國立臺灣大學電機資訊學院電機工程學研究所
碩士論文
Department of Electrical Engineering
College of Electrical Engineering and Computer Science

National Taiwan University
Master Thesis

利用Qsyn實作ZX-calculus對量子電路的動態優化

Dynamic Quantum Circuit Optimization by ZX-calculus using Qsyn

呂承樺
Cheng-Hua (Anita) Lu

指導教授：黃鐘揚 博士
Advisor: Chung-Yang (Ric) Huang, Ph.D.

中華民國 112 年 7 月
July 2023

doi:10.6342/NTU202303169

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

利用Qsyn實作ZX-calculus對量子電路的動態優化

Dynamic Quantum Circuit Optimization by ZX-calculus using Qsyn

本論文係 呂承樺（R10921058） 在國立臺灣大學電機工程學系完成之碩士學位論文，於民國 112 年 7 月 11 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Electrical Engineering on 11 July 2023 have examined a Master's thesis entitled above presented by Cheng-Hua (Anita) Lu (R10921058) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

	梁伯嵩	江介宏
（指導教授 Advisor）		
洪士灝	李建模	

系主任 Director: 李建模

doi:10.6342/NTU202303169

誌謝

特別感謝Ric這兩年多的指導，從老師身上我學到了單純熱愛探索新知的態度，也跟老師學會了很多專業技術的技巧與經驗；此外，老師總是提醒我們在認真研究之餘，別忘了好好享受生活、鍛鍊身體，從老師言傳身教中，真的是獲益良多、受益匪淺。

在碩班的這些日子裡，很有幸能夠與實驗時的精英們一起研究量子電路這個雖然很新潮卻相對冷門的領域，很感謝在這條相對清冷的研究道路中有戰友們彼此砥礪，一起進步、一起學習；特別印象深刻那段沒日沒夜開發軟體的日子，短短幾個月好幾萬行的成就真的是一生中難得的經歷，雖然身心疲累，精神上卻是無比滿足，沒有你們便沒有這樣的人生經歷，實在是萬分感恩！

謝謝Faline、佳妤、萱萱、妮妮、閉閉常常在我研究做不出來的時候陪我吃飯、聊天，幫我加油打氣，謝謝妳們總是那麼相信我，有妳們在真的是我很大很大的幸運。

謝謝家人們的支持與鼓勵，在很多研究遇到瓶頸的時候不斷的鼓勵我，也謝謝你們的體諒與包容，讓我可以無後顧之憂的做研究，在專業上精益求精，謝謝你們。

最後，謝謝充滿勇氣、從來沒有放棄的自己，在一次一次的挫敗中再站起來，在這個自己從未想過的領域中堅持了兩年，這一切的努力與付出真的很值得。

doi:10.6342/NTU202303169

中文摘要

量子電路優化在提高量子計算系統的效率和性能方面起著關鍵作用。本論文以整合量子邏輯閘分解層次調整和動態減縮算法為核心，對量子電路進行了全面的優化研究。為了方便實現和分析這些演算法，我們開發了一個名為Qsyn的C++工具。Qsyn提供了一個靈活的框架，可以高效地執行和探索各種優化策略。我們通過對各種測試案例的應用，展示了我們方法的有效性。實驗結果顯示，與現有方法相比，我們的方法在總量子邏輯閘數、2量子位邏輯閘數、量子電路深度等方面平均上分別達到11.5％、10.5％、14.8％的減少。取得了顯著的改善。此外，我們將Qsyn的性能與廣泛使用的PyZX工具進行了比較，觀察到執行時間大幅減少，特別是對於大規模電路。這些結果突顯了Qsyn的優越性，以及它在質量和計算效率方面優化量子電路的適用性。總體來說，本論文通過提供一個強大的工具和一個優化的工作流程，為量子電路優化領域做出了貢獻，使得電路優化更加有效和可擴展。

關鍵字：量子電路優化、ZX-calculus、量子邏輯閘分解層次調整、動態減縮算法、Qsyn

doi:10.6342/NTU202303169

ABSTRACT

Quantum circuit optimization plays a crucial role in improving the efficiency and performance of quantum computing systems. In this thesis, we present a comprehensive study on optimizing quantum circuits by integrating decomposition level adjustment and dynamic reduction algorithms. To facilitate the implementation and analysis of these algorithms, we have developed a novel quantum circuit optimization framework called Qsyn in C++. Qsyn is very efficient in terms of the execution time and provides the flexibilities in exploring various optimization strategies. We demonstrate the effectiveness of our approach by applying it to a diverse set of test cases. Our experimental results show that our algorithm can lead to significant improvements in quantum circuit gate count, 2-qubit-gate count, and circuit depth. The reduction rate are 11.5%, 10.5%, and 14.8%, respectively, when compared to the existing approaches. Furthermore, Qsyn is much faster than the widely used quantum circuit optimization tool, PyZX, particularly for large-scale circuits. The results highlight the superiority of Qsyn and its suitability for optimizing quantum circuits in both quality and computational efficiency. Overall, this thesis contributes to the field of quantum circuit optimization by providing a powerful tool and an optimized workflow that enable more effective and scalable quantum circuit optimization.

Keywords: Quantum Circuit Optimization, ZX-calculus, Decomposition Level Adjustment, Dynamic Reduction Algorithm, Qsyn

doi:10.6342/NTU202303169

CONTENTS

誌謝 i

中文摘要 ii

ABSTRACT iii

CONTENTS iv

LIST OF FIGURES vi

LIST OF TABLES viii

Chapter 1 Introduction 1

1.1 Quantum Circuit Optimization 1

1.2 Previous Works 2

1.3 Contribution of the Thesis 3

1.4 Organization of the Thesis 3

Chapter 2 Preliminaries 5

2.1 Quantum Circuits 5

2.2 Single-Qubit Gates 6

2.3 Multi-Qubit Gates 9

2.4 Quantum Circuit Synthesis Flow 10

Chapter 3 Quantum Circuit Modeling by ZX-calculus 13

3.1 Introduction to ZX-calculus 13

3.2 PyZX: Automated Diagrammatic Python Tool for Quantum Circuit using ZX-calculus 21

Chapter 4 Optimization by Dynamic Reduction Algorithm 26

4.1 Previous Work 26

doi:10.6342/NTU202303169

4.2 Case Study: 3-Toffoli Quantum Circuit 27
4.3 Decomposition Level Adjustment 29
4.4 Dynamic Reduction Algorithm 32
Chapter 5 Implementation 35
5.1 Qsyn: Quantum Circuit Synthesis Framework 35
5.2 Execution of Dynamic Reduction Algorithm 41
Chapter 6 Experimental Results 45
6.1 Experiment Setup 45
6.2 Comparison of PyZX and Qsyn 48
6.3 Decomposition Level Adjustment Comparison 50
6.4 Dynamic Reduction Algorithm 53
Chapter 7 Conclusion and Future Work 56
REFERENCE 58
APPENDIX 62
A.1 ZX File Format 62
A.2 Command List for Qsyn 67
A.3 Optional Arguments for ZXGSimp 70

doi:10.6342/NTU202303169

LIST OF FIGURES

Fig. 2-1 An example of a quantum circuit diagram 6

Fig. 2-2 A diagram of the CNOT-gate 9

Fig. 2-3 A diagram of the Toffoli gate 10

Fig. 2-4 Quantum circuit synthesis flow [15] 11

Fig. 3-1 The three ZX-diagrams represent the same matrix 14

Fig. 3-2 Z-spider and X-spider interpretation as a linear map 14

Fig. 3-3 Hadamard edge and Hadamard box illustration 15

Fig. 3-4 Transformation between a standard quantum circuit and ZX- diagrams 16

Fig. 3-5 A diagrammatic interpretation for spider fusion rule 17

Fig. 3-6 A diagrammatic interpretation of identity removal rule 17

Fig. 3-7 A diagrammatic interpretation of Hadamard rule 18

Fig. 3-8 A diagrammatic interpretation for local complementation rule 18

Fig. 3-9 A diagrammatic interpretation of the pivot rule 19

Fig. 3-10 A diagrammatic interpretation for pivot gadget rule 19

Fig. 3-11 A diagrammatic interpretation for phase gadget rule 20

Fig. 3-12 An overview of the functionality of PyZX [26] 21

Fig. 3-13 The overall flow for full reduction implementation 25

Fig. 4-1 A original circuit with 3 Toffoli gates 27

Fig. 4-2 The original ZX-diagram with 21 T gates 28

Fig. 4-3 The simplified ZX-diagram after full reduction 28

Fig. 4-4 The simplified ZX-diagram with 15 T gates 28

Fig. 4-5 Circuit implementing a Toffoli gate 30

doi:10.6342/NTU202303169

Fig. 4-6 ZX-diagram implementing a Toffoli gate 30
Fig. 4-7 Optimized ZX-diagram after dynamic reduction algorithm with the density of 9.5676 34
Fig. 5-1 The structure for Qsyn 36
Fig. 5-2 The command line interface after executing Qsyn 41
Fig. 5-3 The command line interface after reading a QASM file.....41
Fig. 5-4 The command line interface after converting QCir to ZXGraph in level 3 ..42
Fig. 5-5 The command line interface after dynamic reduction 43
Fig. 5-6 The command line interface after writing the ZX-graph into a zx file 43
Fig. 5-7 The command line interface after equivalence checking.....44
Fig. 6-1 The workflow for the experimental group and the control group.....45
Fig. 6-2 The experiment workflow for the comparison of different decomposition level.....50
Fig. 6-3 The experiment workflow for the comparison of dynamic reduction algorithm and full simplification using PyZX 53
Fig. A-1 Vertex type in ZX-diagram form 63
Fig. A-2 ZX-diagram implementing a Toffoli gate with labels 63
Fig. A-3 A basic example of a ZX-diagram.....66

doi:10.6342/NTU202303169

LIST OF TABLES

Table 3-1 ZX and Matrix Form of Common Quantum Gates [24].....15

Table 3-2 ZX-rules and their corresponding functions implied in PyZX [26] 22

Table 4-1 The circuit information for the original one and the optimized one.....29

Table 4-2 Decomposition Level for a Toffoli gate.....31

Table 4-3 The circuit information for the original one and the optimized ones by full reduction and dynamic reduction 34

Table 5-1 Some commands and their descriptions for system information.....38

Table 5-2 Some commands and their descriptions for QCir.....39

Table 5-3 Some commands and their descriptions for ZXGraph 39

Table 5-4 Some commands and their descriptions for Tensor.....40

Table 6-1 Circuit information for our benchmark 47

Table 6-2 Runtime comparison for PyZX and Qsyn 49

Table 6-3 Circuit metrics for the control group and optimized metrics for different decomposition levels 51

Table 6-4 Circuit metrics for the control group and our dynamic reduction algorithm ... 54

Table A-1 All commands provided in Qsyn and their description.....67

Table A-2 Optional arguments list for ZXGSimp.....70

doi:10.6342/NTU202303169

Chapter 1 Introduction

This opening chapter provides a comprehensive introduction to the quantum circuit optimization. It outlines the importance of optimizing quantum circuits to enhance the efficiency and performance of quantum computing systems. A review of previous works in the field of quantum circuit optimization is presented, highlighting key techniques and approaches that have been investigated. The chapter then introduces the contributions of the thesis, with a particular emphasis on the development of Qsyn, a novel optimization tool. Furthermore, the integration of decomposition level adjustment and dynamic reduction algorithms for circuit optimization is discussed. Finally, an overview of the organization of the entire thesis is provided, outlining the subsequent chapters and their specific focuses.

1.1 Quantum Circuit Optimization

Quantum circuit optimization refers to the process of improving the implementation of the quantum circuits by reducing their complexities, resource requirements, and overall computational costs. It involves applying various techniques, algorithms, and heuristics [1-4] to minimize the key parameters of the circuits, such as circuit depth, total gate count, and T-count (the number of T-gates or non-Clifford gates).

The optimization of quantum circuits is essential for several reasons. First, it helps mitigate the noise and decoherence, which are the key adverse factors in harming the reliability and accuracy of quantum computations. By reducing the number of gates and

doi:10.6342/NTU202303169

circuit depths, the vulnerability to errors and the impact of noise can be minimized, leading to more reliable and robust quantum computations.

Second, quantum circuit optimization is crucial for practical realization on existing and near-future quantum devices. Quantum processors have limitations in terms of qubit connectivity, gate set availability, and gate operation durations. Optimizing the circuit structure and gate arrangement to match the hardware constraints can improve the feasibility of executing quantum algorithms on specific architectures.

1.2 Previous Works

In the field of quantum circuit optimization, research efforts have been dedicated to enhancing the efficiency and performance of quantum computing systems. A key focus has been on optimizing factors such as gate count, T-count, and circuit depth, which have significant implications for the quality of the quantum circuit design. The reduction of T-count is particularly important as it reduces computational complexity and improves overall circuit performance. On the other hand, optimizing circuit depth is critical for achieving faster execution times and mitigating potential errors resulted from gate interactions and noise.

To address these optimization challenges, several techniques and algorithms have been proposed. The approach in [4] divides the optimization problem into simple subproblems by breaking down complex circuits into simpler ones. However, while this method offers advantages in some scenarios, it might result in increased overall circuit depth in certain cases. Additionally, previous works [1, 5-7] introduced optimization algorithms aimed at minimizing gate count and circuit depth. Nevertheless, these methods may not fully address the interplay between gate count reduction and circuit

doi:10.6342/NTU202303169

depth optimization, potentially leading to suboptimal results. Furthermore, strategies based on gate merging, cancellation, and reordering have been explored to further optimize quantum circuits [4]. Despite their potential, these strategies could occasionally introduce additional gate operations that counteract the expected optimizations. These prior contributions have laid the foundation for the development of advanced techniques and algorithms in the field of quantum circuit optimization, while also highlighting the need for more comprehensive and integrated approaches.

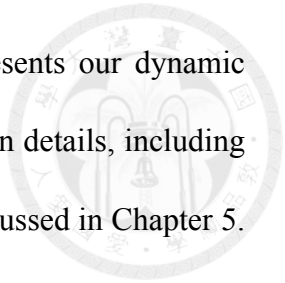
1.3 Contribution of the Thesis

The thesis makes several contributions to the field of quantum circuit optimization. Firstly, it introduces the novel Qsyn framework, developed in C++, which provides a flexible and efficient playground for implementing various optimization algorithms. Qsyn significantly outperforms existing tools like PyZX in terms of execution speed, particularly for large-scale circuits. Secondly, the thesis proposes an integrated workflow of decomposition level adjustment and dynamic reduction algorithm, resulting in reductions of total gate count, 2-qubit gate count, and circuit depth. The experimental results demonstrate the effectiveness of this approach across a wide range of test cases. Overall, the thesis contributes to the advancement of quantum circuit optimization by providing a powerful tool and an optimized workflow for achieving superior results in terms of circuit quality and computational efficiency.

1.4 Organization of the Thesis

The thesis is organized as follows: Chapter 2 provides a background in quantum circuit synthesis. In Chapter 3, the concepts of ZX-calculus and PyZX, a Python library,

doi:10.6342/NTU202303169



are introduced. Chapter 4 defines the decomposition level and presents our dynamic reduction algorithm for quantum circuit optimization. Implementation details, including Qsyn, and the execution of the dynamic reduction algorithm, are discussed in Chapter 5. Chapter 6 presents experimental results showcasing the reduction achieved in gate count, 2-qubit gate count, and circuit depth using our approach. The concluding chapter summarizes the thesis and suggests future research directions. Supplementary materials are provided in the Appendix.

doi:10.6342/NTU202303169

Chapter 2 Preliminaries

This chapter introduces the essential concepts for understanding quantum circuit optimization. It covers the definition of quantum circuits, and the single-qubit and multi-qubit gates. We also discuss operations, and the importance of these quantum gates in quantum circuit design. Additionally, this chapter briefly describes the quantum circuit synthesis flow. It establishes the foundational knowledge needed to comprehend the advanced techniques and algorithms presented in the subsequent chapters, enabling readers to delve into the field of quantum circuit optimization.

2.1 Quantum Circuits

A quantum circuit is a fundamental concept in the field of quantum computing. It is a model used to represent and manipulate quantum information, which is stored in quantum bits or qubits. Similar to classical circuits used in traditional computing, quantum circuits are composed of a series of quantum gates that perform operations on the qubits.

Quantum gates in a quantum circuit are analogous to logic gates in the classical circuit. They can be single-qubit gates that operate on individual qubits or multi-qubit gates that act on multiple qubits simultaneously. These gates enable various types of operations, such as rotations, superpositions, entanglement, and measurements, which are essential for performing quantum computations.

Quantum circuits follow the principles of quantum mechanics, allowing qubits to exist in a superposition of states and undergo quantum entanglement. This unique

doi:10.6342/NTU202303169

characteristic of quantum circuits gives them the potential to solve certain problems computationally more efficient than the classical computers.

To execute a quantum circuit, a sequence of gates are applied to the initial state of the qubits. The gates can manipulate the quantum state, by changing the probabilities of the superpositions in order to achieve specific computations. At the end of the circuit, we usually perform measurements to extract information from the qubits. An example of a quantum circuit is shown in Fig. 2-1.

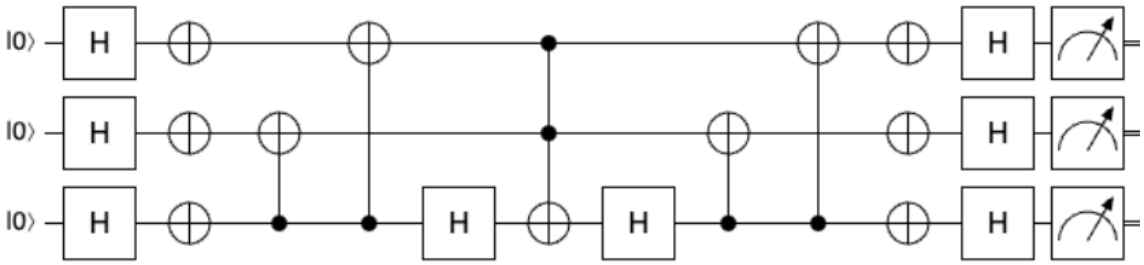


Fig. 2-1 An example of a quantum circuit diagram

Quantum circuits enable applications in various fields, such as cryptography, optimization, and simulation. They are designed to harness the power of quantum mechanics and exploit its computational advantages. However, quantum circuits also face challenges in signal decoherence and error correction. Researchers are actively working on to improve the stability and reliability of quantum computations.

2.2 Single-Qubit Gates

Single-qubit gates are fundamental building blocks in quantum computing that realize the manipulations of individual qubits. In quantum computing, qubits are the quantum counterpart of the classical bits and can exist in a superposition of states, representing multiple possibilities simultaneously. Single-qubit gates enable the

doi:10.6342/NTU202303169

transformation of qubits, altering their quantum states and facilitating the execution of quantum algorithms.

These gates operate on a single qubit at a time and can perform various operations, including rotations, phase shifts, and logical operations. They are represented by 2×2 matrices that describe the transformation applied to the quantum state of the qubit. Common types of single-qubit gates [8, 9] include the Pauli gates (X, Y, and Z), Hadamard gate (H), phase gate (S), and $\frac{\pi}{8}$ gate (T).

- **The Pauli Gates**

The Pauli gates are fundamental gates that perform rotations around the X, Y, and Z axes of the Bloch sphere. The X gate, also known as the NOT gate, flips the qubit's state from $|0\rangle$ to $|1\rangle$ or vice versa. The matrix form of the Pauli-X gate is shown below:

$$X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = |0\rangle\langle 1| + |1\rangle\langle 0|$$

The Y and Z gates induce rotations around the Y and Z axes, respectively, altering the qubit's phase and superposition. The matrix forms of them are shown below:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} = -i|0\rangle\langle 1| + i|1\rangle\langle 0| \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = -|0\rangle\langle 0| - |1\rangle\langle 1|$$

- **The Hadamard Gate**

The Hadamard gate is a widely used single-qubit gate that creates superposition by rotating the qubit's state from the computational basis ($|0\rangle$ and

doi:10.6342/NTU202303169

$|1\rangle$) to the superposition basis ($|+\rangle$and $|-\rangle$). It plays a vital role in generating entangled states and is used in various quantum algorithms, such as the famous quantum algorithm for the Deutsch problem. It has the matrix:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

- **The *S* Gate and *T* Gate**

The *S* gate, also known as the phase gate, introduces a phase shift of $\frac{\pi}{2}$ to the qubit's state. It modifies the phase of the $|1\rangle$ state, leaving the $|0\rangle$ state unchanged. This gate is essential for constructing other gates and quantum algorithms, including Quantum Fourier Transforms.

The *T* gate, represented as the $\frac{\pi}{8}$ gate, introduces a phase shift of $\frac{\pi}{4}$ to the qubit's state. It is particularly useful in implementing certain quantum algorithms, such as Shor's algorithm [10-12] for integer factorization, and is closely related to other gates like the *S* gate.

$$S = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} \end{bmatrix} \qquad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}$$

Single-qubit gates play a critical role in quantum computation, allowing the manipulation and control of individual qubits. They enable the creation of superposition, entanglement, and complex quantum states necessary for quantum algorithms' execution. By combining single-qubit gates with multi-qubit gates, quantum computations can be performed on larger quantum systems, paving the way for solving problems that are beyond the reach of classical computers.

doi:10.6342/NTU202303169

2.3 Multi-Qubit Gates

Multi-qubit gates are fundamental operations in quantum computing that act on multiple qubits simultaneously. They are essential for performing operations such as quantum entanglement, quantum teleportation, and quantum error correction. While single-qubit gates manipulate individual qubits, multi-qubit gates allow the entanglement and interaction between multiple qubits, enabling more complex quantum computations. The two most important types of multi-qubit gates [8, 9] are the *CNOT* gate [13] and the Toffoli gate. We will explain them in details in the rest of the subsections.

- **The *CNOT* gate**

  The *CNOT* gate is a conditional gate that flips the state of the target qubit if the control qubit is in the state $|1\rangle$. On the other hand, it leaves the target qubit unchanged if the control qubit is in the state $|0\rangle$. The *CNOT* gate is fundamental in quantum circuits, used in many quantum algorithms and protocols. The diagram of a *CNOT* gate is represented in Fig. 2-2.

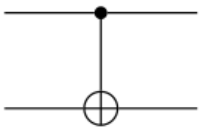


Fig. 2-2 A diagram of the CNOT-gate

Depending on which qubit is the target and which is the control, the *CNOT* gate has one of the two matrices:

doi:10.6342/NTU202303169

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \qquad CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- **The Toffoli Gate**

The Toffoli gate, also known as the Controlled-Controlled-NOT (*CCNOT*) gate, is a multi-qubit gate in quantum computing that operates on three qubits. It is named after its inventor Tommaso Toffoli and is an essential component in many quantum algorithms and quantum error correction codes.

The Toffoli gate performs a controlled *NOT* operation on the target qubit (the third qubit) if and only if both control qubits (the first and second qubits) are in the state $|1\rangle$. In other words, it flips the state of the target qubit if both control qubits are in the $|1\rangle$ state, otherwise, it leaves the target qubit unchanged. The diagram of a Toffoli gate is indicated in Fig. 2-3.

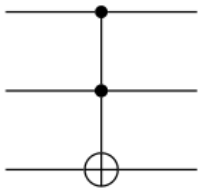


Fig. 2-3 A diagram of the Toffoli gate

2.4 Quantum Circuit Synthesis Flow

Quantum circuit synthesis flow refers to the process of designing and optimizing quantum circuits to efficiently execute a specific quantum algorithm or computation. It involves a series of steps aimed at converting a high-level quantum algorithm into a

doi:10.6342/NTU202303169

physical implementation using a specific set of quantum gates [14]. Fig. 2-4 presents the quantum circuit synthesis flow.

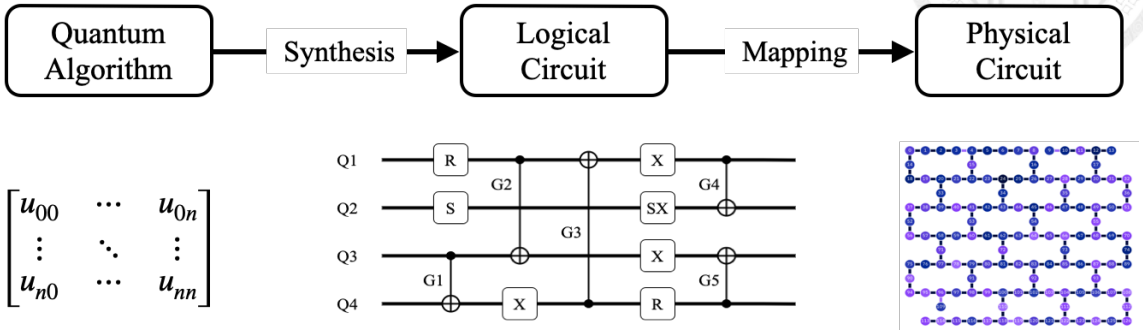


Fig. 2-4 Quantum circuit synthesis flow [15]

The first step is to design the quantum algorithm or computation that solves the desired problem. This involves identifying the problem, designing the logic and structure of the quantum algorithm, and determining the required number of qubits and gates. High-level quantum circuit construction [16] and gate set mapping [17] are implemented to construct a logical quantum circuit in the process of synthesis. The step of synthesis involves various methods that can lead to different outcomes. In other words, different forms of logical quantum circuits may indicate the same quantum operation.

During the status of the logical circuit, the synthesized quantum circuit is optimized to improve its efficiency. This includes applying techniques such as gate cancellation [18], gate merging, and circuit depth reduction [19] to minimize the number of gates. Some transformation strategies have also been defined to map a logical quantum circuit to a more flexible structure for optimization. ZX-calculus [20] is one of the classical examples and will be introduced in Chapter 3. The main purpose of all approaches in quantum circuit optimization is to reduce quantum resource requirements and optimize the overall circuit performance.

doi:10.6342/NTU202303169

The optimized quantum circuits need to adhere to the constraints and limitations of the specific quantum hardware being used. This step, which is called “mapping” [21], involves considering factors such as gate fidelities, connectivity between qubits, and noise characteristics of the hardware when optimizing the circuit.

By following the quantum circuit synthesis flow, quantum algorithm designers can transform high-level quantum algorithms into optimized quantum circuits that are compatible with specific quantum hardware. This process maximizes the efficiency and performance of quantum computations, paving the way for advancements in quantum computing applications.

doi:10.6342/NTU202303169

Chapter 3 Quantum Circuit Modeling by ZX-calculus

This chapter focuses on the utilization of ZX-calculus for quantum circuit modeling. It begins with an introduction of ZX-calculus, explaining its principles and applications in quantum circuit optimization. The chapter also provides a brief introduction to PyZX, a Python library that implements ZX-calculus for circuit analysis and manipulation. By exploring the foundations of ZX-calculus and its integration with PyZX, readers will gain a deeper understanding of the modeling techniques employed in quantum circuit optimization.

3.1 Introduction to ZX-calculus

ZX-calculus is a powerful graphical language and formalism introduced by Coecke and Duncan [22, 23] in 2008. It was proposed for reasoning about quantum information and quantum computations. It provides a visual representation and a set of algebraic rules for manipulating quantum logic operations, making it an intuitive and efficient tool for studying quantum circuit transformation. It was used increasingly in related areas such as quantum circuit optimization, quantum error correction code, and measurement-based quantum computation [24].

- **ZX-diagram**

  ZX-calculus is based on the *ZX-diagram* notation, which is a graphical representation of a standard quantum circuit after the transformation of the ZX-calculus. Unlike quantum circuits, ZX-diagrams do not have a strict temporal ordering. Instead, they focus on capturing the relationships between qubits and the

doi:10.6342/NTU202303169

interactions between quantum gates, emphasizing the concept of quantum interference. In other words, only connectivity matters in a ZX-diagram. See Fig. 3-1 as an example, the following ZX-diagrams all represent the same matrix.

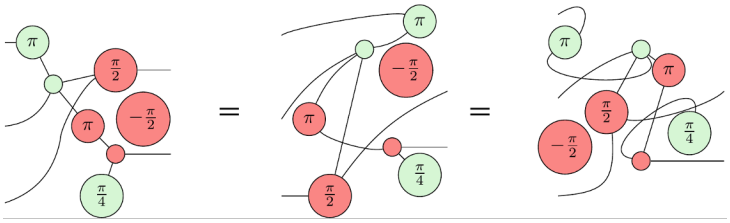


Fig. 3-1 The three ZX-diagrams represent the same matrix

- **Spiders and Wires**

Quantum gates are represented as nodes, and wires connecting these nodes represent qubits. The nodes in the diagram, which are also called “spiders”, are of two types: Z-spiders and X-spiders. The Z-spiders are represented as green dots while the X-spiders are represented as red ones. The phase α carried by the spider is labeled in the dot and α is typically ranged between 0 and 2π. As a supplementary explanation, if α is equal to 0, the phase can be omitted and left unlabeled. The linear map interpretations of both Z-spiders and X-spiders are shown in Fig. 3-2.

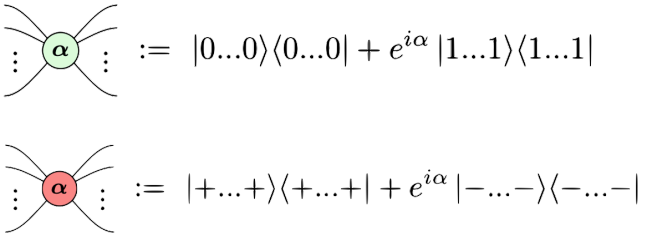


Fig. 3-2 Z-spider and X-spider interpretation as a linear map

Wires are used to connect spiders and two types of wires are defined. The black solid lines are called simple edges while the blue dashed lines are called

doi:10.6342/NTU202303169

Hadamard edges. As Fig. 3-3 shows below, we can transform a Hadamard gate, symboled as a yellow box on the left-hand side into a Hadamard edge as the preprocessing of the simplification of a ZX- diagram.

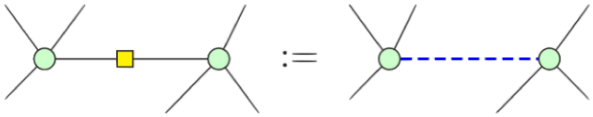
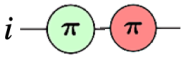
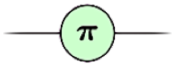
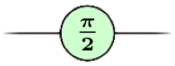
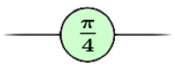
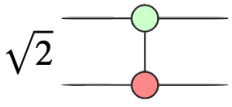


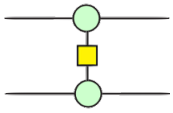
Fig. 3-3 Hadamard edge and Hadamard box illustration

According to the definition of spiders and quantum gates mentioned in Chapters 2.2 and 2.3, the common quantum gates and their representation in the ZX-calculus are listed in Table 3-1. The first column specifies the common name for the quantum gate.

Table 3-1 ZX and Matrix Form of Common Quantum Gates [24]

Name	ZX-diagram Form	Matrix
Identity		$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
Pauli-X gate		$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$
Pauli-Y gate		$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$
Pauli-Z gate		$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$
Hadamard gate		$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$
S gate		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} \end{bmatrix}$
T gate		$\begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}$
CNOT gate		$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

doi:10.6342/NTU202303169

Name	ZX-diagram Form	Matrix
CZ gate	$\sqrt{2}$ 	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$

- **Usage of ZX-calculus**

To make great use of the ZX-calculus, we usually transform the standard quantum circuit into a *graphlike* [20] ZX-diagram. After the transformation, we will use different strategies based on the theories in the ZX-calculus to simplify the ZX-diagram. The simplification strategies are defined distinctively based on simplification targets. For example, T-gates are costly in a quantum circuit so the effective method for T-count minimization [25] has been studied for many years.

Take Fig. 3-4 as an example of the procedure for ZX- diagram transformation. Fig. 3-4 (a) is an original quantum circuit. After the transformation of the ZX-calculus, the ZX-diagram is shown in Fig. 3-4 (b). After the simplification strategies of ZX-calculus, we can finally get the result in Fig. 3-4 (c) and (d).

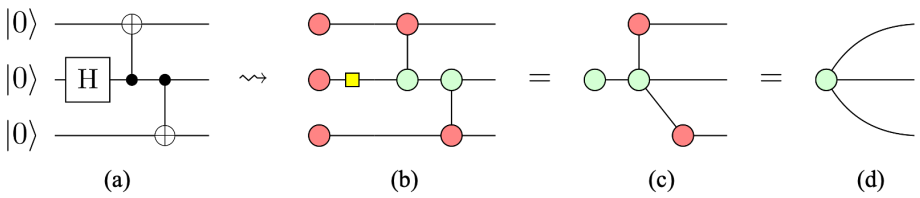


Fig. 3-4 Transformation between a standard quantum circuit and ZX- diagrams

- **ZX-rules**

ZX-rules, also known as the rewriting rules of ZX-calculus, are a set of algebraic rules that govern the manipulation and transformation of ZX-diagrams. These rules enable the simplification, composition, and decomposition of ZX-

doi:10.6342/NTU202303169

diagrams, allowing for the analysis and optimization of quantum circuits and quantum information processes.

There are several ZX-rules, each specifying a specific transformation that can be applied to ZX-diagrams. Some of the commonly used ZX-rules include:

- **Spider Fusion Rule**

This rule allows the merging of two spiders of the same color into a single spider. It represents the combination of quantum states and the interaction between qubits. See Fig. 3-5 for diagrammatic interpretation.

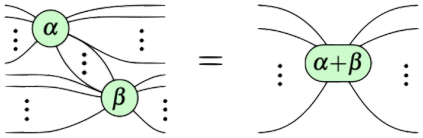


Fig. 3-5 A diagrammatic interpretation for spider fusion rule

- **Identity Removal Rule**

The rule removes the identity spiders from ZX-diagrams. The identity spider is a special type of spider that represents an identity gate, which does not affect the quantum state of the qubits. Different types of identity gates are presented in Fig. 3-6.

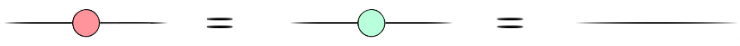


Fig. 3-6 A diagrammatic interpretation of identity removal rule

- **Hadamard Rule**

The rule means conjugating the Pauli-Z gates with Hadamard gates to get Pauli-X gates, which can be interpreted as $HZH = X$. In the ZX form, when an X-spider needs to be transformed into a Z-spider, all Hadamard edges connected

doi:10.6342/NTU202303169

to the X-spider will be converted into simple edges, and all simple edges connected to the X-spider will be converted into Hadamard edges, and vice versa. This transformation is also called as “Color Change Rule”. The diagrammatic interpretation is shown in Fig. 3-7.

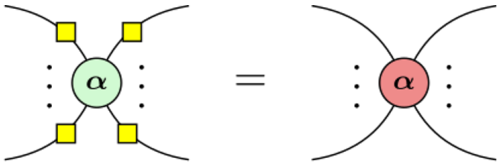


Fig. 3-7 A diagrammatic interpretation of Hadamard rule

- **Local Complementation Rule**

If a spider (denoted by * on the left-hand side in Fig. 3-8) with a $\pm\frac{\pi}{2}$ phase exists within a *graphlike* diagram, which is internally connected to other spiders and not linked to inputs or outputs, it can be eliminated from the diagram by complementing the connectivity within its neighborhood and adjusting certain phases accordingly.

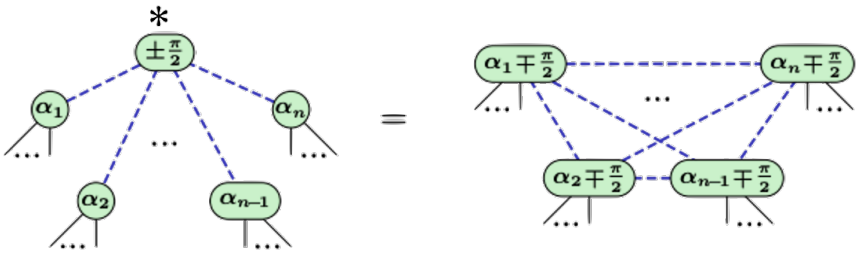


Fig. 3-8 A diagrammatic interpretation for local complementation rule

- **Pivot Rule**

This rule involves the deletion of Pauli spiders pairs, which are spiders with phases that are multiples of π. As shown in Fig. 3-9, when we have a pair of connected Pauli spiders, denoted as u and v, we can split the neighborhood of

doi:10.6342/NTU202303169

$\{u, v\}$ into three distinct parts: U, which consists of spiders only connected to u, V, which consists of spiders only connected to v, and W, which consists of spiders connected to both u and v. Then, delete the pair of spiders u and v, under the condition that we introduce complete bipartite graphs on (U, W), (V, W), and (U, V).

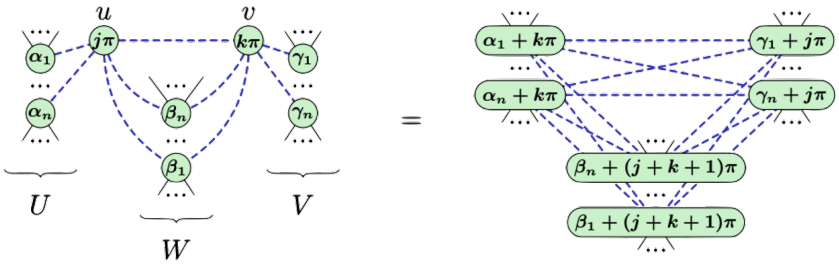


Fig. 3-9 A diagrammatic interpretation of the pivot rule

- **Pivot Gadget Rule**

This rule is similar to the pivot rule but with a specific distinction. While the pivot rule focuses on pairings of Pauli spiders, the pivot gadget rule looks for a different pairing. Specifically, it seeks a pair consisting of an interior Pauli spider and an interior non-Clifford spider within a ZX-diagram. Once this pairing is identified, the pivot gadget rule enables the transformation of the non-Clifford spider into a gadgetized form. See Fig. 3-10 for a diagrammatic representation. This process involves replacing the non-Clifford spider with a specific arrangement of spiders, edges, or other elements that represent the behavior of the original non-Clifford spider.

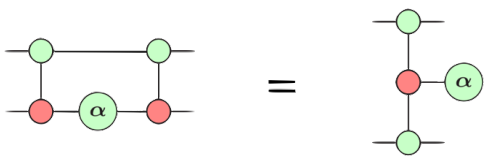


Fig. 3-10 A diagrammatic interpretation for pivot gadget rule

doi:10.6342/NTU202303169

- **Phase Gadget Rule**

A phase gadget refers to a configuration that involves an arity-1 spider with an angle α. This arity-1 spider is connected to another spider, which has no angle, through a Hadamard edge. This rule is to merge the phases of two phase gadgets, α and β, which are connected to the same set of spiders. See Fig. 3-11 for details.

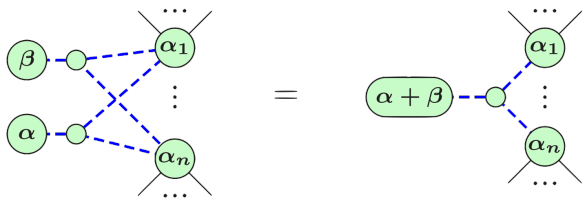


Fig. 3-11 A diagrammatic interpretation for phase gadget rule

doi:10.6342/NTU202303169

3.2 PyZX: Automated Diagrammatic Python Tool for Quantum Circuit using ZX-calculus

PyZX [26] is a Python library (https://github.com/Quantomatic/pyzx) and software package that is specifically designed for working with the ZX-calculus, a graphical calculus for quantum computing. It provides a wide range of functionalities and tools for ZX-diagram manipulation, circuit optimization, and quantum circuit analysis. It allows users to create, modify, and visualize ZX-diagrams using Python programming language. It provides a convenient interface to construct ZX-diagrams, add and remove spiders and edges, apply transformation rules, and perform various operations on the diagrams. Fig. 3-12 shows the overall functionalities of PyZX.

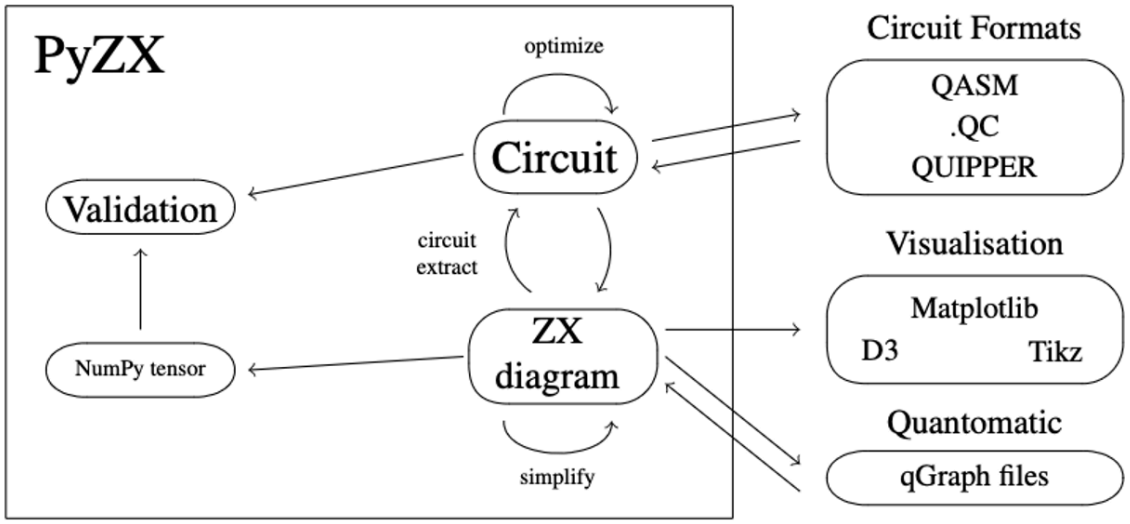


Fig. 3-12 An overview of the functionality of PyZX [26]

One of the key features of PyZX is its ability to perform circuit optimization using ZX-calculus techniques. It offers functions to simplify ZX-diagrams, identify and eliminate redundant elements, exploit symmetries, and optimize quantum circuits for better efficiency and performance. Table 3-2 lists some of the common ZX-rules and the

doi:10.6342/NTU202303169

corresponding functions defined in PyZX. Each simplification function continuously searches for candidates in the ZX-diagram that match its specific rule and updates the ZX-diagram accordingly. This process continues until there are no more candidates, at which point the simplification function stops. Each function outputs the number of iterations for the update of the ZX-diagram.

Table 3-2 ZX-rules and their corresponding functions implied in PyZX [26]

ZX-rule	Function Name
Spider Fusion Rule	`spider_simp`
Identity Removal Rule	`id_simp`
Hadamard Rule	`to_gh`
Local Complementation Rule	`lcomp_simp`
Pivot Rule	`pivot_simp`
Pivot Gadget Rule	`pivot_gadget_simp` `pivot_boundary_simp`
Phase Gadget Rule	`gadget_simp`

By utilizing these simplification rules, quantum circuits can be optimized to varying degrees. The effectiveness of circuit optimization can be enhanced by employing specific sequences of these rules. Here are a few examples of functions implemented in the PyZX package that demonstrate this approach for efficient quantum circuit optimization:

- **Interior Clifford Simplification**

  The primary objective of this simplification strategy is to eliminate as many interior Clifford spiders as possible [25]. These spiders are characterized by having phases that are multiples of $\frac{\pi}{2}$. This strategy simplifies the circuit by iteratively

doi:10.6342/NTU202303169

applying the identity removal rule, spider fusion rule, pivot rule, and local complementation rule that have been mentioned in Chapter 3.1 until none of them can be applied anymore. In PyZX, it is implemented in the function called *interior_clifford_simp* and Algorithm 3-1 presents its pseudocode.

Algorithm 3-1 *Pseudocode of interior Clifford simplification*

Function *interior_clifford_simp(g)*

Input: The ZX-diagram that should be simplified *(g)*
Output: The number of iterations of the while loop *(i)*

```
1   spider_simp(g)
2   Make g a graph-like ZX-diagram
3   matches ← true
4   i ← 0
5   while matches = true do
6       i1 ← id_simp(g)
7       i2 ← spider_simp(g)
8       i3 ← pivot_simp(g)
9       i4 ← lcomp_simp(g)
10      if i1 + i2 + i3 + i4 = 0 then
11          matches ← false
12      else
13          i ← i + 1
14      end if
15  end while
16  return i
```


doi:10.6342/NTU202303169

- **Clifford Simplification**

Clifford simplification is implemented in the function clifford_simp which contained interior Clifford simplification and pivot boundary rule. The algorithm is to remove as many Clifford spiders as possible. In particular, no specific actions are taken to eliminate non-Clifford spiders during this process. See Algorithm 3-2 for the pseudocode.

Algorithm 3-2 *Pseudocode of Clifford simplification*

Function *clifford_simp(g)*

Input: The ZX-diagram that should be simplified *(g)*
Output: The number of iterations of the while loop *(i)*

```
1   matches ← true
2   i ← 0
3   while matches = true do
4       i ← i + interior_clifford_simp(g)
5       i_2 ← pivot_boundary_simp(g)
6       if i_2 = 0 then
7           matches ← false
8       end if
9   end while
10  return i
```

In general, PyZX converts quantum circuits to ZX-diagrams using ZX-calculus and applies a series of simplification strategies to optimize the circuits. Finally, through extraction, the ZX-diagram is transformed back into the form of a quantum circuit, achieving the circuit optimization. The entire process is illustrated in Figure 3-13.

doi:10.6342/NTU202303169

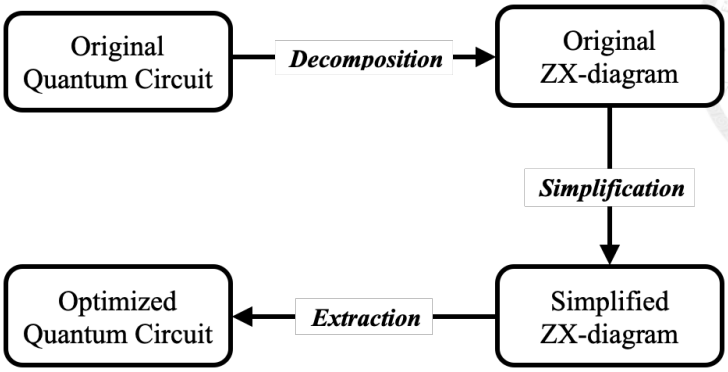


Fig. 3-13 The overall flow for full reduction implementation

doi:10.6342/NTU202303169

Chapter 4 Optimization by Dynamic Reduction Algorithm

This chapter delves into the application of the dynamic reduction algorithm for quantum circuit optimization. It begins with an overview of the previous work utilizing PyZX, a Python library, for circuit optimization. A case study focusing on a 3-Toffoli quantum circuit is presented to illustrate the challenges faced in circuit optimization. The chapter further explores the concept of decomposition level adjustment and its impact on circuit optimization. Finally, the dynamic reduction algorithm is introduced as a powerful technique for reducing gate count, 2-qubit gate count, and circuit depth, and thus improves circuit efficiency and achieved optimized results. By examining these aspects, readers will gain insights into the practical implementation and effectiveness of the dynamic reduction algorithm in quantum circuit optimization.

4.1 Previous Work

The ZX-calculus framework has emerged as a powerful tool for optimizing quantum circuits, and various techniques have been proposed to enhance its effectiveness [20, 25, 27]. One of the general simplification algorithms for reducing T-count, called *full_reduce,* was proposed by Aleks Kissinger et al. in [25]. It uses a combination of Clifford simplification (*interior_clifford_simp* and *clifford_simp*) and the gadgetization strategies such as the pivot gadget rule (*pivot_gadget_simp*) and phase gadget rule (*gadget_simp*) as mentioned. The algorithm of *full_reduce* is shown in Algorithm 3-3.

doi:10.6342/NTU202303169

Algorithm 3-3 *Pseudocode of full reduction*

Function *full_reduce(g)*

Input: The ZX-diagram that should be simplified *(g)*
Output: *None*

```
1   matches ← true
2   interior_clifford_simp(g)
3   while matches = true do
4       clifford_simp(g)
5       i ← gadget_simp(g)
6       interior_clifford_simp(g)
7       j ← pivot_gadget_simp(g)
8       if i + j = 0 then
9           matches ← false
10      end if
11  end while
```

4.2 Case Study: 3-Toffoli Quantum Circuit

Take a circuit with 3 Toffoli gates as an example. The original circuit is shown in Fig. 4-1. Since the decomposition of a Toffoli gate contains 7 *T* gates [7], the original T-count of this circuit is 21.

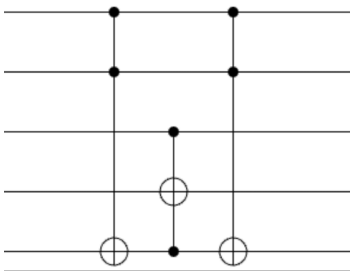


Fig. 4-1 A original circuit with 3 Toffoli gates

doi:10.6342/NTU202303169

By using PyZX to reduce the T-count of the quantum circuit, we first expand the circuit to a ZX-diagram with 21 *T* gates which is shown in Fig. 4-2.

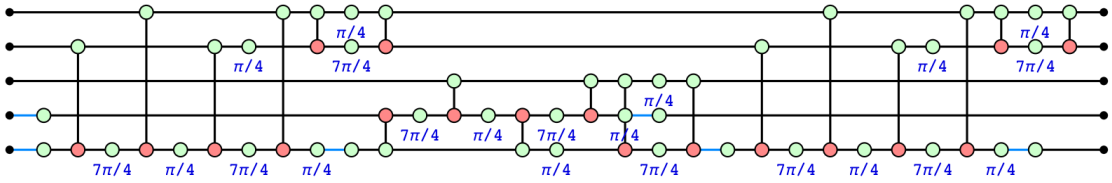


Fig. 4-2 The original ZX-diagram with 21 *T* gates

After applying *full_reduce*, the simplified ZX-diagram is formed as in Fig. 4-3 with 10 phase gadget spiders.

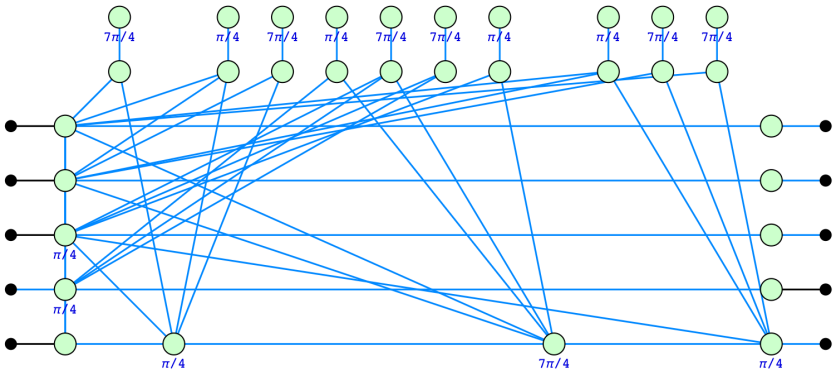


Fig. 4-3 The simplified ZX-diagram after full reduction

Finally, Fig. 4-4 illustrates the process of removing gadgets from Fig. 4-3, representing the resulting diagram in terms of basic gates. From the figure, we can observe a noticeable reduction in the T-count, indicating a significant improvement in the efficiency of the circuit.

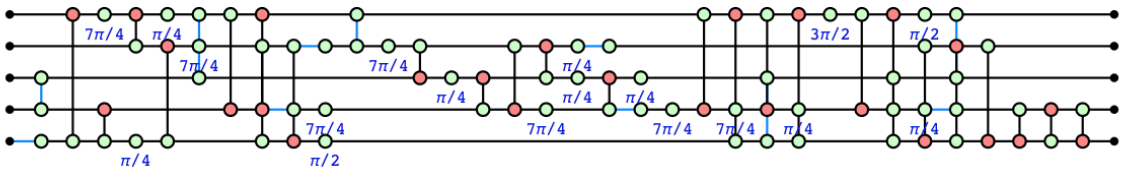


Fig. 4-4 The simplified ZX-diagram with 15 *T* gates

doi:10.6342/NTU202303169

After optimization, the T-count of the circuit decreased significantly from 21 to 15. However, as observed from Table 4-1, the gate count (Σ), 2-qubit gate count (2*Q*), and circuit depth (*D*) increased. This phenomenon becomes more prominent as the circuit size grows. Therefore, it is an important research direction to control the increase in gate count, 2-qubit gate count, and circuit depth while maintaining the reduction in T-count has become an important research direction.

Table 4-1 The circuit information for the original one and the optimized one

Circuit	Σ	2*Q*	*T*	*D*
Original (Fig. 4-2)	45	18	21	31
Optimized (Fig. 4-4)	55	31	15	34

4.3 Decomposition Level Adjustment

According to Fig. 3-13, it can be observed that when converting from the original quantum circuit to the original ZX-diagram, a process of decomposition is required. This step primarily involves transforming each quantum gate into its corresponding ZX form, as shown in Table 3-1, and then connecting them in the correct order. However, for some more complex quantum gates, such as the Toffoli gate, Controlled-CZ gate, and Controlled-RZ gate, there are different ZX representations. Take the Toffoli gate as an example, to implement a Toffoli gate by basic quantum gates [7], it can be represented as Fig. 4-5.

doi:10.6342/NTU202303169

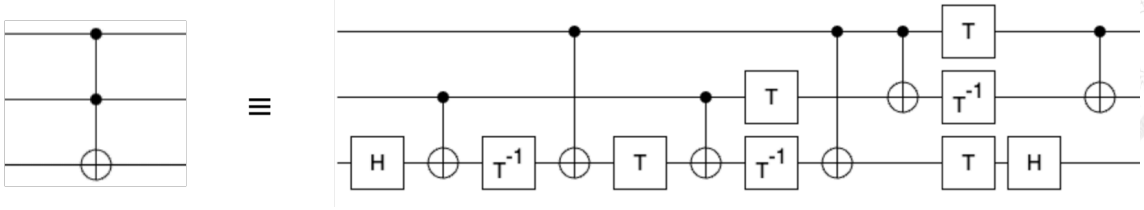


Fig. 4-5 Circuit implementing a Toffoli gate

After transforming each quantum gate to the corresponding ZX form, a Toffoli gate can be interpreted as the ZX-diagram shown in Fig. 4-6.

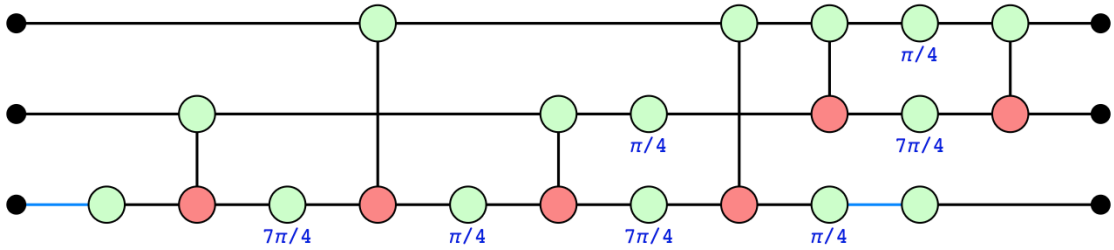
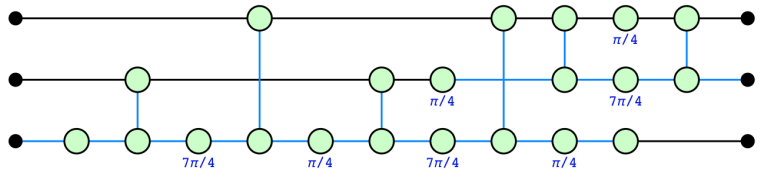
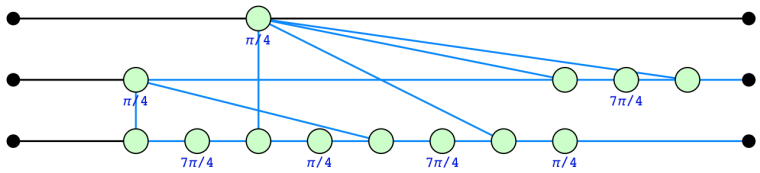
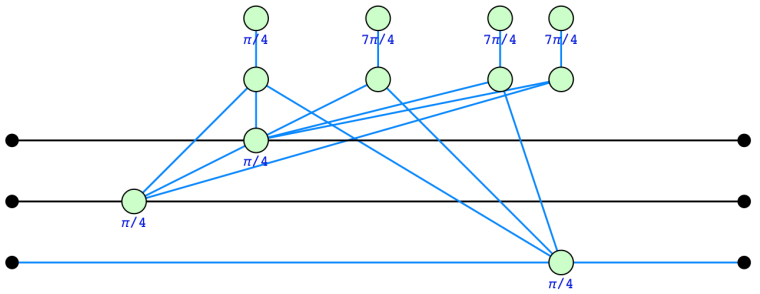


Fig. 4-6 ZX-diagram implementing a Toffoli gate

However, Fig. 4-6 is just one of the representations of a Toffoli gate. Through different types of simplification strategies, we have categorized the decomposition levels of a Toffoli gate into three levels, ranging from the most complex to the simplest, as shown in Table 4-2. For level 1, we apply the Hadamard rule (*to_gh*) to Fig. 4-6. Further employing the spider fusion rule (*spider_simp*) and identity removal rule (*id_simp*) on the ZX-diagram from level 1, we get the ZX-diagram for level 2. For level 3, we obtain the phase gadget for the ZX-diagram by using *pivot_gadget_simp*.

doi:10.6342/NTU202303169

Table 4-2 Decomposition Level for a Toffoli gate

Decomposition Level	ZX Form Representation
1	
2	
3	

Consider the case of Fig. 4-1, which illustrates a configuration involving 3 Toffoli gates. We explore the implications of varying decomposition levels on quantum circuit optimization. For a decomposition level of 1, the initial construction of the ZX-diagram yields a ZX-diagram comprising exclusively Z spiders and Hadamard edges. Elevating the decomposition level to 2 facilitates the exclusion of visually redundant spiders during the initial ZX-diagram construction. Meanwhile, a decomposition level of 3 permits the preliminary selection of potential gadget candidates within the circuit. This proactive selection process holds the potential to exert substantial influence over the subsequent sequence and progression of optimization steps in the ZX-diagram.

doi:10.6342/NTU202303169

4.4 Dynamic Reduction Algorithm

During the simplification process, *full_reduce* aims to minimize the T-count by applying various optimization techniques. However, there are instances where the T-count cannot be further reduced. In such cases, *full_reduce* continues to optimize the ZX-diagram by simplifying spiders and edges. While this approach often results in a reduction of the number of spiders and edges, it can also lead to difficulties when converting the highly optimized ZX-diagram back into a quantum circuit. This conversion process may introduce redundant quantum gates into the optimized quantum circuit.

To address this issue and prevent excessive optimization of the ZX-diagram, we propose a dynamic reduction algorithm. This algorithm effectively balances the goal of minimizing the T-count while avoiding the generation of excessive redundant quantum gates during the extraction process. By dynamically controlling the level of reduction, we ensure that the resulting quantum circuit remains optimized while minimizing the introduction of unnecessary gates.

First, we introduce a self-defined observational metric, which we refer to as "density". Through observations of numerous test data, we have identified a common occurrence in over-optimized circuits, characterized by spiders connected to a large number of edges. Consequently, we define density as the average of the squared sum of the number of edges connected to each spider. The formula is as follows:

$$\text{density} = \frac{\sum_{\text{spider}} \left(\#\text{neighbor}\right)^2}{\#\text{spider}}, \qquad \text{spider} \in \text{ZX-graph}.$$

doi:10.6342/NTU202303169

The most balanced ZX-graph would have each spider connected to exactly two neighbors, resulting in the minimum density. In this case, the minimum density can be approximated as $\left(\frac{2e}{v}\right)^2$, where e represents the number of edges and v represents the number of vertices in the ZX-graph. As the ZX-graph is further simplified, the density gradually increases. Therefore, the objective of the algorithm is to find a ZX-graph with the minimum density when the T-count is minimized.

After each execution of a simplification rule, the density is calculated and recorded, and the current ZX-graph is also saved. At the same time, the T-count is monitored. When the T-count no longer decreases, the ZX-graph with the minimum density is selected from the set of ZX-graphs with the minimum T-count. This ZX-graph with the minimum density is considered the result of the algorithm.

Similarly to the previous section, we exemplify our methodology using Fig. 4-1. Employing the original optimization technique, the full reduction algorithm (full_reduce), on the illustrated ZX-diagram in Fig. 4-2, yields an optimized ZX-diagram shown in Fig. 4-3 with the density of 14.0476. Subsequent extraction results in a noticeable increase in most metrics, except the T-count. In contrast, the application of the dynamic reduction algorithm (dynamic_reduce) for optimization maintains a ZX-graph density of 9.5676, which is shown in Fig. 4-7. The quantum circuit derived from this optimized ZX-graph not only preserves a minimized T-count but also exhibits a reduction in other pertinent metrics. The comparison table is shown in Table 4-3. It is evident that the dynamically optimized quantum circuit exhibits lower total gate count and 2-qubit gate count values compared to those obtained through full reduction. The outcomes of dynamic reduction do not exhibit an enhancement in the depth metric for

doi:10.6342/NTU202303169

the specific modest-sized circuit under consideration. This phenomenon can be attributed to the circuit's limited scale, rendering the influence on the depth parameter inconspicuous. However, in the subsequent sections, it becomes evident that dynamic reduction substantially ameliorates depth for numerous larger-scale circuits.

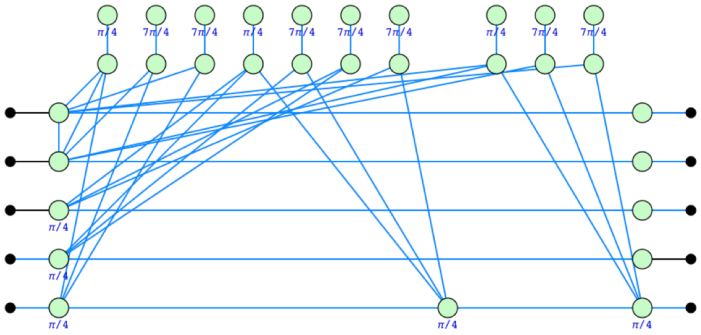


Fig. 4-7 Optimized ZX-diagram after dynamic reduction algorithm with the density of 9.5676

Table 4-3 The circuit information for the original one and the optimized ones by full reduction and dynamic reduction

Circuit	Σ	$2Q$	T	D
Original (Fig. 4-2)	45	18	21	31
Optimized (full reduction)	55	31	15	34
Optimized (dynamic reduction)	44	23	15	34

doi:10.6342/NTU202303169

Chapter 5 Implementation

The chapter begins with an introduction to Qsyn, providing an overview of its design, structure, and key functionalities. It highlights the capabilities of Qsyn as a powerful software system for quantum circuit synthesis, optimization, and verification. The chapter then delves into the execution of the dynamic reduction algorithm within the Qsyn framework, showcasing the integration of this algorithm and its impact on circuit optimization. The implementation details and methodologies employed are discussed, providing insights into the practical aspects of applying the dynamic reduction algorithm in the context of Qsyn.

5.1 Qsyn: Quantum Circuit Synthesis Framework

With the rapid growth of quantum computing, there is a pressing need for efficient techniques to analyze and enhance the performance of complex quantum circuits. In response to this demand, we have designed and implemented Qsyn as a comprehensive solution.

Qsyn is a powerful and versatile software system designed to synthesize, optimize, and verify quantum circuits used in quantum computers. Developed as a C++-based framework, Qsyn offers a comprehensive set of tools and algorithms to address the challenges associated in the quantum circuit synthesis flow. One of the key features of Qsyn is its ability to perform scalable quantum circuit optimization by leveraging the ZX-Calculus and technology mapping techniques. By combining these approaches, Qsyn enables users to effectively optimize quantum circuits, reducing their complexity and improving their performance. Qsyn provides an experimental implementation of

doi:10.6342/NTU202303169

various optimization algorithms, allowing users to explore different strategies and techniques for circuit optimization. Moreover, it offers a programming environment that facilitates simulation and the development of similar applications in the field of quantum computing.

Looking towards the future, Qsyn aims to enhance its existing algorithms and further refine its optimization capabilities. Additionally, the team behind Qsyn is dedicated to improving the visualization of ZX-graphs, providing users with a clearer and more intuitive representation of the optimized circuits.

- **Main-branch Structure**

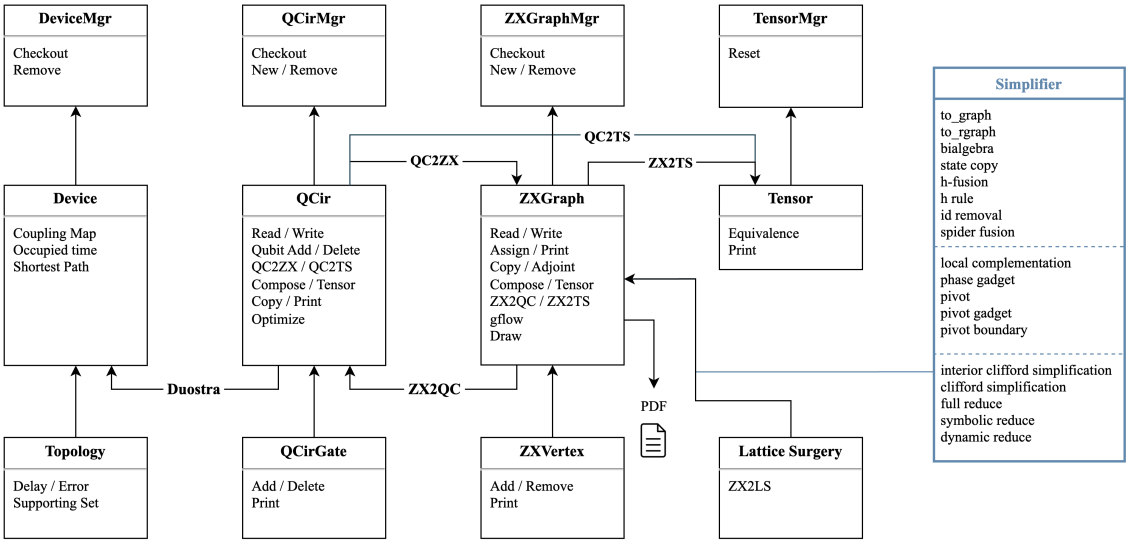


Fig. 5-1 The structure for Qsyn

Qsyn is mainly organized around four key structures: the quantum circuit (*QCir*), ZX-graph (*ZXGraph*), tensor matrix (*Tensor*), and device (*Device*). Fig. 5-1 illustrates the overall design architecture of Qsyn. A brief introduction for each structure is provided below:

doi:10.6342/NTU202303169

- **Quantum Circuit (*QCir*)**

  The quantum circuit structure is implemented using the class *QCir*, which encompasses all the essential characteristics of a quantum circuit. To construct a quantum circuit, different types of quantum gates are defined as subclasses of *QCirGate*.

- **ZX-graph (*ZXGraph*)**

  The ZX-graph structure is represented by the class *ZXGraph*, which allows the construction of a ZX-diagram. Within the ZX-diagram, individual spiders are manipulated using the class *ZXVertex*, representing various types of spiders such as the Hadamard gate (referred to as *H-box* in Qsyn).

- **Tensor Matrix (*Tensor*)**

  Both the quantum circuit and ZX-diagram can be represented as tensor matrices. To ensure the correctness of our implementation, the class *Tensor* serves as a checker, validating the tensor matrix representation of the quantum circuit and ZX-diagram.

- **Device (*Device*)**

  The class *Device* in Qsyn is specifically designed to emulate the execution of a quantum circuit on a physical quantum device. It provides a platform for emulating the behavior and limitations of real-world quantum hardware.

On top of the four main classes described above, to facilitate the simultaneous existence and computation of multiple circuits within Qsyn, we further introduce a manager layer is introduced to manage the access and isolation of different circuits.

doi:10.6342/NTU202303169

The manager layers for the four structures are defined as classes *QCirMgr*, *ZXGraphMgr, TensorMgr*, and *DeviceMgr,* respectively.

- **Common Commands List**

  - **System Information**

    To ensure the smooth execution of the system and the ability to monitor and manipulate system parameters over time, several common commands are available as described in the following Table 5-1:

Table 5-1 Some commands and their descriptions for system information

Command	Description
`usage`	Report the runtime and/or memory usage
`qquit`	Quit Qsyn
`help`	Show helping messages of commands

  - ***QCir***

    In order to construct a quantum circuit, we provide the functionality to read and write QASM (Quantum Assembly Language) [28]. We also offer various commands for editing the circuit. Additionally, since the circuit may require optimization or verification, there are commands related to transforming from *QCir* to *ZXGraph* or *Tensor* as well. Some common commands are listed in Table 5-2.

doi:10.6342/NTU202303169

Table 5-2 Some commands and their descriptions for *QCir*

Command	Description
`qc2ts`	Convert *QCir* to *Tensor*
`qc2zx`	Convert *QCir* to *ZXGraph*
`qccread`	Read a circuit and construct the corresponding setlist
`qccwrite`	Write *QCir* to a QASM file
`qccprint`	Print information of *QCir*

- ***ZXGraph***

To implement the ZX-calculus and apply it to quantum circuit optimization, we provide several commands for editing ZX-graphs. Additionally, we offer a variety of simplification methods to achieve the goal of simplifying ZX-graphs. Here is a list of some commonly used commands in Table 5-3:

Table 5-3 Some commands and their descriptions for *ZXGraph*

Command	Description
`zxgdraw`	Draw *ZXGraph* into a pdf file
`zxgwrite`	Write a *ZXGraph* to a file (.tik / .zx)
`zxgread`	Read a zx file and construct the corresponding *ZXGraph*
`zxgsimp`	Perform simplification strategies for *ZXGraph*

To facilitate broader usage, we have introduced a file format called "ZX" specifically designed to represent the structure of a ZX-diagram. This file format provides a convenient way to describe and store ZX-graphs for further analysis and manipulation within Qsyn. For detailed information on the syntax and usage of the ZX file format, please refer to Appendix A.1.

doi:10.6342/NTU202303169

- ***Tensor***

The class *Tensor* serves as a tool for circuit verification in Qsyn. It enables precise verification of the correctness of circuit simplification processes. By representing a circuit as a tensor, Qsyn can accurately compare the tensors before and after applying simplification rules to ensure that the circuit simplification process is performed correctly. Some commonly used commands are described in Table 5-4.

Table 5-4 Some commands and their descriptions for *Tensor*

Command	Description
`tsadjoint`	Adjoint the specified *Tensor*
`tsequiv`	Check the equivalency of two stored *Tensor*s
`tsprint`	Print information of *TensorMgr*

Please refer to Appendix A.2 for a comprehensive list of additional commands that are available in Qsyn. This appendix provides a detailed description of each command, including its syntax and functionality. It covers a wide range of commands related to quantum circuit editing, ZX-graph manipulation, simplification strategies, and other operations relevant to circuit optimization and verification. The appendix serves as a valuable resource for users to explore and utilize the full capabilities of Qsyn in their quantum circuit optimization workflows.

doi:10.6342/NTU202303169

5.2 Execution of Dynamic Reduction Algorithm

After successful installation and compilation of Qsyn, we can call the command-line interface of Qsyn where users can execute commands implemented into Qsyn. To execute Qsyn, we should enter the command `./qsyn`, the result is shown in Fig. 5-2.

```
> ./qsyn
DV Lab, NTUEE, Qsyn 0.5.0
qsyn>
```

Fig. 5-2 The command line interface after executing Qsyn

First, we need to read a QASM file that describes the quantum circuit. The command for reading the quantum circuit is `qccread`. The usage is as follows:

```
qccread <file_name>
```

The `<file_name>` parameter specifies the name of the file to be read. Take the instance mentioned in Chapter 4.2 as an example, The execution and response of the command line are as shown in Fig. 5-3 below.

```
qsyn> qccread ./benchmark/tof_3.qasm
Create and checkout to QCir 0

qsyn>
```

Fig. 5-3 The command line interface after reading a QASM file

As shown in the response in the figure, the quantum circuit is stored in *QCirMgr* with an index 0.

doi:10.6342/NTU202303169

In the subsequent step, the quantum circuit is transformed into a ZX-graph structure. As described in Chapter 4.3, the decomposition level can be specified during this conversion process by including the optional `[level_parameter]` integer. The `[level_parameter]` ranges from 1 to 3 and is added after the transformation command `qc2zx`. If the [level_parameter] is not provided, the default value is set to 1. The syntax for specifying the decomposition level is as follows:

```
qc2zx [level_parameter]
```

Take level 3 as an example, the response of the command line is as shown in Fig. 5-4 below.

Fig. 5-4 The command line interface after converting *QCir* to *ZXGraph* in level 3

The ZX-diagram converted by the quantum circuit is stored in *ZXGraphMgr* with an index 0.

To apply simplification on the ZX-diagram, we can use the command zxgsimp followed by the desired `[simplification_strategy]`. The usage is as follows:

```
zxgsimp [simplification_strategy]
```

For a detailed list of available simplification strategies, please refer to Appendix A.3. For executing the dynamic reduction algorithm, the required simplification strategy

doi:10.6342/NTU202303169

is called `-dreduce`. During this step, the command line prints out all the executed simplification rules, and the result is displayed as shown in Figure 5-5.

```
qsyn> zxgsimp -dreduce
All rules applied in order:
Spider Fusion Rule            1 iterations.
  1) 2 matches
Phase Gadget Rule             1 iterations.
  1) 1 matches
Local Complementation Rule    2 iterations.
  1) 1 matches
  2) 1 matches

qsyn>
```

Fig. 5-5 The command line interface after dynamic reduction

The simplified graph is stored and updated in *ZXGraphMgr* with the same index 0.

Lastly, we output the optimized ZX-graph as a zx file for further extraction and analysis using PyZX. The command to write the zx file is `zxgwrite`, and its usage is as follows:

```
zxgwrite <file_name.zx>
```

In this case, the result for this command is shown in Fig. 5-6.

```
qsyn> zxgwrite ./zx/tof_3.dr.zx

qsyn>
```

Fig. 5-6 The command line interface after writing the ZX-graph into a zx file

If no error message appears, it indicates that all processes have been executed successfully.

doi:10.6342/NTU202303169

If we want to verify the equivalence of the optimized circuit and the original circuit, we can use *Tensor* for confirmation. First, we convert the quantum circuit into a tensor matrix by `qc2ts` and then convert the ZX-diagram into another tensor matrix by `zx2ts`. For *TensorMgr*, the indices for them are 0 and 1, respectively. Finally, we use the command `tsequiv` to confirm the equivalence of the two tensor matrices. The usage is shown as follows:

```
tsequiv <id_1> <id_2>
```

The result for this case is shown in Fig. 5-7. We can see that the original circuit and the optimized ZX-graph are equivalent.

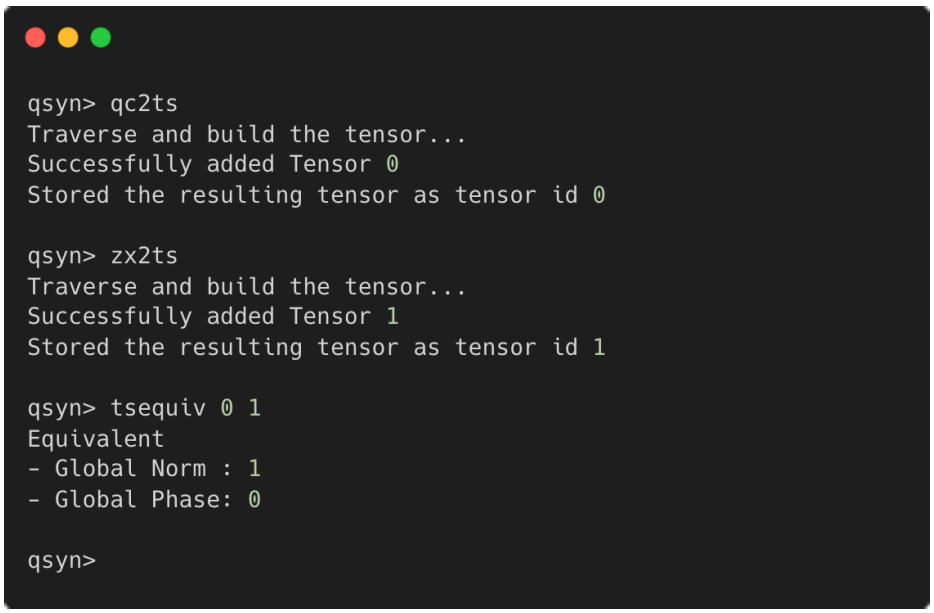


Fig. 5-7 The command line interface after equivalence checking

doi:10.6342/NTU202303169

Chapter 6 Experimental Results

The chapter begins with the discussion of the experimental setup, including the selection of test cases and the configuration of the experimental environment. Subsequently, a thorough comparison is made between the performance of PyZX and Qsyn, highlighting the advantages of the latter in terms of the execution time and scalability. The chapter then presents the results of implementing the decomposition level adjustment technique and dynamic reduction algorithm, showcasing the improvements achieved in terms of gate count, 2-qubit count, and circuit depth.

6.1 Experiment Setup

The experiments are conducted on a machine with Intel(R) Xeon(R) CPU E5-2630 v4 @ 2.20GHz, 131 GB total memory.

According to sections 4.3 and 4.4, our experiments adhere to the workflow illustrated above the dashed line in Fig. 6-1. Within this workflow, our proposed methods alter the decomposition and simplification steps, both of which are carried out using Qsyn, the software developed as detailed in Chapter 5.

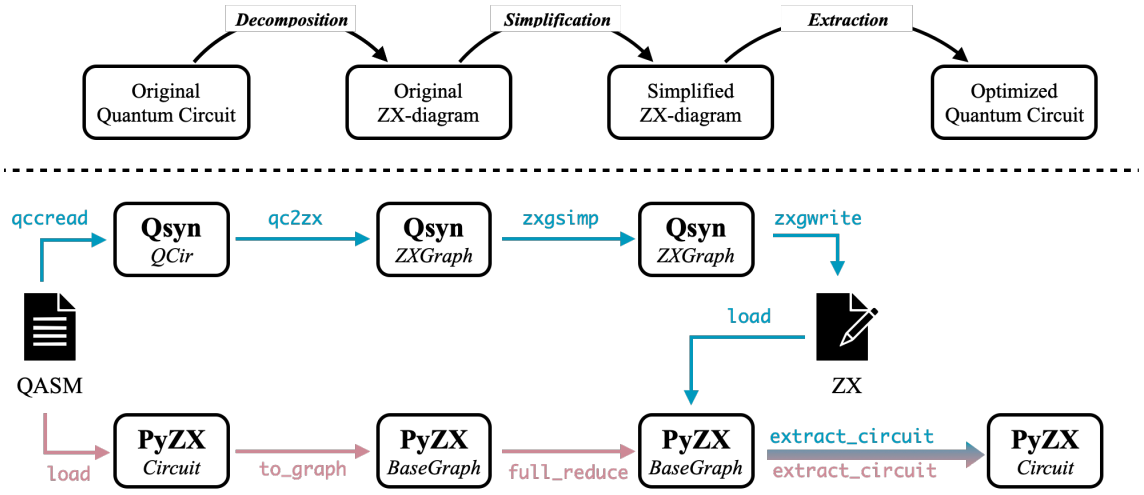


Fig. 6-1 The workflow for the experimental group and the control group

doi:10.6342/NTU202303169

The experimental group utilizes Qsyn to read QASM files and construct a quantum circuit. The quantum circuit is then transformed into a ZX-diagram using the algorithms with different decomposition levels provided by Qsyn, followed by the execution of the dynamic reduction algorithm. It then outputs the optimized ZX-diagram in the ZX format defined in Appendix A.1, which allows for the storage of the ZX-diagram in a *.zx* file. Finally, the ZX file is fed into PyZX, as described in Chapter 3.2, for extraction, resulting in the final optimized quantum circuit. The complete process for the experimental group is illustrated in Fig. 6-1 below the dashed line by the blue flow. In contrast, the control group employs the full simplification method proposed in [25] and utilize PyZX for execution, ultimately obtaining the final optimized quantum circuit. The workflow for the control group is shown in Fig. 6-1 below the dashed line by the red flow.

Our optimization techniques are applied to various types of quantum circuits, including key components of quantum algorithms used in factoring and computing discrete logarithms [29]. These benchmark circuits encompass essential elements such as the quantum Fourier transform, integer adders, and Galois field multipliers. By applying our techniques to these circuits, we can evaluate the effectiveness of our optimizations in improving the performance and efficiency of these important quantum algorithms. The circuit list and the information are provided in Table 6-1 [30, 31].

doi:10.6342/NTU202303169

Table 6-1 Circuit information for our benchmark

Circuit	Circuit Information			
	Σ	$2Q$	T	D
Mod 5_4	79	28	28	59
VEB-Adder$_3$	190	70	70	97
CSLA-MUX$_3$	210	80	70	80
CSUM-MUX$_9$	532	168	196	72
QCLA-Com$_7$	559	186	203	102
QCLA-Mod$_7$	1120	382	413	249
QCLA-Adder$_{10}$	657	233	238	91
Adder$_8$	1128	409	399	282
RC-Adder$_6$	244	93	77	129
Mod-Red$_{21}$	346	105	119	196
Mod-Mult$_{55}$	147	48	49	60
Toff-Barenco$_3$	76	24	28	51
Toff-NC$_3$	57	18	21	38
Toff-Barenco$_4$	146	48	56	99
Toff-NC$_4$	95	30	35	62
Toff-Barenco$_5$	218	72	84	147
Toff-NC$_5$	133	42	49	86
Toff-Barenco$_{10}$	578	192	224	387
Toff-NC$_{10}$	323	102	119	206
GF(2^4)-Mult	289	99	112	133
GF(2^5)-Mult	447	154	175	168
GF(2^6)-Mult	639	221	252	217
GF(2^7)-Mult	865	300	343	259
GF(2^8)-Mult	1139	405	448	307
GF(2^9)-Mult	1419	494	567	336
GF(2^{10})-Mult	1747	609	700	378
GF(2^{16})-Mult	4459	1581	1792	643
GF(2^{32})-Mult	17658	6268	7168	1315
GF(2^{64})-Mult	70075	24765	28672	2659
GF(2^{128})-Mult	279419	98685	114688	5345

doi:10.6342/NTU202303169

6.2 Comparison of PyZX and Qsyn

To increase the flexibility and efficiency of quantum circuit synthesis, we developed Qsyn using C++ as the implementation platform for our research algorithms. Table 6-2 presents a comparison of the execution times between Qsyn and PyZX for all test cases. The execution time is calculated as the total time taken by Qsyn and PyZX to read each circuit, convert it into a ZX-diagram, and perform the full reduction algorithm.

The table records the total gate count (Σ), 2-qubit count ($2Q$), T count (T), and circuit depth (D) for each circuit after the full reduction algorithm, along with the execution times of PyZX and Qsyn. In Table 6-2, the performance of Qsyn compared to PyZX is indicated by green text for improving and red text for inferior results. We observed slight differences in the outcomes of the full reduction algorithm between PyZX and Qsyn. These deviations are attributed to the distinct data structures employed by Qsyn and are considered within the normal range of variation. Additionally, it is observed that the execution time of Qsyn is significantly shorter than that of PyZX, particularly for larger circuits. Indeed, it is well-known that C++ execution time is generally faster than Python. However, the table shows that the average execution time of PyZX is approximately 49 times longer than that of Qsyn. This significant difference in execution time indicates that the built-in structures and algorithms of Qsyn outperform PyZX. Consequently, the reduced execution time of Qsyn can be attributed to its optimized algorithmic implementation and efficient utilization of system resources. As a result, Qsyn demonstrates improved performance and scalability, making it more suitable for handling large-scale quantum circuits.

doi:10.6342/NTU202303169

Table 6-2 Runtime comparison for PyZX and Qsyn

Circuit	PyZX					Qsyn				
	Σ	2Q	T	D	Runtime (s)	Σ	2Q	T	D	Runtime (s)
Mod 5_4	33	23	8	27	0.04	37	27	8	27	0.01
VEB-Adder$_3$	127	74	24	75	0.38	129	79	24	74	0.03
CSLA-MUX$_3$	255	161	62	115	0.05	217	137	62	106	0.01
CSUM-MUX$_9$	474	296	84	116	0.04	481	313	84	113	0
QCLA-Com$_7$	417	222	95	113	0.02	394	221	95	119	0.01
QCLA-Mod$_7$	1146	674	237	416	0.12	1160	706	237	387	0
QCLA-Adder$_{10}$	631	366	162	222	0.06	647	381	162	191	0.01
Adder$_8$	689	366	173	286	0.16	731	405	173	278	0.02
RC-Adder$_6$	214	115	47	136	0.02	220	119	47	128	0
Mod-Red$_{21}$	301	149	73	181	0.02	266	128	73	167	0
Mod-Mult$_{55}$	138	88	35	83	0.11	139	87	35	77	0
Toff-Barenco$_3$	46	21	16	38	0.05	59	37	16	46	0
Toff-NC$_3$	46	25	15	36	0.02	56	32	15	40	0
Toff-Barenco$_4$	102	56	28	82	0.36	92	48	28	76	0.03
Toff-NC$_4$	83	42	23	52	0.1	83	44	23	55	0
Toff-Barenco$_5$	139	71	40	108	0.07	156	85	40	115	0
Toff-NC$_5$	108	54	31	73	0.14	112	54	31	72	0.02
Toff-Barenco$_{10}$	373	199	100	265	0.1	370	208	100	253	0.01
Toff-NC$_{10}$	275	133	71	155	0.01	278	148	71	185	0.01
GF(2^4)-Mult	420	304	68	200	0.01	410	287	68	192	0
GF(2^5)-Mult	569	432	115	235	0.01	612	472	115	242	0
GF(2^6)-Mult	1067	811	150	430	0.04	1083	811	150	398	0
GF(2^7)-Mult	1551	1195	217	564	0.08	1501	1157	217	562	0.01
GF(2^8)-Mult	2133	1691	264	824	0.36	2129	1666	264	769	0.01
GF(2^9)-Mult	2546	1977	351	865	0.29	2575	2018	351	919	0.03
GF(2^{10})-Mult	3514	2790	410	1276	0.39	3432	2717	410	1096	0.03
GF(2^{16})-Mult	9624	7898	1040	2803	2.81	9928	8257	1040	2916	0.1
GF(2^{32})-Mult	37014	32676	4128	9505	34.7	49810	43157	4128	13850	0.65
GF(2^{64})-Mult	226794	201158	16448	54525	206.1	253557	227274	16448	66990	4.07
Sum		-			**246.66**		-			**5.06**
Runtime Rate					**48.75**					

doi:10.6342/NTU202303169

6.3 Decomposition Level Adjustment Comparison

In Chapter 4.3, we mentioned the decomposition level for multiple forms of the *CCNOT* (Toffoli) gate in the ZX-diagram. This allows us to transform the original quantum circuit into different ZX-diagram representations that have the same logical meaning. Since our benchmark circuits heavily feature the *CCNOT* gate, we adjusted the decomposition level to investigate the impact of different decomposition levels on the optimization of ZX-diagrams. As depicted in Fig. 6-2, we alter the parameters 1, 2, and 3 during the `qc2zx` step, representing different decomposition levels. Subsequently, after undergoing full reduction, the optimized ZX-diagrams are input into PyZX for extraction, resulting in the optimized circuits. The results of this extraction process are presented in Table 6-3.

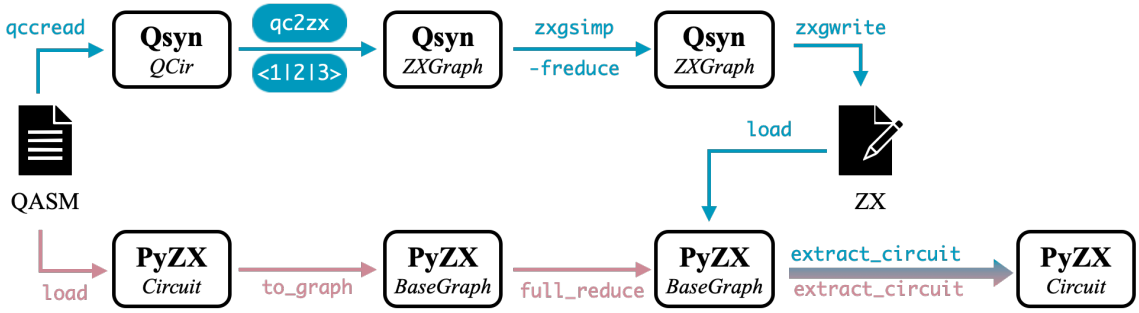


Fig. 6-2 The experiment workflow for the comparison of different decomposition level

For each circuit, we compare the total gate count, the 2-qubit gate count, and the circuit depth. The columns show circuit names, metrics of the existing optimization algorithm by PyZX (the control group), and the metrics of different decomposition levels (experimental group) by Qsyn. In the Qsyn columns, better performance compared to PyZX is indicated with green text, while poorer performance is represented in red. Additionally, cells with a green background represent the best performance achieved within each circuit.

doi:10.6342/NTU202303169

Table 6-3 Circuit metrics for the control group and optimized metrics for different decomposition levels

Circuit	PyZX *full_reduce*			Qsyn-D$_1$ *full_reduce*			Qsyn-D$_2$ *full_reduce*			Qsyn-D$_3$ *full_reduce*		
	Σ	2Q	D	Σ	2Q	D	Σ	2Q	D	Σ	2Q	D
Mod 5$_4$	33	23	27	37	27	27	37	27	27	36	26	23
VEB-Adder$_3$	127	74	75	127	77	71	125	75	68	112	68	72
CSLA-MUX$_3$	255	161	115	217	137	106	211	129	106	222	133	101
CSUM-MUX$_9$	474	296	116	481	313	113	507	337	140	410	266	108
QCLA-Com$_7$	417	222	113	419	250	112	411	225	127	423	241	113
QCLA-Mod$_7$	1146	674	416	1091	632	356	1131	668	446	1065	656	379
QCLA-Adder$_{10}$	631	366	222	647	381	191	671	411	208	648	390	167
Adder$_8$	689	366	286	731	405	278	752	406	310	700	375	235
RC-Adder$_6$	214	115	136	220	119	128	230	120	132	217	123	118
Mod-Red$_{21}$	301	149	181	284	144	189	281	137	176	292	153	188
Mod-Mult$_{55}$	138	88	83	139	87	77	167	99	101	139	86	79
Toff-Barenco$_3$	46	21	38	59	37	46	57	32	44	53	31	43
Toff-NC$_3$	46	25	36	56	32	40	54	30	38	44	23	34
Toff-Barenco$_4$	102	56	82	92	48	76	89	45	69	102	52	88
Toff-NC$_4$	83	42	52	83	44	55	80	40	56	83	44	60
Toff-Barenco$_5$	139	71	108	156	85	115	145	73	114	132	67	101
Toff-NC$_5$	108	54	73	112	54	72	111	53	72	114	58	77
Toff-Barenco$_{10}$	373	199	265	370	208	253	342	165	264	375	202	267
Toff-NC$_{10}$	275	133	155	278	148	185	280	145	177	275	142	172
GF(2^4)-Mult	420	304	200	410	287	192	474	342	237	406	273	177
GF(2^5)-Mult	569	432	235	612	472	242	634	496	269	557	421	243
GF(2^6)-Mult	1067	811	430	1083	811	398	1103	843	414	782	601	293
GF(2^7)-Mult	1551	1195	564	1501	1157	562	1476	1124	521	1294	964	517
GF(2^8)-Mult	2133	1691	824	2129	1666	769	2353	1888	847	1784	1338	622
GF(2^9)-Mult	2546	1977	865	2575	2018	919	2693	2137	992	1942	1557	614
GF(2^{10})-Mult	3514	2790	1276	3432	2717	1096	3625	2923	1249	2507	2046	795
GF(2^{16})-Mult	9624	7898	2803	9928	8257	2916	11046	9275	3754	5725	4633	1571
GF(2^{32})-Mult	37014	32676	9505	49810	43157	13850	46375	42089	13252	32966	28723	7740
GF(2^{64})-Mult	226794	201158	54525	253557	227274	66990	235681	218913	65850	138071	121394	26335
Avg. reduction (%)		-		**-3.8**	**-7.3**	**-1.5**	**-5.1**	**-6.6**	**-7.0**	**7.8**	**6.0**	**10.9**

doi:10.6342/NTU202303169

From the last column in the “Qsyn-D3” section of Table 6-3, it can be observed that approximately 70% of the circuits achieve better optimization results when the decomposition level is set to 3, as compared to the original PyZX workflow. On average, these circuits achieve a 7.8% reduction in gate count, a 6.8% decrease in 2-qubit gate count, and a 10.9% reduction in circuit depth. Furthermore, we also observed from the table that for some smaller circuits, decomposition level 2 tends to outperform level 3, while level 1 shows little improvement compared to PyZX due to the similar underlying structure of CCNOT gates used in both Qsyn and PyZX. Moreover, it has been observed that around 30% of the circuits continue to exhibit better performance under PyZX's original workflow compared to the outcomes achieved through the implementation of the decomposition adjustment. Consequently, a strategic decision was made to introduce the dynamic reduction algorithm as an experimental measure aimed at further enhancing the optimization process.

doi:10.6342/NTU202303169

6.4 Dynamic Reduction Algorithm

In the preceding experiment, it was observed that a subset of ZX-diagrams did not undergo optimal optimization. Building upon this finding, we implemented the dynamic reduction algorithm to prevent over-optimization during full reduction. The aim was to maintain a high T-count reduction rate while obtaining a ZX-diagram with minimal density for subsequent extraction processes. The experimental procedure is illustrated in Fig. 6-3. In this experiment, we select the decomposition level that performs the best as the representative for comparison, denoted by *n*, which ranges from 1 to 3.

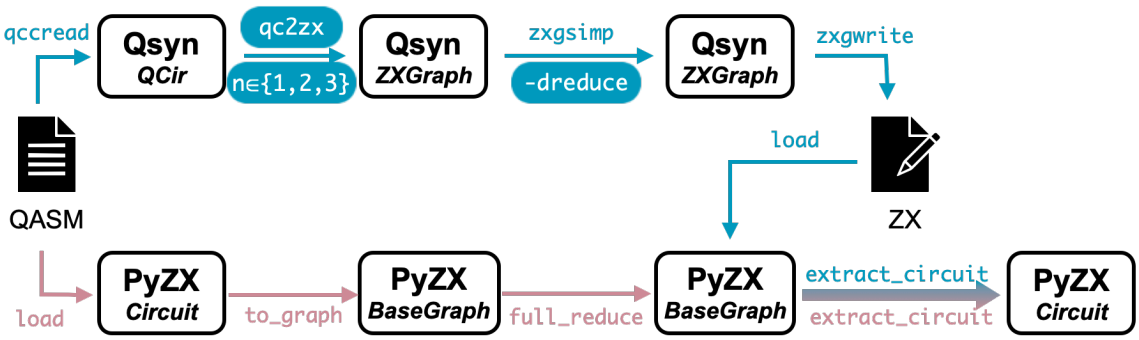


Fig. 6-3 The experiment workflow for the comparison of dynamic reduction algorithm and full simplification using PyZX

The results are summarized in Table 6-4, which includes the circuit name and the metrics for both the control group by PyZX and the dynamic reduction algorithm. The last column, "Reduction Rate," calculate the reduction rates of the gate count, 2-qubit count, and circuit depth for each circuit. The final row in the table represents the average reduction rate across the 29 test cases. We designate a green background color for the data with reduction rates greater than 0.

doi:10.6342/NTU202303169

Table 6-4 Circuit metrics for the control group and our dynamic reduction algorithm

Circuit	PyZX *full_reduce*			Qsyn-D_n *dynamic_reduce*			Reduction Rate		
	Σ	2Q	D	Σ	2Q	D	Σ	2Q	D
Mod 5_4	33	23	27	36	26	23	-9.1%	-13.0%	14.8%
VEB-Adder$_3$	127	74	75	112	68	72	11.8%	8.1%	4.0%
CSLA-MUX$_3$	255	161	115	211	129	106	17.3%	19.9%	7.8%
CSUM-MUX$_3$	474	296	116	410	266	108	13.5%	10.1%	6.9%
QCLA-Com$_7$	417	222	113	411	225	127	1.4%	-1.4%	-12.4%
QCLA-Mod$_7$	1146	674	416	1016	590	344	11.3%	12.5%	17.3%
QCLA-Adder$_{10}$	631	366	222	648	390	167	-2.7%	-6.6%	24.8%
Adder$_8$	689	366	286	700	375	235	-1.6%	-2.5%	17.8%
RC-Adder$_6$	214	115	136	215	121	115	-0.5%	-5.2%	15.4%
Mod-Red$_{21}$	301	149	181	251	125	161	16.6%	16.1%	11.0%
Mod-Mult$_{55}$	138	88	83	139	86	79	-0.7%	2.3%	4.8%
Toff-Barenco$_3$	46	21	38	53	31	43	-15.2%	-47.6%	-13.2%
Toff-NC$_3$	46	25	36	44	23	34	4.3%	8.0%	5.6%
Toff-Barenco$_4$	102	56	82	89	45	69	12.7%	19.6%	15.9%
Toff-NC$_4$	83	42	52	80	40	56	3.6%	4.8%	-7.7%
Toff-Barenco$_5$	139	71	108	132	67	101	5.0%	5.6%	6.5%
Toff-NC$_5$	108	54	73	111	53	72	-2.8%	1.9%	1.4%
Toff-Barenco$_{10}$	373	199	265	342	165	264	8.3%	17.1%	0.4%
Toff-NC$_{10}$	275	133	155	275	142	172	0.0%	-6.8%	-11.0%
GF(2^4)-Mult	420	304	200	335	239	151	20.2%	21.4%	24.5%
GF(2^5)-Mult	569	432	235	550	410	206	3.3%	5.1%	12.3%
GF(2^6)-Mult	1067	811	430	782	601	293	26.7%	25.9%	31.9%
GF(2^7)-Mult	1551	1195	564	1046	806	388	32.6%	32.6%	31.2%
GF(2^8)-Mult	2133	1691	824	1376	1083	502	35.5%	36.0%	39.1%
GF(2^9)-Mult	2546	1977	865	1942	1557	614	23.7%	21.2%	29.0%
GF(2^{10})-Mult	3514	2790	1276	2507	2046	795	28.7%	26.7%	37.7%
GF(2^{16})-Mult	9624	7898	2803	5725	4633	1571	40.5%	41.3%	44.0%
GF(2^{32})-Mult	37014	32676	9505	32966	28723	7740	10.9%	12.1%	18.6%
GF(2^{64})-Mult	226794	201158	54525	138071	121394	26335	39.1%	39.7%	51.7%
Avg. reduction (%)							**11.5%**	**10.5%**	**14.8%**

doi:10.6342/NTU202303169

When combining the decomposition level adjustment with the dynamic reduction algorithm, the degree of optimization is significantly improved, as shown in Table 6-4. On average, the optimizations are 11.5% for gate count reduction, 10.5% for 2-qubit gate count reduction, and 14.8% for circuit depth reduction. Some individual circuits, such as Galois field multipliers (GF(2^k)-Mult), demonstrate even more significant improvements, with a reduction of 40.5% in gate count, 41.3% in 2-qubit gate count (GF(2^{16})-Mult), and a remarkable 51.7% reduction in circuit depth (GF(2^{64})-Mult). These findings highlight the effectiveness of combining the decomposition level and dynamic reduction algorithm in achieving substantial optimization gains for various circuits, particularly in the case of larger circuits.

doi:10.6342/NTU202303169

Chapter 7 Conclusion and Future Work

In conclusion, we have developed the Qsyn tool as a flexible and efficient implementation platform for quantum circuit synthesis. By integrating decomposition level adjustment and dynamic reduction algorithm for ZX-diagram, Qsyn has demonstrated remarkable improvements in terms of the gate count, 2-qubit gate count, and circuit depth reductions compared to the PyZX framework. The experimental results also indicate that Qsyn exhibits significantly faster execution times, especially for larger circuits.

Our findings demonstrate that by customizing the decomposition level and applying dynamic reduction algorithms, we can effectively optimize quantum circuits. The ability to fine-tune the decomposition level based on circuit characteristics allows for better optimization outcomes. Additionally, the utilization of dynamic reduction algorithms further enhances the reductions of gate count, 2-qubit gate count, and circuit depth.

The development and utilization of Qsyn have demonstrated significant potential in advancing the field of quantum circuit optimization. Qsyn offers a versatile and customizable framework that can be extended to explore further optimization strategies and techniques. Its notable advantage over PyZX lies in its superior execution speed, rendering it highly suitable for analyzing and optimizing large-scale quantum circuits. The built-in structures and algorithms of Qsyn contribute to its exceptional performance and scalability, enabling efficient and effective circuit optimization.

Future work can involve investigating new decomposition methods, refining the dynamic reduction algorithms, and integrating more efficient simplification strategies

doi:10.6342/NTU202303169

with the algorithms of circuit extraction. Also, continuing to enhance and expand the functionalities and capabilities of Qsyn, including improvement of user interface and visualization, as well as integration with other quantum software libraries would enable researchers to advance in the field and contribute to the development of efficient and practical quantum computing technologies.

doi:10.6342/NTU202303169

REFERENCE

[1] M. Amy, "Algorithms for the optimization of quantum circuits," University of Waterloo, 2013.

[2] A. Shafaei, M. Saeedi, and M. Pedram, "Optimization of quantum circuits for interaction distance in linear nearest neighbor architectures," in *Proceedings of the 50th annual design automation conference*, 2013, pp. 1-6.

[3] N. Abdessaied, M. Soeken, and R. Drechsler, "Quantum circuit optimization by Hadamard gate reduction," in *Reversible Computation: 6th International Conference, RC 2014, Kyoto, Japan, July 10-11, 2014. Proceedings 6*, 2014: Springer, pp. 149-162.

[4] Y. Nam, N. J. Ross, Y. Su, A. M. Childs, and D. Maslov, "Automated optimization of large quantum circuits with continuous parameters," *npj Quantum Information,* vol. 4, no. 1, p. 23, 2018.

[5] A. Bocharov and K. M. Svore, "Resource-optimal single-qubit quantum circuits," *Physical Review Letters,* vol. 109, no. 19, p. 190501, 2012.

[6] A. G. Fowler, "Time-optimal quantum computation," *arXiv preprint arXiv:1210.4626,* 2012.

[7] M. Amy, D. Maslov, M. Mosca, and M. Roetteler, "A meet-in-the-middle algorithm for fast synthesis of depth-optimal quantum circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 32, no. 6, pp. 818-830, 2013.

[8] M. A. Nielsen and I. Chuang, "Quantum computation and quantum information," ed: American Association of Physics Teachers, 2002.

doi:10.6342/NTU202303169

[9] M. Coggins, *Introduction to quantum computing with qiskit*. Scarborough Quantum Computing Ltd, 2021.

[10] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th annual symposium on foundations of computer science*, 1994: Ieee, pp. 124-134.

[11] B. P. Lanyon *et al.*, "Experimental demonstration of a compiled version of Shor's algorithm with quantum entanglement," *Physical Review Letters,* vol. 99, no. 25, p. 250505, 2007.

[12] T. Monz *et al.*, "Realization of a scalable Shor algorithm," *Science,* vol. 351, no. 6277, pp. 1068-1070, 2016.

[13] J. O'Gorman and E. T. Campbell, "Quantum computation with realistic magic-state factories," *Physical Review A,* vol. 95, no. 3, p. 032338, 2017.

[14] C.-C. Lin, S. Sur-Kolay, and N. K. Jha, "PAQCS: Physical design-aware fault-tolerant quantum circuit synthesis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems,* vol. 23, no. 7, pp. 1221-1234, 2014.

[15] B. Gerard and M. Kong, "String Abstractions for Qubit Mapping," *arXiv preprint arXiv:2111.03716,* 2021.

[16] A. Paler, I. Polian, K. Nemoto, and S. J. Devitt, "Fault-tolerant, high-level quantum circuits: form, compilation and description," *Quantum Science and Technology,* vol. 2, no. 2, p. 025003, 2017.

[17] A. Zulehner, A. Paler, and R. Wille, "An efficient methodology for mapping quantum circuits to the IBM QX architectures," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 38, no. 7, pp. 1226-1236, 2018.

doi:10.6342/NTU202303169

[18] G. Burkard and D. Loss, "Cancellation of spin-orbit effects in quantum gates based on the exchange coupling in quantum dots," *Physical review letters,* vol. 88, no. 4, p. 047903, 2002.

[19] N. Abdessaied, R. Wille, M. Soeken, and R. Drechsler, "Reducing the depth of quantum circuits using additional circuit lines," in *Reversible Computation: 5th International Conference, RC 2013, Victoria, BC, Canada, July 4-5, 2013. Proceedings 5*, 2013: Springer, pp. 221-233.

[20] R. Duncan, A. Kissinger, S. Perdrix, and J. Van De Wetering, "Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus," *Quantum,* vol. 4, p. 279, 2020.

[21] A. Paler, S. J. Devitt, K. Nemoto, and I. Polian, "Mapping of topological quantum circuits to physical hardware," *Scientific reports,* vol. 4, no. 1, p. 4657, 2014.

[22] B. Coecke and R. Duncan, "Interacting quantum observables," in *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II 35*, 2008: Springer, pp. 298-310.

[23] B. Coecke and R. Duncan, "Interacting quantum observables: categorical algebra and diagrammatics," *New Journal of Physics,* vol. 13, no. 4, p. 043016, 2011.

[24] J. van de Wetering, "ZX-calculus for the working quantum computer scientist," *arXiv preprint arXiv:2012.13966,* 2020.

[25] A. Kissinger and J. van de Wetering, "Reducing the number of non-Clifford gates in quantum circuits," *Physical Review A,* vol. 102, no. 2, p. 022406, 2020.

doi:10.6342/NTU202303169

[26] A. Kissinger and J. van de Wetering, "PyZX: Large scale automated diagrammatic reasoning," *arXiv preprint arXiv:1904.04735,* 2019.

[27] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, "t| ket⟩: a retargetable compiler for NISQ devices," *Quantum Science and Technology,* vol. 6, no. 1, p. 014003, 2020.

[28] L. S. Bishop, "Qasm 2.0: A quantum circuit intermediate representation," in *APS March Meeting Abstracts*, 2017, vol. 2017, p. P46. 008.

[29] M. Amy, D. Maslov, and M. Mosca, "Polynomial-time T-depth optimization of Clifford+ T circuits via matroid partitioning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems,* vol. 33, no. 10, pp. 1476-1489, 2014.

[30] M. Amy. "T-par GitHub." https://github.com/meamy/t-par.git (accessed.

[31] N. J. R. Yunseong Nam, Yuan Su, Andrew M. Childs, Dmitri Maslov. "optimizer GitHub." https://github.com/njross/optimizer.git (accessed.

doi:10.6342/NTU202303169

APPENDIX

A.1 ZX File Format

The names of files are supposed to have a "*.zx*" extension. Each line in the file represents individual vertex information in the format as follows:

```
< H | I | O | X | Z ><ID> <optional_information>
```

The positional information consists of a vertex type which is either "H", "I", "O", "X" or "Z" on the first character position, followed by a vertex ID, which is not separated by a space. After a space character, optional information may contain qubit number, column index, neighbors' information, and carried phase. The information mentioned above should provide sequentially in the following format: type, ID, qubit number, column index, neighbors information, and phase.

- **Mandatory Information**

  Since vertex type and vertex ID are positional information, they should be provided without space between them and always provided at the beginning of a line.

  We use a single character to represent different vertex types, "H", "I", "O", "X" or "Z". We accept both uppercase and lowercase characters. The interpretation of the characters is as follows:

```
H = H-box              (H_BOX Type)
I = Input              (BOUNDARY Type)
O = Output             (BOUNDARY Type)
X = X-spider           (X Type)
Z = Z-spider           (Z Type)
```


doi:10.6342/NTU202303169

The graphical representation of different vertex types in the ZX-diagram is shown in Fig. A-1. The `"BOUNDARY"` corresponds to the left top black dot while the `"H_BOX"` corresponds to the right top yellow square in the figure. The left bottom green dot is called `"Z"` and the right bottom red dot is called `"X"`.

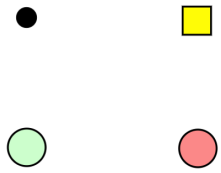


Fig. A-1 Vertex type in ZX-diagram form

After a single-character representation of the vertex type, a non-negative integer as the vertex `ID` should be provided. Note that vertex `ID` should be unique in a file, so duplicated vertex `ID` is not allowed.

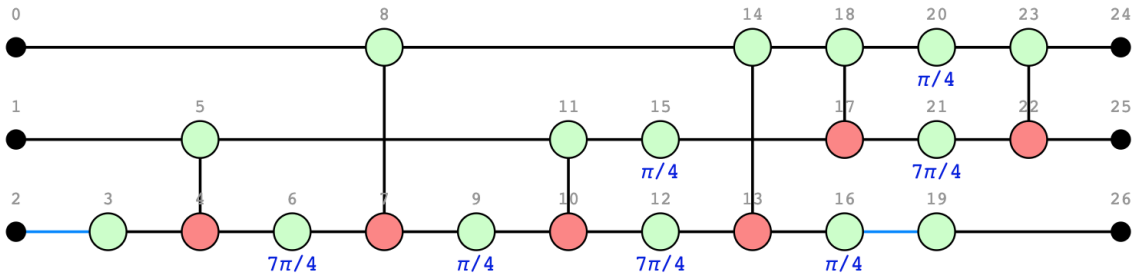


Fig. A-2 ZX-diagram implementing a Toffoli gate with labels

Take Fig. A-2, which is the labeled Toffoli ZX-diagram as in Fig. 4-6, as an example, the leftmost black dots will be represented as `I0`, `I1`, and `I2` respectively. Similarly, the rightmost black dots will be stated as `O24`, `O25`, and `O26` from the top to the bottom. The green vertex connected to `I2` should be noted as `Z3` and the red vertex connected to `Z3` is supposed to be `X4`.

doi:10.6342/NTU202303169

- **Optional Information**

The optional information of a vertex is composed of three parts, which are separated by spaces. The first part provides the qubit number and the column index which are separated by a comma, `","`, and enclosed in parentheses. The qubit number should be an integer and the column index is supposed to be a non-negative integer. The main purpose of providing the qubit number and the column index is to draw the ZX-diagram accurately. Thus, such information is not necessary without visualization. Since the qubit number and the column index are optional, they should be left as `"-"` if no information is given. This part can be left out if both the qubit number and the column index are `"-"`. For example, in Fig. A-2, the qubit number and column index of `I0` can be stated as `(0,0)` and those of `Z8` can be expressed as `(0,4)`. If the information is not provided, it can be stated as `(-,-)` or omitted entirely.

The second part records the information of all the neighbors connected to the vertex. Each neighbor's information should be in the format as follows:

```
< S | H ><ID>
```

That is a single character of either `"S"` or `"H"` followed by the `ID` of the neighbor without a space between them. Add `"S"` before the `ID` of the neighbor represents the vertex connects with the neighbor by a `"Simple Edge"` which is usually visualized as the black solid line. Similarly, if the vertex and the neighbor are connected by a `"Hadamard Edge"`, an `"H"` should be added in the front of the `ID` of the neighbor. If the vertex is connected to many neighbors, all the neighbor information may be listed in any order and each one should be separated by a space

doi:10.6342/NTU202303169

" ". As an added note, the two endpoints of an edge do not need to record each other as its neighbor at the same time. For example, if vertex B is recorded as a neighbor of vertex A, while A is not recorded as a neighbor of B, A and B are treated as connected. In Fig. A-2, since `Z9`, `Z11`, and `Z12` are all neighbors of `X10`, the neighbor information of `X10` can be stated as "`S9 S11 S12`".

The last part marks the phase carried by the vertex. It is recommended to provide the phase in the range of 0 to 2π. If the phase carried by the vertex is 0, it can be omitted. The format of the phase should follow the form of "`a*pi/b`" that "`a`" must be an integer and "`b`" should be a non-zero integer. If "`a`" is equal to 1, "`a*`" can be omitted. Also, if "`b`" is equal to 1, "`b`" can be left out. For instance, the phase of `Z12` in Fig. A-2 should be noted as `7*pi/4`.

- **An example**

Below is a zx file describing the ZX-diagram in Fig. A-3.

```
I0 (0,0) S8
I1 (1,0) S5
I2 (2,0) H3
Z3 (2,1) S4
X4 (2,2) S5 S6
Z5 (1,2) S12
Z6 (2,3) S7 7*pi/4
X7 (2,4) S8 S9
Z8 (0,4) S11
Z9 (2,5) S13 pi
O11 (0,6)
O12 (1,6)
O13 (2,6)
```


doi:10.6342/NTU202303169

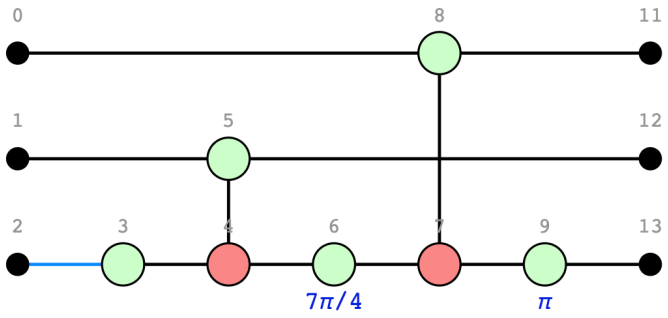


Fig. A-3 A basic example of a ZX-diagram

If the file is not intended to be visualized as a ZX-diagram, it may be simplified as follows:

```
I0 (0,-) S8
I1 (1,-) S5
I2 (2,-) H3
Z3 S4
X4 S5 S6
Z5 S12
Z6 S7 7*pi/4
X7 S8 S9
Z8 S11
Z9 S13 pi
O11 (0,-)
O12 (1,-)
O13 (2,-)
```

It is important to emphasize that the qubit number of inputs and outputs are necessary and all the qubit numbers of inputs and outputs should be unique respectively.

doi:10.6342/NTU202303169

A.2 Command List for Qsyn

Table A-1 provides a comprehensive overview of the commands available in Qsyn and their corresponding descriptions. Qsyn is a powerful software system designed for the synthesis, optimization, and verification of quantum circuits used in quantum computers. The table lists the various commands that users can utilize within the Qsyn framework to perform specific tasks. Each command is accompanied by a description that highlights its functionality and purpose. This table serves as a handy reference for users, allowing them to easily access and understand the different commands provided by Qsyn. With this information, users can effectively navigate the software and leverage its capabilities to perform various quantum circuit operations and optimizations.

Table A-1 All commands provided in Qsyn and their description

Command	Description
color	Toggle Colored printing
dofile	Execute the commands in the dofile
dtcheckout	Checkout to *Device* <id> in *DeviceMgr*
dtdelete	Remove a *Device* from *DeviceMgr*
dtgprint	Print information of *Device* topology
dtgread	Read a *Device* topology
dtprint	Print information of *DeviceMgr*
dtreset	Reset *DeviceMgr*
duoprint	Print Duostra parameters
duoset	Set Duostra parameters(s)
duostra	Map logical circuit to physical circuit
extprint	Print information of extracting ZX-graph
extract	Perform step(s) in extraction
extset	Set extractor parameters

doi:10.6342/NTU202303169

Command	Description
help	Show helping messages of commands
history	Print command history
mpequiv	Check equivalence of the physical and the local circuits
optimze	Optimize *QCir*
qc2ts	Convert *QCir* to *Tensor*
qc2zx	Convert *QCir* to *ZXGraph*
qcbadd	Add qubit(s) to *QCir*
qcbdelete	Delete qubit to *QCir*
qccheckout	Checkout to *QCir* <id> in *QCirMgr*
qccompose	Compose a *QCir*
qccopy	Copy a *QCir* to *QCirMgr*
qccprint	Print information of *QCir*
qccread	Read a circuit and construct the corresponding setlist
qccwrite	Write *QCir* to a QASM file
qcdelete	Remove a QCir from QCirMgr
qcgadd	Add quantum gate to QCir
qcgdelete	Delete quantum gate in QCir
qcgprint	Print gate information in QCir
qcnew	Create a new QCir to QCirMgr
qcprint	Print information of QCirMgr or settings
qcreset	Reset QCirMgr
qcset	Set QCir parameters
qctensor	Tensor a QCir
qquit	Quit Qsyn
seed	Set the random seed
tsadjoint	Adjoint the specified Tensor
tsequiv	Check the equivalency of two stored Tensors
tsprint	Print information of TensorMgr
tsreset	Reset TensorMgr

doi:10.6342/NTU202303169

Command	Description
usage	Report the runtime and/or memory usage
verbose	Set verbose level to 0-9 (default: 3)
zx2qc	Extract QCir from ZXGraph
zx2ts	Convert ZXGraph to Tensor
zxcheckout	Checkout to ZXGraph <id> in ZXGraphMgr
zxcompose	Compose a ZXGraph
zxcopy	Copy a ZXGraph into ZXGraphMgr
zxdelete	Remove a ZXGraph from ZXGraphMgr
zxgadjoint	Adjoint ZXGraph
zxgassign	Assign quantum states to input/output *ZXVertex*
zxgdraw	Draw *ZXGraph* into a pdf file
zxgedit	Edit *ZXGraph*
zxggflow	Calculate the generalized flow of current *ZXGraph*
zxgprint	Print information of *ZXGraph*
zxgread	Read a zx file and construct the corresponding *ZXGraph*
zxgsimp	Perform simplification strategies for ZXGraph
zxgtest	Test ZXGraph structures and functions
zxgtraverse	Traverse ZXGraph and update topological order of all ZXVertices
zxgwrite	Write a ZXGraph to a file
zxnew	Create a new ZXGraph to ZXGraphMgr
zxopt	Dynamic optimization for ZXGraph
zxoptprint	Print parameter of optimizer for ZXGraph
zxoptr2r	Set r2r parameter of optimizer for ZXGraph
zxopts2s	Set s2s parameter of optimizer for ZXGraph
zxprint	Print information of ZXGraphMgr
zxreset	Reset ZXGraphMgr
zxtensor	Tensor a ZXGraph

doi:10.6342/NTU202303169

A.3 Optional Arguments for `ZXGSimp`

Table A-2 provides a list of optional arguments for the command `ZXGSimp` in Qsyn. These arguments allow users to customize the simplification process of the ZX-diagram, enabling fine-grained control over the optimization procedure. Users can tailor the simplification algorithm to their specific requirements and achieve desired optimization outcomes by specifying these optional arguments.

Table A-2 Optional arguments list for *ZXGSimp*

Flag	Description
`-dreduce`	Perform dynamic reduction
`-freduce`	Perform full reduction
`-sreduce`	Perform symbolic reduction
`-interclifford`	Perform interior clifford simplification
`-clifford`	Perform clifford simplification
`-bialgebra`	Apply bialgebra rule
`-gadgetfusion`	Fuse phase gadgets connected to the same set of vertices
`-hfusion`	Remove adjacent H-boxes or H-edges
`-hrule`	Convert H-boxes to H-edges
`-idremoval`	Remove Z/X-spiders with no phases
`-lcomp`	Apply local complementation to vertices with phase of $\pm\frac{\pi}{2}$
`-pivotrule`	Apply pivot rule to vertex pairs with phase of 0 or π
`-pivotboundary`	Apply pivot rule to vertex pairs connected to the boundary
`-pivotgadget`	Unfuse the phase and apply pivot rules to form gadgets
`-spiderfusion`	Fuse spiders of the same color
`-stcopy`	Apply state copy rule
`-tograph`	Convert to green (Z) graph
`-torgraph`	Convert to red (X) graph

doi:10.6342/NTU202303169