Machine Learning Optimization of Quantum Circuit Layouts

ALEXANDRU PALER, Aalto University, Finland, University of Texas at Dallas, USA, and Transilvania University of Braşov, Romania
LUCIAN SASU and ADRIAN-CĂTĂLIN FLOREA, Transilvania University of Braşov, Romania
RĂZVAN ANDONIE, Central Washington University, USA and Transilvania University of Braşov, Romania

The quantum circuit layout (QCL) problem involves mapping out a quantum circuit such that the constraints of the device are satisfied. We introduce a quantum circuit mapping heuristic, QXX, and its machine learning version, QXX-MLP. The latter automatically infers the optimal QXX parameter values such that the laid out circuit has a reduced depth. In order to speed up circuit compilation, before laying the circuits out, we use a Gaussian function to estimate the depth of the compiled circuits. This Gaussian also informs the compiler about the circuit region that influences most the resulting circuit's depth. We present empiric evidence for the feasibility of learning the layout method using approximation. QXX and QXX-MLP open the path to feasible large-scale QCL methods.

CCS Concepts: • **Hardware** → **Quantum computation**; **Software tools for EDA**; • **Software and its engineering** → Compilers;

Additional Key Words and Phrases: Machine learning, quantum circuits, optimization

ACM Reference format:
Alexandru Paler, Lucian Sasu, Adrian-Cătălin Florea, and Răzvan Andonie. 2023. Machine Learning Optimization of Quantum Circuit Layouts. *ACM Trans. Quantum Comput.* 4, 2, Article 12 (February 2023), 25 pages.
https://doi.org/10.1145/3565271

1 INTRODUCTION

The quantum circuit layout problem is deeply related to the topology of the device used to execute the circuit: instructions cannot be applied between arbitrary hardware registers. Before executing a quantum circuit, this is adapted to the device's register connectivity during a procedure called *compilation*. Quantum circuit compilation is often called *quantum circuit layou*t (QCL). The interest in efficient QCL methods is motivated by the current generation of quantum devices, called *NISQ devices* [29]. Very recent work presents worrisome evidence that even very small and shallow circuits are difficult to execute on NISQ [37].

NISQ circuit compilation includes, for example, error-mitigation strategies [11], flag-qubits [8] and not just QCL methods, but the latter definitely play a significant role in the compilation of

AP was supported by Google Faculty Research Awards and the project NUQAT funded by Transilvania University of Braşov.
Authors' addresses: A. Paler, Konemiehentie 2 02150 Espoo Finland; email: alexandru.paler@aalto.fi; L. M. Sasu and A.-C. Florea, Str. Iuliu Maniu nr.50 Braşov, Romania; R. Andonie, 400 East University Way, Ellensburg, WA 98926, USA.

2643-6817/2023/02-ART12 $15.00
https://doi.org/10.1145/3565271

large-scale fully error-corrected circuits. However, scalable compilation is not possible with current state-of-the-art QCL methods. Fast and scalable QCL allows laying out a circuit with multiple QCL parameter values and selecting the best compiled circuit. Without going into detail, our preliminary analysis showed that at the time of this writing, when optimizing aggressively, compilation can take up to 1 hour for most QCL methods when presented circuits of approximately 50 qubits. It is imperative to have efficient and configurable QCL methods. We focus on three research questions:

(I) How can we determine the best QCL parameter values to minimize the depth of the compiled output circuit?

(II) Choosing good parameter values for the QCL method should be very fast. How can we establish a time-performance trade-off between searching optimal parameter values and the minimization of the depth?

(III) Can QCL be sped up by machine learning?

More formally, QCL takes as input a circuit C_{in} incompatible with a device's (can be error corrected) register connectivity and outputs a circuit C_{out} that is compatible. *Additional gates* are used to overcome the connectivity limitations and compile a device-compatible circuit C_{out}. We define the *Ratio* function between the depths of two circuits, where $|C| > 0$ is the depth of circuit C. We formalize the QCL problem as follows: *QCL optimization is the minimization of the Ratio function.*

$$Ratio(C_{in}, C_{out}) = \frac{|C_{out}|}{|C_{in}|} \tag{1}$$

1.1 Related Work

QCL is already a wide topic, and we will not be providing a thorough exposition of the field. We refer the reader to the works of the authors of [9, 31, 32, 34, 35] for detailed and careful overviews of some of the studies that influenced and shaped QCL.

Some of the first discussions about circuit optimality and gate counts appeared in the seminal paper of Barenco et al. [4]. After quantum circuits started being analyzed as reversible circuits formed from Toffoli gates, many exact methods and heuristics were proposed —a not so recent but complete review is provided by Saeedi and Markov [30]. The complexity class of QCL was discussed first in [19], which has been used as a foundation for proving the complexity of different QCL variations, such as in [7, 32, 34].

In general, there are heuristic QCL methods and exact QCL methods. A recent exact method is found in [34] (which includes a discussion about bottlenecks of exact methods). Automatic quantum circuit compilation does not always generate the best possible circuit. One of the first attempts to design full algorithm circuits considering hardware connectivity limitations is described in [14]. Another recent example of a hand-optimized circuit is found in [26], in which, interestingly, the optimization was achieved by using results known from the automatic optimization of circuits.

Most quantum circuit design automation tools treat QCL as a sequence of steps, like initial placement and gate scheduling. The authors of [31] present a complete discussion of the theoretical implications of placement and scheduling. In practice, QCL has been solved by introducing SWAP gates, but there are more refined methods, such as those described in [23]. Solving QCL through search algorithms has been recently presented in [24] (beam search) and [41] (A*). The work of [31] introduced a parameterizable search algorithm for QCL, Bounded Mapping Tree.

In this work, we focus on QCL as an instance of register allocation as introduced by [32] and use the by now classic graph view of the quantum device layout. Recently, graph-based QCL approaches (motivated by [19]) seem to achieve the necessary performance for compiling very large circuits to complicated device topologies. Some QCL approaches were proposed using hypergraphs [3], but more classical approaches can be found in [10].

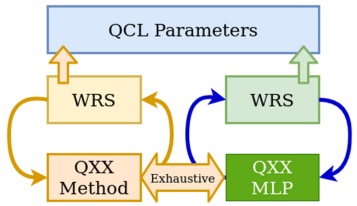

Fig. 1. Continuous learning QCL: It is possible to learn our QCL method, which is called QXX (yellow), and replace it with a machine learning model (e.g., neural network — green). Optimal parameter values (blue) for adapting QXX performance are chosen by automatically executing weighted random search (WRS) in a loop.

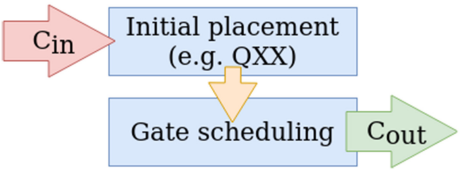

Fig. 2. The quantum circuit compilation (QCL) procedure takes an input circuit C_{in} and transforms it into the functionally equivalent C_{out} circuit. The QXX method computes an optimal assignment of circuit qubits (registers) to device qubits (registers).

Machine learning has started being widely applied in different aspects of quantum computing. Nevertheless, compared with the work in [40], we are not compiling only single-qubit gates after needing impractical numbers of hours to train a very large network. In parallel and independent to the preparation of this article, machine learning methods for QCL have been presented in [28]. Compared with the work in [28], our method is capable of learning continuously.

QCL includes two steps [31]. The first step is *initial placement* (e.g., [25]), in which a mapping of the circuit qubits to device registers is computed. This step is also called *qubit allocation* [32]. The second QCL step is *gate scheduling*, in which the circuit C_{in} is traversed gate-by-gate.

1.2 Contribution

We present a machine learning method (Figure 1) for the initial placement (i.e. mapping, see Figure 2) of quantum circuits. In order to test the feasibility of learning, we develop an initial placement heuristic, which we called QXX for no particular reason.

From a technical perspective, our methods are designed and implemented in great detail and capable of being deployed in practice. The machine learning model of QXX is called QXX-MLP, which is the significantly faster implementation of the QXX. As of this writing, our method was the first to effectively learn a QCL heuristic.

QXX is configurable over multiple parameters, and our goal is to have a method that can automatically choose the best parameter values in order to achieve optimally compiled circuits. The novelties of QXX and QXX-MLP (Section 2) are as follows.

(1) Automatic feature selection–focusing on the important subcircuit using a configurable Gaussian function (Section 2.3);
(2) Automatic QCL configuration–we use weighted random search (Section 2.5) to optimize QXX parameter values. The parameters also influence the speed of the QCL;
(3) Demonstrating QCL learnability–the machine learning version QXX-MLP works as an approximation method for circuit layout depth (Section 2.6);

(4) Scalability–the almost instantaneous layout performance (Sections 3.2 and 3.4 and Table 3). We show that QCL execution time can be shortened while maintaining the *Ratio* optimality.

The goal of this work is to highlight the generality and wide applicability of learning QCL methods. For this reason, we focus on benchmarking circuits that are compatible with both non-error-corrected and error-corrected machines. We use synthetic benchmarking circuits, QUEKO [35] (Section 3.1), which capture the properties of Toffoli+H (e.g., arithmetic, quantum chemistry, and quantum finance) circuits without being specific for a particular application. Section 3 discusses experimental results performed with the QXX and QXX-MLP approaches. We present our conclusions in Section 4.

2 METHODS

In this section, we describe the QXX method and the machine learning techniques that are using it. Section 2.1 gives the details of QXX. Figure 1 illustrates the approach followed by this work during the QCL parameter optimization stage. The parameters of the normal QXX method (orange) are optimized using Weighted Random Search (WRS). To further speed up the QXX, we train a machine learning model, called QXX-MLP, to predict the *Ratio* values obtained by using the QXX for a given circuit and a particular set of parameter values.

We employ three methods to evaluate how the parameter optimization of QXX influences the compiled circuits. First, an efficient parameter optimizer is WRS (Section 2.5). Second, we use exhaustive search to collect training data to obtain the QXX-MLP neural network (Section 2.6). The latter is used to estimate optimal parameter values.

The WRS method starts from an initial set of parameter values and adapts the values in order to minimize the obtained *Ratio*. This procedure forms a feedback loop between WRS and the QXX method. Running WRS multiple times for a set of benchmark circuits is equivalent to *almost* executing an exhaustive search of the parameter space. The exhaustive search data are collected and used to train QXX-MLP.

From a methodological point of view, device variabilities are very important and seem to fluctuate in their behavior [37]. However, in this work we do not consider that NISQ qubits have variable fidelities [36] or that crosstalk is a concern during NISQ compilation [22].

We will use the following notation. The first QCL step computes a list M, where $M[q] = r$ refers to circuit qubit q being stored on device register r. Computing the list M is the analogue to determining a good starting point for the gate scheduling procedure. We represent two-qubit gates by the tuples (q_i, q_j). Scheduling executes the current two-qubit gate if $(M[q_i], M[q_j])$ is an edge of device connectivity graph, which will be called *DEVICE*. Otherwise, the gate qubits are moved across the device and stored in registers connected by an edge from *DEVICE*. The movement introduces additional gates, such as SWAP gates, in order for all tuples (q_i^{out}, q_j^{out}) to be edges of *DEVICE*. The mapping is updated accordingly, as illustrated in Figure 7. In general, the depth of C_{out} is lower bounded by $|C_{in}|$.

2.1 The QXX Mapping Heuristic

QXX is a fast search algorithm to determine a qubit mapping (allocation), and is called by the subsequent gate scheduler to compute a good qubit allocation/mapping/placement. QXX uses an estimation function to predict how a C_{out} with minimum depth would have to be initially mapped.

A novelty is that QXX uses a Gaussian-like function called *GDepth* to estimate the resulting depth (cost) of the laid out circuit, called C_{out}. The qubit mapping is found using the minimum

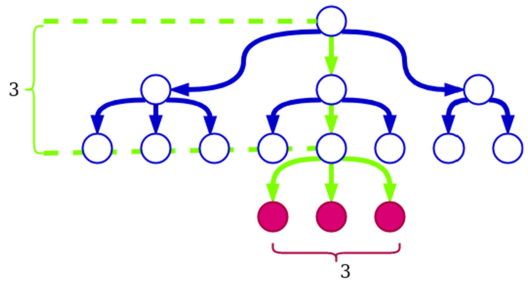

Fig. 3. The search space dimensions of QXX can be configured by adapting the parameters *MaxDepth* (green) and *MaxChildren* (red). Each node of the search tree stores a list of possible mappings for which a minimum cost was computed. The maximum length of the list is *MaxChildren* (e.g., 3 blue levels). The list is emptied if a lower minimum cost is computed or if *MaxDepth* has been achieved. In the latter case, the path with the minimum cost (green) is kept and all other nodes are removed.

estimated value of *GDepth*. QXX uses *three types of parameters*, which we will explain in the following three sections:

(1) for configuring the search space;
(2) for adapting *GDepth* to the circuit C_{in};
(3) for adapting QXX to the second step of QCL, namely, the scheduler/router of gates.

2.2 Search Space Configuration

QXX is a combination of breadth-first search and beam search. The search space is a tree (Figure 3). Constructing a qubit-to-register mapping is an iterative approach: qubits are selected one after the other and so are the registers where the qubits mapped initially.

For example, consider $C_{in} = \{(q_1, q_2), (q_2, q_3), (q_3, q_4)\}$ a circuit of three CNOTs and a mapping $M = [r_1, r_2]$. This means that q_1 is allocated to register r_1, and q_2 to r_2. After the first qubit is mapped, we have that $|M| = 1$. After all qubits are mapped, we have that $|M| = Q$. The maximum depth of the tree is Q. In the worst case, each node has Q children.

The tree is augmented one step at a time by adding a new circuit qubit a to the mapping (in the order of their index, in the current version — we do not analyze the influence of this choice). This increases the tree's depth: at each existing leaf node, all possible N mappings of a are considered. Consequently, all of the new leaves of a tree are the result of appending a to the previous level's leaves, which now are usual nodes.

New depth estimation values are computed using the *GDepth* function each time leaves are added to the tree. Each tree node has an associated *GDepth* cost. The level in the tree equals the length of the mapping for which the cost was computed.

The search is stopped after computing a complete mapping with the minimum *GDepth* cost. Thus, the maximum number of leaves per node is added in the unlikely case that all values of *GDepth* are equal.

The search space will easily explode for large circuits. We introduce two parameters to prune the search space. In Figure 3, the result of pruning the search space tree is represented by the green path and the green bounding box.

The first parameter is *MaxChildren*, whose job is to limit the number of children of equal minimum *GDepth* values. For an arbitrary value of $MaxChildren < Q$, the tree will include at level l at most $l \cdot MaxChildren$ nodes.

The second parameter is the cut-off threshold *MaxDepth*, which specifies that, at levels indexed by multiples of *MaxDepth*, all of the nodes are removed from the tree except for the ancestors of

the minimum cost leaf. This is because for large circuits (e.g., more than 50 qubits) it is not practical to evaluate all of the new combinations of Q registers for $l \cdot MaxChildren$ leaves.

2.3 Configuration of Circuit Depth Estimation

The *GDepth* function is used to estimate the depth of the compiled circuit without laying it out.

The value of *GDepth* can be calculated once at least two qubits are mapped. Equivalently, the value of *GDepth* is computed only for CNOTs whose qubits are mapped. The *GDepth* function is a sum of a Gaussian function whose goal is to model the importance of CNOT sub-circuits from C_{in}. In other words, *GDepth* is the sum of the costs estimated for scheduling the CNOTs of a circuit:

$$GDepth = \sum_{i=0}^{N_c} dist_i \exp\left(-B \cdot \left\| \frac{i}{N_c} - C \right\|^2\right) \quad \text{where } N_c \leq |C| \tag{2}$$

In this formula, N_c is the number of CNOTs from C_{in} whose qubits were already mapped. The B and C parameters control the spread and position of the Gaussian function. The value of i is the index of the CNOT from the resulting circuit and $dist_i$ is the cost of moving the qubits of the CNOT.

For example, for the circuit

$$C_{in} = \{(q_1, q_2), (q_2, q_3), (q_3, q_4)\}$$

and the mapping $M = [r_1, r_2]$, $N_c = 1$, the *GDepth* can be computed for only the first CNOT because q_3 and q_4 were not mapped.

The scheduled CNOTs are indexed, and *we assume that all gates are sequential* (parallel gates are executed sequentially, considering the circuit truly a gate list). For example, after scheduling two CNOTs of an hypothetical circuit, the two CNOTs are numbered 1 and 2. In the exponent of *GDepth*, the value of $\frac{i}{N_c}$ is always in the range $[0, 1]$.

For $dist_i$, we use the shortest distance on an undirected graph. An example is Figure 7, where $dist_i = 5$. For two qubits a, b, where $(M[a], M[b])$ is already an edge of *DEVICE*, the $dist_i = 1$. Otherwise, the distance between two qubits is computed based on the edge weights attached to the *DEVICE* graph.

If the intention is to allow CNOTs at the middle of the circuit to have longer movements on the device, we can set parameters to generate a function such as that seen in the top right panel of Figure 5. For example, for $B = 5$ and $C = 0.5$, the $dist_i$ from *GDepth* is weighted with almost zero at the start of the circuit (Figure 4). The opposite situation is illustrated in the middle panel. For $B = 0$, the Gaussian is effectively a constant function, such that the cost is the sum of all of the CNOT distances.

2.4 Configuring QXX for the Scheduler

QXX is effectively estimating the output of the gate scheduler, which *selects the best edge* of *DEVICE* at which to execute the CNOTs of a circuit. The scheduler is modeled as a black box and its functionality is unknown. QXX can work with Qiskit's StochasticSwap, tket, or other tools such as the those featured in [18, 39].

Starting from the initial placement, QXX estimates the total depth of the circuit after repeatedly mapping the circuit. Without loss of generality, QXX assumes that the scheduler will move both qubits of a CNOT across the *DEVICE* toward the selected edge. Instead of updating the mapping, the movements of the qubits are *accumulated* into an *offset* variable.

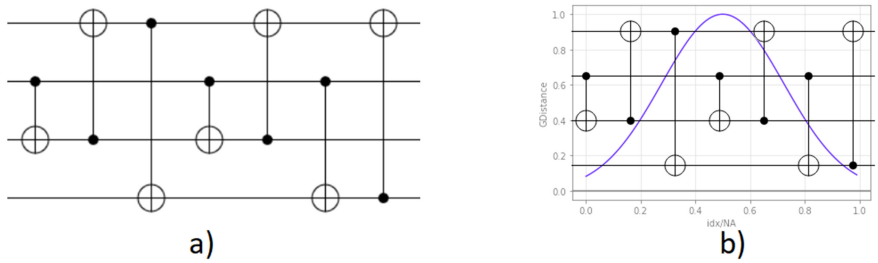

Fig. 4. A quantum circuit and a Gaussian function. (a) A 4-qubit quantum circuit consisting of a sequence of CNOTs; (b) The $dist_i$ uses a Gaussian function. The latter is drawn superimposed in order to highlight the weight generated by the Gaussian at each CNOT position in the circuit. The weights are minimal for the first and last CNOT. The maximum value is for the CNOT at the middle of the circuit.

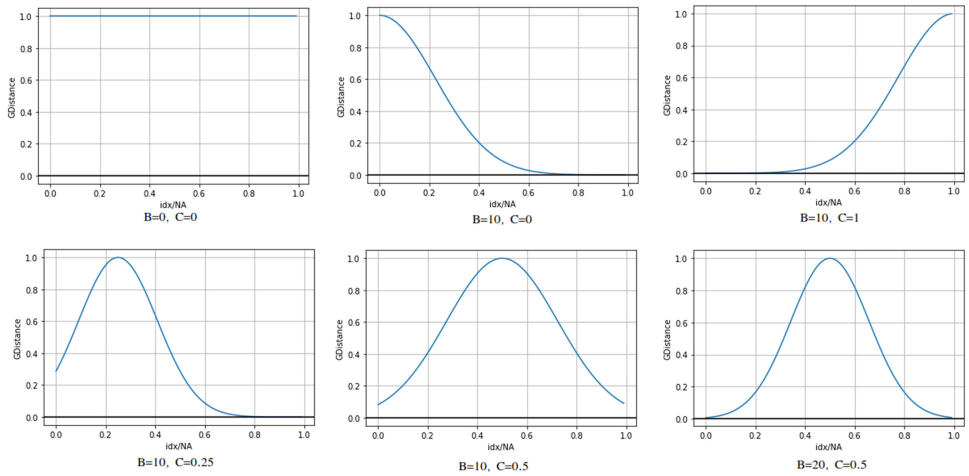

Fig. 5. The value of the Gaussian used by *GDepth* for different values of parameters B and C. The horizontal axis illustrates the input value to the Gaussian: the gates' integer index in the circuit is scaled with the total number of gates in the circuit. The importance (weight) of a gate is highest whenever the value of the Gaussian approaches the maximum value. A gate has low importance when the Gaussian has low values.

Qubit movements are captured by the *MovementFactor* parameter. The *MovementFactor* is asymmetric, and when processing CNOT qubits, it moves on the *DEVICE* the qubit with the lowest index by the fraction $\frac{1}{MovementFactor}$, and the qubit with the highest index by $\frac{MovementFactor-1}{MovementFactor}$.

As we will show in the Results section, for deep circuits we determine *MovementFactor* values closer to 2 (favors high- and low-indexed qubits equally), and for shallow circuits the values are higher (favors low-indexed qubits).

After moving the qubits on the device (Figure 6), the offset of a qubit is an estimation of how much the qubit was moved by the scheduler. For example, the offset of an arbitrary qubit used in 3 CNOTs is the sum of the three movement updates that are obtained after scaling each CNOT's $dist_i$ with the corresponding *MovementFactor* expression.

The movement of qubits on the device is controlled by an additional parameter, called *EdgeCost*: higher values imply a larger estimation of the movement heuristic. Changing the value of *EdgeCost* is equivalent to scaling the total value of *GDepth*, because *EdgeCost* is a common factor in the calculation of $dist_i$. For the weight scale factor $EdgeCost = 1$, as shown in Figure 7, the minimum

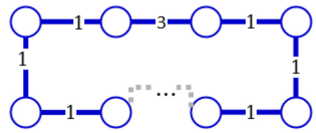

Fig. 6. Device connectivity graph is blue; the weight of all but one edge is 1. For example, the high cost might be due to higher physical error rates.

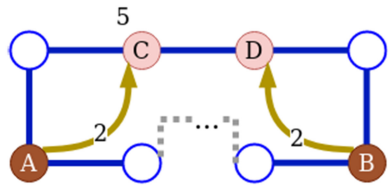

Fig. 7. The effect of the *MovementFactor*: Device connectivity graph is blue, and the two brown qubits A and B have to interact. If not specified, all edges have weight 1. The shortest path between A and B has cost L = 5, because there are 5 edges between A and B. There are multiple options to bring A and B together. One is to assume that A and B move to the pink C and D. Assuming that CD is at the middle of the path connecting A and B, then *MovementFactor = 2* because both A and B are moved on average L/2. A higher movement factor implies that one of the qubits moves less, whereas the other moves more.

Table 1. Parameter Values

	QXX Parameters			WRS Obtained		Exhaustive Search		
Name	Min	Max	Increment	WRS-Weight	Prob. of Change	Start	Stop	Increment
MaxDepth	1	55	1	9.35	0.62	1	9	4
MaxChildren	1	55	1	8.00	0.53	1	9	4
B	0	500	0.1	7.76	0.52	0	20	2
C	0	1	0.01	15.06	1.00	0	1	0.25
MovementFactor	1	55	1	3.52	0.23	2	10	4
EdgeCost	0.1	1	0.1	10.59	0.70	0.2	1	0.4

The *QXX Parameters* columns represent the min. and max. value together with the increments be to used with the heuristic. The *WRS Obtained* columns illustrate the results of the weighted random search method for optimal parameter values. The trainings of QXX-MLP as well as WRS were performed for a smaller search space whose parameter ranges are presented in the *Exhaustive Search* columns.

distance between the CNOT qubits corresponds to the sum of the edge weights separating them (5 in Figure 7).

2.5 Weighted Random Search for Parameter Configuration

The QXX parameters from Sections 2.2–2.4 have to be tuned for optimal performance of the compilation. We call the evaluation for one set of parameter values a *trial*. In previous work [13], we introduced the WRS method,[1] a combination of Random Search (RS) and probabilistic greedy heuristic. Instead of a blind RS, the WRS method uses information from previous trials to guide the search process toward the next interesting trials. We use WRS to optimize the following QXX parameters (introduced in Section 2.1): *MaxDepth*, *MaxChildren*, *B*, *C*, *MovementFactor*, and *EdgeCost* (see Table 1).

[1] https://github.com/acflorea/goptim.

Within the same number of trials, different optimization methods achieve different scores depending on how "smart" they are. Due to the nature of QXX, the trial execution times are variable since they depend on the defined quantum architecture, the topology of the circuit to lay out, and the values of the parameters (see Figure 3). Search space reduction and search strategy are interconnected. For the exhaustive search parameter ranges from Table 1, we limit the time spent evaluating a parameter configuration by introducing a *timeout* parameter. WRS uses the obtained data to run an instance of functional analysis of variance (fANOVA) [15].

2.6 Learning QXX - Training QXX-MLP

The previous section was about searching parameter values. Herein, we go one step further and learn the behavior of the gate scheduler in relation to the mapper parameters and the *GDepth* function used by QXX. We describe the method to learn specific tuples consisting of circuit and QXX parameters, and how to estimate the depth of the mapped circuits.

Parameter optimization and, for that reason, efficient initial mapping of the circuit to the device, is a regression problem. The training data for learning was obtained as follows: for the parameters from Table 1, we choose smaller ranges and larger increments, as illustrated in the *Exhaustive Search* columns. There are three possible values for $MaxDepth$ and three values for $MaxChildren$. For a given value of $MaxDepth$, there are $3 \times 11 \times 5 \times 3 \times 3 = 1485$ possible parameter configurations. There is a total of $90 \times 1485 = 133{,}650$ layouts for each $MaxDepth$. Overall, there are 400,950 parameter configurations of the 90 circuits that are being evaluated.

Table 1 illustrates the parameter ranges for which we collected data that allows us to compute the ratio $Ratio(C_{in}, C_{out})$ for every combination of circuit layout and QXX parameters. The generated exhaustive search data are available in the project's online repository.

We considered three candidate models to learn QXX: k Nearest Neighbors (KNN), Random Forest (RF) and MultiLayer Perceptron (MLP). Each model has a different inductive bias [21]: a local–based predictor, an ensemble model built by bootstrapping, and a connectionist model, respectively.

Despite their conceptual simplicity, KNN predictors are easily interpretable and passed the test of time [38]. RFs were found as the best models for classification problems [12], and we wanted to investigate their performance on this regression problem as well. Finally, MLPs create new features through nonlinear input feature transformations, unlike KNN and RF, which use raw input attributes. Nonlinear transformations and the non-local character of MLPs are considered the premises for the successful deep learning movement [5].

We found the MLP to be the best model and use it during the parameter optimization stage (as illustrated in Figure 1) as an approximator for the functionality of QXX. More details are in the Appendix.

3 RESULTS

We present empirically obtained results about the performance of QXX, the quality of the QXX-MLP model, and the performance of optimizing QXX parameters with WRS. We select QXX parameter values using exhaustive search, WRS with the QXX method (orange in Figure 1), and WRS on the QXX-MLP (green in Figure 1).

The QXX method was implemented and is available online.[2] The current implementation is agnostic of the underlying quantum circuit design framework (e.g., Cirq or Qiskit). QXX is implemented in Python. The exhaustive search was executed on an i7 7700K machine with 32 GB of RAM. The QXX model was trained on Intel Xeon W-2145 3.70GHz with 16 cores and 256 GB RAM.

[2] https://github.com/alexandrupaler/qxx.

The WRS parameter optimization was performed on a laptop grade i5 processor with 16 GB RAM. For the benchmarks and comparisons, we used Cirq 0.9, IBM Qiskit 0.25, and tket 0.2.

In the following, we describe how the WRS heuristic is used to evaluate the QXX method and its MLP implementation. We then present a series of plots that support empirically the performance of QXX. We analyze the influence of the parameters and offer strong evidence in favor of learning quantum circuit layout methods. In particular, we will show that the *GDepth* Gaussian can shorten the time necessary to run QXX. This is achieved by pruning the search space and focusing on the most important region of the circuit. We will present examples of how the Gaussian is automatically adapted for deep and shallow circuits.

3.1 Benchmark Circuits

We evaluate the *Ratio* fitness of the QXX method using the QUEKO benchmark suite [35]. These circuits abstract Toffoli-based and quantum supremacy–like circuits, as well as a variety of NISQ chip layouts. Such benchmarks complement the libraries of reversible adders and quantum algorithms [17].

Automatic compilation/mapping of circuits, although applicable to NISQ applications, for example, Figure 9, is of little practical importance by itself when taking fault-tolerance into account. NISQ circuits, such as for supremacy or for VQE/QAOA, are co-designed and a method such as ours is just one piece in a much larger workflow. We focused on Toffoli+H circuits, QUEKO TFL, because such circuits are not co-designed and are also very representative for large-scale error-corrected computations.

The 90 QUEKO TFL circuits include circuits with known optimal depths of $[5, 10, 15, 20, 25, 30, 35, 40, 45]$ (meaning that the input circuit and the layout circuit have equal depths $|C_{out}| = |C_{in}|$). For each depth value, there are 10 circuits with 16 qubits. The NISQ machine to map the circuits to is Rigetti Aspen. A perfect QCL method will achieve $Ratio = 1$ on the QUEKO benchmarks.

In our experimental setup, the layout procedure uses QXX for the initial placement (first QCL step) and the Qiskit `StochasticSwap` gate scheduling (second QCL step). The results depend on both the initial placement and the performance of the `StochasticSwap` scheduler. We do not configure the latter and use the same randomization seed. We assume that this is the reason why the QXX performance is close to that of Qiskit.

3.2 Resulting Depths and Scalability

Our goal is to show that the behavior of the mapper/compiler can be learned and that compilation can be sped up using machine learning. QXX achieves *Ratio* values around 30% lower (which means better/best *Ratio* is 1) than Qiskit on the low depth (up to 15 gates) QUEKO TFL circuits. In general, as shown in Figure 8, the performance of QXX is between Qiskit and tket [33]. tket outperforms Qiskit and QXX on the QUEKO benchmarks, because it has a much smarter scheduler. QXX is not a scheduler; rather, it is a mapper (see Section 2.4).

The results from Figure 8 are encouraging, because (a) QXX performs better than most compilers; (b) there is known variability in the compiler's performance with respect to benchmark circuits, such that for other circuits the classification might look completely different; and (c) our results were obtained very fast when using the MLP approach —we get the laid out circuit almost instantaneously (Section 3.4).

For NISQ gate error rates (realistically) upper bounded by 10^{-2}, only circuits with a maximum depth of 30 are of practical importance. For shallow circuits, the mapper is more important than the scheduler (QXX is better than Qiskit); for deeper circuits, the importance of the mapper vanishes

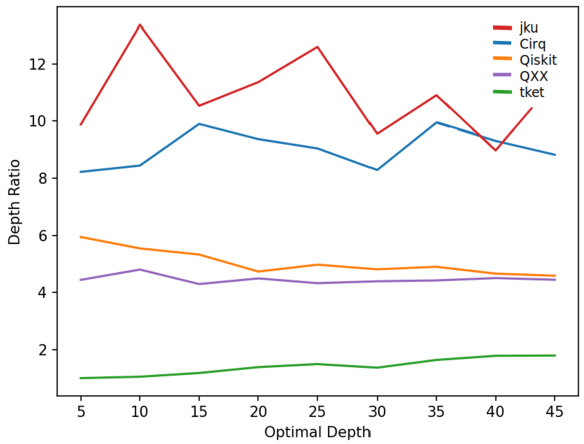

Fig. 8. Comparison with state-of-the-art methods using QUEKO TFL benchmark circuits. All curves except QXX are from [35]. The curves are computed after averaging the depth ratios for 10 circuits for each known optimal depth from the benchmark. The horizontal axis is the known optimal depth of the TFL circuits. The vertical axis illustrates the achieved depth *Ratio*. We have used QXX together with the Qiskit scheduler, and we assume that this is the reason why the QXX curve is close to the Qiskit curve. For shallow circuits, the mapper is more important than the scheduler (QXX is better than Qiskit), and for deeper circuits the importance of the mapper vanishes (QXX and Qiskit perform close to each other). QXX is not a scheduler; rather, it is a mapper (see Section 2.4).

(QXX and Qiskit perform close to each other). Figure 10 presents results with parameters chosen specifically for shallow circuits.

With respect to the scalability of our methods, one interesting aspect is the timeouts. For extreme parameter values (e.g., when *MaxChildren* and *MaxDepth* are 9), the execution time of QXX is high although the method has polynomial complexity. We introduced a timeout of 20 s for the QXX executions and collected data accordingly. Table 2 illustrates the increasing execution times. It offers the motivation to learn the method — the model will have constant execution time irrespective of the parameter value configuration.

Additionally, we notice that computing a good initial placement takes, in the best case, a small fraction of total time spent laying out. Without considering the optimality of the generated circuits, in the worst case, computing the initial placement using QXX can take between 2% and 99% of the total layout time. For more details see Table 3 and Section 3.4. For $MaxDepth = 1$, the maximum time fraction is 10%; for $MaxDepth = 5$, the maximum is 85%; and for $MaxDepth = 9$, it is 99%. These values are also in accordance with the execution times presented in Table 2. However, when considering the 100 fastest QXX execution times, for each of the *MaxDepth* values, the maximum mapping duration is 4% of the total layout duration.

3.3 QXX Parameter Optimization using WRS

Initially, we ran WRS for a total of 1,500 trials within the parameter space defined in Table 1. For *MaxDepth*, *MaxChildren*, and *MovementFactor*, we use the same limits and steps defined in the table. For *B*, *C*, and *EdgeCost*, we generated the values by drawing from a uniform distribution in the specified range.

We ran the classical RS step for 550 trials and computed the weight (importance) of each of the parameters using fANOVA, obtaining the values in Table 1. The weight of a parameter measures its importance for the optimization of the fitness function.

Table 2. The Number of WRS Timeouts (0.05 s, 0.5 s, 5 s, 20 s) is a Measure of QCL Execution Time

Search Space Params.		Timeout			
MaxDepth	MaxChildren	0.05 s	0.5 s	5 s	20 s
1	1	25	0	0	0
1	5	43	0	0	0
1	9	36	3	0	0
5	1	25	0	0	0
5	5	23,899	0	0	0
5	9	41,014	2,457	0	0
9	1	46	0	0	0
9	5	44,550	37,365	2,501	172
9	9	44,550	44,494	36,288	28,798

We count the number of timeouts when using different search space pruning strategies configured by $MaxDepth$ and $MaxChildren$.

Table 3. Parameter Values Obtained Using WRS on a Batch of 90 QUEKO Circuits

	TFL-45				TFL-25				MLP
Name	20s	5s	0.5s	0.05s	20s	5s	0.5s	0.05s	
MaxDepth	9	9	9	6	8	9	9	3	9
MaxChild	4	3	2	2	4	3	2	2	9
B	5	17.3	8	3.5	6.9	15.7	6.10	2	1.5
C	0.61	0.25	0.02	0.31	0.86	0.91	0.65	0.74	0.32
Mov.Factor	2	4	2	6	6	10	6	7	10
EdgeCost	0.2	0.2	0.2	0.9	0.2	0.2	1	0.2	0.8
Duration(s)	35170	28800	21785	11438	10583	8110	6497	4102	~2
Avg. Ratio	4.093	4.138	4.328	4.465	4.043		3.966	4.279 4.537	4.423

The TFL-45 and TFL-25 are for QXX and WRS on the TFL circuits of depths 25 and 45. The MLP column is for when using QXX-MLP with WRS. The last two rows represent the durations and the recorded average Ratios. The MLP approach is very fast, taking approximately 2 s.

We use the $mean(Ratio) = mean(|C_{out}|/|C_{in}|)$ fitness measure. WRS ran with eight workers for a total time of 4 hours and 11 minutes and the best result (3.99) was obtained at iteration 1,391 and again, at a later trial, for a different value of $MaxDepth$. Table 3 shows the combination of parameter values that yield the best results.

The subsequent WRS executions use the exhaustive search parameter space defined in Table 1. In this parameter space, we use WRS to optimize several configurations for which we have changed either the evaluation timeout with values from Table 2 or the maximum TFL Depth (either 25 of 45). Interestingly, the 5-s timeout achieves a better performance than the WRS parameter optimization with a 20-s timeout (see large number of timeouts in Table 2).

In general, WRS selects high $MaxDepth$ values. This is explainable by the fact that optimal $Ratio$ values are easier to be found using large search spaces.

3.4 QXX-MLP using WRS: Fast and Scalable QCL

One of the research questions was whether it is feasible to learn the QCL methods in general and QXX in particular. We also answer the question "Would it be possible to choose optimal values by a rule of thumb instead of searching for them?". For example, is it possible to obtain good $Ratio$ in a timely manner by using low $MaxDepth$ values? During our experiments, the found parameters

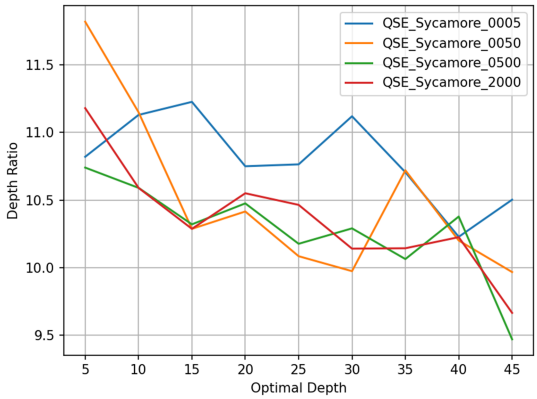

Fig. 9. Transfer of learning: Laying out QUEKO QSE (supremacy experiment) circuits using parameters learned from QUEKO TFL circuits. Each parameter evaluation executed by WRS was timed out after 5, 50, 500, and 2,000 $1/100th$ s (10 ms).

did not differ significantly between QXX and QXX-MLP. The results from Figure 10 and Table 3 show that the MLP model of QXX performs well —within a 10% performance decrease compared with the normal QXX.

We used a combination of WRS and QXX-MLP in an attempt to minimize the time required to identify an optimal configuration. QXX-MLP and WRS are almost instantaneous: under 2 s for the entire batch of 90 circuits. In contrast, the execution time of WRS using the normal QXX was about 3 hours to find the optimal parameters for the 90 circuits. A detailed analysis of the speed/optimality trade-offs is available in the Appendix (Figure 17).

We also tested the *transfer of learning*: the possibility of training QXX on a set of circuits and then applying to a different type of circuit. Figure 9 illustrates the results of applying QXX-MLP on quantum supremacy circuits (QSE)[34].

The performance of QXX-MLP is more than encouraging: it had the performance of a timed-out WRS optimization. The WRS parameter optimization did not time out, because MLP inference is a very fast constant time operation. Table 3 shows that QXX-MLP performs similarly to the WRS, with a 0.05-s timeout. For example, when applied to QXX-MLP, the values chosen by WRS for *EdgeCost* and *MovementFactor are consistent* with the exhaustive search results (Figure 20 in the Appendix): in general, an $EdgeCost = 0.2$ is preferable for all of the TFL-depth values, and for $MovementFactor > 2$ is preferable. The preference for large movement factors is obvious for the shallow TFL circuits.

Regarding QCL speed, the WRS and QXX-MLP optimization takes a few seconds compared with the hours (see Table 3 for laying out all of the circuits from Section 3.1) necessary for WRS and QXX. This is a great advantage that comes with the cost of obtaining a trained MLP, which is roughly the same order of magnitude to a WRS parameter optimization. MLP training is performed only once. In the case in which the MLP model is not used, WRS parameter optimization has to be repeated. In a setting in which quantum circuits are permanently laid out and executed (as in quantum computing clouds) incremental online learning is a feasible option.

The parameter optimization of QXX-MLP seems to be very predictable because the corresponding curves in Figure 10 are almost flat. Table 3 provides evidence that the MLP is very conservative with the choice of the values of B and C: the Gaussian curve is almost flat and positioned slightly towards the beginning of the circuit. The flatness of the curve and its position could explain the almost constant performance from Figure 10 and the 10% average performance degradation of QXX-MLP.

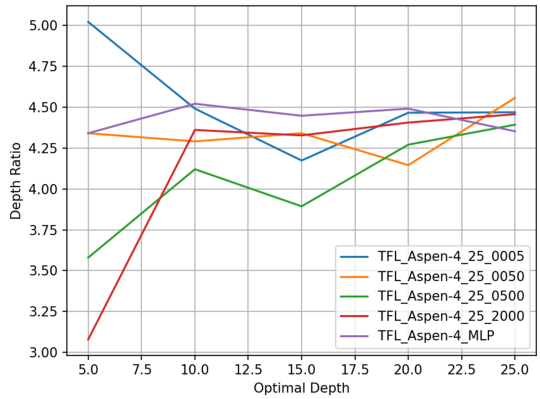

Fig. 10. QXX-MLP achieves approximately 90% of the QXX performance when laying out TFL circuits with depths up to 25 — the most compatible with current NISQ devices. WRS was applied on the normal QXX and timing out too long evaluations and the QXX-MLP model.

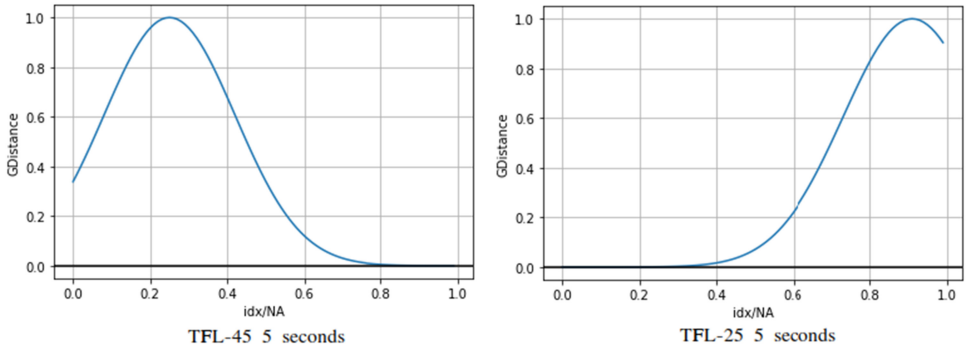

Fig. 11. Two Gaussian curves obtained using WRS. Left: on TFL-45 circuits with a timeout of 5 s; right: on TFL-25 circuits with a timeout of 5 s. (see Table 3).

3.5 Automatic Subcircuit Selection

Our results support the thesis that WRS can adapt the parameters of the QXX Gauss bell curve in order to select the region of the circuit that influences the total cost of laying it out. The parameter controlling the center of the bell curve is relevant with respect to the resulting layout optimality as well as the speed of the layout method (see Figures 19 in the Appendix for more details). Figure 11 is a comparison of the Gaussian curves obtained for the different TFL circuit depths.

It is surprising that values of $C > 0$ are found given that these assume an upfront cost of re-routing some early gates. For shallow circuits, WRS prefers C values close to the end of the benchmark circuits, meaning that the last gates are more important than the others. This is in accordance with Figure 12, in which the green curve ($MaxDepth = 9$) is over the orange curve ($MaxDepth = 1$) for large values of $C > 0.75$ in the range of TFL depths from 5 to 30.

For deeper circuits, WRS sets the center C of the Gaussian to be closer to the beginning of the circuit. This is in accordance with Figure 12, in which the vertical distance between the green and orange curves is maximum for $C < 0.25$.

Figure 12 answers the following question: Considering the different values of $MaxDepth$, where should the Gaussian bell be placed relative to the start of the compiled circuit? Intuitively, this

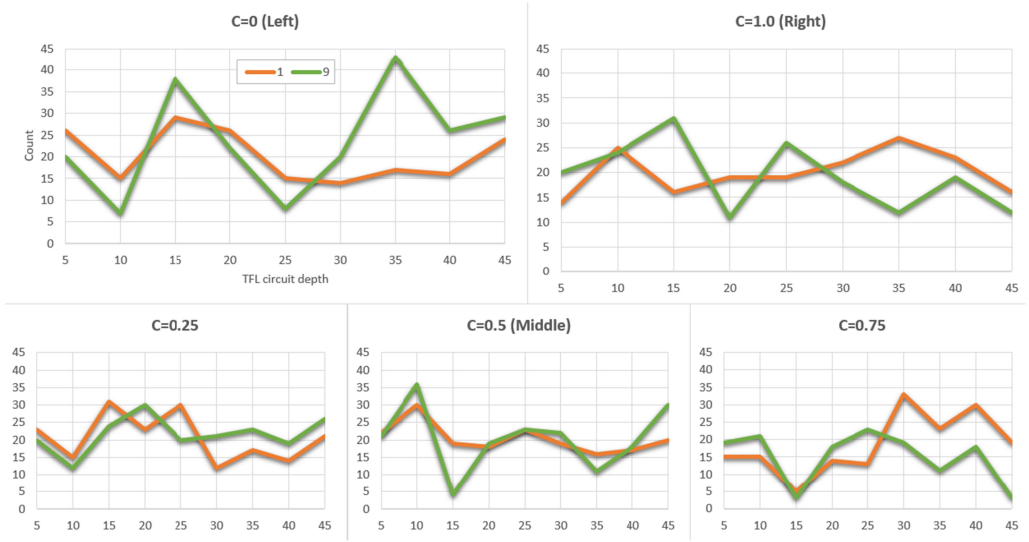

Fig. 12. The *Count* (the number of times that a parameter value was counted in the best 100 parameter combinations obtained for QXX) values to assess the influence of the Gaussian center C parameter on the *Ratio*. The Gaussian curve is automatically adapted to circuit depth. The horizontal axis in the plots represents the different types of QUEKO TFL circuits with known depths. The vertical axis is the *Count* value for the corresponding circuit types. There are two curves, the brown one for $MaxDepth = 1$ and the green one for $MaxDepth = 9$. For circuits with depths up to 30, the green $MaxDepth = 9$ performs better than brown $MaxDepth = 1$ (lower *Count* value) when the Gaussian curve is positioned to the right ($C = 1.0$ represents the end of the circuit)— the last gates are more important. For circuits of depths larger than 30, the opposite is true — the first gates are more important. The function $Count(P = v, |C_{TFL}|, MaxDepth)$ is the number of times that a parameter P bounded to v was counted in best 100-parameter combinations obtained for QXX executed for a particular value of $MaxDepth$ and for QUEKO TFL circuits C_{TFL} of depth $|C_{TFL}|$. More details can be found in the Appendix.

means to answer this question: Are the first gates more important than the last ones, or vice versa? Increasing values of C influence the performance of QXX with decreasing $MaxDepth$–the orange ($MaxDepth = 1$) and green ($MaxDepth = 9$) curves swap positions along the vertical axis, with the intersection between them being around TFl depth 30.

4 CONCLUSIONS

Scalable, configurable, and fast QCL methods are an imperative necessity. In the context of quantum computing clouds, continuous learning is a real possibility because a large batch of circuits is permanently sent and executed on mainframe-like machines. It is feasible to consider machine learning QCL methods for fast and accurate QCL.

We introduced QXX, a novel and parameterized QCL method. The QXX method uses a Gaussian function whose parameters determine the circuit region that influences most of the layout cost. The optimality of QXX is evaluated on the QUEKO benchmark circuits using the *Ratio* function, which expresses the factor by which the number of gates in the laid out circuit has increased. We illustrate the utility of QXX and its employed Gaussian. We show that the best results are achieved when the bell curve is non-trivially configured. QXX parameters are optimized using weighted random search (WRS).

To increase the speed of the parameter search, we train an MLP that learns QXX and apply WRS on the resulting QXX-MLP to crosscheck the quality of the WRS optimization and of the MLP model. This work brought empirical evidence that (1) the performance of QXX (resulting depth *Ratio* and speed) is on par with state-of-the-art QCL methods; (2) it is possible to learn the QXX method parameter values and the performance degradation is an acceptable trade-off with respect to achieved speed-up compared with WRS (which per se is orders of magnitude faster than exhaustive search); (3) WRS is finding parameter values that are in accordance with the very expensive exhaustive search.

We conjecture that, in general, new cost models are necessary to improve the performance of QCL methods. Using the Gaussian function, we confirmed the observation that the cost of compiling deep circuits is determined only by some of the gates (either at the start or the end of the circuit). From this perspective, the Gaussian function worked as a simplistic feature extraction. Future work will focus on more complex techniques to extract features to drive the QCL method.

APPENDICES

A BACKGROUND

From an abstract point of view, register connectivity is encoded as a graph. In the simplest form possible, the graph edges are not weighted. The graph edges are unique tuples of circuit registers (q_i, q_j).

For example, in Figure 13, the register connectivity of the circuit C_{in} is the red graph. The unique tuples of device registers are the edges of the device graph. For example, in Figure 13, the device register connectivity is the blue graph.

Figure 4(a) includes a quantum circuit example: the qubits are represented by horizontal wires, the two qubit gates are vertical lines, the control qubit is marked with a •, and the target with ⊕. The red graph from Figure 13 is obtained by replacing all wires from Figure 4 with vertices and the CNOTs with edges.

The aim of *parameter optimization* is to find the parameters of a given model that return the best performance of an objective function evaluated on a validation set. In simple terms, we want to find the model parameters that yield the best score on the validation set metric.

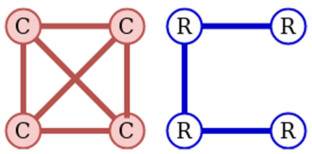

Fig. 13. The quantum circuit (red) has to be executed on the quantum device (blue). The circuit uses four qubits (vertices marked with C) and the hardware also has four registers (blue vertices). The circuit assumes that operations can be performed between arbitrary pairs of registers (the edges connecting the registers). The device supports operations only between a reduced set of register pairs.

In machine learning, we usually distinguish between the training parameters, which are adapted during the training phase, and the hyperparameters (or meta-parameters), which have to be specified before the learning phase [1]. In our case, since we do not train (adjust) inner parameters on specific training sets, we have only hyperparameters, which we will simply call here *parameters*.

Parameter optimization may include a budgeting choice of how many CPU cycles are to be spent on parameter exploration and how many CPU cycles are to be spent evaluating each parameter choice. Finding the "best" parameter configuration for a model is generally very time-consuming. There are two inherent causes of this inefficiency: one is related to the search space, which can

be a discrete domain. In its most general form, discrete optimization is NP-complete. The second cause is the potential high cost of evaluation of the objective function. We call this evaluation for one set of parameter values a *trial*.

There have been several recent attempts to optimize the parameters of quantum circuits. Machine learning optimizers tuned for usage on NISQ devices were recently reviewed by Lavrijsen et al. [16]. Several state-of-the-art gradient-free optimizers were compared that were capable of handling noisy, black-box, cost functions. They were stress-tested using a quantum circuit simulation environment with noise injection capabilities on individual gates. Their results indicate that specifically tuned optimizers are essential to obtaining valid results on quantum hardware. Parameter optimizers have a range of applications in quantum computing, including the Variational Quantum Eigensolver and Quantum Approximate Optimization algorithms. However, this approach has the same weaknesses as classical optimization —global optima are exponentially difficult to achieve [20].

Currently, the most common parameter optimization approaches are [1, 2, 6, 13] Grid Search, Random Search, derivative-free optimization (Nelder-Mead, Simulated Annealing, Evolutionary Algorithms, Particle Swarm Optimization), and Bayesian optimization (Gaussian Processes, Random Forest Regressions, Tree Parzen Estimators, etc.). Many software libraries are dedicated to parameter optimization or have parameter optimization capabilities: BayesianOptimization, Hyperopt-sklearn, Spearmint, Optunity, and more [1, 13]. Cloud-based highly integrated parameter optimizers are offered by companies such as Google (Google Cloud AutoML), Microsoft (Azure ML), and Amazon (SageMaker).

B TECHNICAL DETAILS

B.1 Weighted Random Search for Parameter Optimization

There are two computational complexity aspects that have to be addressed in order to find good QXX parameters: (1) reduce the search space and implicitly the number of trials; and (2) reduce the execution time of each trial. In general, the performance of a parameter optimizer is determined by [1, 2]

- F1, The execution time of each trial
- F2, The total number of trials — search space size
- F3, The performance of the search

Search space reduction (F2) and search strategy (F3) are interconnected and can be addressed in a sequence: F2 is a quantitative criterion (how many). For instance, we can first reduce the number of parameters (F2) and create more flexibility in the following stage for F3. In this work, we do not reduce the number of parameters.

F3 is a qualitative criterion (how "smart"). For instance, we can first rank and weight the parameters based on the functional analysis of the variance of the objective function (F3) and then reduce the number of trials (F2) by giving more chances to the more promising trials.

There is a trade-off between F2 and F3. To address these issues and reduce the search space, we use the following standard techniques:

- *Instance selection:* Reduce the dataset based on statistical sampling (relates to F1).
- *Feature selection* (relates to F1).
- *Parameter selection*: Select the most important parameters for optimization (relates to F2 and F3).
- *Parameter ranking*: Detect which parameters are more important for the model optimization and weight them (relates to F3 and F2).

- *Use additional objective functions:* Number of operations, optimization time, and so forth (relates to F3 and F2).

On average, the WRS method converges faster than RS [13]. WRS outperformed several state-of-the-art optimization methods: RS, Nelder-Mead, Particle Swarm Optimization, Sobol Sequences, Bayesian Optimization, and Tree-Structured Parzen Estimator [2].

In the RS approach, parameter optimization translates into the optimization of an objective function F of d variables by generating random values for its parameters and evaluating the function for each of these values [6]. The function computes some quality measure or score of the model (e.g., accuracy), and the variables correspond to the parameters. The assumption is to maximize F by executing a given number of trials.

Focusing on factor F3, the idea behind the WRS algorithm is that a subset of candidate values that already produced a good result has the potential, in combination with new values for the remaining dimensions, to lead to better values of the objective function. Instead of always generating new values (as in RS), the WRS algorithm uses for a certain number of parameters the so far best obtained values. The exact number of parameters that actually change at each iteration is controlled by the probabilities of change assigned to each parameter. WRS attempts to have a good coverage of the variation of the objective function and determines the parameter importance (the weight) by computing the variation of the objective function.

B.2 Training QXX-MLP

For the training and validation stages, we had 12 input features: circuit features (extracted using the Python package network) —max_page_rank,[3] nr_conn_comp, $edges$, $nodes$, $efficiency$,[4] $smetric$[5] — merged with QXX's parameters–$MaxDepth$, $MaxChildren$, B, C, $MovementFactor$, $EdgeCost$. We have chosen PageRank, Smetric, and the other metrics in order to capture as much information as possible about the circuits. The more features used for learning, the better the trained model is. The value to be predicted by the models was the $Ratio$ between the depth of the known optimal circuit and resulting circuit.

The performance of KNN, RF, and MLP was assessed through tenfold cross-validation (CV) over the whole dataset. In tenfold CV, the available dataset resulted in exhaustive search, which is split into 10 folds. Each fold is used, in turn, as a validation subset and the other nine folds are used for training. Finally, the ten performance scores obtained on the validation subsets are averaged and used as an estimation of the model's performance.

For each of the ten train/validation splits, the optimal values of their specific hyperparameters were sought via grid search using fivefold CV; the metric to be optimized was mean squared error. The models' specific hyperparameters and candidate values are given in Table 4. As both KNN and MLP are sensitive to the scales of the input data, we used a scaler to learn the ranges of input values from the train subsets. The learned ranges were subsequently used to scale the values on both train and validation subsets. We used the reference implementations from scikit–learn [27], version 0.22.1. Except for the hyperparameters in Table 4, the hyperparameters of all other models are kept to their defaults.

The lowest average values for mean squared error were obtained by RF, closely followed by MLP and KNN. From RF and MLP, we preferred the latter due to its higher inference speed and smaller memory footprint.

[3]Ranking nodes based on the structure of the incoming links.
[4]The efficiency of a pair of nodes in a graph is the multiplicative inverse of the shortest path distance between the nodes.
[5]The sum of the node degree products for every graph edge.

Table 4. Hyperparameter Names and Candidate Values

Model	Hyperparameter	Values
KNN	Neighbors sought	$\{2, \ldots, 8\}$
KNN	Minkowski metric's p	$\{1, 2\}$
MLP	Hidden layer's size	$\{3, 10, 20, 50, 100\}$
MLP	Activation function	ReLU, tanh
RF	Maximum depth of a tree	$\{2, 3, 4, 5, auto\}$
RF	Number of trees	$\{2, 5, 10, 20\}$

The final MLP model was prepared by doing a final grid search for the optimal hyperparameters from Table 4, choosing the best model through fivefold CV. The resultant network is as follows. The input layer has 12 nodes, fully connected with the (only) hidden layer, which hosts 100 neurons. This layer is fully connected with the output neuron. ReLU and identity were used as activation functions for the hidden and output layers, respectively. For the hidden layer, both the number of neurons and the activation function were optimized through grid search.

C EVALUATION

We use the exhaustive search raw data and introduce metrics to evaluate the parameter importance of *GDepth*. The function has six parameters (see Section 2.1); we analyzed their *individual* importance using WRS (see Section 2.5).

For example, Table 1 lists the importances (weights) of the individual QXX parameters. These weights were computed under the strong (naïve) independence assumption between the parameters. Usually, parameters are statistically correlated; we prefer a finer-grained understanding of the QXX's performance.

To compare how parameter pairs influence the *Ratio* function, we introduce two metrics called *Count* and *Rank*. To compute these metrics, we execute an exhaustive search for the three values of *MaxDepth* and consider all of the parameter configurations from Table 1 — a six-dimensional grid search.

For a given value of *MaxDepth* and a parameter configuration (all of the other five parameters), we average the resulting depth of the compiled circuit, C_{out}, over the circuits existing in the *TFL* benchmark. From the total 1,485 averages, we sample the lowest 100 values, leading to an approximate 7% sampling rate, $\frac{100}{1485} \approx 0.067$, from the total number of parameter configurations.

The function

$$Count(P = v, |C_{TFL}|, MaxDepth)$$

is the number of times that a parameter P bounded to v was counted in the best 100-parameter combinations obtained for QXX executed for a particular value of *MaxDepth* for QUEKO TFL circuits C_{TFL} of depth $|C_{TFL}|$. For example, $Count(B = 0, 1, 30)$ is the number of parameter configurations where $B = 0$ and QXX was used with a search tree of $MaxDepth = 30$ to lay out TFL circuits of depth 1. The *Count* function can compare how, for different circuit depths, *MaxDepth* influences the optimal values of P.

The *Rank* function aggregates how different values of P are ranked against each other when considering different TFL circuit depths: for the same $|C_{TFL}|$, higher rank values are better. The *Rank* is used to suggest parameter value ranges:

$$Rank(P, |C_{TFL}|) = \sum_{\substack{i=0 \\ MaxDepth_i \in \{1,5,9\}}}^{2} Count(P, |C_{TFL}|, MaxDepth_i). \tag{3}$$

C.1 Optimum Parameters vs. Circuit Depth

To speed up QXX, we are interested in finding parameter values that keep execution times as low as possible without massively impacting the obtained *Ratio* values. According to WRS, *MaxDepth* is one of the most important parameters. However, it is not immediately obvious if it is possible to achieve optimal *Ratio* values using low *MaxDepth* values. In the following, we form parameter pairs between $MaxDepth \in \{1, 5, 9\}$ and the other five QXX parameters.

The best layouts are obtained for large values of *MovementFactor* and for shallow circuits; the *EdgeCost* has to be preferably low for the latter. These observations explain the `StochasticSwap` gate scheduling method (see Figure 2). The *MovemenFactor* value shows that the scheduler prefers to move a single qubit on the coupling graph.

The way that *EdgeCost* values are influenced by the TFL depth indicates that there is a relation between the number of gates in the circuit and the number of edges in the coupling graph. This relation could be modelled through a density function such as $\frac{nr\_gates}{nr\_edges}$. To the best of our knowledge, the effect of this density function on the coupling graph edge weights has not yet been investigated in the literature.

Figures 14 and 15 show how randomly chosen parameter configurations influence the depth *Ratio* of shallow circuits with depths up to 25.

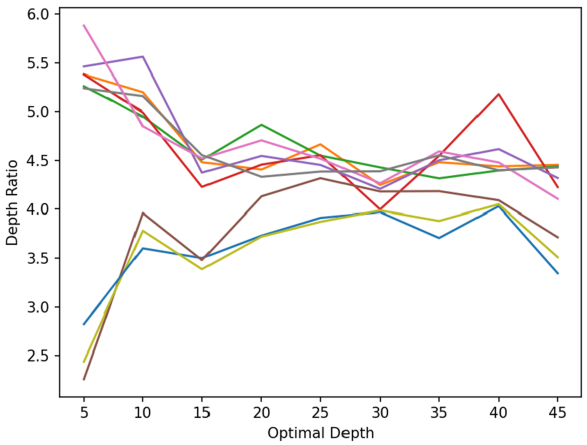

Fig. 14. Random parameter configurations and their influence on TFL circuit depth optimality. The axes have the same interpretation as that in Figure 8. Each line corresponds to a random parameter value configuration.

Figure 17 (optimal parameter configuration irrespective of the value of *MaxDepth*) is supported by the results from Figures 18, 19 (discussed in the next section —the Gaussian influences *MaxDepth*), and 20. The best layouts are obtained for large values of *MovementFactor* and for shallow circuits; the *EdgeCost* has to be preferably low for the latter.

We conjecture that the variability in Figures 18–20 is mostly due to the correlations that exist between the search space size and the timeouts. For example, it can be seen that, with a small exception for medium-depth circuits, the best performing value of *MaxChildren* is correlated with the one of *MaxDepth*. The number of timeouts we obtained for high values of $MaxDepth = 9$ is an indication of this observation. We conclude that it does not seem to be necessary to increase the breadth of the search if the depth of the search tree is shallow.

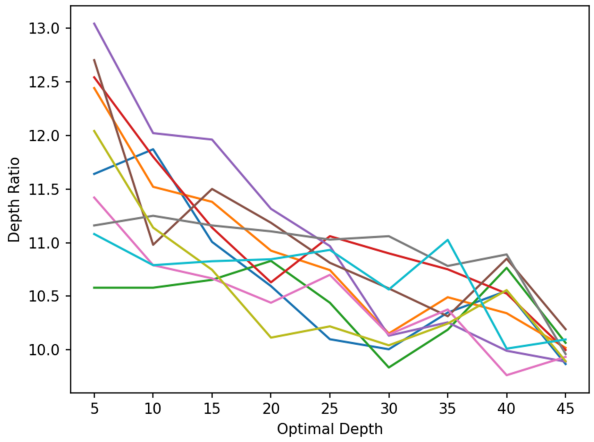

Fig. 15. Random parameter configurations and their influence on QSE (supremacy experiment) circuit depth optimality. The axes have the same interpretation as in Figure 8. Each line corresponds to a random parameter value configuration.

C.2 GDepth Parameters

Figure 19 answers the following question. What is the best performing value of B considering the depth ($MaxDepth$) of the QXX search space? *How many gates of a circuit* are important is answered by the value of B. Out of the 11 used values (see Table 1), the first 4 $(0.0, 0.2, 0.4, 0.6)$ are considered being $Flat$, whereas the last four $(1.4, 1.6, 1.8, 2.0)$ are $Narrow$. The remaining three values are $Wide$. Due to these ranges, the values on the vertical axis are normalized to 1. The width of the bell curve from Figure 5 is configurable and indicative of which circuit gates are the most important with regard to $Ratio$ optimality.

The number of preferred gates seems to be a function of the used timeout. The more time spent searching for optimal parameters, the thinner the Gaussian bell. As observed in Tables 3 and 2, the number of timeouts for $MaxDepth = 9$ is high such that the B values for timeout at 20 s seem not to obey the scaling observed for the other timeouts. This confirms the results of the exhaustive search as presented in Figure 19 in the Appendix, where the curve for $MaxDepth = 9$ has a high variation along the vertical axis.

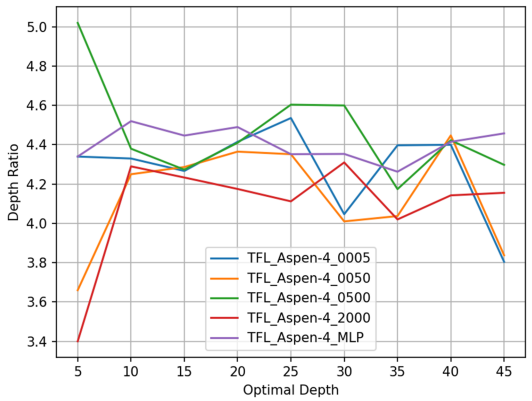

Fig. 16. Precision: Laying out TFL circuits with depths up to 45.

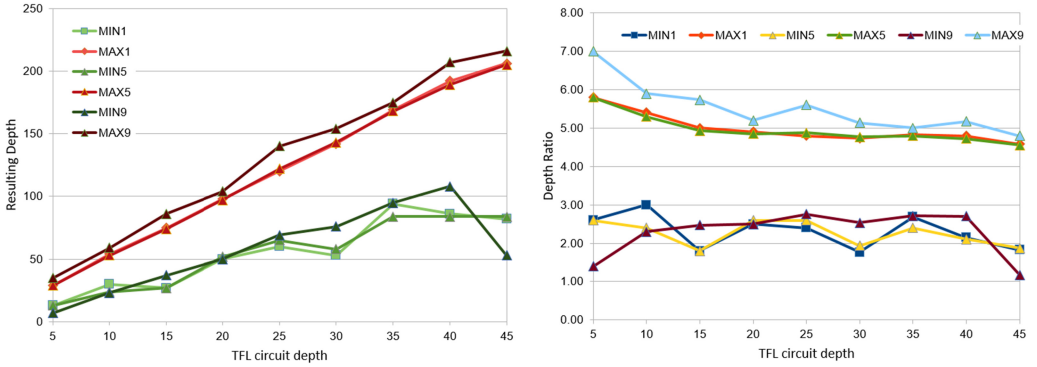

Fig. 17. It should be possible to find an optimal parameter configuration irrespective of the value of *MaxDepth*: Plotting the best depth (left) and *Ratio* (right) obtained per TFL circuit depth and *MaxDepth* parameter. The red (MAX1, MAX5, MAX9) and green (MIN1, MIN5, MIN9) curves are the highest and lowest depths achieved for each *MaxDepth* value (1, 5, 9).

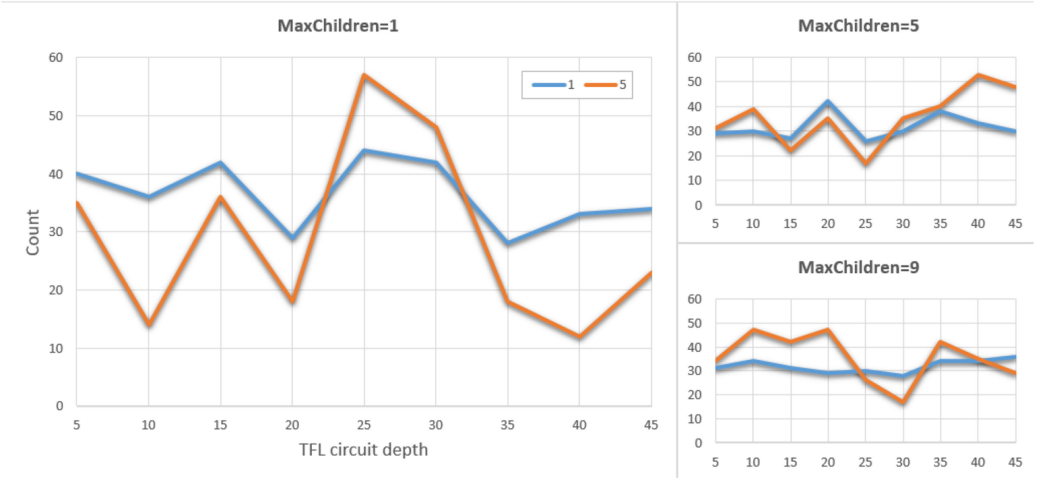

Fig. 18. Exhaustive Search: The *Count* values to assess the influence of the *MaxChildren* parameter on the *Ratio*. For this, we plot the *Count* obtained for two values of *MaxDepth*: blue for 1 and orange for 5. Due to the high number of timeouts during the exhaustive search with $MaxDepth = 9$, the corresponding curve was not plotted. For $MaxChildren = 1$, for almost all circuits (with the exception of depth 20, 25, and 30), the best *Ratios* are obtained for $MaxDepth = 1$. As *MaxChildren* is increased, the better results are achieved by larger *MaxDepth* — the orange curve is above the blue one.

This diagram in Figure 16 is similar to the one from Figure 10, but the WRS was executed on all benchmarks — parameters were chosen to be compatible with a much larger range of circuits. The curves do not seem to converge as well as they did for the depth 25 circuits.

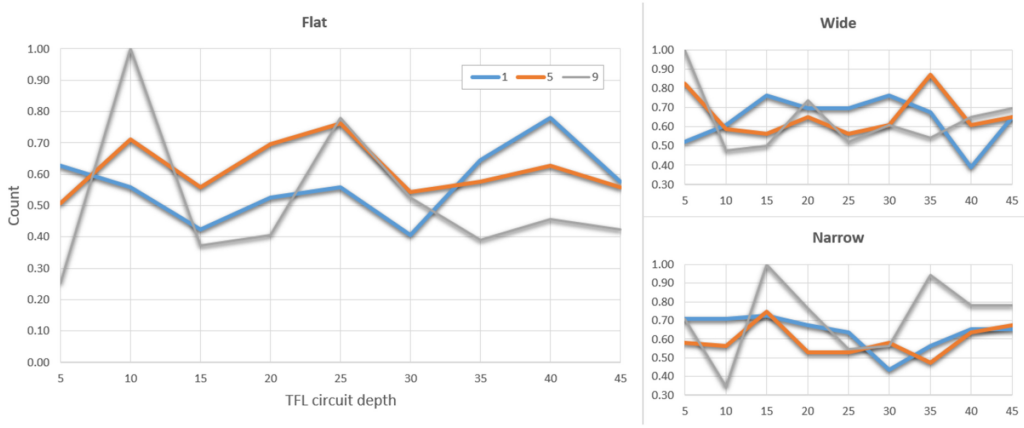

Fig. 19. The value of B can both improve the mapping (higher *Count*) and speed up the search due to lower *MaxDepth*: Normalized *Count* values to assess the influence of the B parameter on the *Ratio*. For this, we plot the *Count* obtained for three values of *MaxDepth*: blue for 1, orange for 5, and gray for 9. Better results are obtained with decreasing *MaxDepth*, as the value of B is increasing. For example, in the left panel, *Flat* performs better for $MaxDepth = 5$ for circuits with a depth up to 30. Moreover, *Wide* achieves the best depth ratios for $MaxDepth = 1$.

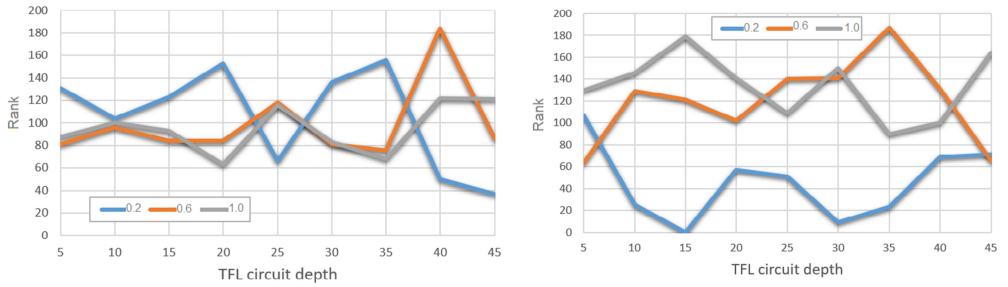

Fig. 20. Exhaustive Search. Left: *Rank* values for the *EdgeCost* parameter — up to circuits of depths 35 QXX achieves better *Ratio* values for edge costs of 0.2, whereas for deeper circuits, a higher value of the edge cost delivers better *Ratio* values. Right: *Rank* values for the *MovementFactor* parameter —higher parameter values perform significantly better than lower values, meaning that it is better to move a single qubit instead of two across the graph (see Figure 7).

ACKNOWLEDGMENTS

We are grateful to Bochen Tan for his feedback on a first version of this manuscript, explaining the QUEKO benchmarks and offering the scripts to generate and plot the presented results.

REFERENCES

[1] Razvan Andonie. 2019. Hyperparameter optimization in learning systems. *Journal of Membrane Computing* 1, 4 (2019), 279–291. https://doi.org/10.1007/s41965-019-00023-0

[2] Razvan Andonie and Adrian-Catalin Florea. 2020. Weighted random search for CNN hyperparameter optimization. *International Journal of Computers, Communications and Control* 15, 2 (2020). https://doi.org/10.15837/ijccc.2020.2.3868

[3] Pablo Andrés-Martínez and Chris Heunen. 2019. Automated distribution of quantum circuits via hypergraph partitioning. *Physical Review A* 100, 3 (2019), 032308.

[4] Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. 1995. Elementary gates for quantum computation. *Physical Review A* 52, 5 (1995), 3457.

[5] Yoshua Bengio. 2009. Learning deep architectures for AI. *Foundations and Trends in Machine Learning* 2, 1 (Jan. 2009), 1–127. https://doi.org/10.1561/2200000006

[6] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for hyper-parameter optimization.. In *NIPS*, John Shawe-Taylor, Richard S. Zemel, Peter L. Bartlett, Fernando C. N. Pereira, and Kilian Q. Weinberger (Eds.). 2546–2554. http://dblp.uni-trier.de/db/conf/nips/nips2011.html.

[7] Adi Botea, Akihiro Kishimoto, and Radu Marinescu. 2018. On the complexity of quantum circuit compilation. In *11th Annual Symposium on Combinatorial Search*.

[8] Rui Chao and Ben W. Reichardt. 2018. Quantum error correction with only two extra qubits. *Physical Review Letters* 121, 5 (2018), 050502.

[9] Andrew M. Childs, Eddie Schoute, and Cem M. Unsal. 2019. Circuit transformations for quantum architectures. In *14th Conference on the Theory of Quantum Computation, Communication and Cryptography*.

[10] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John Van De Wetering. 2020. Graph-theoretic simplification of quantum circuits with the ZX-calculus. *Quantum* 4 (2020), 279.

[11] Suguru Endo, Simon C. Benjamin, and Ying Li. 2018. Practical quantum error mitigation for near-future applications. *Physical Review X* 8, 3 (2018), 031027.

[12] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. 2014. Do we need hundreds of classifiers to solve real world classification problems? *Journal of Machine Learning Research* 15, 1 (Jan. 2014), 3133–3181.

[13] Adrian-Catalin Florea and Razvan Andonie. 2019. Weighted random search for hyperparameter optimization. *International Journal of Computers, Communications and Control* 14, 2 (2019), 154–169. https://doi.org/10.15837/ijccc.2019.2.3514

[14] A. G. Fowler, S. J. Devitt, and L. C. L. Hollenberg. 2004. Implementation of Shor's algorithm on a linear nearest neighbour qubit array. *Quantum Information and Computation.* 4, quant-ph/0402196 (2004), 237–251.

[15] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. 2014. An efficient approach for assessing hyperparameter importance. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32* (Beijing, China) *(ICML'14)*. JMLR.org, I-754-I-762.

[16] Wim Lavrijsen, Ana Tudor, Juliane Müller, Costin Iancu, and Wibe de Jong. 2020. Classical optimizers for noisy intermediate-scale quantum devices. *arXiv preprint arXiv:2004.03004* (2020).

[17] Ang Li and Sriram Krishnamoorthy. 2020. QASMBench: A low-level QASM benchmark suite for NISQ evaluation and simulation. *arXiv preprint arXiv:2005.13018* (2020).

[18] Gushu Li, Yufei Ding, and Yuan Xie. 2019. Tackling the qubit mapping problem for NISQ-era quantum devices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 1001–1014.

[19] Dmitri Maslov, Gerhard W. Dueck, D. Michael Miller, and Camille Negrevergne. 2008. Quantum circuit simplification and level compaction. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27, 3 (2008), 436–444.

[20] Jarrod R. McClean, Sergio Boixo, Vadim N. Smelyanskiy, Ryan Babbush, and Hartmut Neven. 2018. Barren plateaus in quantum neural network training landscapes. *Nature Communications* 9, 1 (2018), 1–6.

[21] Tom M. Mitchell. 1980. *The Need for Biases in Learning Generalizations*. Technical Report. Rutgers University, New Brunswick, NJ. http://dml.cs.byu.edu/~cgc/docs/mldm_tools/Reading/Need.%20for%20Bias.pdf.

[22] Prakash Murali, David C. McKay, Margaret Martonosi, and Ali Javadi-Abhari. 2020. Software mitigation of crosstalk on noisy intermediate-scale quantum computers. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 1001–1016.

[23] Beatrice Nash, Vlad Gheorghiu, and Michele Mosca. 2020. Quantum circuit optimizations for NISQ architectures. *Quantum Science and Technology* 5, 2 (2020), 025010.

[24] Shin Nishio, Yulu Pan, Takahiko Satoh, Hideharu Amano, and Rodney Van Meter. 2019. Extracting success from IBM's 20-qubit machines using error-aware compilation. *arXiv preprint arXiv:1903.10963* (2019).

[25] Alexandru Paler. 2019. On the influence of initial qubit placement during NISQ circuit compilation. In *International Workshop on Quantum Technology and Optimization Problems*. Springer, 207–217.

[26] Sam Pallister. 2020. A Jordan-Wigner gadget that reduces T count by more than 6x for quantum chemistry applications. *arXiv preprint arXiv:2004.05117* (2020).

[27] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[28] Matteo G. Pozzi, Steven J. Herbert, Akash Sengupta, and Robert D. Mullins. 2020. Using reinforcement learning to perform qubit routing in quantum compilers. *arXiv preprint arXiv:2007.15957* (2020).
[29] John Preskill. 2018. Quantum computing in the NISQ era and beyond. *Quantum* 2 (2018), 79.
[30] Mehdi Saeedi and Igor L. Markov. 2013. Synthesis and optimization of reversible circuits—a survey. *ACM Computing Surveys* 45, 2 (2013), 1–34.
[31] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Caroline Collange, and Fernando Magno Quintão Pereira. 2019. Qubit allocation as a combination of subgraph isomorphism and token swapping. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–29.
[32] Marcos Yukio Siraichi, Vinícius Fernandes dos Santos, Sylvain Collange, and Fernando Magno Quintão Pereira. 2018. Qubit allocation. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. 113–125.
[33] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 2020. t| ket>: A retargetable compiler for NISQ devices. *Quantum Science and Technology* (2020).
[34] Bochen Tan and Jason Cong. 2020. Optimal layout synthesis for quantum computing. In *2020 IEEE/ACM International Conference on Computer Aided Design (ICCAD'20)*. IEEE, 1–9.
[35] B. Tan and J. Cong. 2020. Optimality study of existing quantum computing layout synthesis tools. *IEEE Transactions on Computers* early access (2020), 1–1.
[36] Swamit S. Tannu and Moinuddin K. Qureshi. 2019. Not all qubits are created equal: A case for variability-aware policies for NISQ-era quantum computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 987–999.
[37] Ellis Wilson, Sudhakar Singh, and Frank Mueller. 2020. Just-in-time quantum circuit transpilation reduces noise. *arXiv preprint arXiv:2005.12820* (2020).
[38] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. 2007. Top 10 algorithms in data mining. *Knowledge and Information Systems* 14, 1 (Dec. 2007), 1–37. https://doi.org/10.1007/s10115-007-0114-2
[39] Chi Zhang, Ari B. Hayes, Longfei Qiu, Yuwei Jin, Yanhao Chen, and Eddy Z. Zhang. 2021. Time-optimal qubit mapping. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 360–374.
[40] Yuan-Hang Zhang, Pei-Lin Zheng, Yi Zhang, and Dong-Ling Deng. 2020. Topological quantum compiling with reinforcement learning. *Physical Review Letters* 125, 17 (2020), 170501.
[41] Alwin Zulehner, Alexandru Paler, and Robert Wille. 2018. An efficient methodology for mapping quantum circuits to the IBM QX architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 7 (2018), 1226–1236.

Received 16 November 2021; revised 26 July 2022; accepted 24 September 2022