Quantum Science and Technology

PAPER • OPEN ACCESS

Quanto: optimizing quantum circuits with automatic generation of circuit identities

Quantum Science and Technology

OPEN ACCESS

RECEIVED
19 December 2023

REVISED
26 March 2024

ACCEPTED FOR PUBLICATION
24 June 2024

PUBLISHED
12 July 2024

PAPER

Quanto: optimizing quantum circuits with automatic generation of circuit identities

Jessica Pointing[1,2,*], Oded Padon[2,3], Zhihao Jia[2,4], Henry Ma[5], Auguste Hirth[5], Jens Palsberg[5] and Alex Aiken[2]

[1] Department of Physics, University of Oxford, Oxford OX1 3PU, United Kingdom
[2] Department of Computer Science, Stanford University, Stanford, CA 94305, United States of America
[3] VMware Research, Palo Alto, CA 94304, United States of America
[4] Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, United States of America
[5] Department of Computer Science, University of California, Los Angeles, CA 90095, United States of America
[*] Author to whom any correspondence should be addressed.

E-mail: contact@jessicapointing.com

Keywords: quantum compilation, quantum optimization, quantum computing

Abstract

Existing quantum compilers focus on mapping a logical quantum circuit to a quantum device and its native quantum gates. Only simple circuit identities are used to optimize the quantum circuit during the compilation process. This approach misses more complex circuit identities, which could be used to optimize the quantum circuit further. We propose Quanto, the first quantum optimizer that automatically generates circuit identities. Quanto takes as input a gate set and generates provably correct circuit identities for the gate set. Quanto's automatic generation of circuit identities includes single-qubit and two-qubit gates, which leads to a new database of circuit identities, some of which are novel to the best of our knowledge. In addition to the generation of new circuit identities, Quanto's optimizer applies such circuit identities to quantum circuits and finds optimized quantum circuits that have not been discovered by other quantum compilers, including IBM Qiskit and Cambridge Quantum Computing Tket. Quanto's database of circuit identities could be applied to improve existing quantum compilers and Quanto can be used to generate identity databases for new gate sets.

1. Introduction

Quantum programs are expressed as circuits of primitive gates; the specific *gate set* available varies depending on the quantum computer. Compilers are used to optimize the circuit in an effort to reduce the execution time, the noise that naturally arises in quantum computations [22], or both.

A quantum algorithm is implemented on a quantum computer with a quantum circuit, which consists of a sequence of quantum gates. A quantum gate applies a transformation to a qubit and can be represented as a matrix. Generally, the larger the circuit depth (length of the longest path from the input to the output), the noisier the computation, leading to more errors in the results. Because of the possibility of incorrect results and the inherent statistical nature of quantum computation, quantum programs must be executed many times to obtain an accurate picture of their results. Therefore, reducing the depth of a quantum circuit not only makes a quantum program faster, but also reduces the potential for noise and reduces the number of program evaluations required. Thus, finding equivalent but smaller depth circuits is a core problem in the implementation of quantum programs.

Quantum hardware imposes many constraints on quantum compilers, including the layout and connectivity of the qubits. Such aspects are addressed by the compilers built by the hardware vendor (such as IBM or Google). Less attention has been given to optimizing the logical circuit itself—finding circuit identities for a given gate set.

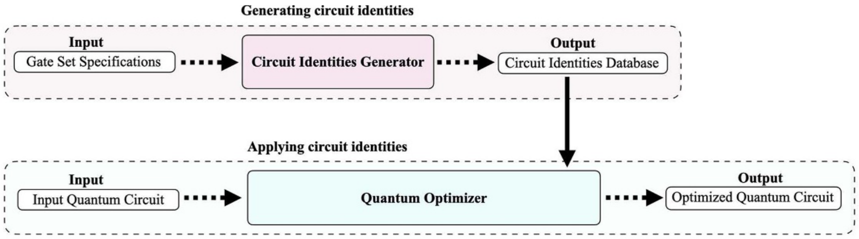

Generating circuit identities
Input
Gate Set Specifications
Circuit Identities Generator
Output
Circuit Identities Database
Applying circuit identities
Input
Input Quantum Circuit
Quantum Optimizer
Output
Optimized Quantum Circuit

Figure 1. Overview of Quanto (inspired by the architecture of TASO, shown in figure 1 in [17]).

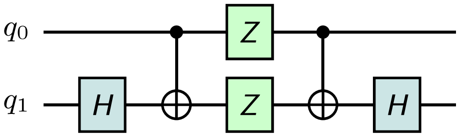


Figure 2. Example of a quantum circuit that can be optimized with Quanto.

Circuit identities have been found manually [15] and automatically for single-qubit gates [19], but as far as we know there is no system today that automatically finds identities for full gate sets and no compiler that optimizes quantum circuits based on such circuit identities.

We present Quanto, the first quantum circuit optimizer that *automatically generates* circuit identities. Figure 1 shows an overview of Quanto. In a first, offline phase, Quanto generates a database of circuit identities. The Quanto compiler takes a quantum circuit and identity database as inputs and uses a cost-based search algorithm to replace the quantum circuit with an optimized quantum circuit.

Figure 2 shows an example of a quantum circuit with two qubits and a depth of five. The circuit contains the two-qubit gate *CX* and the single-qubit gates *H* and *Z*. The single-qubit gates are represented by 2×2 unitary complex matrices and the two-qubit gates are represented by 4×4 unitary complex matrices. This quantum circuit is a *logical quantum circuit*, which is a quantum circuit defined in terms of unitary matrices and is agnostic to the quantum hardware.

If we wish to run this logical quantum circuit on the quantum hardware, we would have to compile the quantum circuit into the hardware's *native quantum gates*. These are the set of quantum gates that have been implemented on particular quantum hardware. For example, IBM's native quantum gate set was [21] {U1,U2,U3,CX} where

$$\begin{aligned}
U1\left(\lambda\right) &= \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix} \\
U2\left(\psi,\lambda\right) &= \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & -e^{i\lambda} \\ e^{i\psi} & e^{i(\lambda+\psi)} \end{pmatrix} \\
U3\left(\theta,\psi,\lambda\right) &= \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}\sin\left(\frac{\theta}{2}\right) \\ e^{i\psi}\sin\left(\frac{\theta}{2}\right) & e^{i(\lambda+\psi)}\cos\left(\frac{\theta}{2}\right) \end{pmatrix} \\
CX &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.
\end{aligned}$$

In this example, the *source gates* are the gates present in the logical quantum circuit {X, H, CZ} and the *target gates* are the native quantum gates {U1, U2, U3, CX}. Therefore, in order to run this logical quantum circuit on the quantum hardware, we must find circuit identities for the source gates in terms of the target gates. In this case $Z = U1(\pi)$ and $H = U2(\pi, 0)$.

Another step for running a logical quantum circuit on the hardware is to map the qubit and gate connections to those of the physical constraints of the hardware. Most compilers focus on these two aspects: (1) mapping source gates to target gates and (2) mapping the quantum circuit to the hardware constraints. Then, generally, a set of simple manual rules are used to optimize the logical quantum circuit itself. For example, voqc (Verified Optimizer for Quantum Circuits) [16] includes five rules for optimizations – (0) not

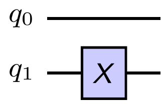


Figure 3. The optimal circuit of the quantum circuit in figure 2 for the given set.

propagation, (1) Hadamard reduction, (2) single-qubit gate cancellation, (3) two-qubit gate cancellation, and (4) rotation merging. The voqc optimizer applies these rules in the following order in order to optimize a circuit: 0, 1, 3, 2, 3, 1, 2, 4, 3, 2.

Applying a limited number of optimization rules limits the opportunity to find other optimizations. Instead of defining a set of optimization rules, our compiler is the first to automatically generate circuit identities. Quanto finds an optimized logical quantum circuit that is equivalent to the input logical quantum circuit.

The first phase in the compiler is generating *circuit identities*. Circuit identities are quantum circuits that are mathematically equivalent. A quantum circuit can be represented as a matrix and two quantum circuits are equivalent when their matrices are equal (for a certain floating point precision).

Quanto's circuit identity generator enumerates all possible quantum circuits over a given set of gates up to a fixed depth and number of qubits. In the example in figure 2, the circuit identity generator would find all possible circuits with two qubits and a depth of five containing the gates {X, H, CZ} (or another gate set specified). To efficiently find all such circuit identities, Quanto constructs a hash table where quantum circuits are stored based on the hash of their *unitary matrix*. The operation of a quantum gate can be represented using a unitary complex matrix. The operation of a quantum circuit can be represented by a unitary matrix obtained from the matrices of individual gates using matrix multiplication and tensor products. In the example in figure 2, the unitary matrix is $U = (I \otimes H) \cdot CX \cdot (Z \otimes Z) \cdot CX \cdot (I \otimes H)$.

The second phase is the optimizer. When given an input quantum circuit, our optimizer searches for quantum circuits equivalent to the input quantum circuit that minimize the cost function. In this paper we use circuit depth as the cost; that is, a circuit will be replaced by the lowest-depth equivalent circuit known. In general, however, the cost function can be chosen to optimize any quantifiable criteria of a circuit. For example, a cost function could include the total number of gates, total number of specific gates, such as T gates or two-qubit gates, and/or gate errors. Because we generate all possible circuits, Quanto returns the optimal quantum circuit with respect to the cost function. In the example in figure 2, our optimizer returns an equivalent quantum circuit shown in figure 3. Quanto has reduced the depth of the circuit from five to one. This circuit has the minimum depth possible for that given gate set.

The circuit identities are obtained by simulating the quantum circuits. One problem, however, is that simulating a quantum circuit is expensive if the quantum circuit has a large number of qubits and circuit depth. The promise of quantum computing is dependent on the fact that simulating quantum circuits is classically inefficient. Therefore, the larger the quantum circuit becomes (i.e. more qubits and greater depth), and the more gates that are added, the more configurations there are to generate, which eventually can become infeasible. In order to solve this issue in our optimizer, we introduce *tiles*. A *tile* is a subset of a quantum circuit with a specified length and width. A tile is the source quantum sub-circuit that is matched to equivalent quantum circuits. In the example in figure 2, if our tile width is equal to three and our tile length is equal to two, then the tiles in figure 4 would be generated.

Quanto searches for quantum circuits equivalent to the tiles in the circuit identities database. A cost-based search algorithm is used to apply the substitution, which is a quantum circuit in the circuit identities database that replaces a tile in the quantum circuit. A substitution is the target quantum circuit that defines an equivalent new quantum circuit to replace the matched quantum circuit (i.e. the tile). Figure 5 shows how the circuit gets optimized when it is split into tiles.

Our main contribution is presenting Quanto, which is the first quantum optimizer to automatically generate quantum circuit identities. Our automatic generator finds identities used by existing compilers, and in addition finds novel identities (to the best of our knowledge). Quanto can generate novel identities for any given gate set, which is especially useful for quantum hardware with native gate sets that are not standard (for example, the two-qubit iSwap gate is less common than the two-qubit CX gate). In our evaluation we demonstrate that the IBM Qiskit compiler [1] can not find these novel identities. Quanto also includes a tile-based method to optimize a large quantum circuit using the generated identities. We demonstrate that Quanto's optimizer finds equivalent circuits with smaller depth compared to IBM's Qiskit and Cambridge Quantum Computing's Tket [23] compilers.

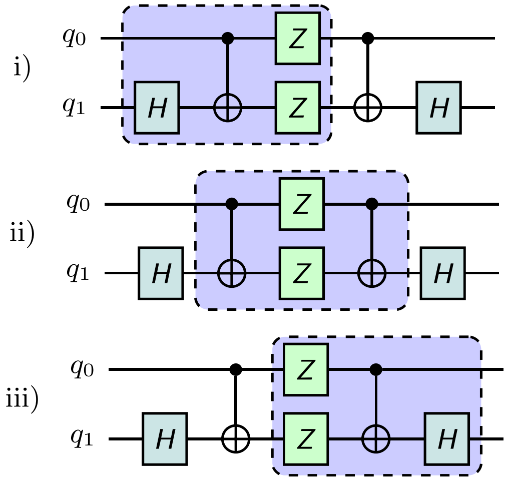


Figure 4. Splitting the circuit into tiles of width 3 and height 2.

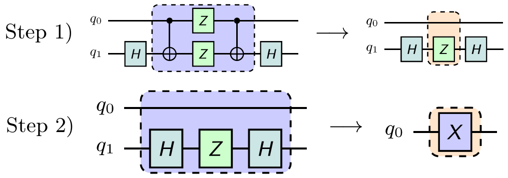


Figure 5. Steps to optimize the circuit when the circuit is split into tiles. The boxes shaded in blue are the tiles. The boxes shaded in orange are the substitutions.

2. Results

2.1. Generating existing circuit identities

Quanto can automatically discover identities that are known from previous work. [11, 15] We ran the compilers (Quanto, IBM's Qiskit, and Cambridge Quantum Computing's Tket) on a set of quantum circuits with known optimizations. The results show that Quanto is able to find the known optimizations for all of the input quantum circuits, whereas other compilers fail to find some of the identities. The other compilers could find simple identities, such as a circuit with the same gate in a row, but they failed to find more complicated identities.

These results are shown in tables 1–5. References [11, 15] give a list of equivalence rules, which lead to equivalent quantum circuits (i.e. circuit identities). The circuits that are shown in our results are chosen from [11, 15] because these are known circuit identities. The second column of the tables show the input (uncompiled) circuit and the other columns show the compiled circuit after running on the compiler specified in that column. The last columns in tables 1 and 4 show the circuits that are generated when the uncompiled circuits are run through the Quanto compiler and then the IBM compiler. The reason we show this column is because our optimizer is at the logical level and does not map the qubits to the hardware, a task we leave to the IBM compiler. We show that for the circuits in table 1 and some of the circuits in table 4 the circuits are unchanged by the mapping to the hardware. In particular, it is interesting to look at Circuit A and Circuit C in table 4. One might say that the IBM compiler circuit increased the depth of the uncompiled circuit from depth three to six in Circuit A and from depth three to seven in Circuit C because it needs to map to the hardware, but when we run our compiled circuit (which reduced the depth of the circuit to two in both circuits), the IBM compiler left the optimization unchanged. These results demonstrate that optimizing at the logical level is a useful task that can be done before mapping to the specific quantum hardware.

The Quanto compiled circuits of Circuits B and D in table 4 are changed when they are mapped to the hardware. In particular, two-qubit gates that are not applied to neighboring qubits are changed. In order to address this, we implemented a two-qubit gate neighboring constraint in Quanto that can be turned on or off. If this constraint is on, it only returns optimized circuits that include two-qubit gates on neighboring

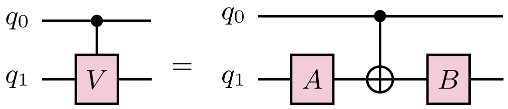


Figure 6. Known circuit identity from [8] where A and B are unitary if and only if V is of the form $\mathrm{V} = R_z(\alpha) \cdot R_y(\theta) \cdot R_z(\alpha) \cdot X$.

qubits. This is a simple hardware constraint. Quanto could be improved by implementing more specific hardware constraints. The last column in table 4 shows the compiled circuits when the constraint is on.

2.1.1. Example of known circuit identities

Known circuit identities are presented in the quantum computation literature, including [11, 15], which focus on circuit identities for specific quantum gates (e.g. the Hadamard gate), and [8], which presents circuit identities for any unitary gate or gates with a specific form (e.g. a gate which is composed of a sequence of rotation and other gates). Our results present known circuit identities for specific quantum gates. In this section, we want to show that some of these known circuit identities with specific gates are also equivalent to the generic circuit identities presented in [8]. One of the circuit identities presented in [8] is shown in figure 6. This two-qubit circuit identity is $CV = (I \otimes A) \cdot CX \cdot (I \otimes B)$ where CV is a controlled-V gate. A and B are unitary if and only if V is of the form $V = R_z(\alpha) \cdot R_y(\theta) \cdot R_z(\alpha) \cdot X = \begin{pmatrix} \sin(\theta/2) & e^{i\alpha}\cos(\theta/2) \\ e^{-i\alpha}\cos(\theta/2) & -\sin(\theta/2) \end{pmatrix}$ and where

$$R_y(\theta) = \begin{pmatrix} \cos(\theta/2) & \sin(\theta/2) \\ -\sin(\theta/2) & \cos(\theta/2) \end{pmatrix}$$

$$R_z(\alpha) = \begin{pmatrix} e^{i\alpha/2} & 0 \\ 0 & e^{-i\alpha/2} \end{pmatrix}.$$

Quanto is able to find this known circuit identity as shown in circuits A and B in table 5. For circuit A, the circuit identity is $CY = (I \otimes S^\dagger) \cdot CX \cdot (I \otimes S)$. Therefore, this is equivalent to the circuit identity from [8] as shown in figure 6 because $V = Y, A = S^\dagger, B = S$ where Y is the pauli-Y gate and has the form of V as:

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} = R_z(\pi/2) \cdot R_y(2\pi) \cdot R_z(\pi/2) \cdot X.$$

For circuit B, the circuit identity is $CZ = (I \otimes H) \cdot CX \cdot (I \otimes H)$. Therefore, this is equivalent to the circuit identity from [8] as shown in figure 6 because $V = Z, A = H, B = H$ where Z is the pauli-Z gate and has the form of V as:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} = R_z(0) \cdot R_y(\pi) \cdot Rz(0) \cdot X.$$

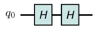
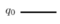
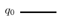
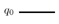
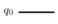
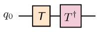
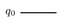
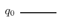
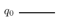
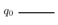
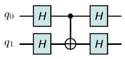
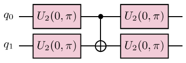
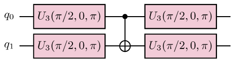
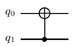
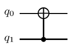
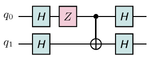
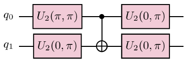
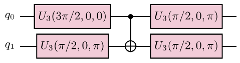
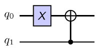
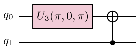
2.2. Generating novel circuit identities

Quanto can also generate new identities, especially when given custom gates, as it is more likely that identities for custom gates have not been previously discovered. Existing compilers have a fixed repertoire of identities using common gates. The advantage of Quanto is that it can find identities for any custom gate, leading to optimizations that may be missed by other compilers. In table 6 we show some of these novel identities (these identities were found from our identity database and we are not aware of any previous reports of these identities). We searched through our identity database to find circuit identities we believe have not been identified based on our review of the literature. We compare our novel identity optimizations to the IBM Qiskit compiler, as it allows us to specify basis gates for compiling. IBM's compiler is unable to find these circuit identities.

2.3. Optimizing quantum circuits

Other than generating existing and novel identities, Quanto can also optimize any quantum circuit as shown in table 1. For quantum circuits where we are able to produce tiles with the same dimension as the quantum circuit, our optimizer will always find the most optimal equivalent quantum circuit (given a particular gate

Table 1. Comparing Quanto to other compilers for optimizing quantum circuits with known optimizations. Some of these circuits are chosen from [15]. IBM and Tket backend is ibmq essex.

Circuit	Uncompiled circuit	IBM compiled circuit	Tket compiled circuit	Quanto compiled circuit	Quanto-IBM compiled circuit
A					
Depth	2	0	0	0	0
B					
Depth	2	0	0	0	0
C		 q_0 q_1 $U_2(0,\pi)$ $U_2(0,\pi)$ $U_2(0,\pi)$ $U_2(0,\pi)$	 q_0 q_1 $U_3(\pi/2,0,\pi)$ $U_3(\pi/2,0,\pi)$ $U_3(\pi/2,0,\pi)$ $U_3(\pi/2,0,\pi)$		
Depth	3	3	3	1	1
D		 q_0 q_1 $U_2(\pi,\pi)$ $U_2(0,\pi)$ $U_2(0,\pi)$ $U_2(0,\pi)$	 q_0 q_1 $U_3(3\pi/2,0,0)$ $U_3(\pi/2,0,\pi)$ $U_3(\pi/2,0,\pi)$ $U_3(\pi/2,0,\pi)$		
Depth	4	3	3	2	2

set). Therefore our optimizer will perform the same or better than other optimizers; there would be no instances where other optimizers outperform.

For quantum circuits where we can only produce tiles with dimensions smaller than the quantum circuit, our optimizer is not guaranteed to find the most optimal equivalent quantum circuit. Therefore, there may be some instances where the other optimizers outperform our optimizer.

2.4. Benefits of optimization

Our results show that Quanto can reduce the depth of quantum circuits. Two notable benefits of reducing the depth include (1) faster quantum program execution times, both for simulations and hardware and (2) noise reduction in the quantum program. In this section, we show why reducing the depth leads to these benefits and we show a practical example of the benefits on a quantum circuit.

2.4.1. Execution times

Each quantum gate takes a certain duration to execute both on quantum simulators and hardware. A quantum simulator executes a quantum gate by performing matrix multiplication. A quantum gate, which we denote as U, operating on n qubits has dimension $2^n \times 2^n$. The quantum circuit is a sequence of these unitary gates, so that the unitary matrix of the quantum circuit is equivalent to $U = U_{d-1} \cdot U_{d-2} \cdots U_0$ where d is the number of layers in the circuit (i.e. the depth).

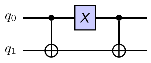
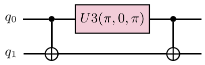
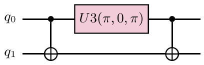
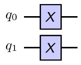
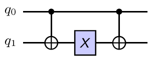
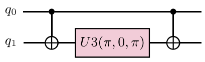
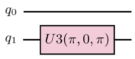
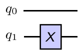
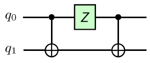
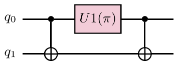
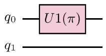
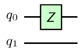
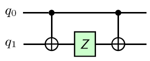
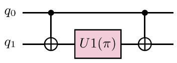
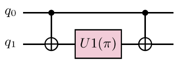
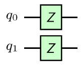
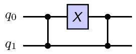
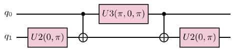
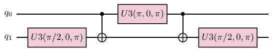
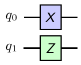
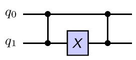
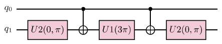
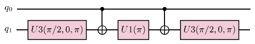
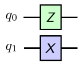
Therefore, the number of $2^n \times 2^n$ matrix multiplications in a quantum circuit simulation is equivalent to d. (There are other methods for simulating quantum circuits that may be faster—for example by taking advantage of Clifford gates as described in [9], but we are using the default method for our analysis). If our quantum circuit optimization algorithm reduces the depth of a quantum circuit by k, the number of $2^n \times 2^n$ matrix multiplications in a quantum circuit simulation is now equivalent to $d - k$. If each matrix multiplication takes time t_s, the execution time of the quantum program has been reduced from $d \cdot t_s$ to $(d - k) \cdot t_s$. Table 7 shows an example of this reduction in execution time in practice.

For quantum hardware, each quantum gate also has an execution time t_h, which depends on the specific properties of the hardware. For example, the ECR gate, which is a two-qubit gate, on the IBM Kyoto device has a gate time of 660 nanoseconds [4]. Reducing the depth of the input quantum circuit also reduces the execution time of the quantum program from $d \cdot t_h$ to $(d - k) \cdot t_h$.

2.4.2. Noise reduction

Another benefit of reducing the depth is reducing the noise for the quantum program. For our analysis, we use a simple noise model to show the effect of our optimizer on noise. We also show an example of this reduction in noise in practice in table 7.

Table 2. Comparing Quanto to other compilers for optimizing quantum circuits with known optimizations. IBM and Tket backend is Aerbackend.

Circuit	Uncompiled circuit	IBM compiled circuit	Tket compiled circuit	Quanto compiled circuit
A				
Depth	3	3	3	1
B				
Depth	3	3	1	1
C				
Depth	3	3	1	1
D				
Depth	3	3	3	1
E		 q_0 q_1 $U2(0,\pi)$ $U3(\pi,0,\pi)$ $U2(0,\pi)$	 q_0 q_1 $U3(\pi/2,0,\pi)$ $U3(\pi,0,\pi)$ $U3(\pi/2,0,\pi)$	
Depth	3	5	5	1
F			 q_0 q_1 $U3(\pi/2,0,\pi)$ $U1(\pi)$ $U3(\pi/2,0,\pi)$	
Depth	3	5	5	1

Each quantum gate implemented on quantum hardware has an error rate e. The error rate is different depending on the quantum gate, but we can take the average and state that the average gate error is $\tilde{e}$. If a quantum program consists of x gates, then the probability of no error in our quantum program is $P_s = (1-\tilde{e})^x$. If an optimizer reduces the number of gates by k, then the probability of no error in the quantum program becomes $P_s = (1-\tilde{e})^{(x-k)}$ [18].

We can use this simple noise model to estimate the error rates for a quantum program on a particular quantum device. In this example, we are going to use the quantum program in table 8. There are four Hadamard gates and one CNOT gate, which the IBM compiler compiles to eight R_z gates, four $\sqrt{X}$ gates, and one CNOT gate. At a particular snapshot in time, the gate error rates for the IBM Vigo device are shown in table 9.

Using the simple noise model and the error rates in table 9, we can calculate that the probability of no error for the circuit compiled by IBM's compiler is $P_{s_{\text{IBM}}} = (1-e_0^{R_z})^4 \cdot (1-e_1^{R_z})^4 \cdot (1-e_0^{\sqrt{X}})^2 \cdot (1-e_1^{\sqrt{X}})^2 \cdot (1-e^{\text{CNOT}_{0,1}}) = (1-4.1352\cdot 10^{-4})^2 \cdot (1-5.0203\cdot 10^{-4}) \cdot (1-1.2012\cdot 10^{-2}) = 98.668\%$ where e_q^g is the error rate for the gate g on the qubit(s) q.

Quanto, however, compiles this circuit to just one CNOT gate. We can estimate that the probability of no error is $P_{s_{\text{Quanto}}} = (1-e^{\text{CNOT}_{0,1}}) = (1-1.2012\cdot 10^{-2}) = 98.799\%$. Using this simple noise model, Quanto has reduced the error by 0.131%. This simple noise model, however, underestimates the true amount of noise and in practice Quanto can provide a greater noise reduction.

Table 3. Comparing Quanto to other compilers for optimizing quantum circuits with known optimizations. IBM and Tket backend is Aerbackend.

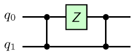
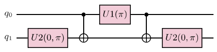
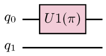
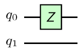
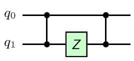
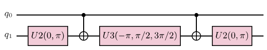
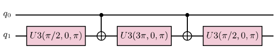
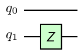
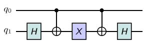
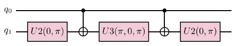
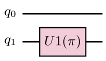
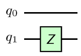
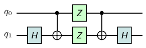
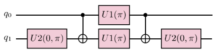
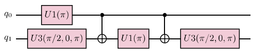
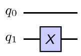
Circuit	Uncompiled circuit	IBM compiled circuit	Tket compiled circuit	Quanto compiled circuit
A				
Depth	3	3	1	1
B		 q_0 q_1 $U2(0,\pi)$ $U3(-\pi,\pi/2,3\pi/2)$ $U2(0,\pi)$	 q_0 q_1 $U3(\pi/2,0,\pi)$ $U3(3\pi,0,\pi)$ $U3(\pi/2,0,\pi)$	
Depth	3	5	5	1
C		 q_0 q_1 $U2(0,\pi)$ $U3(\pi,0,\pi)$ $U2(0,\pi)$		
Depth	5	5	1	1
D		 q_0 q_1 $U2(0,\pi)$ $U1(\pi)$ $U1(\pi)$ $U2(0,\pi)$	 q_0 q_1 $U1(\pi)$ $U3(\pi/2,0,\pi)$ $U1(\pi)$ $U3(\pi/2,0,\pi)$	
Depth	5	5	5	1

Table 4. Comparing Quanto to other compilers for optimizing quantum circuits with known optimizations. Some of these circuits are chosen from [15]. IBM backend is ibmq lima and Tket backend is Aerbackend.

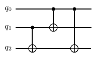
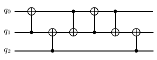
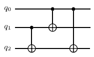
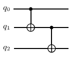
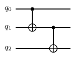
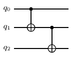
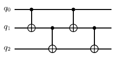
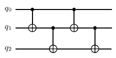
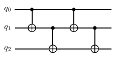
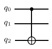
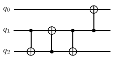
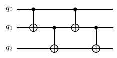
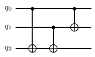
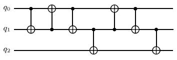
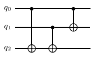
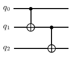
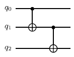
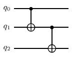
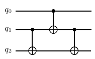
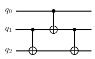
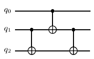
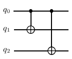
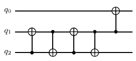
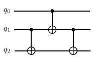
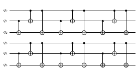
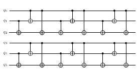
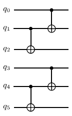
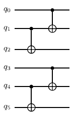
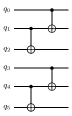
Circuit	Uncompiled circuit	IBM compiled circuit	Tket compiled circuit	Quanto compiled circuit	Quanto-IBM compiled circuit	Quanto compiled circuit (eith constraint)
A						
Depth	3	6	3	2	2	2
B						
Depth	4	4	4	1	4	4
C						
Depth	3	7	3	2	2	2
D						
Depth	3	3	3	2	5	3
E		Circuit too large for displaying image				
Depth	9	31	9	2	2	2

Table 5. Comparing Quanto to other compilers for optimizing quantum circuits for unique gate sets. Some of these circuits are chosen from [11].

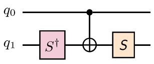
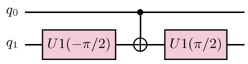
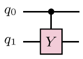
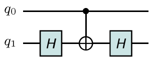
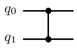
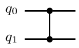
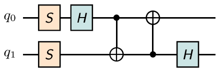
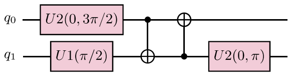
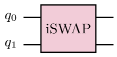
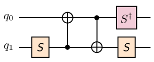
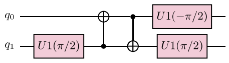
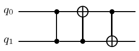
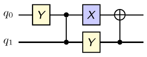
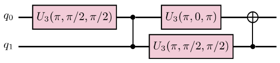
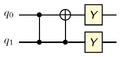
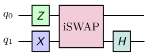
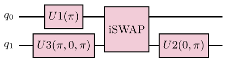
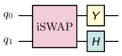
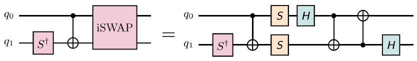
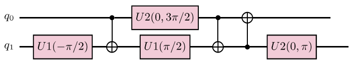
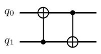
Circuit	Uncompiled circuit	IBM compiled circuit	Quanto compiled circuit
A			
Depth	3	3	1
B			
Depth	3	1	1
C		 q_0 q_1 $U2(0,3\pi/2)$ $U1(\pi/2)$ $U2(0,\pi)$	
Depth	5	4	1

Table 6. Comparing Quanto to other compilers for optimizing quantum circuits with unknown optimizations.

Circuit	Uncompiled circuit	IBM compiled circuit	Quanto compiled circuit
A		 q_0 q_1 $U1(\pi/2)$ $U1(-\pi/2)$ $U1(\pi/2)$	
Depth	4	4	3
B		 q_0 q_1 $U_3(\pi,\pi/2,\pi/2)$ $U_3(\pi,0,\pi)$ $U_3(\pi,\pi/2,\pi/2)$	
Depth	4	4	3
C		 q_0 q_1 $U1(\pi)$ $U3(\pi,0,\pi)$ iSWAP $U2(0,\pi)$	
Depth	3	3	2
D	 q_0 q_1 $S^\dagger$ iSWAP = q_0 q_1 $S^\dagger$ S H S H	 q_0 q_1 $U1(-\pi/2)$ $U2(0,3\pi/2)$ $U1(\pi/2)$ $U2(0,\pi)$	
Depth	7	6	2

We can perform a noisy simulation to see the noise reduction for this circuit in practice. To perform noisy simulations of quantum circuits, we can use the IBM Qiskit Aer device noise model [3], which generates a simplified noise model for a real device, by using the calibration information for a device, taking into account the gate error, time to execute the gate, relaxation times of the qubits, and readout errors.

To obtain the probability of error in the noisy simulation, we execute the circuit 10^6 times and calculate the probability of obtaining the correct output.

Using our simulation on the IBM FakeVigo device (a noisy simulation of the real IBM Vigo device), as shown in table 7, the IBM-compiled circuit has $P_{s_{\mathrm{IBM}}} = 89.85\%$ and $P_{s_{\mathrm{Quanto}}} = 91.45\%$. In this case, Quanto has reduced the error by 1.6%.

The noise in the experiment is greater than our simple noise model because our simple noise model only takes into account the gate error, whereas there are other sources of noise in practice such as the decoherence time of the qubits—the longer the execution of the quantum program (which depends on the depth), the more the qubits decohere and the more noise in the quantum program.

Table 7. Comparing the compilation and execution time of the circuit compiled by Quanto versus by IBM.

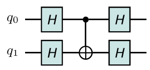
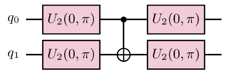
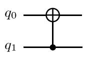
Circuit	Number of qubits	Uncompiled circuit	IBM compiled circuit	Quanto compiled circuit
A	2		 q_0 q_1 $U_2(0,\pi)$ $U_2(0,\pi)$ $U_2(0,\pi)$ $U_2(0,\pi)$	
Depth	-	3	3	1
Compilation Time (seconds)	-	-	0.3017	0.0140
Execution Time (seconds)	-	-	0.0120	0.0054

Table 8. Comparing the noise of the circuit compiled by Quanto versus compiled by IBM, using a noisy simulation of the IBM FakeVigo device.

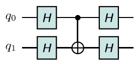
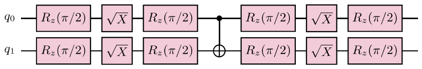
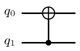
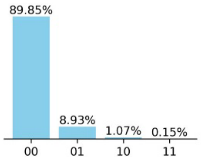
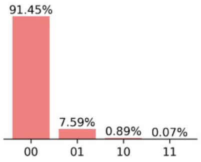
Circuit	Number of qubits	Uncompiled circuit	IBM compiled circuit	Quanto compiled circuit
A	2		 q_0 q_1 $R_z(\pi/2)$ $\sqrt{X}$ $R_z(\pi/2)$ $R_z(\pi/2)$ $\sqrt{X}$ $R_z(\pi/2)$	
Depth	-	3	7	1
Counts		-		

Table 9. Gate error rates for the IBM Vigo device.

Gate	Qubit(s)	Error rate
CNOT	[0,1]	$1.2012 \cdot 10^{-2}$
$\sqrt{X}$	0	$4.1352 \cdot 10^{-4}$
$\sqrt{X}$	1	$5.0203 \cdot 10^{-4}$
R_z	0	0
R_z	1	0

The magnitude of the noise reduction depends on the properties of the device. For example, the IBM London device has a different set of basis gates and therefore the IBM compiler compiles the circuit to a different set of gates (as shown in table 7). This compiled circuit has a shorter depth compared to the compiled circuit for the IBM Vigo device and therefore we can expect the noise reduction to be less significant for the Quanto-compiled circuit. An experiment on this device demonstrates $P_{s_{\text{IBM}}} = 96.19\%$ and $P_{s_{\text{Quanto}}} = 96.59\%$ leading to a noise reduction of 0.4%.

3. Discussion

3.1. Related work

3.1.1. Automatically generating identities

To the best of our knowledge, previous work on automatically discovering quantum circuit identities is limited to single-qubit gates [19]. In addition, these automatically generated identities have not been applied to optimizing quantum programs.

3.1.2. Superoptimization

Superoptimization is a compiler technique that was originally designed to find the optimal machine code for a sequence of instructions [6, 7, 20]. Our approach is inspired by TASO [17]. TASO applies superoptimization to deep learning by optimizing tensor computation graphs. For a given set of tensor operators (e.g. addition, matrix convolution) TASO automatically finds identities and uses these as substitutions to optimize computation graphs of deep learning applications. There are some differences: In the TASO paper, they generate a graph and then go through the list of substitutions and try to see whether the substitution is a sub-graph in the original graph. They have a limited number of substitutions, in the thousands, so they can do that fairly efficiently. In our case, however, we have can have hundreds of thousands or millions of substitutions and the time complexity of that technique would grow large. Instead we split our circuit into sub-circuits, tiles, and then look for the hash of that tile in the hash table, which is a constant lookup. Therefore, our technique is more efficient for our case and only depends on the number of tiles in the circuit, which grows modestly.

3.1.3. Quantum optimization

There are other compilers, [1, 16, 23] but their focus is mapping the logical circuit to the quantum device. Therefore, they do not pay much attention to the potential logical transformations. This can be shown in our results—other optimizers fail to recognize certain circuit identities. These optimizers have manual rules for logical substitutions and therefore are not finding the full range of optimization.

3.2. Limitations and future work

3.2.1. Scalability

One limitation of Quanto is the scalability of generating a large number of circuit identities, which limits the size (depth and number of qubits) of the circuit identities. This issue could be mitigated by applying pruning techniques during the generation of substitutions to remove redundant substitutions or by using heuristics to produce substitutions.

3.2.2. Missed substitutions and larger circuits

Another limitation is missing out on potential substitutions, due to invalid tiles in the quantum circuit as shown in figure 13. In addition, substitutions between distant pairs of qubits could be missed if the tiles only consider its nearest neighbor qubits. This could be mitigated by reordering the qubits (as we describe in the Methods section) at the cost of increasing the time complexity of the algorithm. Due to these limitations, Quanto's performance on larger quantum circuits is not as effective. We believe that further work to reduce these limitations could improve the effectiveness of Quanto when applied to larger quantum circuits.

3.2.3. Parameterized gates

To use gates with parameters in our optimizer, we can set the parameter to a small set of specific values when generating identities. Since many parameters are used for qubit rotations, and certain angles of rotation are very common in quantum algorithms, this approach gives some reasonable coverage of potential optimizations involving parameters. However, our optimizer would be improved if it included symbolic identities, where substitutions for parametrized gates are generated regardless of the value of the parameter.

3.2.4. Noise effects

It could also be interesting to do an in-depth analysis of the effects on noise in quantum hardware after using the methods presented here, thereby providing further insight into how these methods could be used as an error-mitigation technique.

3.3. Conclusion

Quanto is the first quantum optimizer that automatically generates general circuit identities. Quanto finds novel optimizations and finds better optimizations for quantum circuits, compared to other compilers. This work contributes to reducing the circuit depth of quantum circuits, which could lead to less noise and more accurate execution of quantum algorithms.

4. Methods

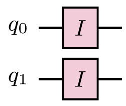
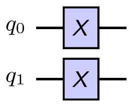
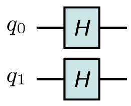
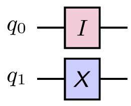
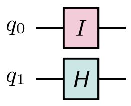
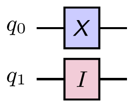
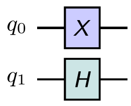
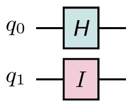
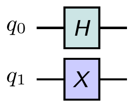
4.1. Circuit identity generator

This section describes the Quanto circuit identity generator, which automatically generates circuit identities given a gate set, as well as the number of qubits and depth bounds. This generation algorithm finds all circuit identities given the number of qubits, circuit depth, and gate set.

Table 10. Inputs of the generation algorithm.

Symbol	Definition	Description
n	number of qubits	maximum number of qubits of the circuit identities generated
d	depth of the circuit	the maximum depth of the circuit identities generated
dp	decimal point accuracy	the decimal points of the floating-point numbers in the matrices of the gates
gs	gate set	a list of gates to use in the circuit identities, including the identity gate

Table 11. Possible circuits for single-qubit gates given gate set $[I, X, H]$.

		
$[I,\ I]$	$[X,\ X]$	$[H,\ H]$
		
$[I,\ X]$	$[I,\ H]$	$[X,\ I]$
		
$[X,\ H]$	$[H,\ I]$	$[H,\ X]$

4.1.1. Generation algorithm

Table 10 shows the inputs to the generation algorithm.

Step 1: represent the quantum circuit as a grid. We represent a quantum circuit as a grid, where the number of rows m is the number of layers in the quantum circuit, also known as the circuit depth. The number of columns n is the number of qubits in the quantum circuit. If there is no gate present at a particular row for a particular column, we place the Identity gate in that position. Figure 7 shows this quantum circuit representation.

Step 2: for a single layer (depth = 1), generate all the possible circuits of quantum circuits using the gate set.

a) Single-qubit gates only

If there are only single-qubit gates in the gate set, then each gate is applied to a single qubit. If n is the number of qubits and g is the number of single-qubit gates, then there are g^n possible circuits.

Example: gate set $= [I,\ X,\ H]$ and $n = 2$ where

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, H = \begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}.$$

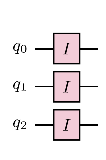
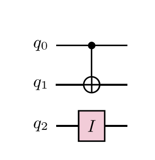
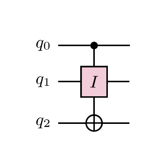
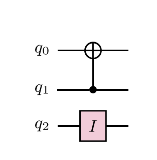
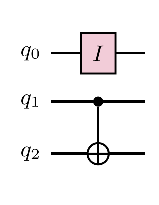
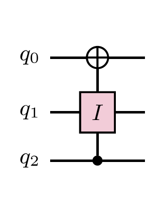
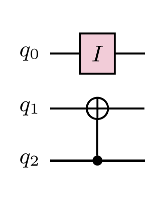
Table 11 shows all the possible circuits that would be generated for the gate set $[I, X, H]$

In the case of the possible circuit $[H, X]$ the H gate is applied to the first qubit and the X gate is applied to the second qubit as shown in table 11.

b) Two-qubit and single-qubit gates

If there are two-qubit gates, then we split the two-qubit gate into the first and second qubits it operates on.

Table 12. Possible circuits for two-qubit gates given gate set $[I, CX]$.

			
$[I,\ I,\ I]$	$[CXC,\ CXT,\ I]$	$[CXC,\ I,\ CXT]$	$[CXT,\ CXC,\ I]$
			
$[I,\ CXC,\ CXT]$	$[CXT,\ I,\ CXC]$	$[I,\ CXT,\ CXC]$	

Example: gate set = $[I,\ CX]$ and $n = 3$ where

$$CX = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

In the notation below, *CXC* denotes applying the *CX* gate to the *control* qubit and *CXT* denotes applying the *CX* gate to the *target* qubit.

Table 12 shows all the possible circuits that would be generated for the gate set $[I, CX]$. In the case of the possible circuit $[CXC, CXT, I]$ the *CX* gate is applied to the first and second qubit where the first qubit is the *control* qubit and the second qubit is the *target* qubit. The *I* gate is applied to the third qubit.

Verifying the validity of the two-qubit gates

CXC and *CXT* must both be present in the possible circuits; it is not possible to have one present without the other. For example, $[CXC, I, I]$ would be an invalid configuration. Quanto verifies that *CXC* and *CXT* are present in each possible circuit.

Step 3: for multiple layers (depth > 1), generate all the possible circuits of the quantum circuits using the gate set. For an additional layer in the circuit, we add all the possible circuits for a single layer onto the previous layer.

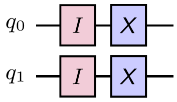
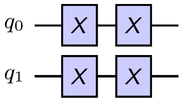
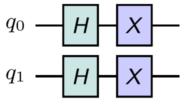
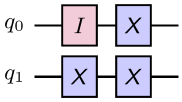
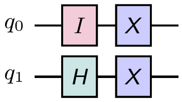
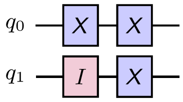
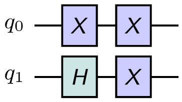
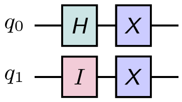
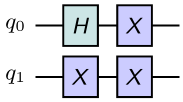
Example: for the gate set presented in table 11, we take each possible circuit and append it to the end of each possible circuit. Table 13 illustrates appending the possible circuit $[X, X]$ to each possible circuit. This is repeated for $[H, H]$, $[I, X]$, *etc*...

Step 4: calculate the unitary matrices and generate their fingerprints. For each possible circuit, we find the *unitary matrix* of the possible circuit. We adopt an idea from compiler superoptimization [6] and compute a *fingerprint* for each possible circuit, which is a hash of the unitary matrix. Figure 8 shows a possible circuit and how to calculate the unitary matrix. The matrices in part (c) correspond to the gates in part (b) and (a) of figure 8.

Step 5: create hash table of fingerprints. The hash table of fingerprints includes the quantum circuit as the key and the fingerprint of the quantum circuit as the value. This allows us to look up a quantum circuit's fingerprint in constant time during the optimizer stage, without having to recalculate the unitary matrix and the fingerprint of the quantum circuit. Figure 9 shows the sequence of steps to generate a hash table of fingerprints for each possible circuit. Table 14 is an example hash table with one entry which shows the quantum circuit representation as the key and its corresponding fingerprint as the value.

Step 6: create hash table of circuit identities. The hash table of substitutions includes the fingerprints as keys and an array of matching circuit identities as values. For each possible circuit we generate, we check if its fingerprint is in the hash table of circuit identities. If it is, we add the possible circuit to the matching fingerprint. If it is not, we add the fingerprint to the hash table and add the possible circuit to the array of

Table 13. Possible circuits for multiple layers.

		
$[I, I], [X, X]$	$[X, X], [X, X]$	$[H, H], [X, X]$
		
$[I, X], [X, X]$	$[I, H], [X, X]$	$[X, I], [X, X]$
		
$[X, H], [X, X]$	$[H, I], [X, X]$	$[H, X], [X, X]$

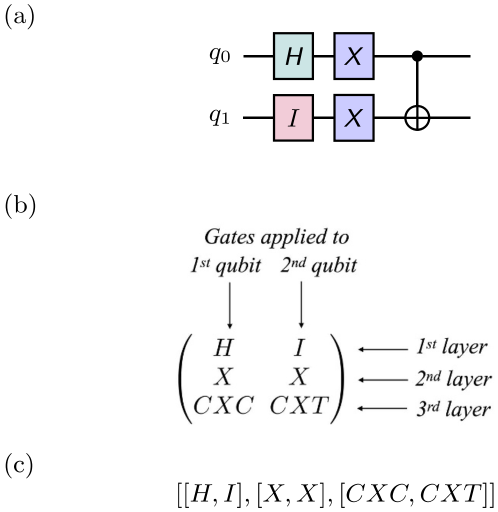

(a)
q_0
q_1
H
X
I
X
(b)
Gates applied to
1st qubit
2nd qubit
$\begin{pmatrix} H & I \\ X & X \\ CXC & CXT \end{pmatrix}$
1st layer
2nd layer
3rd layer
(c)
$[[H, I], [X, X], [CXC, CXT]]$

Figure 7. (a) Quantum circuit which can be represented as (b) a matrix, which can be represented computationally as (c).

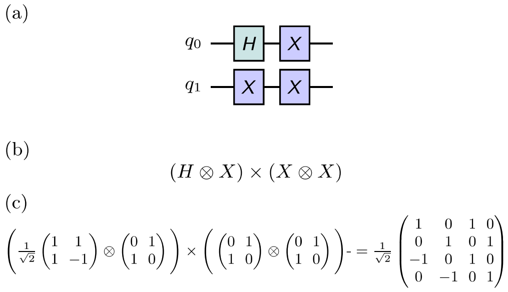

(a)
q_0
q_1
H
X
X
X
(b)
$(H \otimes X) \times (X \otimes X)$
(c)
$\left(\frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\right) \times \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}\right) - = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \end{pmatrix}$

Figure 8. (a) A possible circuit generated from a gate set (b) the gate representation of the circuit (c) the matrix calculation of the circuit to obtain the unitary matrix.

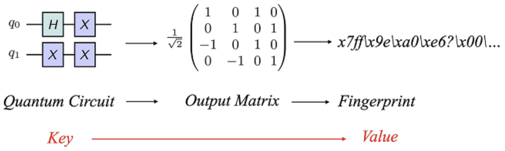

q_0
q_1
H
X
X
X
$\frac{1}{\sqrt{2}}\begin{pmatrix}1&0&1&0\\0&1&0&1\\-1&0&1&0\\0&-1&0&1\end{pmatrix}$
x7ff\x9e\xa0\xe6?\x00\...
Quantum Circuit
Output Matrix
Fingerprint
Key
Value

Figure 9. A quantum circuit can be represented by an unitary matrix. The hash of this unitary matrix is its fingerprint.

Table 14. Example of a hash table with one entry demonstrating the quantum circuit representation as the key and the fingerprint as the value. Note that the fingerprint in this table is only for illustrative purposes (as the actual fingerprint is too long to be displayed).

Key	Value
$[[H,X],[X,X]]$	$x7ffx9exa0xe6?x00\ldots$

Table 15. Hash table with one entry demonstrating the fingerprint as the key and the array of substitutions as the value.

Key	Value
$x7ffx9exa0xe6?x00\ldots$	$[[X,X],[X,X]]$ $[[H,H],[H,H]]$

Table 16. Symbols for scaling formula.

Symbol	Definition
S	total number of possible circuits
S_l	number of possible circuits per depth
g	number of single-qubit gates
n	number of qubits
t	number of two-qubit gates
S	number of possible circuits for the depth
d	depth

matching circuit identities. The identities are stored implicitly in the hashtable—any set of matrices associated with the same key are equal and can be substituted for each other. The identities are up to some (configurable) floating point precision. Table 15 is an example hash table with one entry which shows the fingerprint as the key and all the circuit identities that match to the fingerprint as the value.

4.2. Scaling

To estimate the computational resources to generate circuit identities, we derived a formula for the total number of possible circuits to be generated. The symbols are defined in table 16.

$$S_l = \sum_{r=0}^{\lfloor n/2 \rfloor} \frac{n!}{r!\,(n-2r)!} g^{n-2r} t^r \tag{1}$$

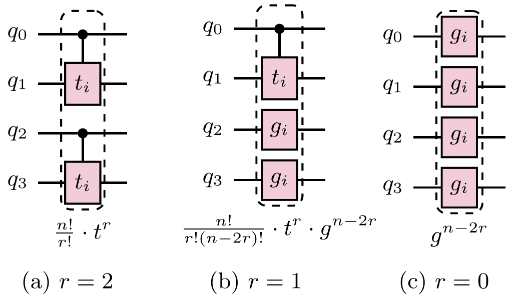
$$S = S_l^d. \tag{2}$$

As one can see, the number of possible circuits scales quickly with n which is to be expected. We always use the optimizer, and if the size of the tile is equivalent to the size of the input circuit, there is only one tile, meaning we enumerated this circuit during generation.

We derived the scaling formula by calculating the permutations of two-qubit and single-qubit gates in a quantum circuit. In particular, we can fix the number of two-qubit gates in a single layer and calculate the permutations of two-qubit and single-qubit gates in that layer. The formula is then derived by calculating the summation of the permutations across all possible number of two-qubit gates in a single layer. The maximum number of two-qubit gates in a single layer of a n-qubit circuit is $\lfloor n/2 \rfloor$. The number of two-qubit gates in a single layer of a n-qubit circuit is r.

If $r = \lfloor n/2 \rfloor$ then there are $\frac{n!}{r!}$ ways to arrange the two-qubit gates in a single layer. For example, for $n = 4$, the maximum number of two-qubit gates in a single layer is $\lfloor 4/2 \rfloor = 2$ as shown in figure 10(a) and there are $\frac{4!}{2!} = 12$ ways to arrange those two two-qubit gates.

To account for multiple types of two-qubit gates, we multiply $\frac{n!}{r!}$ by t^r where t is the number of types of two-qubit gates. For example, if our quantum circuit contains both CNOT and CZ gates, $t = 2$ and the



(a) $r = 2$ (b) $r = 1$ (c) $r = 0$

Figure 10. Permutations of single-qubit and two-qubit gates in $n = 4$ circuit.

Table 17. Possible circuits for multiple layers.

Symbol	Definition
quantum circuit	quantum circuit to optimize
i	the maximum number of qubits of the tiles
j	the maximum depth of the tiles

number of ways to arrange two two-qubit gates in a single layer from the set of CNOT and CZ becomes $\frac{4!}{2!} \cdot 2^2$. For this example, where we have CNOT and CZ gates, $t_i \in [X, Z]$ for the gates in figure 10(a).

If $r = 0$, this means the number of two-qubit gates in a single layer is zero and therefore the layer only contains single-qubit gates, which can include the identity gate. If g is the number of types of single-qubit gates, then g^n is the number of possible ways to arrange n gates in a single layer from a set of g types of single-qubit gates. For example, for $n = 4$, and $g = 3$, where the set of single-qubit gates is $g_i \in [I, X, H]$, there are 3^4 possible ways as shown in figure 10(c).

If $r = 1$, then there is one two-qubit gate and two single-qubit gates in a circuit where $n = 4$ as shown in figure 10(b). We now have two positions to place the two-qubit gate, so the number of possible ways to arrange them is $\frac{n!}{r!(n-2r)!}$ and we multiply by t^r to account for multiple types of two-qubit gates. We also have two positions to place the two single-qubit gates, so the number of possible ways to arrange them is $\frac{g^{n-2r}}{r!}$.

The scaling formula is the sum of permutations over all possible r's.

4.3. Optimizer

This section describes the Quanto optimizer, which applies the automatically generated circuit identities to optimize a quantum circuit.

4.3.1. Algorithm

Table 17 shows the inputs.

Step 1: generate tiles of the input quantum circuit. If the input circuit, which has depth (or width) m and number of qubits (or length) n, is larger than the size of the circuit identities generated, we split a circuit into sub-circuits, called *tiles*. The tile width, j, is the maximum depth of the circuit identities and the tile length, i, is the maximum number of qubits of the circuit identities. We iterate through the rows and columns in the quantum circuit matrix and generate the indices of all possible tiles. The number of possible $i \times j$ tiles for an $n \times m$ circuit is $(n - i + 1) \cdot (m - j + 1)$.

Figure 11 shows the dimensions of the tiles and quantum circuit. Figure 12 shows all the possible tiles generated if we chose a tile with $j = 2$ and $i = 2$ in a quantum circuit with $m = 3$ and $n = 2$. Using the formula for the number of possible tiles, we see that there are four possible tiles, which corresponds with figure 12.

Step 2: check for valid tiles. Not all tiles generated from the circuit will be valid. If a two-qubit gate is not placed at the boundaries of the tile, then the tile is invalid if the two-qubit gate is cut off by the tile. In this case, we discard such tiles. We can keep a tile if the two-qubit gate is placed at the boundary of the tile even it has been cut off by the tile. Figure 13 shows an example of an invalid tile, whereas figure 14 shows an example of a valid tile.

Step 3: search for tile in hash table of circuit identities. For each tile we have generated, we first search in the hash table of fingerprints. If a tile contains a two-qubit gate that has been cut off *only at the boundary* of the

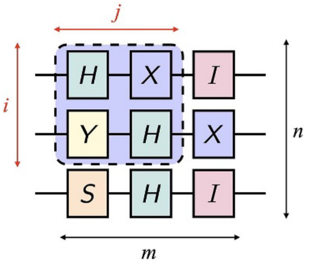


Figure 11. Dimensions of a tile compared to the dimensions of a quantum circuit.

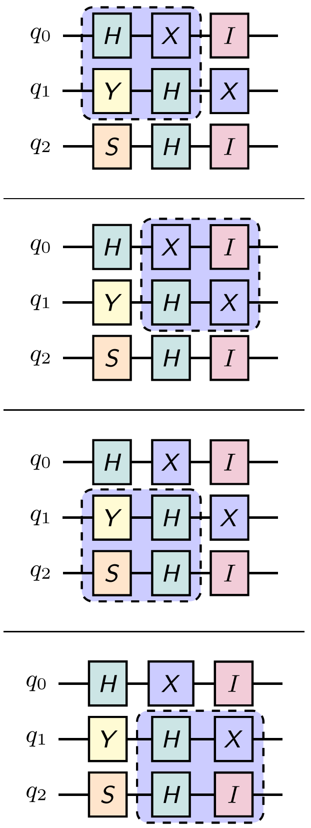


Figure 12. All the possible tiles (shaded in blue) generated for a tile width and length = 2 for a quantum circuit with width and length = 3.

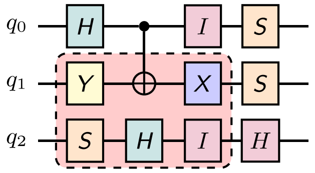


Figure 13. An example of an invalid tile (shaded in red), because the *CX* gate is cut off and is not first or last on its wire.

tile, then we first have to replace the two-qubit gate with an Identity gate in order to make the tile valid and searchable in the hash table of circuit identities. Part (a) in figure 15 shows this transformation. The now-valid tile is represented as a matrix, as shown in Part (b) in figure 15 and we can search for this matrix in the hash table of fingerprints. This gives us the fingerprint of the tile, without needing to calculate the unitary

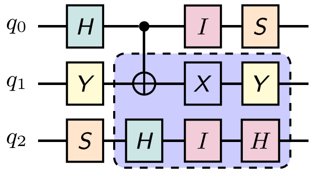
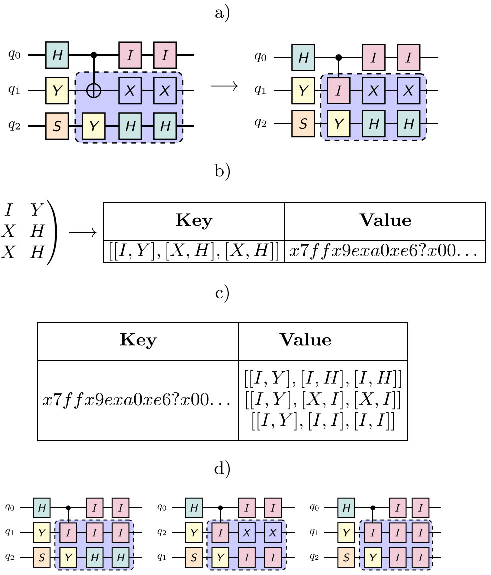


Figure 14. An example of a valid tile (shaded in blue), because even though the *CX* gate is cut off, it is first or last on its wire.



b)

$$\begin{pmatrix} I & Y \\ X & H \\ X & H \end{pmatrix} \longrightarrow$$

Key	Value
$[[I, Y], [X, H], [X, H]]$	$x7ffx9exa0xe6?x00...$

c)

Key	Value
$x7ffx9exa0xe6?x00...$	$[[I, Y], [I, H], [I, H]]$ $[[I, Y], [X, I], [X, I]]$ $[[I, Y], [I, I], [I, I]]$

d)

Figure 15. (a) The invalid *CX* gate in the tile is replaced with an Identity gate to make it a valid tile (b) the matrix representation of the tile is searched for in the hash table of fingerprints, which returns its fingerprint (c) the fingerprint is used to find the circuit identities for the tile in the hash table of circuit identities (d) the circuit identities for the tile.

matrix of the tile. After we have obtained the fingerprint, we can search for circuit identities in the hash table of circuit identities as shown in Part (c) in figure 15. Part (d) in figure 15 shows circuit identities for the tile.

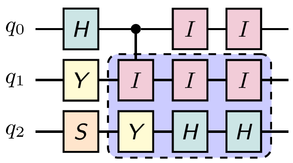
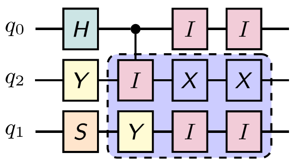
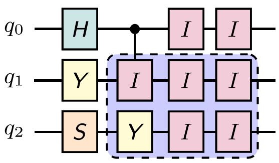
Once we have the fingerprint, we then look up the possible substitutions by its fingerprint in the hash table of circuit identities.

Step 4: apply substitutions via cost-based search. We then iterate through the circuit identities for each tile and evaluate the cost of each circuit identity. In this paper, we simply use the depth of the circuit (ignoring identity gates) as the circuit's cost; the cost function can be customized if desired. Table 18 shows the cost for the circuit identities presented in figure 15. In this case, circuit (iii) in table 18 has the lowest cost and will be used to substitute the tile in the quantum circuit.

Step 5: check for valid substitutions. Adding a substitution might make the entire quantum circuit invalid. For example, two-qubit gates may be cut off or not aligned properly. Additional steps, explained below, are needed to ensure the final circuit has valid two-qubit gates.

Step 6: apply the substitution to the circuit. We apply the substitution to the quantum circuit. If the original tile contained an invalid *CX*, then we have to do a couple more transformations before applying the substitution. Figure 16 shows the steps. Step (a) is equivalent to step (a) in figure 15. In step (b), we apply the substitution with the lowest cost, which was the circuit identity (iii) in table 18. We now need to transform this new circuit so that it is consistent with the gates in the original circuit. In particular, this new circuit has the *CX* target gate replaced with an Identity gate. Step (b) replaces the Identity gate with the original *CX* target gate. Step (c) shows the total transformation from the original circuit to an optimized circuit, after only one tile substitution. The Identity gates have been omitted to show that the depth of the circuit has decreased and therefore the circuit has been improved. We then continue to the next tile with the newly replaced substitution.

Table 18. Circuit identities and their cost.

	Circuit Identity	Cost
i		3
ii		3
iii		1

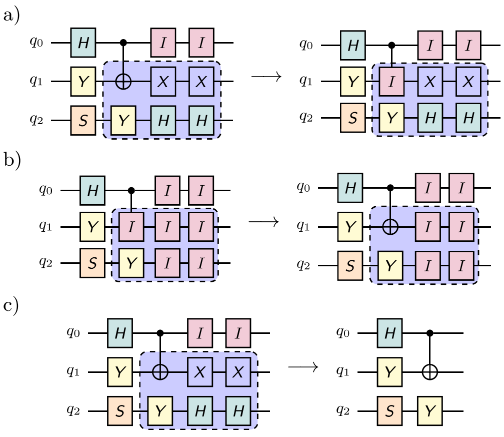


Figure 16. Applying the substitution.

Step 7: repeat process. Steps 1–6 are repeated for a fixed number of times. In our implementation, ten times were sufficient.

4.3.2. Reordering qubits

In order to find more potential optimizations in the circuit, we can also reorder the qubits. Figure 17 shows an example of a circuit with width 5 and height 3 (this is a modification of the example circuit in figure 4). In this circuit, we have a couple of long-range two-qubit gates, which are CNOT gates between qubit q_0 and qubit q_2. If the tiles have width 5 and height 2, which is shown in figure 18, then optimizations between distant pairs of qubits, in this case q_0 and q_2 would be missed.

One way to address this issue is to reorder the qubits. Figure 19 shows the quantum circuit where the qubits have been reordered. Now, with a tile of width 5 and height 2, we can find an optimization for the circuit and reduce its width from 5 to 1, which is shown in figure 20.

4.4. Runtime

The optimization process involves finding an optimum sub-circuit for all of the possible tiles in the input circuit. For each possible tile, the optimum circuit is found in $O(1)$ because of our construction of the hash

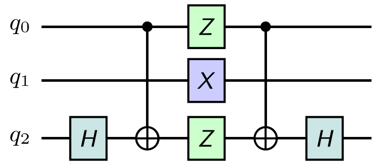


Figure 17. Example of a quantum circuit with a long-range two-qubit gate.

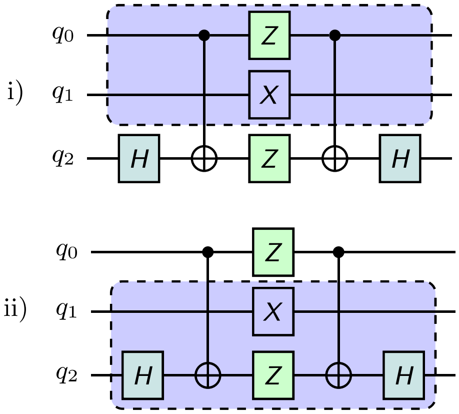


Figure 18. Splitting the circuit into tiles of width 5 and height 2.

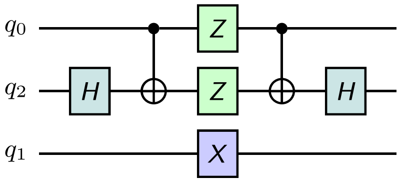


Figure 19. Example of a quantum circuit with qubits reordered.

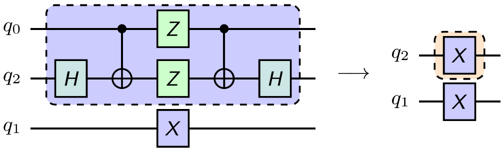


Figure 20. Steps to optimize the circuit when the circuit is split into tiles. The boxes shaded in blue are the tiles. The boxes shaded in orange are the substitutions.

table of circuit identities. The number of possible $i \times j$ tiles for an $n \times m$ circuit is $(n-i+1)\cdot(m-j+1)$. Therefore, the runtime of the optimization algorithm is $O(m \cdot n)$. If qubits are reordered to improve the effectiveness of the optimization, then this adds $O(n!)$ to the algorithm and therefore the algorithm with qubit reordering, takes time $O(m \cdot n!)$. Table 7 shows an example of the runtime of optimising a quantum circuit.

4.5. Implementation

Quanto is implemented in Python and can be used as an API in principle. In particular, Quanto has the following modules:

4.5.1. Circuit module

Quanto can take *qasm* (quantum assembly language) files [13] as input, [5, 10, 12, 14, 24], which is a popular way to construct circuits and makes our optimizer compatible with other quantum circuit frameworks, such as Qiskit [1] and Cirq [2], which can input/output *qasm* files. Our code converts the *qasm* file into our object representation of a circuit as shown in figure 7. We represent a circuit as a grid, where the rows represent the layers and the columns represent the gates applied to the qubits.

4.5.2. Subs module

The user can specify a gate set to generate circuit identities. Quanto also has several pre-set gate set options apart from customized gate sets. If no gate set is specified, then our code automatically detects all the gates used in the input circuit and uses that as the gate set. This module generates the circuit identities and builds the circuit identities database is used for the optimizer module.

4.5.3. Optimize module

The optimize module takes the generated circuit identities database and optimizes the input circuit. It outputs an optimized circuit, along with metrics, such as the final circuit depth. The circuit can be saved as a *qasm* file or a qiskit circuit (applicable for certain gates).

Data availability statement

All data that support the findings of this study are included within the article (and any supplementary files).

The circuit optimisations from previous work are available at [11, 15]. The datasets generated during and/or analysed during the current study are available from the corresponding author on reasonable request.

Conflict of interest

The Authors declare no Competing Financial or Non-Financial Interests.

Author Contributions

Jessica Pointing: Conceptualization, Methodology, Software, Validation, Formal Analysis, Investigation, Resources, Data Curation, Writing—Original Draft, Writing—Review and Editing, Visualization, Project administration. **Oded Padon**: Conceptualization, Methodology, Validation, Formal Analysis, Writing—Review and Editing. **Zhihao Jia**: Conceptualization, Methodology, Validation, Formal Analysis, Resources. **Henry Ma**: Conceptualization, Methodology, Software. **Auguste Hirth**: Conceptualization. **Jens Palsberg**: Conceptualization, Supervision. **Alex Aiken**: Conceptualization, Supervision, Writing—review and editing.

ORCID iDs

Jessica Pointing https://orcid.org/0000-0002-3651-5941
Jens Palsberg https://orcid.org/0000-0003-4747-365X

References

[1] Qiskit 2017 (available at: https://qiskit.org/)
[2] Cirq 2018 (available at: https://quantumai.google/cirq)
[3] Device backend noise model simulations 2024 (available at: https://qiskit.org/ecosystem/aer/tutorials/2_device_noise_simulation.html)
[4] Get backend information with qiskit 2024 (available at: https://docs.quantum.ibm.com/run/get-backend-information)
[5] Balensiefer S, Kreger-Stickles L and Oskin M 2005 Quale: quantum architecture layout evaluator *Proc. SPIE* **5815** 103–14
[6] Bansal S and Aiken A 2006 Automatic generation of peephole superoptimizers *Proceedings of the 12th Inter, Conf. on Architectural Support for Programming Languages and Operating Systems* (*San Jose, California, USA*) pp 394–403
[7] Bansal S and Aiken A 2008 Binary translation using peephole superoptimizers *Proc. 8th USENIX Conf. on Operating Systems Design and Implementation, OSDI'08* (USENIX Association) pp 177–92
[8] Barenco A, Bennett C H, Cleve R, DiVincenzo D P, Margolus N, Shor P, Sleator T, Smolin J A and Weinfurter H 1995 Elementary gates for quantum computation *Phys. Rev.* A **52** 3457–67
[9] Bravyi S and Gosset D 2016 Improved classical simulation of quantum circuits dominated by clifford gates *Phys. Rev. Lett.* **116** 250501
[10] Chuang I 2005 qasm2circ (available at: www.media.mit.edu/quanta/qasm2circ/)
[11] Crooks G E 2024 Gates, states, and circuits (available at: https://threeplusone.com/pubs/on_gates.pdf)
[12] Cross A 2005 qasm-tools (available at: www.media.mit.edu/quanta/quanta-web/projects)
[13] Cross A, Bishop L, Smolin J and Gambetta J 2017 Open quantum assembly language (arXiv:1707.03429)

[14] Dousti M, Shafaei A and Pedram M 2016 Squash 2: a hierarchical scalable quantum mapper considering ancilla sharing *Quantum Information & Computation* **16** 332–56

[15] Garcia-Escartin J C and Chamorro-Posada P 2011 Equivalent quantum circuits (arXiv:1110.2998)

[16] Hietala K, Rand R, Hung S-H, Wu X and Hicks M 2021 A verified optimizer for quantum circuits *Proc. ACM Program. Lang.* **5** 1–29

[17] Jia Z, Padon O, Thomas J J, Warszawski T, Zaharia M and Aiken A 2019 Taso: optimizing deep learning computation with automatic generation of graph substitutions *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (*Huntsville, Ontario, Canada)* pp 47–62

[18] Johnstun S and Van Huele J-F 2021 Understanding and compensating for noise on IBM quantum computers *Am. J. Phys.* **89** 935–42

[19] Lomont C 2003 Quantum circuit identities (arXiv:quant–ph/0307111)

[20] Massalin H 1987 Superoptimizer: a look at the smallest program *ACM SIGARCH Comput. Archit. News* **15** 122–6

[21] IBM recently changed their native gate set to {CX, ID, RZ, SX, X.}

[22] Preskill J 2018 Quantum computing in the NISQ era and beyond *Quantum* **2** 79

[23] Sivarajah S, Dilkes S, Cowtan A, Edgington W and Duncan R 2020 Tket: a retargetable compiler for nisq devices *IOP Quantum Sci. Technol.* **6** 014003

[24] Svore K, Aho A, Cross A, Chuang I and Markov I 2006 A layered software architecture for quantum computing design tools *Computer* **39** 74–83