Bachelor's Thesis (Academic Year 2019)

Controlled Gate Compilation for IBMQ

Keio University
Faculty of Policy Management

Shin Nishio

Tokuda/Murai/Kusumoto/Nakamura/Takashio/
Van Meter/Uehara/Mitsugi/Nakazawa/Takeda Labs
Advancing Quantum Architecture project

January 2019

Bachelor's Thesis Abstract   Academic Year 2019

Controlled Gate Compilation for IBMQ

Summary

Quantum circuits for useful quantum applications such as Grover's algorithm and quantum walk include many multi-controlled gates, but these require the implementation of many two-qubit gates. The two-qubit gate is more affected by noise than the single-qubit gate, and requires a longer calculation time, and thus has a higher cost. Therefore, in this research, we design and implement a transpiler pass (a kind of compiler that treats quantum circuits as DAG) based on the specifications of IBMQ and Qiskit for optimizing quantum circuits that include multiple controlled gates. The transpiler performs gate sorting and simplification with the reduction of the number of control qubits using the QuineMcCluskey method. The results of this study can be evaluated by the number of gates and the circuit depth after decomposing the compiled quantum circuit into IBMQ's minimum instruction set (u1, u2, u3, and cx).

The results are reported in this binding. This result may increase the size of the executable algorithm.

Keywords

Quantum Circuit Optimization, Reversible Computing, Quantum Compilation

Faculty of Policy Management
Keio University

Shin Nishio

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Backgrounds

Quantum algorithms that will shown computational superiority to classical computers often include complicated controlled gates when described as quantum circuits. In this study, we aim to help programmers easily call these complicated controlled gates and execute them at the lowest possible cost (the number of gates).

1.2 Contributions

In this research, we extended the function of Qiskit, an important Quantum Computing SDK, and released it as CongX, a library that executes and optimizes a complex controlled gate. The extension of the quantum gate class and the complex set of controlled gates and their information can now be passed to the transpiler pass as a DAG. The transpiler pass enables us to optimize complex Controlled Gates with an algorithm called Common Target Rule.

1.3 Organization of this Dissertation

In Chapter 1, we described the background of the paper and the specific contributions of this study. In Chapter 2, we systematically describe the background of this research and the prerequisite knowledge. Chapters 3 describe methods for reducing the number of gates required to execute a controlled gate by using a compiler (transpiler). In Chapter 3, we will extend the `gate` class of Qiskit and define complex Controlled Gates. Next, Chapter **??** describes the proposed method of rearranging and sorting the quantum gates as a preparation before optimizing the quantum circuit.Chapter 4 describes a logic synthesis method that is effective when multiple Controlled Gates are adjacent. Chapter **??** describes a method for determining in what order transpiler passes created based on the method of Chapter 4 and Chapter **??** can be applied to a circuit to perform effective optimization.

In Chapter 5, we evaluate the entire set of transpiler passes created in this study, and in Chapter 9, we summarize and discuss future prospects.

Appendix **??** describes and implements some quantum gates added in CongX. Appendix A describes a tool that converts Reversible Logic Synthesis Benchmarks, a tool for benchmarking compilers developed by the author, into Qiskit and CongX.

Chapter 2

Preliminary

This chapter covers the basic knowledge related to this study.

2.1 Quantum Information

Quantum information science is an information science that uses a quantum mechanical state and its behavior as information. In quantum information science, there are fields such as quantum information processing that performs communication and computation using quantum information, and quantum information theory that uses quantum information as an information source. This section describes quantum information processing, especially quantum computation.

2.1.1 Bit and Qubit

Bit is an abbreviation for binary digit, and can represent one binary number using two discrete states. The names of the two discretized states are mainly 0 and 1.

Qubit is an extension of bit. The states corresponding to the 0 state and the 1 state in the bit can be regarded as the following states in the linear space. The numbers or symbols in the ket have no meaning and serve as labeling (name the vectors).

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

2.1.2 No-cloning

There is no unitary gate to replicate unknown quantum states[1][2].

Theorem 1. *No cloning theorem*

$$\neg \exists U for \forall |\psi\rangle, U(|\psi\rangle \otimes |0\rangle) = |\psi\rangle \otimes |\psi\rangle$$

The Bloch sphere's z axis represents the probability of observing $|0\rangle$ and $|1\rangle$, and the angle φ on the xy plane represents the phase[1].

2.1.3 Superposition

A qubit can take a 0 state, a 1 state, and a superposition of them, which can be expressed as a linear combination of a 0 state and a 1 state.

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle$$

At this time, there are the following restrictions.

$$\sqrt{|\alpha|^2 + |\beta^2|} = 1$$

2.2 Quantum Computation

Quantum computation uses quantum mechanical phenomena such as quantum entanglement and superposition. A digital computer encodes data into bits and performs calculations, whereas a quantum computer encodes data into qubits (quantum bits) and performs quantum information processing.

2.3 Quantum Circuit and Quantum Gates

Some quantum computers, including IBMQ, describe the manipulation of qubits using what is called a quantum circuit. This is an extension of the logic circuit of electronic circuits in existing computers. A quantum circuit is an ordered (or partially ordered) sequence of quantum gates. The quantum gate is described graphically by placing a box on the line as shown in the figure. One time line represents one qubit. Fig. 2.1 is an example of quantum circuit.

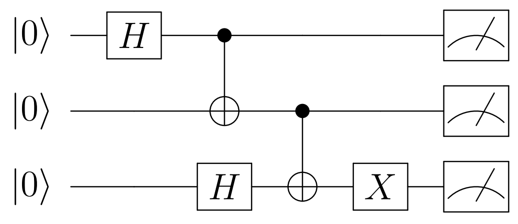


Figure 2.1: An example of Quantum Circuit

2.4 Graphs

A graph is a type of data structure that can handle the structure of things.Graphs are roughly divided into directed graphs and undirected graphs. The directed graph is defined[3].

[1]It is also possible to output the phase as a probability by rotating the qubit.

Definition 1. *Directed Graph*
A directed graph is a set of E, V and f where E is edges, V is vertices, and f is a map.

$$G := (f, V, E)$$

Each element of f is an ordered pair consisting of two elements of V to an element of E.

$$f : E \to V \times V$$

DAG is an abbreviation of Directed Acyclic Graph and is defined as following definition 2.

Definition 2. *Directed Acyclic Graph(DAG)*
A directed acyclic graph is a directed graph with no cycles.

Edges represent dependencies. A DAG is a partial ordering.

2.5 Existing Quantum Computing System

In recent years, the development of science and technology has made it possible to handle dozens of qubits[4]. Fig. 2.2 shows the architecture of a quantum computer created by IBM and Rigetti.

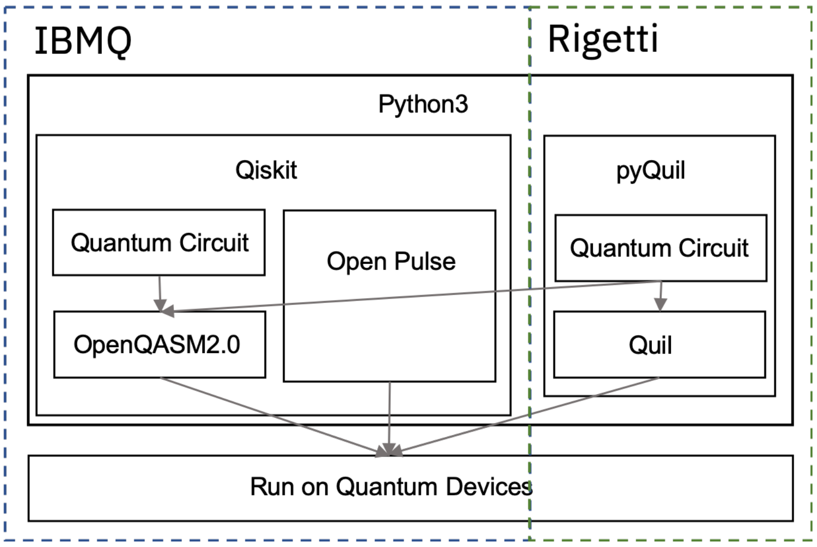


Figure 2.2: Existing Quantum Computer Architecture

Chapter 3

Extend Gate class

This chapter briefly describes how quantum circuits are handled inside the existing Qiskit(ver 0.14.1), and then describes classes and instructions added by CongX to easily handle more complicated Controlled Gates.

3.1 Circuit and classes

Qiskit treats instructions and gates as Python classes. Tree.1 shows the class dependencies, including the classes proposed by the authors.

Tree.1: Dependency of Qiskit Gate class

```
qiksit.circuit
 |__Instruction class
 |   |__c_if function
 |__Measure class
 |__Gate class
     |__ControlledGate class(added in Qiskit 0.14.0)
         |__BinaryControlledGate class (Proposal)
         |__GeneralizedControlledGate class (Proposal)
```

3.2 gate.control() method and ControlledGate class

`gate.control()`(hereafter referred as just `control()`) is a new method released in Qiskit `0.14.0` on 10th December, 2019[5]. A `gate` object can create a gate to which a control qubit is added by calling a `control()` method. The argument of `control()` is an integer, which can be used to represent the number of control qubits. When the programmer calls the `control()` method, a function called `add_control()` is executed, creating a gate for the `ControlledGate` class. `ControlledGate` class is a child class of `Gate` class.

As shown in Listring 3.1, the gate created by the control method can be taken into the quantum circuit by `append`.

Listing 3.1: Control methods taken in CongX

```
qc.append(ugate.control(3),[0,1,2],[3])
```

3.3 c_if and condition

c_if is a method of the Instruction class. By using this method, it is possible to add a classical condition on classical register. This method has classical register and value as arguments.

The gate called by c_if has a parameter condition. The variable q_condition of binary_control() and basis_control() described later is implemented based on condition.

3.4 classes added in CongX

BinaryControlledGate class GeneralizedControlledGate class

q_condition: num_ctrl_qubits(int), bin_str(str), basis_str(str)

3.5 Methods added in CongX

I added binary_control() as a new method of the gate class. It takes one integer and one string as arguments. The integer represents the number of control qubits, and the string representing the binary (called bin$_str)fordeterminecontrolandanti-control.Binariesareimplementedasstrinsteadofinteger,bec$
$Naurform$[7].

$$\begin{array}{rcl} <bin\_str> & ::= & '0' \\ & | & '1' \\ & | & <bin\_str><bin\_str> \end{array}$$

I added basis_control() as a new method of gate class. It takes one integer and two strings. The integer and one string is used for same notation in binary_control(). An additional string (called basis_str) is treated as the basis for control qubits. Our implementation accepts X, Y, Z bases as input, and you can add additional bases like diagonal basis by extending q_condition. basis_str is a character string as defined below in Backus-Naur form.

$$\begin{array}{rcl} <basis\_str> & ::= & 'X' \\ & | & 'Y' \\ & | & 'Z' \\ & | & <basis\_str><basis\_str> \end{array}$$

binary_control() and basis_control() can be called as shown in Listing 3.2.

Listing 3.2: Control methods added in CongX

```
qc.append(ugate.binary_control(3, '101'),[0,1,2],[3])
qc.append(ugate.basis_control(3, '101', 'xzy'),[0,1,2],[3])
```

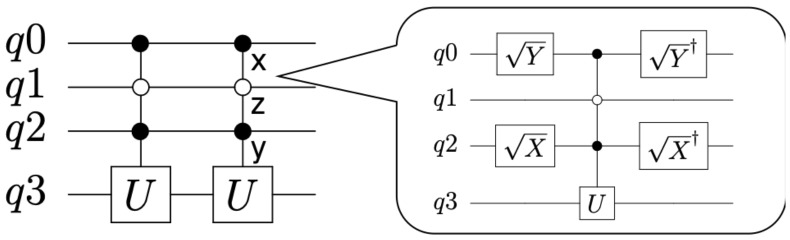


Figure 3.1: The circuit on the left has two examples of controlled gates. These gates can be called as shown in Listing 3.2. The subscripts of the control qubit correspond to the axes of the Bloch sphere. The circuit on the right is a circuit obtained by unrolling the gate on the right side of the circuit on the left.

3.6 Gates added in CongX

I defined the following requirements and added gates in CongX to satisfy them.

1. It can be executed on the backends (quantum devices and simulators) simply by applying Unroller without selecting an option such as Ancilla qubits.
2. Use short and simple names.

In order to satisfy the first requirement, a decomposition method was always inserted inside the definition of the gate.

The following design and implementation were performed to satisfy the second requirement.

- A gate is called ”a¡name¿gate” if it is anti-controlled by one qubit.
- When the number of (positive) control qubits of a `ControlledGate` is *n* (a positive integer of 2 or more), it is called `cn`¡name¿gate.
- When a gate allows control and anti-control, it is called `b`¡name¿gate, omitting binary-controlled to express control and anti-control.
- When there are plural target qubits of a certain `Controlled gate`, m (multi) is inserted after the control information (a, b, cn), and is referred to as (a, b, cn) mgate.

The following gates shown in Table 3.1, 3.2 have been implemented in CongX.

Table 3.1: Quantum Gates added in CongX

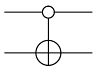
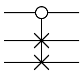
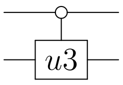
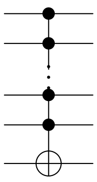
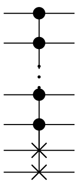
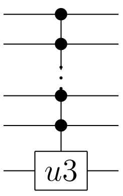
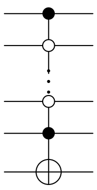
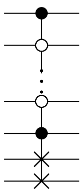
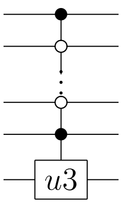
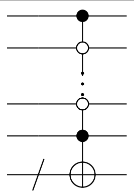
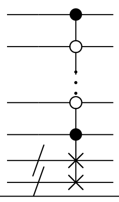
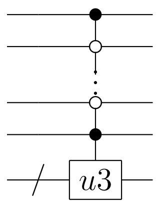
Gate(Usage Guide)	name and class	sample code
	ax, `AnotGate(BinaryControlledGate)`	qc.ax(0,1)
	aswap, `ASWAPGate(BinaryControlledGate)`	qc.aswap(0,1,2)
	au3, `Au3Gate(BinaryControlledGate)`	qc.au3(theta, phi, lam, 0, 1)
	cnx, `CNXGate(ControlledGate)`	qc.cnx(0, 1, 2, 3)
	cnswap, `CNSWAPGate(ControlledGate)`	qc.cnswap(0, 1, 2, 3)
	cnu3, `CNu3Gate(ControlledGate)`	qc.cnu3(theta, phi, lam, [0, 1, 2, 3], [4])
	bx, `BXGate(BinaryControlledGate)`	qc.bx('1001', 0, 1, 2, 3, 4)
	bswap, `BSWAPGate(ControlledGate)`	qc.bswap('1001', [0, 1, 2, 3],[4, 5])
	bu3, `Bu3Gate(BinaryControlledGate)`	qc.bu3('1111',theta, phi, lam, [0, 1, 2, 3], [4])

Table 3.2: Quantum Gates added in CongX

Gate(Usage Guide)	name	sample code
	bmx, `BMXGate(BinaryControlledGate)`	qc.bmx('1001',[0, 1, 2, 3], [4, 5])
	bmswap, `BMSWAPGate(BinaryControlledGate)`	qc.bmswap('1001',[0, 1, 2, 3], [4, 5])
	bmu3, `BMu3Gate(BinaryControlledGate)`	qc.bmu3('1001',theta, phi, lam, [0, 1, 2, 3], [4

3.7 DAG circuit

DAG can be used to represent a set of tasks that need to be ordered. This can also be applied to quantum computer tasks. Qiskit defines DAG corresponding to quantum circuits as `DAGCircuit` class.

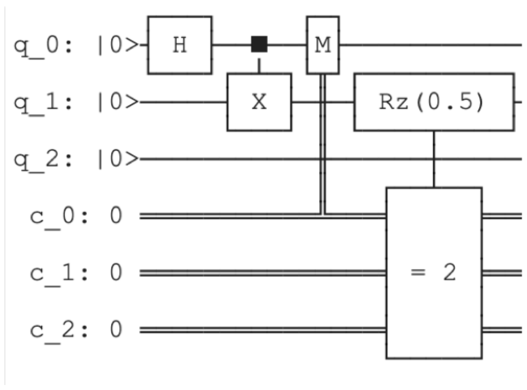
`DAGCircuit` treats the following as nodes.

- Inputs: qubit or classical bit
- Instructions: Instructions such as gates and measurements.
- Outputs: qubit or classical bit

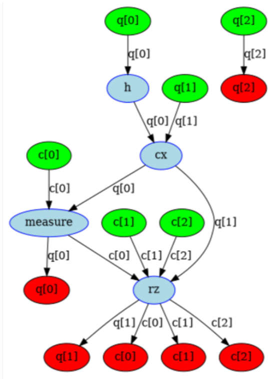
Transitions of qubits and classical bits are treated as edges. The figure shows a quantum circuit and the corresponding `DAGCircuit`.

The transpiler passes of Qiskit converts the quantum circuit into `DAGCircuit` before processing because of the ease of ordering. At this time, `DAGCircuit` node is treated as `DAGNode` class, and the variables of DAGNode class are listed below[8].

1. `name`: name of nodes
2. `op`: gate definition
3. `qargs`: qubits(edges)
4. `cargs`: bits(edges)
5. `condition`: condition for `c_if`



(a) A Quantum Circuit written in Qiskit



(b) The DAGCircuit converted from Fig. 3.2a

Figure 3.2: (a) shows an example of Quantum Circuit, and (b) shows the DAGCircuit converted from (a).

To preserve the correspondence between quantum circuits and `DAGCircuit`, and to pass complicated `ControlledGate` information to the transpiler passes, we had to do the following:

1. Add the `q_condition` variable to `DAGNode` class
2. Enable the `circuit_to_dag()` function and `dag_to_circuit()` function to convert quantum circuits including `q_condition`.

Chapter 4

Multi-Multi-Controlled Gates optimization

When multiple control quantum gates are adjacent, optimization is possible based on several rules

4.1 Common Target Rule

The Common Target Rule(CTR)[9] is a rule that simplifies reversible logic, proposed by Arabzadeh et al. This method can be used when the same gate is consecutive with the same target bit (qubit).

Fig.5.2 shows an example of a quantum circuit to which CTR can be applied. In the quantum gates of the quantum circuit, first and second qubit are control qubits, and third qubit is a target qubit. Fig. 4.2 is the Karnaugh map corresponding to this quantum circuit.

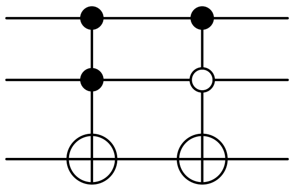


Figure 4.1: An example for a quantum circuit which can be simplified by CTR.

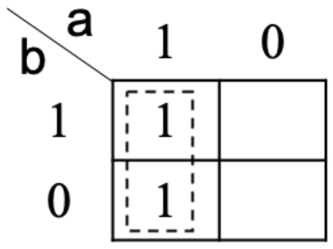


Figure 4.2: Karnaugh map for Fig. 5.2. The broken line represents the Disjunctive normal form for the circuit.

The Karnaugh map is constructed so that the Hamming distance between adjacent cells is 1, and the logical expression can be visually simplified. Disjunctive normal forms can be generated by grouping adjacent true parts in the Karnaugh map.

Since visualization is difficult when the number of dimensions increases, we used the Quine–McCluskey method developed by Willard V. Quine[10][11] and extended by Edward J. McCluskey [12], which is more suitable for computer calculations. In this study, we use the Quinema-Clusky method implemented by Thomas Pircher et al.[13]

4.2 Commutation

To simplify multiple quantum gates with CTR, the set of quantum circuits to be optimized must be adjacent. To create such a situation, an algorithm for rearranging quantum gates and sub-circuits is required.

Chapter 5

Evaluation

We evaluate this research using a quantum circuit implementation of the Quantum Walks on Complete Graphs shown in Listlings. 5.1.

Reversible Logic Synthesis Benchmarks[14]are also going to be evaluated.

5.1 Execution order of transpiler pass

- Optimization of the proposed method
- Optimization of the proposed method + optimization by Qiskit compiler
- Optimization with Qiskit compiler + Optimization of the proposed method

The following program is a quantum walk on a complete graph. The quantum register t0 indicates the initial state of the quantum walker, and the quantum register coin indicates the coin unitary. The result of the transition of the walker using t0 and coin as the control qubit group is stored in t1.

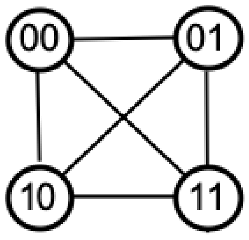


Figure 5.1: The complete graph with 4 nodes

Listing 5.1: Quantum Walk on Complete Graph

```
# import CongX.extensions.standard
from CongX.extensions.standard.bmx import BMXGate
import numpy as np
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import execute
from qiskit import IBMQ

t0 = QuantumRegister(2)
```

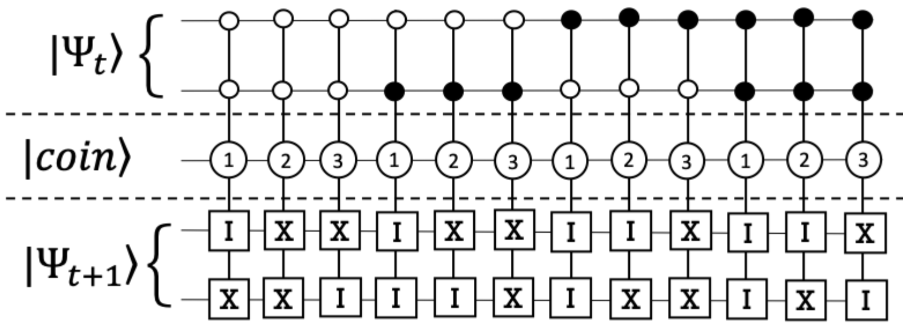


Figure 5.2: Quantum Circuit for Quantum Walk on graph shown in Fig.**??**

```
t1 = QuantumRegister(2)
coin = QuantumRegister(2)
result = ClassicalRegister(2)
qc = QuantumCircuit(t0, t1, coin, result)
qc.h(coin[0])
qc.h(coin[1])

qc.bmx('0001',[t0[0],t0[1],coin[0],coin[1]],[t1[1]])
qc.bmx('0010',[t0[0],t0[1],coin[0],coin[1]],[t1[0]])
qc.bmx('0011',[t0[0],t0[1],coin[0],coin[1]],[t1[0],t1[1]])

qc.bmx('0101',[t0[0],t0[1],coin[0],coin[1]],[t1[1]])
qc.bmx('0110',[t0[0],t0[1],coin[0],coin[1]],[t1[0]])
qc.bmx('0111',[t0[0],t0[1],coin[0],coin[1]],[t1[0],t1[1]])

qc.bmx('1001',[t0[0],t0[1],coin[0],coin[1]],[t1[1]])
qc.bmx('1010',[t0[0],t0[1],coin[0],coin[1]],[t1[0]])
qc.bmx('1011',[t0[0],t0[1],coin[0],coin[1]],[t1[0],t1[1]])

qc.bmx('1101',[t0[0],t0[1],coin[0],coin[1]],[t1[1]])
qc.bmx('1110',[t0[0],t0[1],coin[0],coin[1]],[t1[0]])
qc.bmx('1111',[t0[0],t0[1],coin[0],coin[1]],[t1[0],t1[1]])

qc.measure(t1, result)
qc.draw(output = 'mpl')
```

The Quantum Cost (Maslov)[15] proposed for Reversible Logic Synthesis is a measure of how many CNOT gates a circuit can decompose. Since this is originally designed for a reversible circuit, it does not include the cost of single qubit gates. Therefore, the Quantum Cost (QChallenge) used in IBM Quantum Challenge [16] is also described. This is obtained by adding 1/10 the number of single qubit gates (u1, u2, and u3) to the number of CNOT gates.

Transpiler Pass	Quantum Cost(Maslov)	Quantum Cost (QChallenge)	time
Unroller	$768 + \alpha$	847.1	
BMXUnroller + CTR + Unroller			
BMXUnroller + CTR + Unroller + 1q opt			

Chapter 6

Conclusion and Future Works

In this research, we proposed and implemented a function for calling a complex Controlled Gate and a compiler specialized for its optimization. This research has made it possible to optimize quantum gates with up to about 30 control qubits.

The remaining problem is to develop heuristic algorithms to optimize larger quantum circuits when they need to be handled in the future.

In this study, the number of CNOT gates in the quantum circuit was optimized. When quantum information processing is performed using quantum error correction in the future, it will be necessary to set other optimization targets such as the number of T gates and the depth of the circuit.

When the rules used to optimize quantum circuits (corresponding to the CTR in this study) become more sophisticated in the future, software tools such as compiler compilers will be required. How to incorporate a mechanism that optimizes quantum circuits without defining rules, such as machine learning, into existing systems is a topic for future research.

The outcome of this research is a toolset that exists on top of the SDK called Qiskit. It is desirable that processing requiring high speed such as a compiler be performed on a lower layer language on toolsets such as LLVM. However, since this research is a prototype, it was implemented in Python in order to mesh smoothly with Qiskit.

Acknowledgment

I would like to thank my supervisor, Professor Rodney Van Meter. I have been in his lab for three years since I was sophomore. He gave me not only the method of research but also the attitude of research and various lessons. I thank him for giving me the opportunity to present at conferences and write dissertations. Thanks to his financial support, I have been able to participate and present at a number of wonderful academic and hackathon events, including CQIS2018, AQIS2018, IWQC2019, QiskitCamp2019, QiksitCampAsia2019.

I would like to thank my supervisor, Project Research Associate Takahiko Satoh. He worked with me on almost every project I worked on. He carefully discussed each of my childish proposals for research, provided theoretical support and new perspectives.

Thanks to Jun Murai. He taught me a serious attitude towards the Internet and technology.

Thanks to my seniors Takaaki Matsuo and Yulu Pan. They showed me how to work in university life and lab activities. Takaaki consulted me at many times and provided appropriate advice. Yulu showed me a good programmer by co-authoring the dissertation.

Thanks to Ryosuke Satoh and Yasuhiro Okura. I grew up by working with them for Mitou Target. Ryosuke also strongly supported my activities as a Kenkyu Group Leader and taught me a lot of things. It was great for me that Yasuhiro gave me many questions and advice from a keen point of view.

Thanks to Ryusei Siiba. Talking to him in the lab always gave me witty stories and I was able to change my mind.

Thanks to Yuki Matsumoto. She cheer-uped me and enjoyed my student life together.

Thanks to Yuto Maekawa. He told me how interesting physics is.

Thanks to Yinjie Zhou, Shigetora Miyashita, Nozomi Tanetani, Sitong Liu, Makoto Nakai, Yutaro Yoshii, Shaimay Shah, Kosuke Onishi, and Shuta Kobayashi for their active contribution to AQUA.

Thanks to the members of the IBMQ team, Qiskit Community. Thanks to them, I was able to conduct various experiments and learn.

Thanks to Atsushi Matsuo and Yuri Kobayashi from IBM Research for working with me for IBM Quantum Challenge.

Thanks to Rudy Raymond, Takashi Imamichi, Naoki Kanazawa, Toshinari Itoko, and Luciano Bello from IBM Research for useful discussions.

Finally, I thank the Faculty and all members of the Tokuda, Murai, Kusumoto, Nakamura, Takashiom, Van Meter, Uehara, Mitsugi, Nakazawa, Takeda Joint Research Group (RG) for all the supports.

Bibliography

[1] William K Wootters and Wojciech H Zurek. A single quantum cannot be cloned. *Nature*, Vol. 299, No. 5886, pp. 802–803, 1982.

[2] DGBJ Dieks. Communication by epr devices. *Physics Letters A*, Vol. 92, No. 6, pp. 271–272, 1982.

[3] Dieter Jungnickel. *Graphs, Networks and Algorithms*. Springer Publishing Company, Incorporated, 3rd edition, 2010.

[4] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, Vol. 574, No. 7779, pp. 505–510, 2019.

[5] Qiskit Development Team. Qiskit api documentation ¿ release notes, 2020. https://qiskit.org/documentation/release_notes.htmlqiskit0140 (Accessed on Wednesday, 22th January, 2020 at 02.30 am).

[6] Python Software Foundation. Python 3.8.1 documentation ¿the python standard library ¿ built-in functions, 2020. https://docs.python.org/3/library/functions.html?highlight=intint (Accessed on Wednesday, 22th January, 2020 at 16.30 pm).

[7] Daniel D. McCracken and Edwin D. Reilly. *Backus-Naur Form (BNF)*, p. 129–131. John Wiley and Sons Ltd., GBR, 2003.

[8] Qiskit Development Team. Qiskit api documentation: qiskit.dagcircuit ¿ dagnode, 2020. https://qiskit.org/documentation/api/qiskit.dagcircuit.DAGNode.html (Accessed on Wednesday, 22th January, 2020 at 16.30 pm).

[9] Mona Arabzadeh, Mehdi Saeedi, and Morteza Saheb Zamani. Rule-based optimization of reversible circuits, 2010.

[10] Willard V Quine. The problem of simplifying truth functions. *The American mathematical monthly*, Vol. 59, No. 8, pp. 521–531, 1952.

[11] Willard V Quine. A way to simplify truth functions. *The American Mathematical Monthly*, Vol. 62, No. 9, pp. 627–631, 1955.

[12] Edward J McCluskey. Minimization of boolean functions. *The Bell System Technical Journal*, Vol. 35, No. 6, pp. 1417–1444, 1956.

[13] Thomas Pircher. A python implementation of the quine mccluskey algorithm, 2012. available at https://github.com/tpircher/quine-mccluskey.

[14] Shin Nishio. RLSB-CongX-Qiskit, 2020. RLSB-CongX-Qiskit is available at https://github.com/sfc-aqua/RLSB-CongX-Qiskit.

[15] Dmitri Maslov. Reversible logic synthesis benchmarks page: Quantum cost, 2009. available at https://webhome.cs.uvic.ca/ dmaslov/definitions.html.

[16] Takahiko Satoh Yuri Kobayashi, Atsushi Matsuo and Shin Nishio. Ibm quantum challenge, 2019. available at https://quantumchallenge19.com/.

[17] D. MASLOV. Reversible logic synthesis benchmarks page. *http://www.cs.uvic.ca/maslov/*, 2005. Reversible logic synthesis benchmarks page is available at https://webhome.cs.uvic.ca/ dmaslov/.

[18] Andrew W. Cross, Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. Open quantum assembly language, 2017.

Appendix A

Convert Reversible Logic Synthesis Benchmarks to Qiskit

In this chapter, I describe a set of reversible circuits for benchmarking logic synthesis, called the Reversible Logic Synthesis Benchmark, and the converter I have implemented for using it in Qiksit and CongX. My implementation is available on the GitHub repository[14].

A.1 Overview of Reversible Logic Synthesis Benchmarking

Reversible Logic Synthesis Benchmarks Page[17] (Hereinafter referred to as RLSB) is a set of reversible circuits written and maintained by Dmitri Maslov et al. RLSB can be used for benchmark logic synthesis. Examples of the RLSB circuit include an adder and Hamming coding functions. The circuits included in the RLSB describe a reversible circuit using a characteristic machine-readable format called .tfc. There are roughly 15 types of algorithms, and the number of circuits is about 90.

A.2 Notation and Gates used in RLSB

The .tfc file is described using the notation shown in the table **??**. The feature is that there are only two functions used to describe the quantum circuit itself: Generalized Toffoli gate and Generalized Fredkin gate.

A Generalized Toffoli gate defined in RLSB is a Toffoli gate in which the number of control qubits is extended to a non negative integer. It can be defined by indicating the sum of the number of control qubits and the number of target qubits after `t`. For example, `t3 a, b, c` is a CX gate that uses a and b as control qubits and c as a target qubit. X gates without control qubits can also be called as `t1 a`.

A Generalized Fredkin gate defined in RLSB is a Fredkin gate in which the number of control qubit is extended to a non negative integer. It can be defined by indicating the sum of the number of control qubits and the number of target qubits after `f`. For example, `f4 a,b,c,d` is a CCSWAP gate that uses a and b as control qubits and c and d as target(SWAP) qubits.

notation	function	example
`#`	Comment out	`# Comment`
`.v`	Declare the list of variables(Qubits).	`.v a,b,c,d,e`
`.i`	Declare the list of variables used as input.	`.i a,b,c`
`.o`	Declare the list of variables used as output.	`.o d,e`
`.c`	Declare values for constant input variables.	`.c 0`
`t`	Generalized Toffoli	`t2 a,b`
`f`	Generalized Fredkin	`f3 f,d,g`

Table A.1: Notations used in .tfc file

A.3 Convert .tfc to .py

In this study, RLSB was used to evaluate the transspira path. In this research, Qtkit and its extension CongX class are used for compiling, so it is necessary to convert .tfc to the corresponding form. The following methods can be used for this conversion.

1. Convert .tfc to low-level language (.QASM) such as OpenQASM and import into Qiskit and CongX.
2. Convert directly to Qiskit / CongX (written as .py) quantum circuit
3. Use other language processing system or SDK such as Q and Cirq.

Since dealing with other language processing systems would be very expensive for humans, I rejected third option and considered first and socond option in particular. The advantages of using the first option are that OpenQASM[18] has fewer updates than Qiskit itself, and its grammar is defined in simple Backus-Naur form, so the cost of managing code is low. On the other hand, the advantage of using the second option is that it can easily handle more complicated circuits and instructions. As a result, I used the technique listed in the second option in this study.

A.4 A brief tutorial

This section shows how to convert RLSB to .py using a converter and import it into Qiksit / CongX. Because it is wrapped in a single function, it can be converted very easily.

Install the required packages. Open Terminal etc. and execute the following commands A.1. Qiskit and CongX are installed in `pythonX.X/site-packages`.

Listing A.1: Install required packages

```
pip install qiskit
pip install CongX
```

Clone the repository. `git clone git@github.com:parton-quark/RLSB-OpenQASM.git`

Prepare an appropriate tfc fileA.2.

Listing A.2: test.tfc.txt

```
# (c) Copyright 2020 Shin Nishio
# Distributing this file without an express written permission or erasing this note is prohibited.
.v a,b,c,d
.i a,b,c,d
.o a,b,c,d
BEGIN
t1 a
t1 d
t2 d,c
t4 a,c,d,b
t3 c,d,b
t3 a,d,b
t2 a,d
t4 b,c,d,a
t2 b,c
t3 a,d,c
END
```

Import the function `tfc_to_qiskit()` in the repository and specify the path of the .tfc file the code A.3.

Listing A.3: Conversion

```
from tfc_to_qiskit import tfc_to_qiskit
tfc_to_qiskit('test.tfc.txt')
```

The following file is saved as `RLSBCQ _(input_file_name).py`. Variables inside the .tfc are converted to linear quantum registers.

Listing A.4: Generated file RLSBCQ_test.py

```
# Reversible Logic Synthesis Benchmarks Page is a set of reversible circuits written and
    maintained by Dmitri Maslov et al.
# This file is converted by RLSB-CongX-Qiskit. RLSB-CongX-Qiskit is written by Shin Nishio.
    bib file is https://github.com/parton-quark/RLSB-CongX-Qiskit/blob/master/RLSB-CongX-
    Qiskit.bib
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from CongX.extensions.standard import cnx
# Can be run by installing Qiskit and CongX.
# !pip install qiskit
# !pip install CongX
# (c) Copyright 2020 Shin Nishio
# Distributing this file without an express written permission or erasing this note is prohibited.
# valuables{'a': 0, 'b': 1, 'c': 2, 'd': 3}
q = QuantumRegister(4)
qc = QuantumCircuit(q)
# inputs{'a': 0, 'b': 1, 'c': 2, 'd': 3}
# outputs{'a': 0, 'b': 1, 'c': 2, 'd': 3}
# BEGIN
qc.cnx(0)
qc.cnx(1)
qc.cnx(3)
qc.cnx(3, 2)
qc.cnx(0, 2, 3, 1)
qc.cnx(2, 3, 1)
qc.cnx(0, 3, 1)
qc.cnx(0, 3)
qc.cnx(1, 2, 3, 0)
qc.cnx(1, 2)
qc.cnx(0, 3, 2)
```

```
# END
```

After importing and visualizing in the code A.5, you can see that it is working properly in Fig. A.1. I plan to improve visualization in Qiskit and CongX.

Listing A.5: Import .py and visualization

```
import RLSBCQ_test
# The imported circuit can be handled as RLSBCQ_filename.qc.
RLSBCQ_test.qc.draw(output='mpl')
```

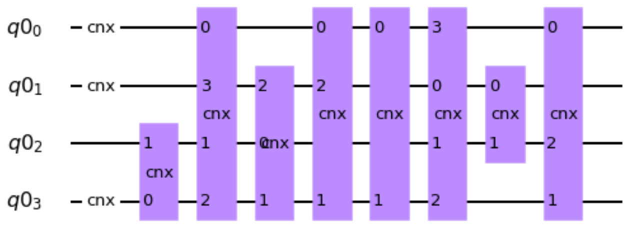


Figure A.1: Visualized Reversible Circuit