Department of Informatics
Technical University of Munich

Guided Research - Project Report

Optimization of Quantum Circuits using Diagrammatic Calculi

Fabian Putterer

Department of Informatics
Technical University of Munich

Guided Research - Project Report

Optimization of Quantum Circuits using Diagrammatic Calculi

Optimierung von Quantenschaltkreisen mithilfe diagrammatischer Kalküle

Author:	Fabian Putterer
Supervisor:	Prof. Dr. Christian Mendl
Advisor:	Qunsheng Huang
Submission Date:	October 18, 2021

Contents

1 Introduction

1.1 Quantum Optimization

In Quantum Computing, computations are commonly expressed using circuits. Those consist of multiple gates followed up by measurement operations that yield the final results. As these gates usually represent a unitary transformation, the entire circuit therefore also just represents a unitary transformation. There are multiple different ways to express this transformation and thereby multiple different circuits that perform the same computation. Those different representations might have vastly different sizes. As quantum computers are still very susceptible to errors, it is crucial to minimize the number of computations performed and therefore the number of gates in the circuit while maintaining the equality of the circuit and results.

Multiple different approaches can be used to optimize a circuit. The most common approach is to attempt to rewrite the circuit directly. This can be done by for example exploiting some mathematical properties of the gates involved, like the Hadamard gate being its own inverse or the ability to shift around and exchange certain gates. Another approach is using graphical (diagrammatic) representations including rewrite rules that allow transforming equivalent representations into each other. One of those so-called diagrammatic calculi is the ZX-Calculus, which this report will focus on mainly.

1.2 The ZX-Calculus

The ZX-Calculus is a graphical notation for representing linear maps between quantum states as diagrams. It generalizes quantum circuits by also allowing non-unitary transformations to be represented. Additionally, the calculus contains so-called rewrite rules that allow transforming different diagrams into each other while maintaining the equality of the underlying linear map.

1.3 Optimization using the ZX-Calculus

A circuit can be translated into a diagram representing the same unitary map. The rewrite rules can then be used to obtain a simplified version of the diagram that represents the same transformation. Afterward, a circuit can be extracted from the new diagram to arrive at an optimized version of the original circuit.

So far, most other research has focused on a specific optimization procedure using a few algorithmic rules. This work provides an implementation of the presented flow of operation:

1. Importing a circuit from OpenQASM
2. Translating the circuit to a diagram
3. Optimizing the diagram
4. Validating all performed transformations

It aims at providing a generalized framework for using any ruleset/calculus to perform, compare and evaluate optimization using diagrammatic calculi. Further research is required to extract a circuit from any resulting, arbitrary diagram.

First, a data structure for representing and working with circuits and diagrams as well as importing them from the common OpenQASM notation is proposed. An approach to diagram visualization is presented. Afterward, an algorithm for circuit to diagram translation is described. Then, validation approaches for circuits and diagrams are discussed and implemented. Lastly, a matching and rewriting system for applying an arbitrary ruleset for optimization of any diagram is proposed and implemented. The used rewrite rules do not have to be manually written but can be specified as a static rule data structure declaring the desired patterns before and after rewriting as well as their properties. Correctness of the rewriter is shown based on the default ZX-Calculus rules using unit testing and the validity checker developed beforehand.

2 Related Work

Duncan et al.[9] describe a full optimization procedure for quantum circuits using the ZX-Calculus. After translating the circuit to a diagram, they additionally convert it into a so-called graph-like state. The resulting diagram then only contains Z spiders, no self-loops or parallel wires and all wires contain a Hadamard gate. The optimization procedure is using only a few selected rules, mainly based on local complementation and pivoting. This yields an optimized diagram with all X spiders eliminated. All non-Clifford spiders will then be present only as the so-called "inner spiders". They present a circuit extraction procedure for Clifford-only and general circuits which is based on assumptions fulfilled by the used optimization procedure. It is therefore not applicable to any general diagram yet.

This optimization procedure is implemented by Kissinger and van de Wetering[12]. Their work, called PyZX, deals with implementation-related issues like the representation of circuits, removal of self-loops and parallel wires as well as the optimization procedure using matching and rewriting. They demonstrate successful circuit extraction based on the work by Duncan et al.[9] by implementing the simplification strategy and extending the extraction algorithm. They implement all matchers and rewriters for simplification manually without using a generalized framework.

There have been other attempts at generalizing extraction, like the work by Backens et al.[5], but they require additional properties like preserving so-called "gflow" during optimization. So far there is no known procedure that can reconstruct a circuit from any given ZX-diagram.

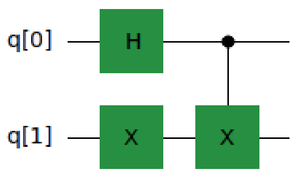
Therefore research has also concentrated on other approaches for exploiting the insight gained from the ZX-Calculus for circuit optimization. In their work on T-count reduction, Kissinger and van de Wetering[13] describe how to optimize the circuit directly. Based on the ZX-diagram representation, they gain information on possible rewrites which they exploit using a trick called "phase teleportation" to modify the original circuit directly, thereby sidestepping the extraction problem.

The goal of this work is to provide a generalized framework for reading and translating circuits and optimizing the resulting diagrams using an arbitrary user-defined ruleset. Compared to a fixed set of rules that have to be manually implemented as used in other research, the presented algorithms allow specifying rewrite rules based on their source and target structures in the resulting diagrams. Matchers and rewriters for applying them and their inverse are then automatically generated. This allows optimizing ZX-diagrams with different types of rulesets and calculi. Further research will be required on how to extract a circuit from an arbitrary diagram.

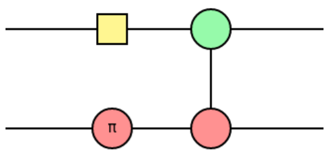
3 The ZX-Calculus

The ZX-Calculus is a graphical notation for representing linear maps between states introduced by Coecke and Duncan [6]. It generalizes quantum circuits, which are only able to represent unitary maps between quantum states using the so-called ZX-diagrams. An example for a quantum circuit and an equivalent ZX-diagram can be seen in Figure 3.1

The ZX-Calculus, being a calculus, also contains rewrite rules. Those allow transforming one diagram into another while maintaining the equality of the represented linear map. They can be used for deriving and proving equalities of different diagrams and therefore in extension circuits.



(a) The quantum circuit



(b) The ZX-diagram

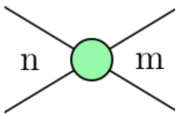
Figure 3.1: A quantum circuit and an equivalent ZX-diagram

3.1 Generators

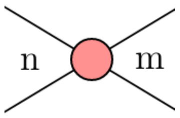
A ZX-diagram is a type of tensor network composed of a few basic building blocks called generators. The most fundamental of those generators is the spider, which got its name due to its legs. All spiders have a color, green or red, determining which operator they represent. Additionally, each spider has a phase. If the phase is not specified, it is assumed to be 0 as can be seen in Figure 3.1b.

The other basic building blocks of ZX-diagrams are wires. Wires connect spiders to each other. They always have two ends, although those ends do not necessarily have to be connected to a node. An open end represents a boundary of the map where it can be contracted with another map or which can be used as an in- or output.

In most literature, another generator can be found, the Hadamard node. It is usually indicated by a yellow rectangle. Technically it is not a part of the mathematical definition of the ZX-Calculus but can be generated using three spiders as part of the Euler decomposition of the Hadamard gate. It can be used in visualizations as a shorthand.



$$\begin{cases} |0\rangle^{\otimes n} & \mapsto |0\rangle^{\otimes m} \\ |1\rangle^{\otimes n} & \mapsto e^{i\alpha} |1\rangle^{\otimes m} \\ else & \mapsto 0 \end{cases}$$



$$\begin{cases} |+\rangle^{\otimes n} & \mapsto |+\rangle^{\otimes m} \\ |-\rangle^{\otimes n} & \mapsto e^{i\alpha} |-\rangle^{\otimes m} \\ else & \mapsto 0 \end{cases}$$

Figure 3.2: Definition of the tensor represented by a spider with a phase of α

3.2 Rewrite Rules

The rewrite rules specify modifications that can be applied to diagrams while preserving the equality of the underlying linear map. They can be applied to subgraphs of the diagram by matching them with one side of the rule and replacing the affected part of the diagram with the other side. This can be seen in Figure 3.3 for the Bialgebra Law B2 rule.

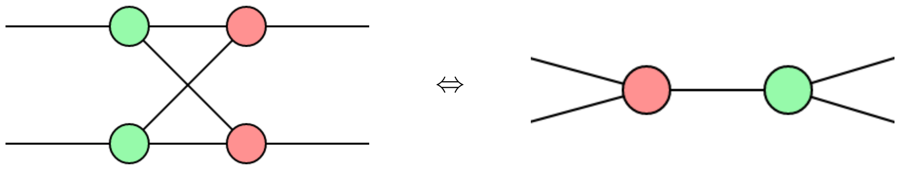
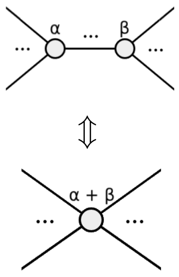


Figure 3.3: An application of the B2 rule to a ZX-diagram

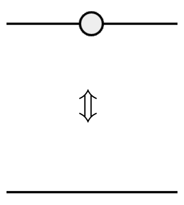
When used to rewrite diagrams, these rules can introduce a constant phase offset which can be ignored when optimizing diagrams. Such diagrams are therefore considered equivalent in this work and the rules usually dealing with those are not listed. The ruleset of the ZX-Calculus[6] can be found in Figure 3.4.

Additional rules, like the self inverse property of the Hadamard gate, can and will be added in other contexts but can also be derived by using the other rules of the calculus.

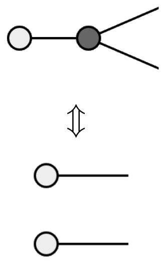
Figure 3.4: The rewrite rules of the ZX-Calculus



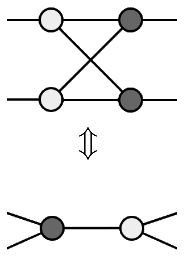
(a) Spider rule S1



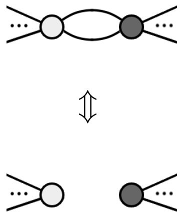
(b) Spider rule S2



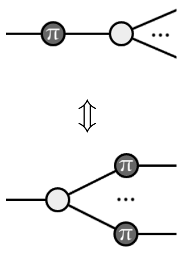
(c) Copy rule B1



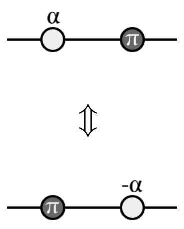
(d) Bialgebra law B2



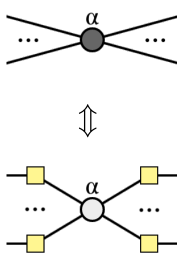
(e) Hopf law H



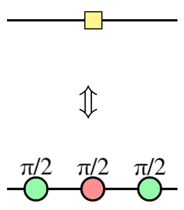
(f) π-copy rule K1



(g) π-commutation rule K2



(h) Color rule C



(i) Euler decomposition of the Hadamard gate

4 Implementation

As part of this research project, the data structures, processing methods, and algorithms presented in this report were implemented and evaluated. More details on how to use the implementation can be found in Appendix 9.1.

4.1 Toolset

Quantum gates, circuits, and diagrams usually represent underlying numerical and linear structures like matrices and tensors. To ease working with those, the Python[19] programming language was chosen. It supports a large collection of libraries, e.g. `scipy`[11] and `numpy`[10], that help in dealing with scientifically oriented, mathematical concepts. Additionally, this ensures the possible interoperability with other projects in the field which are often built based on Python as well.

One of the main drawbacks of this choice of language is related to performance. Other compiled languages that produce native executables have faster execution speeds compared to Python. This can be alleviated though by shifting the most intensive algorithms into a native implementation and just interfacing them using slower Python. This is done in this project's implementation by using `graph-tool`[3] for working with graphs which offloads the very costly subisomorphism search in graphs to a native executable.

The implementation supports working in an interactive environment like IPython[16] and Jupyter[15] as well as a standalone application window that can be used to interact with circuits and diagrams. PyGTK[4] is used as the windowing toolkit.

4.2 Quantum Circuits

Quantum circuits are stored based on their quantum and classical registers as well as individual components, the gates, barriers, and measurement operators. As they are quite different from graphs, a graph library is not used. Instead, all components are simply stored alongside each other and get assigned a "step" which represents its temporal location in the circuit. When a new component is added, all of the registers it will affect are searched for the largest step they were last involved in and the component is placed at the step after that. This ensures that there is only one component acting on each register at a time while maintaining the same order they were added in.

4.2.1 OpenQASM

Using the circuit structure defined above, it is possible to define and process a variety of quantum circuits. They can then be used for testing and benchmarking, allowing different optimization strategies to be evaluated. However, manually defining all circuits makes comparisons with other implementations complicated. All circuits as well as algorithms that generate entire families of benchmarking circuits would have to be transcribed into a different format. To simplify this process, parsing a circuit using a standardized notation used in lots of existing research benchmarks is desirable.

It was therefore decided to add support for reading circuits from files using the Open Quantum Assembly Language v2 (OpenQASM) as introduced by Cross et al. [8]. This provides a standardized way for importing sets of circuits as well as for outputting automatically generated circuits for benchmarking.

Listing 4.1: A circuit preparing a Bell state described in OpenQASM

```
OPENQASM 2.0;
include "qelib1.inc";

qreg q[2];
creg c[2];

h q[0];
cx q[0],q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
```

OpenQASM aims to be a simple, assembly-like language for defining quantum circuits. It is easy to write and understand. Technically it distinguishes itself quite a lot from a regular assembly language for a classical computer though. Additionally to containing a few basic gates, it supports defining new gates and therefore instructions which can themselves be composed of more self-defined gates. Those gates can use parameters that can be combined using unitary and binary operators, thereby forming an algebraic system. Such a self-defined gate as present in the standard library can be seen in Listing 4.2 It therefore requires a different approach for parsing compared to traditional assembly which can be read from top to bottom while executing each command step by step.

Listing 4.2: The controlled Z rotation (CRZ) gate from qelib1.inc [8]

```
gate crz(lambda) a,b
{
   u1(lambda/2) b;
   cx a,b;
   u1(-lambda/2) b;
   cx a,b;
}
```

Parsing

To simplify processing, `ANTLR`[1] was used to automatically read in OpenQASM-based circuits based on a specified grammar.

Most commonly used gates are not part of the actual OpenQASM specification but are defined in `qelib1.inc` which is usually imported at the start of each file. This library then defines those gates based on the unitary and controlled not gates. The implemented OpenQASM parser of this project fully supports reading those gate definitions as well as evaluating expressions. To simplify further processing, visualization, and translation to ZX-diagrams though, the `H, X, Y, Z, S, T` gates are hard-coded and detected in the parser.

4.3 Diagrams

To simplify the representation of diagrams in memory, special boundary generators which are not part of the ZX-Calculus itself have been used in this work to denote in- and outputs. Additionally, Hadamard gates are not stored as generators. Instead, each wire is given a property specifying if it contains a Hadamard gate. This reduces the expressiveness of the stored diagrams as it removes the ability to add multiple Hadamard gates sequentially. It does not affect the equivalence of the underlying map or the processing algorithms though, as multiple Hadamard gates just cancel themselves. This property will be indicated in visualizations by using blue color as can be seen in Figure 4.1. This representation is based on the work by Kissinger and van de Wetering in PyZX[12, p. 231].

The ZX-Diagrams themselves could be stored in the implementation by using a simple graph containing some node-based properties. To speed up processing as well as ease the computation of graph subisomorphisms used in the matcher (chapter 7) though, a library is used for storing and dealing with diagram graphs. In this case, `graph-tool`[3] was chosen. It is used to store the underlying graph of the diagram, using vertices as spiders and undirected edges as wires. Additional diagram properties like color and phase are stored separately.

Most of the rewrite rules do not require a specific color but change the color of spiders in a certain way or require multiple spiders to have the same/different color. Therefore the stored color may also contain a placeholder which is indicated in all figures of this report as well as the implementation by using white and grey colors.

Graph Library

`graph-tool`[3] is implemented as a native library and just exposes a Python interface. It is therefore able to maintain the performance of a native application for processing algorithms like subisomorphism search.

It allows storing additional properties such as color and phase of a spider or the Hadamard property of a wire in so-called `PropertyMaps`. Those can then also be used for the matching procedure to ensure equality of the Hadamard property on found subgraphs. Due to its performance-oriented implementation though, special care needs to be taken in case of diagram manipulating algorithms that e.g. remove spiders and therefore vertices from the graph. This causes all pointers to vertices and edges to be invalidated due to their index changing.

Additionally, `graph-tool` allows rendering graphs using `cairo`[2] which simplifies visualization as that library is also used for the rendering of circuits.

Additional properties

For each vertex, multiple additional properties are stored:

- **Vertex identifier:** Used to identify the vertices after a rewrite as the handles used by the graph library may have changed.
- **Vertex type:** Specifies whether the vertex is a spider or boundary as well as the type of boundary or the color of the spider.
- **Phase:** Specifies the phase of the given spider.
- **Qubit index:** Indicates the qubit register this spider or boundary was placed on in the original circuit which is useful for rendering.

Edges and therefore wires only have one property, specifying whether the wire contains a Hadamard gate or not.

4.4 Visualization

For development, testing, and evaluation purposes it will be necessary to display and interact with generated circuits and diagrams. The implementation therefore supports rendering those. This allows investigating the processing steps of each algorithm or calculus ruleset.

The output can be displayed as an image in an inline context within an interactive notebook or be rendered to a standalone PyGTK window. In the case of diagrams, the second option allows interacting with the diagram in real-time.

4.4.1 Circuit

Circuits are rendered using `cairo`[2] which can target an inline context as well as a PyGTK drawing area. First, the individual quantum and classical registers are rendered. Then, each component is added to the qubit it mainly affects. Finally, other affected qubits like control bits and measurement targets are connected.

4.4.2 Diagram

Rendering diagrams is a bit more complicated compared to circuits. They do not have a natural order by themselves and may be placed in any arrangement that maintains the given graph structure without changing their meaning. Rendering this can be achieved using functionality from the graph library.

To make working with them more intuitive, an algorithm was developed that tries to guess the qubit each spider should belong to and therefore to align it similarly to the original circuit.

It works by first placing the input boundary vertices based on the qubit registers that were stored when translating the circuit into a diagram. Afterward, it places all remaining spiders and boundaries one by one. In each step, it attempts to place as many spiders as possible. It looks at all vertices which neighbor an already placed one and tries to place

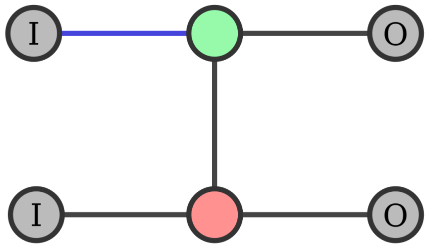


Figure 4.1: A visualized diagram preparing the Bell state $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$

them along with all of their placeable neighbors. A vertex is considered ready for placement if all of its unplaced neighbors on other registers are placeable within the current iteration.

This ensures spiders are placed flowing from inputs to outputs therefore maintaining the temporal flow of the original circuit. Additionally, looking at neighbors on other qubits aligns structures like controlled Pauli gates similarly to their placement in a quantum circuit.

This procedure most of the time produces an intuitive image that clearly shows the structure of the rendered diagram. In some cases, however, this approach will fail and may overlap wires therefore hiding them. To mitigate those cases and enable further investigation, diagrams shown in a standalone window can also be manipulated manually.

5 Circuit Translation

To optimize a circuit that was read from OpenQASM as described in Chapter 4, it needs to be translated into a diagram. An algorithm that achieves this is described in the following sections. The resulting diagram will be structured as described in Section 4.3 and therefore does not contain Hadamard generators anymore. This also automatically removes any duplicate Hadamard gates as can be seen in Figure 5.1c.

5.1 Translation Algorithm

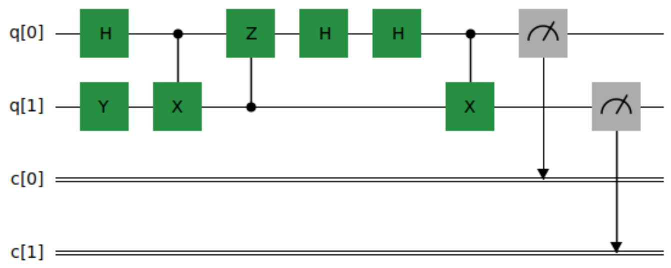
First, input boundaries are created for each register qubit. The algorithm maintains a frontier that points to the generator that was last added for each register. That way any newly added components can be added to the diagram in the correct position based on the registers they operate on. Additionally, another map is maintained, describing the Hadamard status of each register. When a Hadamard gate is applied to a register, the status is flipped. When a new component is added to a register, the newly added wire gets assigned the current Hadamard status of that wire.

Then, the algorithm proceeds through the circuit step by step. When it encounters a new gate, it will add it to the affected registers. This is achieved by mapping them to their current representation in the diagram based on the frontier and appending the new generators in that location. When it encounters a Hadamard gate, the Hadamard status is flipped as described above. Gates are mapped to their corresponding ZX representations based on the definition described by van de Wetering[18, p. 87]. The current implementation supports X, Y, Z, S, and T single-qubit gates as well as controlled X and controlled Z gates.

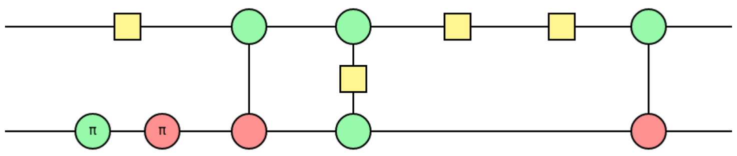
After adding each component for the current step, the frontier is advanced to the newly added generators and the correct Hadamard status is set to the wire and reset. This continues until the entire circuit has been processed and the output boundaries can be added, thereby completing the diagram.

5.1.1 Further Processing

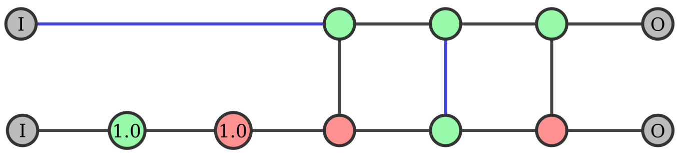
The produced diagram can be further processed if desired. While the implementation supports duplicate wires and recursive wires between spiders, a lot of simplification strategies require them to be removed. Therefore the implementation also supports resolving and removing any duplicate wires between two spiders as well as recursive wires on a single spider as described by PyZX[12]. This is achieved in part by using rewrite rules as specified in Chapter 7, in case of self-loops by simply searching for them and changing the phase of the affected spiders.



(a) A quantum circuit



(b) An equivalent translated ZX-diagram



(c) A simplified diagram indicating H gates using wires and removing duplicate gates

Figure 5.1: A quantum circuit getting translated into an equivalent ZX-diagram

6 Transform Validation

Using the matching and rewriting algorithms from Chapter 7, ZX-diagrams can be rewritten into different diagrams. This way the number of generators can be reduced and therefore the underlying circuit will be optimized. To ensure the validity of the implemented rewrite rules, the equality of the linear map of two different diagrams needs to be validated.

Validation is also required after translating a circuit to a diagram as described in Chapter 5 to ensure there were no errors in the translation and the diagram describes the same, in this case unitary, map as the input circuit.

6.1 Validation using the ZX-Calculus

Validation and therefore proving the equality of two ZX-diagrams can be achieved by using the ZX-Calculus itself without even looking at the represented linear map as proposed by Kissinger and van de Wetering[13]. This is achieved by exploiting the property of an inverted diagram representing the inverse linear map. Combining a diagram with its inverted self will therefore yield a diagram representing the identity which can be reduced to a single wire using the rules of the ZX-Calculus.

This way, the equality of two diagrams can be proven by inverting one and combining them. If a matching and rewriting algorithm is then able to reduce the resulting diagram to a single wire, the diagrams do indeed represent the same linear map.

The main disadvantage of this approach is that it uses the unit it is trying to test for verification. In case the matcher or rewriter is faulty or uses an incorrect ruleset, it may be able to prove the equality of two diagrams it just produced itself which do represent two different linear maps. Additionally, there is no guarantee this procedure will terminate in case it fails to find the correct transformation for reducing the diagram.

A different procedure will therefore be needed for verifying the correctness of the matching and rewriting and therefore optimization algorithms. This can then also be used for the validation of a circuit translation, as the previous technique is not able to investigate equality between a circuit and a diagram.

6.2 Validation using Tensor Contraction

Another approach for validation is computing the represented linear maps in the form of a tensor. This can be done for both, circuits as well as diagrams. This approach works for comparison with any structure that represents a linear map.

In case of a circuit, obtaining the unitary map is trivial. For diagrams, a tensor contraction can be used to obtain those linear maps. Afterward, they can be compared using some minor tolerance to cope with numerical errors.

6.2.1 Circuit validation

The tensor/matrix representing the unitary map of a circuit can be easily obtained by iterating over it step by step, where a step is defined as in Section 4.2. For each time step, the matrix representing the transform at that step is calculated and then composed together with all other matrices in order. The only challenge is obtaining the gate unitary matrix for arbitrary controlled gates.

For each step $s \in S$, first, the transformations of all non controlled gates N_s including the applied gates A_s as well as identities I_s for the qubits q without a gate are collected:

$$\begin{aligned} N_s &= \{U_{(s,q)} \mid q \in Q \land \neg C(q)\} \\ A_s &= \{U_{(s,q)} \mid q \in Q, U_{(s,q)} \in G_s\} \\ I_s &= \{I \mid q \in Q, U_{(s,q)} \notin G_s\} \end{aligned}$$

where $U_{(s,q)}$ is the unitary applied to qubit $q \in Q$ at step $s \in S$ and $C(q)$ indicates whether the unitary gate is a controlled gate.

They are joined using the Kronecker product to obtain the combined, non-controlled transformation $T_s^{\neg c}$ at that time step.

$$T_s^{\neg c} = \bigotimes_{n \in (A_s \cup I_s)} n$$

The controlled gate unitaries $T_s^c = \{C_{(s,q))} | q \in Q\}$ are obtained by summing both possible cases for the control bit.

$$C_{(s,q))} = C^0_{(s,q)} + C^1_{(s,q)}$$

First, using the projector $|0\rangle \langle 0|$ with the controlling qubit "set to $|0\rangle$" and the target qubit applying the identity.

$$C^0_{(s,q)} = I \otimes ... \otimes |0\rangle \langle 0| \otimes ... \otimes I$$

Secondly, using the projector $|1\rangle \langle 1|$ with the controlling qubit "set to $|1\rangle$" and the target qubit applying the controlled gate.

$$C^1_{(s,q)} = I \otimes ... \otimes |1\rangle \langle 1| \otimes U_{(s,q)} \otimes ... \otimes I$$

In both cases, the projector is at the location of the control qubit and $U_{(s,q)}$ at the location of the controlled qubit. Afterward, the combined transformation at that time step can be obtained as:

$$U_s = T_s^{\neg c} \times \prod_{C_{(s,q)} \in T_s^c} C_{(s,q)}$$

The circuit transformation can then be found as:

$$T = \prod_{s \in S} U_s$$

6.2.2 Diagram validation

As a diagram is just a tensor network, tensor contraction can be used to obtain the linear map it represents.

To ease the computational process, all red spiders are converted to green spiders using the Hadamard rule. This simplifies the tensors used for all spiders. As the used data structure uses a flag for each wire to store Hadamard gates, those need to be converted back to individual gates whose tensors will be used during the contraction as well.

Afterward, the tensor for each node needs to be determined. In case of Hadamard gates, it simply equals the 2-dimensional H matrix. Those nodes always have two adjacent edges, one "in-" and one "output". As $H = H^T$, the order of those also does not matter during contraction.

The tensor representation of all other nodes, being green spiders, is according to the ZX-Calculus definition[18, p. 13]

$$T^{j_1 \dots j_n}_{i_1 \dots i_m} = \begin{cases} 1 & \text{if } i_a = j_b = 0, \ \ \forall\, a \in [m] \ \ \forall\, b \in [n] \\ e^{i\alpha} & \text{if } i_a = j_b = 1, \ \ \forall\, a \in [m] \ \ \forall\, b \in [n] \\ 0 & \text{else} \end{cases}$$

where n and m are the in- and output dimensions. It therefore only contains two non-zero values. It additionally equals its transpose in any dimension due to being symmetric, meaning the order of in- and outputs does not matter for the spiders either.

Afterward, each edge gets assigned an index. Boundary indices get negative values. As discovered before, the dimension of contraction does not matter as all participating tensors are equal to their transpose in any dimension. We can therefore simply obtain the edges connected to each tensor and compute the contraction. In this case, the `tensornetwork`[17] library's `ncon` function was used to achieve this.

6.3 Implementation

The current implementation supports computing tensors representing the underlying transformation from circuits as well as diagrams. For the tensor contraction process it uses the `tensornetwork`[17] library. The resulting value will be a matrix expressing the transform from input qubits to output qubits. Equivalent transformations may differ by a scalar factor though. This means they can not be compared directly.

One solution to checking equivalence between two representations, e.g. a circuit and a diagram, is to generate a random input state $|s_{\text{in}}\rangle$ and apply the linear transformations U_1 and U_2 to it. Afterward, all measurements using random observables $O_1, \dots, O_n$ on the obtained states $|s_1\rangle = U_1 |s_{\text{in}}\rangle$, $|s_2\rangle = U_2 |s_i\rangle$ will yield equivalent results. As the states represent a quantum state though, the sum of their squared entries, representing a probability, will have to sum up to one. Therefore it is sufficient to compute the probabilities $\langle s_i | s_i \rangle, i \in \{1, 2\}$ for both states and then normalize them. The equivalence of the resulting two vectors then corresponds to the equivalence of the compared circuits and diagrams, taking into account numerical errors due to rounding.

7 Matching and Rewriting

One big challenge when using a calculus for automatic optimization is that applicable rules of the calculus need to be detected and locations in the diagram where they can be applied need to be found. The decision on which rule to apply where can then be made by the optimization algorithm (see Chapter 8).

Afterward, the diagram needs to be rewritten by replacing or moving the individual spiders while maintaining all their properties like color and phase as well as wires to other components not part of the replaced rule structure.

7.1 Rewrite Rules

The matching and rewriting algorithms require a ruleset specification, like the ZX-Calculus (see Section 7.4), which they can use for performing the actual restructuring. Those rules have to specify the "source" graph structure to look for in the diagram, its properties, as well as the target structure with which to replace it. Additionally, they need to specify how to apply properties gathered from the source to the new target, like the original colors, the phases as well as the wires connecting to spiders outside of the subsection affected by the replacement.

This allows defining an arbitrary ruleset and therefore using different calculi for implementing and experimenting with different optimization strategies.

7.1.1 Rule

The rules have the following structure:

- **Source structure:** The source structure describing the subgraph of the diagram the matcher should search for and that will be replaced by the rewriter.
- **Target structure:** The target structure describing the subgraph that will be inserted by the rewriter when applying the rule.
- **Variable mapping:** A mapping of variables from the source to the target structure. Variables will capture phase values from the diagram on matching and apply them to the target on rewriting.
- **Connecting wire mapping:** A mapping of spiders from the source to the target structure. Wires leading to diagram spiders that are not part of the rule will be mapped to the specified target spider on a rewrite.

In case of more complicated rules like the copying rule, the connecting wire mapping may also specify a list of target spiders for each spider in the source rule. In that case, wires leading to the spider are distributed among the target spiders equally.

7.1.2 Rule structure

The rule structure represents one half of the rule as being realized as a part of a diagram. It either describes the diagram part that will be replaced (**source**) or the part it will be replaced with (**target**).

- **Spiders:** The spiders that are part of this structure.
- **Spider phases:** A phase for each spider, specified by a phase expression (see Section 7.1.3).
- **Inner wires:** The wires between spiders of this structure.
- **Hadamard property:** Which of the inner wires contain a Hadamard gate.
- **Connecting wires:** How many spiders not part of the structure is a spider connected to. Can be unlimited. The number of connecting wires that will flip their Hadamard property also has to be specified.
- **Colors:** What colors do the spiders have. Apart from the usual colors *green* and *red*, *black* and *white* can be used to signify arbitrary, but identical/different colors. An arbitrary color is indicated by *gray*.
- **Variables:** Variables can be used as the phases of spiders, they need to be resolved during matching before the phases of the target structure can be determined.

7.1.3 Phase expression

Phase expressions represent the phase of a spider in the source or target structures. They can have arbitrary or constant values or represent a variable or binary operation. Arbitrary or constant values are only used during matching or for directly setting the phase value after the rewrite.

Variables will be resolved during matching while ensuring that all spiders with the same variable have the same phase. They will then be used for determining the phase of spiders of the target structure. This can be achieved by using an operation expression, which consists of two other expressions and an operation. This allows capturing two variables from the source part of the diagram before rewriting, summing them up, and setting the value of a new spider from the target to their sum.

7.1.4 Inverting

For each rule, an inverted rule is automatically computed and used if desired. This is achieved by simply exchanging **source** and **target** structures and inverting the variable and connecting wire mappings. Sometimes it may not be possible to invert a rule or doing so would cause a rule that matches in every possible location like **S2**. Those cases need to be handled separately.

7.2 Matching

The matching algorithm takes as input a diagram as well as a rule. It then searches the diagram for occurrences of the rule's source structure, verifies all necessary properties, and returns the occurrence location for further processing like rewriting.

Additionally, it captures the necessary, variable properties from the matched location that will be needed for rewriting.

1. First of all, the graph of the selected diagram is searched for subisomorphisms. This search is performed one by one checking the properties of possible matching candidates first instead of generating all subisomorphisms at once. For this, `graph-tool`[3]'s implementation of the VF2 algorithm[7] is used.

   Additionally, the Hadamard property of wires is also checked in this step, ensuring equality of Hadamard gates between the diagram and source rule wires.
2. Afterward the colors and phases of the matched subgraph are checked. If they contain wildcards like *black*, *white*, *grey* or a variable in case of the phase, the corresponding value is noted in the rule for rewriting. It is also used to verify that spiders with the same color/phase in the source part of the rule match in their values.
3. Then the number of wires to "external" spiders in the original diagram connecting to spiders of the source rule is checked. The "connecting neighbors" are noted down by the spider in the rule they connect to for rewriting. Additionally, for each connecting wire, the Hadamard property is captured and combined with the number of connecting wires to be flipped to determine if the wire after the rewrite should contain a Hadamard gate.
4. If all properties matched, the matching subgraph of the original diagram can be returned. Otherwise, the algorithm will reset the captured properties of the rule, restart at step 1 and look at the next generated subgraph.

An overview of the algorithm as pseudo-code can be seen in Listing 7.1.

Listing 7.1: The matching algorithm as pseudo-code

```
def match(diagram, rule):
   # find graph structure, including hadamards
   isomorphisms = diagram.search_graph_subisomorphisms(rule)

   for subgraph in isomorphisms:
      check_and_resolve_spider_colors()
      check_and_resolve_spider_phases()
      check_and_resolve_external_connecting_wire_count()
      determine_connecting_wire_hadamard_property()

      if all matched:
         return subgraph # containing resolved properties
      else:
         rule.reset()

   return NotFound
```

7.3 Rewriting

The rewrite algorithm can be performed after the matching algorithm has found a subgraph suitable for rewriting and will replace the found match representing the rule's source structure with the rule's target structure. It does so while maintaining the colors resolved from the match as specified in the rule. It also has to deal with connecting the replaced structure with external spiders while correctly determining if a Hadamard gate is needed.

It takes the original diagram, a rule, and the found subgraph as input and produces a rewritten diagram.

1. Captured properties from the matching process are resolved and copied over to the target structure. This involves resolving expressions such as rewrite variables which can be added to each other as well as obtaining the actual colors assigned to the placeholders *black*, *white* and *grey*. Additionally, unresolved colors for which the opposing color has been resolved need to be determined in cases where the rule e.g. flips the color of a spider like it is the case with the Hadamard rule.
2. The spiders of the rule's target structure are added.
   i. For visualization purposes, the register index of the added spider needs to be determined. This is achieved by guessing based on the source rule spider whose connecting wires get mapped to the newly added target spider.
   ii. The spider's color is determined based on either a fixed color value in the target structure or by looking at the resolved colors.
   iii. The spider's phase is determined based on the resolved phase variables.
3. The inner wires of the rule's target structure are added setting their Hadamard gate property according to the rule.
4. External neighbors in the diagram are connected to the newly added structure using wires. Each original spider in the subgraph of the diagram is mapped to its relative newly added target spider as specified in the rule's connecting wire mapping. For each new wire, its Hadamard property is determined by looking at its old property as well as the rule's structure specifying if it should get flipped.
5. The spiders of the rule's source structure are removed from the diagram.

Due to the way the used graph library, `graph-tool`[3], handles the internal representation of vertices, spiders from the source rule structure are only removed at the end so pointers to them used for determining the connecting neighbors are not invalidated beforehand.

The entire algorithm can be seen using pseudo code in Listing 7.2.

Listing 7.2: The rewrite algorithm as pseudocode

```
def rewrite(diagram, rule, matching_subgraph):
    rule.target.variables = source.resolve_variables()
    rule.target.colors = source.resolve_colors()
    rule.target.colors.resolve_unknown_placeholders()

    # add spiders and wires with correct properties
    add_target_structure(diagram, rule, matching_subgraph)

```

```
9      for spider in matching_subgraph:
10        neighbors = matching_subgraph.external_neighbors(spider)
11        target_spider = rule.connecting_wire_map[spider]
12
13        if target_spider exists:
14           for n in neighbors:
15              is_hadamard = was_hadamard[n] ^ should_flip[n]
16              diagram.add_new_wire(is_hadamard)
17        else:
18           for (n1, n2) in neighbors:
19              is_hadamard = was_hadamard[n1] ^ should_flip[n1] ^
                    was_hadamard[n2] ^ should_flip[n2]
20              diagram.add_new_wire(is_hadamard)
21
22     return diagram
23
24  def add_target_structure(diagram, rule, subgraph)
25     for spider in rule.target:
26        determine_qubit_index() # from matched subgraph
27        determine_color()
28        determine_phase()
29        diagram.add_new_spider()
30
31     for wire in rule.target:
32        diagram.add_new_wire(wire.is_hadamard)
```

7.4 Implementation of the ZX-Calculus Rules

Using the rule structure defined in Section 7.1.1 and Section 7.1.2 a lot of rewrite rules can be specified. This includes quite many of the ZX-Calculus' rules such as S1, S2, B1, B2, P2, and C.

But as creating a calculus gives near endless possibilities on how to design rewrite rules, not all possible rules are covered by the system. This includes e.g. the Pi Copy Rule K1, which requires creating a dynamic amount of spiders based on the number of connecting wires of a spider. Those edge cases need to be taken care of during the implementation.

7.4.1 Edge cases

Empty target structure

In some cases, like the S2 Spider rule, the resulting diagram might not contain a spider for the matched subgraph, in such a case the connecting neighbors need to be connected while correctly determining their Hadamard property. This is taken care of in line 24 of the pseudocode in Listing 7.2.

Multiple connecting wire targets

In other cases like the B1 rule, connecting wires from one spider might be mapped to multiple targets. In that case, the algorithm will have to additionally take care of distributing them equally among target spiders. This is not shown in Listing 7.2 for readability reasons.

Dynamic amount of spiders

Some rules like the Pi Copy Rule K1 can remove or add an arbitrary amount of spiders based on the number of connecting wires. To support this, the algorithm will have to additionally store a property for each spider of the rule specifying the type of spider added/removed on that connecting wire and apply that during matching and rewriting. As this requires generating spiders during rewriting, detecting spiders on connecting wires during matching, and supporting normal non-spider containing wires at the same time, it is currently not yet implemented.

7.4.2 Decision policies

Some rules are non-deterministic but require some kind of decision to be made such as splitting a spider into two, therefore distributing its phase over two new values such as S1. The same applies to rules that can be applied in lots of locations such as the Hopf Law H, the Color Rule C, and the Spider Rule S2. The latter one can be applied in any location to create a new spider, the color of which is arbitrary.

Therefore a decision policy needs to be passed to the matching and rewriting algorithms when constructing the rule. It will then be used by the algorithm to resolve such non-deterministic decisions, e.g. by obtaining the split phases of an inverse Spider Rule S1.

8 Optimization

Using the data structures and algorithms described in the previous chapters, full circuit translation and diagram optimization can now be implemented in one pipeline. First, the circuit is read using the OpenQASM parser from Chapter 4. Then, it is translated into a diagram according to Chapter 5. Afterward, the diagram is optimized by applying the rewrite rules of the desired calculus according to Chapter 7. Every step, including translation and rewriting, is validated as described in Chapter 6. The entire process can be seen in Figure 8.1

8.1 Optimization

The optimization procedure uses the rules of the calculus, as an example in this case the ZX-Calculus. It applies them using a trial and error approach while maintaining a ranking of the most and least desired rules to be applied.

The optimizer relies on a user-defined optimization strategy in conjunction with a simplifier to tell it which rule to apply next. Inspired by the work of Kissinger and van de Wetering in PyZX[12], simplifiers are categorized into two types, single rule and compound simplifiers. A single rule simplifier just contains a single rule, a compound one consists of multiple other simplifiers. This allows defining hierarchies of rules to be applied, therefore prioritizing them. This is desirable as some rules like the Bialgebra Law may only match in a few certain instances but simplify the diagram quite a lot. Others, on the other hand, might introduce new, undesired complexity and should only be chosen in a few scenarios where there is no other option.

To prevent the optimizer from getting stuck, randomized simplifiers are also implemented. They order their rules in a random way each time they are asked about which rules to try. This allows the optimizer to backtrack and reverse any rewrite it has performed and then afterward chose a different path.

In the current implementation, the only strategy used is ranked optimization, which simply uses the rules in the order they are supplied in by the simplifier. Once a rule matches, it starts back at the highest-ranking rule again. It terminates when no rule matches anymore.

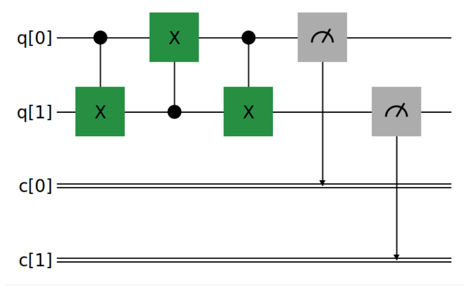
8.2 Extraction

After the diagram has been optimized, a circuit will have to be extracted. This has not been implemented as part of this research project due to time constraints. As described by Duncan et al.[9] and Backens et al.[5], there are approaches to circuit extraction. Those are limited to cases though, where additional conditions are met by the diagram, that have been taken into account by the previous optimization strategy.

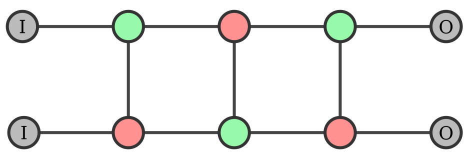
There are also other approaches like sidestepping the problem entirely by using the diagram merely as a guide and performing optimization on the original diagram directly as shown by Kissinger and van de Wetering[13].

As this work focused on developing a generalized optimization framework using diagrams for arbitrary rulesets, those extraction strategies are not applicable without substantial modification. Further research will be necessary on how to extract a circuit from any simplified diagram.

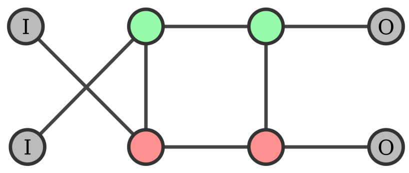
Figure 8.1: The entire pipeline automatically performing a validated translation and simplification of a swap circuit



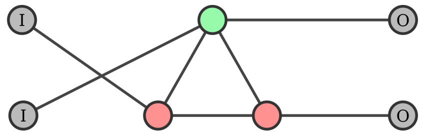
(a) The input circuit



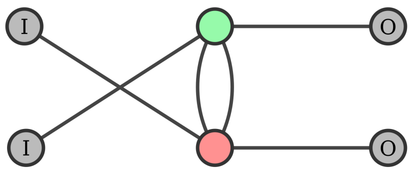
(b) The translated diagram



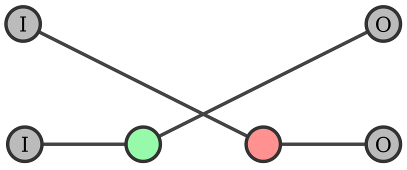
(c) Bialgebra Law applied



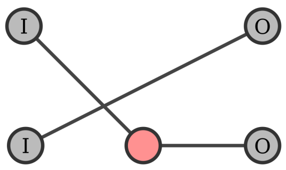
(d) Spider Rule 1 applied



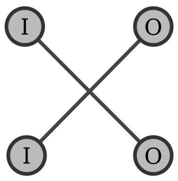
(e) Spider Rule 1 applied



(f) Hopf Law applied



(g) Spider Rule 2 applied



(h) The simplified diagram

9 Conclusion

As part of this project, a generalized processing and simplification framework for diagrams based on the ZX-Calculus or similar calculi was developed and implemented. It shows how a circuit represented using the OpenQASM[8] notation can be translated into a diagram. A solution for validating this transform as well as further rewrites on the resulting diagram was presented.

A matching and rewriting framework was proposed, that allows arbitrary, user-defined rulesets to be applied to a diagram, which enables further experimentation with different optimization strategies. The correctness of the rewriter was verified using the transform validation method as well as individual unit testing.

Finally, a simple optimization strategy using the matching and rewriting system was shown, that can successfully simplify diagrams while validating all performed transformations.

9.1 Further work

As discussed in Section 8.2, extraction has not been implemented yet and further research will be required on how to extract a circuit from a diagram in the general case.

The correctness of possible optimization using the matcher and rewriter was proven using a simple optimization strategy and simplifier rule hierarchy. Further work will be required on how to design such a strategy more intelligently. Other possible solutions include alternating between rules instead of just picking the most prioritized one and using the decision policies (see Section 7.4.2) as part of the strategy itself.

Additional work will also be required in developing a mechanism or heuristic for terminating the optimization procedure. So far it only stops when no rule can be matched, but this is, apart from trivial identity circuits, nearly never the case. The optimizer therefore needs to be able to decide when it has reached the end of the simplification process. It is then just repeating the same rules over and over and has to detect that it should terminate the optimization procedure.

Bibliography

[1] ANTLR. `https://www.antlr.org/`. Accessed: 2021-10-15.

[2] Cairo. `https://www.cairographics.org/documentation/`. Accessed: 2021-10-15.

[3] graph-tool. `https://graph-tool.skewed.de/`. Accessed: 2021-10-06.

[4] PyGTK. `https://gitlab.gnome.org/GNOME/pygobject`. Accessed: 2021-10-13.

[5] Miriam Backens, Hector Miller-Bakewell, Giovanni de Felice, Leo Lobski, and John van de Wetering. There and back again: A circuit extraction tale. *Quantum*, 5:421, Mar 2021.

[6] Bob Coecke and Ross Duncan. Interacting quantum observables: categorical algebra and diagrammatics. *New Journal of Physics*, 13(4):043016, Apr 2011.

[7] L.P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, 2004.

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

[9] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John van de Wetering. Graph-theoretic Simplification of Quantum Circuits with the ZX-calculus. *Quantum*, 4:279, June 2020.

[10] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[11] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–.

[12] Aleks Kissinger and John van de Wetering. PyZX: Large scale automated diagrammatic reasoning. *Electronic Proceedings in Theoretical Computer Science*, 318:229–241, May 2020.

[13] Aleks Kissinger and John van de Wetering. Reducing T-count with the ZX-calculus. *Physical Review A*, 102(2), Aug 2020.

[14] Aleks Kissinger and Vladimir Zamdzhiev. Quantomatic: A proof assistant for diagrammatic reasoning. *Lecture Notes in Computer Science*, page 326–336, 2015.

[15] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain

Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.

[16] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[17] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. Tensornetwork: A library for physics and machine learning, 2019.

[18] John van de Wetering. ZX-calculus for the working quantum computer scientist, 2020.

[19] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

Appendix - Software Documentation

The entire implementation can be found within the git repository:

`https://github.com/putterer/zx-optimization-framework`

It contains the data structures and algorithms described in this report as well as some test cases showcasing how it can be interfaced.

The provided code can be run using Python by installing the dependencies specified in the `requirements.txt` file. A system-wide installation of `graph-tool` and `PyGTK` is required as well. Additionally, the ANTLR based parser needs to be generated based on the provided grammar by executing the `compile.sh` script.

The entire code can be used interactively within an IPython[16] / Jupyter[15] environment. Alternatively, it can be run as a standalone application, therefore, creating a window using PyGTK[4]. This also allows interacting with generated diagrams.

1 Test Cases

All test cases are located within the top-level `test` directory.

1.1 Circuits

The circuit test case in `data_structures/circuit/circuit_test.py` showcases how individual circuits can be manually constructed according to Section 4.2. The code in `visualization/test_circuit_renderer.py` then shows how to render the given circuit as a visualization using a standalone window as described in Section 4.4. The example provided in `openqasm/test_open_qasm_parser.py` demonstrates how to read a circuit from an OpenQASM-based file, as in Section 4.2.1.

Some sample circuits are provided in the OpenQASM format in the top-level `circuits` directory.

1.2 Diagrams

`visualization/test_diagram_renderer.py` explains how a diagram can be manually created as specified in Section 4.3. Afterward, it uses the corresponding renderer to show a visualization of the created diagram. Instead of creating the diagram manually, it can also be automatically created from a given circuit based on Chapter 5 as shown in `translation/test_circuit_translator.py`.

1.3 Validation

The validation of circuits and diagram equality according to Chapter 6 is implemented and tested within the `validation` directory.

The `test_circuit_diagram_translation_equality.py` test case showcases how the circuit and diagrams can be converted to a linear matrix representation and how the equivalence of the circuit to diagram translation can be ensured.

1.4 Matching and Rewriting

The implementation supports matching and rewriting based on the rewrite rules and algorithms specified in Chapter 7. Apart from the Pi-Copy rule, all ZX-Calculus rules are implemented and can be used. They are all unit tested to ensure correct functionality in `rewriting/matcher_rewriter_test.py`. This can also be used as a reference on how to perform matching and rewriting using the implemented algorithms and calculus rules.

1.5 Entire Pipeline

Putting it all together, the goal of the project and implementation is quantum circuit optimization. The entire pipeline, reading in a circuit from OpenQASM, rendering it, translating it to a ZX-diagram, and optimizing that using the matcher and rewriter in combination with a ruleset can be seen in `optimization/test_optimization.py`. The resulting diagram is then visualized. As described in Chapter 8, circuit extraction is not yet implemented.