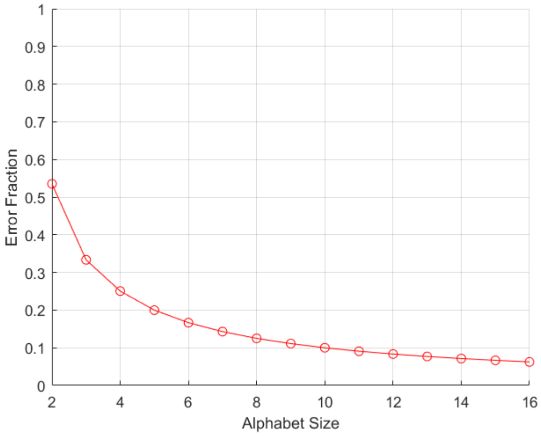


Figure 8: Failure probability for various alphabet sizes

5 Conclusion

In this work we presented an implementation of quantum pattern matching using the OpenQL compiler and the QX simulator platform. The algorithm is presented with an example case, and the circuit construction is presented with gate-level design details. Some modifications of the algorithm are justified. The limitations are calculated and future extensions for overcoming them are suggested.

This project gave us a good exposure to the OpenQL and QX simulator platform. More importantly, it gave us a first hand experience of understand an existing quantum algorithm and modifying the quantum primitives for a specific application. We would like to acknowledge the course instructors for this opportunity.

References

[1] P Mateus and Y Omar. Quantum pattern matching. *arXiv preprint quant-ph/0508237*, 2005.

[2] David Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. In *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, volume 400, pages 97–117. The Royal Society, 1985.

[3] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[4] Dave Bacon. All you compute, and all you qft, is all your life will ever be. Available at http://dabacon.org/pontiff/?p=1291, 2018.

[5] Christof Zalka. Grover's quantum searching algorithm is optimal. *Physical Review A*, 60(4):2746, 1999.

[6] Lov K Grover. Quantum mechanics helps in searching for a needle in a haystack. *Physical review letters*, 79(2):325, 1997.

[7] George F Viamontes, Igor L Markov, and John P Hayes. Is quantum search practical? *Computing in science & engineering*, 7(3):62–70, 2005.

[8] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT Press, 2009.

[9] Nader Khammassi, I Ashraf, X Fu, Carmen G Almudever, and Koen Bertels. Qx: A high-performance quantum computer simulation platform. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 464–469. IEEE, 2017.

[10] Aritra Sarkar. Quantum innovation environment. Available at https://gitlab.com/prince-ph0en1x/QuInE, 2018.

[11] Stephen Jordan. Quantum algorithm zoo. Available at http://math.nist.gov/quantum/zoo/, 2018.

[12] Craig Gidney. Constructing large controlled nots. Available at http://algassert.com/circuits/2015/06/05/Constructing-Large-Controlled-Nots.html, 2018.

A OpenQL Code

```python
## Reference: Quantum Pattern Matching - P. Mateus and Y. Omar
## arXiv: quant-ph/0508237

## \date: 01-12-2017 - 20-01-2017
## \repo: https://gitlab.com/prince-ph0en1x/QuInE
## \proj: Quantum-accelerated Genome-sequencing

from openql import openql as ql
from random import random
from math import *
import os

def QPM():

    N = 9            # Reference String size
    w = "111000000"    # Reference String        #randStr(2,N)
    M = 2            # Search String size
    p = "10"         # Search String           #randStr(2,M)
    s = ceil(log2(N))

    config_fn = os.path.join('gateConfig.json')
    platform = ql.Platform('platform_none', config_fn)

    total_qubits = 2*s*M-2
    prog = ql.Program('qg', total_qubits, platform)

    # Initialization
    qk1 = ql.Kernel('QCirc1',platform)
    Circ1(qk1,s,M)
    prog.add_kernel(qk1)

    # Oracle Kernels
    qk2_0 = ql.Kernel('QCirc2_0',platform)
    qk2_1 = ql.Kernel('QCirc2_1',platform)

    # Grover Amplitude Amplification
    qk3 = ql.Kernel('QCirc3',platform)
    Circ3(qk3,s,M)

    f0 = []
    f1 = []
    for wi in range(0,2**s):
        if wi >= N:
            f0.append(False)
            f1.append(False)
        elif w[wi] == '0':
            f0.append(True)
            f1.append(False)
        elif w[wi] == '1':
            f0.append(False)
            f1.append(True)

    for pi in range(0,M):
        if p[pi] == '0':
            Circ2(qk2_0,f0,s,pi*s,s*M)
            prog.add_kernel(qk2_0)
        elif p[pi] == '1':
            Circ2(qk2_1,f1,s,pi*s,s*M)
            prog.add_kernel(qk2_1)
        # Improve: Reset Kernels
        prog.add_kernel(qk3)

    prog.compile(False, "ASAP", False)
    display()
    #showQasm(1)

def Circ1(k,s,M):
    for Qi in range(0,(s+1)*M):
        k.prepz(Qi)
    for si in range(0,s):
        k.gate("h",si)
    for Mi in range(0,M-1):
        for si in range(0,s):
            k.gate("cnot",Mi*s+si,Mi*s+s+si)
        for si in range(0,s):
            k.gate("x",Mi*s+s-1-si)
            nc = []
            for sj in range(Mi*s+s-1,Mi*s+s-1-si-1,-1):
                nc.insert(0,sj)
            for sj in range(Mi*s+s+s-1,Mi*s+s+s-1-si-1,-1):
                nCX(k,nc,sj,s*M)
            k.gate("x",Mi*s+s-1-si)

def Circ2(k,f,s,q,anc):
    for fi in range(0,len(f)):
        if f[fi]:
            fis = format(fi,'0'+str(s)+'b')
            for fisi in range(0,s):
                if fis[fisi] == '0':
                    k.gate("x",q+fisi)
            k.gate("h",q+s-1)
            nc = []
            for qsi in range(q,q+s-1):
                nc.append(qsi)
            nCX(k,nc,q+s-1,anc)
            k.gate("h",q+s-1)
            for fisi in range(0,s):
                if fis[fisi] == '0':
                    k.gate("x",q+fisi)

def Circ3(k,s,M):
    for si in range(0,s*M):
        k.gate("h",si)
        k.gate("x",si)
    k.gate("h",s*M-1)
    nc = []
    for sj in range(0,s*M-1):
        nc.append(sj)
    nCX(k,nc,s*M-1,s*M)
    k.gate("h",s*M-1)
    for si in range(0,s*M):
        k.gate("x",si)
        k.gate("h",si)
    return

def nCX(k,c,t,anc):
    nc = len(c)
    if nc == 1:
        k.gate("cnot",c[0],t)
    elif nc == 2:
        k.toffoli(c[0],c[1],t)
    else:
        k.toffoli(c[0],c[1],anc)
        for i in range(2,nc):
            k.toffoli(c[i],anc+i-2,anc+i-1)
        k.gate("cnot",anc+nc-2,t)
        for i in range(nc-1,1,-1):
            k.toffoli(c[i],anc+i-2,anc+i-1)
        k.toffoli(c[0],c[1],anc)
    return

def display():
    file = open("test_output/qg.qasm","a")    # Append display at end (simulator directive)
    file.write("display")
    file.close()

def showQasm(r):
    file = open("test_output/qg.qasm","r")
    print("\nCODE FILE\n\n")
    n = r
    for line in file:
        if line[0:7] == '.QCirc2':
            n = 1
        if n == 1:
            print (line,end='')
        if line[0:7] == '.QCirc3':
            n = r
    print ()
    file.close()

def randStr(szA,sz):
    # Generates a random string of length 'sz' over the alphabet of size 'szA' in decimal
    bias = 1/szA    # Improve: add bias here
    rbs = ""
    for i in range(0,sz):
        rn = random()
        for j in range(0,szA):
            if rn < (j+1)*bias:
                rbs = rbs + str(j)    # Improve: BCD version
                break
    return rbs

if __name__ == '__main__':
    QPM()
```

B QASM Code

```
qubits 14

.QCirc1
    prepz q0
    prepz q1
    prepz q2
    prepz q3
    prepz q4
    prepz q5
    prepz q6
    prepz q7
    prepz q8
    prepz q9
    h q0
    h q1
    h q2
    h q3
    cnot q0,q4
    cnot q1,q5
    cnot q2,q6
    cnot q3,q7
    x q3
    cnot q3,q7
    x q3
    x q2
    toffoli q2,q3,q7
    toffoli q2,q3,q6
    x q2
    x q1
    toffoli q1,q2,q8
    toffoli q3,q8,q9
    cnot q9,q7
    toffoli q3,q8,q9
    toffoli q1,q2,q8
    toffoli q1,q2,q8
    toffoli q3,q8,q9
    cnot q9,q6
    toffoli q3,q8,q9
    toffoli q1,q2,q8
    toffoli q1,q2,q8
    toffoli q3,q8,q9
    cnot q9,q5
    toffoli q3,q8,q9
    toffoli q1,q2,q8
    x q1
    x q0
    toffoli q0,q1,q8
    toffoli q2,q8,q9
    toffoli q3,q9,q10
    cnot q10,q7
    toffoli q3,q9,q10
    toffoli q2,q8,q9
    toffoli q0,q1,q8
    toffoli q0,q1,q8
    toffoli q2,q8,q9
    toffoli q3,q9,q10
    cnot q10,q6
    toffoli q3,q9,q10
    toffoli q2,q8,q9
    toffoli q0,q1,q8
    toffoli q0,q1,q8
    toffoli q2,q8,q9
    toffoli q3,q9,q10
    cnot q10,q5
    toffoli q3,q9,q10
    toffoli q2,q8,q9
    toffoli q0,q1,q8
    toffoli q0,q1,q8
    toffoli q2,q8,q9
    toffoli q3,q9,q10
    cnot q10,q4
    toffoli q3,q9,q10
    toffoli q2,q8,q9
    toffoli q0,q1,q8
    x q0

.QCirc2_1
    x q0
    x q1
    x q2
    x q3
    h q3
    toffoli q0,q1,q8
    toffoli q2,q8,q9
    cnot q9,q3
    toffoli q2,q8,q9
    toffoli q0,q1,q8
    h q3
    x q0
    x q1
    x q2
    x q3
    x q0
    x q1
    x q2
    h q3
    toffoli q0,q1,q8
    toffoli q2,q8,q9
    cnot q9,q3
    toffoli q2,q8,q9
    toffoli q0,q1,q8
    h q3
    x q0
    x q1
    x q2
    x q0
    x q1
    x q3
    h q3
    toffoli q0,q1,q8
    toffoli q2,q8,q9
    cnot q9,q3
    toffoli q2,q8,q9
    toffoli q0,q1,q8
    h q3
    x q0
    x q1
    x q3

.QCirc3
    h q0
    x q0
    h q1
    x q1
    h q2
    x q2
    h q3
    x q3
    h q4
    x q4
    h q5
    x q5
    h q6
    x q6
    h q7
    x q7
    h q7
    toffoli q0,q1,q8
    toffoli q2,q8,q9
    toffoli q3,q9,q10
    toffoli q4,q10,q11
    toffoli q5,q11,q12
    toffoli q6,q12,q13
    cnot q13,q7
    toffoli q6,q12,q13
    toffoli q5,q11,q12
    toffoli q4,q10,q11
    toffoli q3,q9,q10
    toffoli q2,q8,q9
    toffoli q0,q1,q8
    h q7
    x q0
    h q0
    x q1
    h q1
    x q2
    h q2
    x q3
    h q3
    x q4
    h q4
    x q5
    h q5
    x q6
    h q6
    x q7
    h q7

.QCirc2_0
    x q4
    x q5
    h q7
    toffoli q4,q5,q8
    toffoli q6,q8,q9
    cnot q9,q7
    toffoli q6,q8,q9
    toffoli q4,q5,q8
    h q7
    x q4
    x q5
    x q4
    x q6
    x q7
    h q7
    toffoli q4,q5,q8
    toffoli q6,q8,q9
    cnot q9,q7
    toffoli q6,q8,q9
    toffoli q4,q5,q8
    h q7
    x q4
    x q6
    x q7
    x q4
    x q6
    h q7
    toffoli q4,q5,q8
    toffoli q6,q8,q9
    cnot q9,q7
    toffoli q6,q8,q9
    toffoli q4,q5,q8
    h q7
    x q4
    x q6
    x q4
    x q7
    h q7
    toffoli q4,q5,q8
    toffoli q6,q8,q9
    cnot q9,q7
    toffoli q6,q8,q9
    toffoli q4,q5,q8
    h q7
    x q4
    x q7
    x q4
    h q7
    toffoli q4,q5,q8
    toffoli q6,q8,q9
    cnot q9,q7
    toffoli q6,q8,q9
    toffoli q4,q5,q8
    h q7
    x q4
    x q5
    x q6
    x q7
    h q7
    toffoli q4,q5,q8
    toffoli q6,q8,q9
    cnot q9,q7
    toffoli q6,q8,q9
    toffoli q4,q5,q8
    h q7
    x q5
    x q6
    x q7

.QCirc3
    h q0
    x q0
    h q1
    x q1
    h q2
    x q2
    h q3
    x q3
    h q4
    x q4
    h q5
    x q5
    h q6
    x q6
    h q7
    x q7
    h q7
    toffoli q0,q1,q8
    toffoli q2,q8,q9
    toffoli q3,q9,q10
    toffoli q4,q10,q11
    toffoli q5,q11,q12
    toffoli q6,q12,q13
    cnot q13,q7
    toffoli q6,q12,q13
    toffoli q5,q11,q12
    toffoli q4,q10,q11
    toffoli q3,q9,q10
    toffoli q2,q8,q9
    toffoli q0,q1,q8
    h q7
    x q0
    h q0
    x q1
    h q1
    x q2
    h q2
    x q3
    h q3
    x q4
    h q4
    x q5
    h q5
    x q6
    h q6
    x q7
    h q7
display
```